HumbleDB Documentation

Release 6.0.0

Jake Alheid

Mar 16, 2018

Contents

HumbleDB is a thin wrapper around Pymongo for MongoDB that is designed to make working with flexible schema documents easy and readable.

- **Readable:** Short document keys can be mapped to long attribute names to keep document size small and efficient, while providing completely readble code.
- **Flexible:** A `Document` is also a dictionary, so you have maximum flexibility in your schema - there are no restricitons.
- **Concurrent:** HumbleDB is thread-safe and greenlet-safe and provides a connection paradigm which minimizes the amount of time a socket is used from the connection pool.

Why HumbleDB?

With so many excellent MongoDB Python ORMs, ODMs, and interfaces out there such as MongoEngine, MongoAlchemy, MiniMongo, and of course Pymongo itself, why would I write yet another one? The answer is, as usual, that those excellent software projects didn't do exactly what I want. I enjoy that Python and MongoDB are both completely flexible in their ability to be modified on the fly, and be adapted very easily to unique problems. I feel like Python and MongoDB are almost mirrors of each other in that way - a perfect pair. After all, at their heart, both Python and MongoDB objects are just dictionaries.

But there's a problem with MongoDB, which is that keys are repeated in every document. When you create a MongoDB schema that's verbose, readable, and easily understandable, you end up with key names that take up more space than the data you're trying to store! When you use short, single character or double character key names, you save lots of space, but your schema becomes almost completely unintelligible.

Of course, MongoEngine and MongoAlchemy let you map your attributes to different or shorter key names, but they also are heavy - instantiating a large number of ORM objects can be 10x or 100x slower than doing the same query with Pymongo.

This is the problem that HumbleDB tries to solve - it provides a clean, readable interface for the shortest keys you can use, saving your database RAM for more documents, using resources more efficently, and exposing all the power and flexibility of Pymongo underneath. It's called "Humble" because "humble" is another word for "small", which is what it tries to be.

Example: The humblest document

```
>>> from humbledb import Mongo, Document
>>> class HumbleDoc(Document):
...     # config_database and config_collection are required configuration
...     # for every Document subclass
...     config_database = 'humble'
...     config_collection = 'examples'
...     # The 'd' document key is mapped to the description attribute
...     description = 'd'
...     # Same for the 'v' key and the value attribute
...     value = 'v'
...
>>> # Create a new empty document
>>> doc = HumbleDoc()
>>> # Set some values in the document
>>> doc.description = "A humble example"
>>> doc.value = 3.14159265358979
>>> # with Mongo: tells HumbleDoc to use the default MongoDB connection
>>> with Mongo:
...     # The insert method (and others) are the same as the pymongo methods
```

```
...     HumbleDoc.insert(doc)
...
>>> # Newly created documents will have their _id field set, and you can see
>>> # what the raw document would look like in MongoDB
>>> doc
HumbleDoc({'_id': ObjectId('50c3e72c6112798c3bcde02d'),
   'd': 'A humble example', 'v': 3.14159265358979})
```

What's going on here?

- `config_database = 'humble'` - This tells the document that it's stored in the `'humble'` database.
- `config_collection = 'examples'` - This tells the document that it's part of the `'examples'` collection.
- `description = 'd'` - This maps the `description` attribute to the document key `'d'` (see *Attribute Mapping*).
- `value = 'v'` - This maps the `value` attribute to the document key `'v'`.
- `with Mongo:` - This *Mongo* context manager tells the document which MongoDB connection to use (see *Connecting to MongoDB*).
- `HumbleDoc.insert(doc)` - This inserts `doc` into the HumbleDoc collection (see *Working with a Collection*).

Download and Install

Install the latest stable release via PyPI (`pip install -U humbledb`). HumbleDB requires Pymongo (`>=2.2.1`), Pytool (`>=1.1.0`) and Pyconfig, and runs on Python 2.7 or newer (though maybe not 3.x).

The code for HumbleDB can be found on github.

CHAPTER 1

User's Guide

1.1 Tutorial

This tutorial will introduce you to the concepts and features of the HumbleDB micro-ODM and covers basic and advanced usage. It will teach you how to install HumbleDB, how to create `Document` and `Mongo` subclasses that fit your needs, and the HumbleDB way of manipulating documents. The *API Documentation* covers more details, but has less explanation.

1.1.1 Installation

HumbleDB requires Pymongo (`>=2.0.1`), Pytool (`>=3.0.1`) and Pyconfig. These are installed for you automatically when you install HumbleDB via pip or easy_install.

```
$ pip install -U humbledb    # preferred
$ easy_install -U humbledb   # without pip
```

To get the latest and greatest development version of HumbleDB, clone the code via github and install:

```
$ git clone http://github.com/shakefu/humbledb.git
$ cd humbledb
$ python setup.py install
```

1.1.2 Quickstart: The humblest document

This tutorial assumes you already have HumbleDB *installed* and working. Let's start with a very basic `Document`:

```
from humbledb import Document

class HumbleDoc(Document):
    config_database = 'humble'
    config_collection = 'example'
```

```
    description = 'd'
    value = 'v'
```

The `config_database` and `config_collection` attributes are required to tell the HumbleDoc class which database and collection that it lives in.

HumbleDB's basic attribute access works by looking for class attributes whose values are `str` or `unicode` objects, and mapping those values to the attribute names given.

We see above that the `description` attribute is mapped to the `'d'` key, and the `value` attribute is mapped to the `'v'` key. These keys can bet set by assigning their attributes:

```
doc = HumbleDoc()
doc.description = "The humblest document"
doc.value = 3.14
```

In addition to any keys you specify, every Document is given a `_id` attribute which maps to the `'_id'` key.

Accessing Keys and Values

When you access a mapped key on your document class, it returns the key for you, so you can reference your short key names more readably:

```
>>> HumbleDoc.description
'd'
>>> HumbleDoc.value
'v'
```

When querying or setting keys you should use these attributes, rather than the short key names, for more understandable code:

```
HumbleDoc.find({HumbleDoc._id: 'example'})

HumbleDoc.update({HumbleDoc._id: 'example'},
        {'$set': {HumbleDoc.value: 4}})

HumbleDoc.find_one({HumbleDoc.value: 4})
```

When these same attributes are accessed on a document instance, they return the current value of that key:

```
>>> with Mongo:
...     doc = HumbleDoc.find_one()
...
>>> doc.description
u'A humble example'
>>> doc.value
3.14159265358979
```

Connecting to MongoDB

The *Mongo* class is a context manager which takes care of establishing a `pymongo.connection.Connection` instance for you. By default, the Mongo class will connect to `'localhost'`, port `27017` (see *Configuring Connections* if you need different settings).

When doing any operation that hits the database, you always need to use the `with` statement with *Mongo* (or a *subclass*):

```
with Mongo:
    HumbleDoc.insert(doc)
    docs = HumbleDoc.find({HumbleDoc.value: {'$gt': 3}})
```

The Mongo context ensures any operations you do are within a single request (for consistency) and that the socket is released back to the connection pool as soon as possible (for concurrency).

Working with a Collection

For your convenience, all of the `pymongo.collection.Collection` methods are mapped onto your document class (but not onto class instances). Because these methods imply using the MongoDB connection, they're only available within a *Mongo* context.

Within a *Mongo* context, all the `Collection` methods are available on your document class:

```
with Mongo:
    doc = HumbleDoc.find_one()
```

Without a context, a `RuntimeError` is raised:

```
>>> HumbleDoc.insert
Traceback (most recent call last):
  ...
RuntimeError: 'collection' not available without context
```

Document instances do not have collection methods and will raise a `AttributeError`:

```
>>> doc.insert
Traceback (most recent call last):
  ...
AttributeError: 'HumbleDoc' object has no attribute 'insert'
```

1.1.3 Working with Documents

Document subclasses provide a clean attribute oriented interface to your collection's documents, but at their heart, they're just dictionaries. The only required attributes on a document are `config_database`, and `config_collection`.

Example: Documents are dictionaries

```
from humbledb import Document

class Basic(Document):
    # These are required
    config_database = 'humble'
    config_collection = 'basic'

# Documents are dictionaries
doc = Basic()
doc['my-key'] = 'Hello'
```

```
# Documents can be initialized with dictionaries
doc = Basic({'key': 'value'})
doc['key'] == 'value'

# You also can query using arbitrary keys
with Mongo:
   docs = Basic.find({'my-key': {'$exists': True}})
```

Attribute Mapping

Attributes are created by assigning string key in your class definition to attribute names. These attributes are mapped to the dictionary keys internally. In addition to any attributes you specify, a `_id` attribute is always available.

Attributes names with a leading underscore (_) are not mapped to keys.

When an mapped attribute is accessed from the class, the short key is returned, and when accessed from an instance, that instance's value for that key is returned.

If a document doesn't have a value set for a mapped attribute, `{}` is returned (rather than raising an `AttributeError`), so you can easily check whether an attribute exists. This also allows you to create embedded documents whose keys are not mapped.

When a document is inserted, its `_id` attribute is set to the created `ObjectId`, if it wasn't already set.

Changed in version 3.0: Unset attributes on a Document return `{}` rather than `None`

Example: Attribute mapping

```
class MyDoc(Document):
    config_database = 'humble'
    config_collection = 'mydoc'

    # Keys are mapped to attributes
    my_attribute = 'my_key'

    # Private names are ignored
    _my_str = 'private'

    # Non string values are ignored
    an_int = 1

doc = MyDoc()

# Unset attributes return {}, which evaluates to False
if not doc.my_attribute:
   # Attribute assignment works like normal
   doc.my_attribute = 'Hello'
   # Attribute deletion works like normal too
   del doc.my_attribute

# You can explicitly check if you expect to assign values which also
# evaluate to False
if doc.my_attribute == {}:
   doc.my_attribute = 'Hello World'
```

```
# Class attributes return the key
MyDoc.my_attribute # 'my_key'

# Instance attributes return the value
doc.my_attribute # 'Hello World'

# Private names aren't mapped
doc._my_str # 'private'

# Neither are non-string values
doc.an_int # 1

doc._id # {}

if not doc._id:
   with Mongo:
      MyDoc.insert(doc)

doc._id # ObjectId(...)
```

Giving Documents Default Values

Sometimes it's useful to allow a document to provide a default value for a missing key. HumbleDB provides a convenient syntax for specifying both persisted and unpersisted default values.

New in version 5.2.0.

Both persisted and unpersisted values are declared by assigning a 2-tuple to an attribute where the first item is a string document key, and the second item is a default value.

If the second item is a callable, then that indicates that the value should be persisted. It will be called once on first access or first save and persisted with the document.

By convention, HumbleDB uses the no-parentheses form of tuple declaration when declaring default values.

Lastly, when serializing to JSON via the *for_json()* method, the default values will be inserted into the resulting JSON.

- `attr = 'key', value` - If `value` is not callable, provide an unpersisted default value, which is available through attribute access only, and not part of the document.

  Examples:

```
class MyDoc(Document):
    config_database = 'humble'
    config_collection = 'docs'

    attr = 'a'  # Regular mapping

    # Any non-callable as the second item is used as a default value
    truth = 't', False
    number = 'u', 1
    name = 'n', "humble"

    # Any expression may be used to declare an unpersisted default
    some_value = 'v', func()  # Where func() returns a non-callable
    one_day = 'd', 60*60*24
```

- `attr = 'key', default` - Where `default` is a callable, provide a persisted default value, which will become part of the document the first time it's accessed via attribute or inserted or saved. Updates do not trigger default values.

  Examples:

```
class MyDoc(Document):
    config_database = 'humble'
    config_collection = 'docs'

    attr = 'a'  # Regular mapping

    # A callable makes this a persisted default - as soon as it's
    # accessed, it is assigned to the doc, or when saved or inserted

    # Here, my_func() will be called without arguments to provide a
    # default value which will be saved with the document
    my_value = 'v', my_func

    # Here, uuid.uuid4() will be called - note the lack of parens, we're
    # not calling it (which would return a value) - we're providing the
    # function itself
    uid = 'u', uuid.uuid4

    # Here, the datetime.now() function will be called on save which is
    # a convenient way to provide a creation timestamp
    created = 'c', datetime.now
```

Example: Declaring default values for keys

```
class BlogPost(Document):
    config_database = 'humble'
    config_collection = 'posts'

    title = 't'  # Normal attribute
    author = 'a'  # Still normal

    public = 'p', False  # Default value that is not saved automatically

    created = 'c', datetime.now  # This will be saved to the document


# Create a post
post = BlogPost()
post.title = "A post"
post.author = "HumbleDB"

# The default value is provided on the document object when accessed via an
# attribute
post.public  # False

# But it isn't part of the document itself, so dict key access will raise a
# KeyError
post[BlogPost.public]  # KeyError
post['p']  # KeyError

# After the first access, the persisted default is called and the returned
```

```
# value is stored in the document and will be consistent from then on
post.created  # datetime(2014, 2, 14, 6, 59, 0)

# Saving the post would also call the persisted default and store the value

# Save the post and reload it
with Mongo:
    _id = BlogPost.save(post)
    post = BlogPost.find_one(_id)

# The unpersisted default value is not stored with the document
post  # BlogPost({'_id': ObjectId(), 't': "A post", 'a': "HumbleDB",
      #         'c': datetime(2014, 2, 14, 6, 59, 0)})

# But it's still available on the document object
post.public  # False

# Once modified, the value will be saved and retrieved like normal
post.public = True
with Mongo:
    BlogPost.save(post)

post  #  BlogPost({'_id': ObjectId(), 't': "A post", 'a': "HumbleDB",
      #         'p': True, 'c': datetime(2014, 2, 14, 6, 59, 0)})
```

Introspecting Documents

Sometimes it's useful to be able to introspect a document schema to find out what attributes or keys are mapped. To do this, HumbleDB provides two methods, *mapped_keys()* and *mapped_attributes()*. These methods will return all the mapped dictionary keys and document attributes, respectively, excluding the `_id` key/attribute.

Example: Introspecting documents

```
class MyDoc(Document):
    config_database = 'humble'
    config_collection = 'mydoc'

    my_attr = 'k'
    other_attr = 'o'

MyDoc.mapped_keys() # ['k', 'o']
MyDoc.mapped_attributes() # ['my_attr', 'other_attr']

# Mapping an arbitrary dict, while restricting keys
some_dict = {'spam': 'ham', 'k': True, 'o': "Hello"}

# Create an empty doc
doc = MyDoc()

# Iterate over the mapped keys, assigning common keys
for key in MyDoc.mapped_keys():
    if key in some_dict:
        doc[key] = some_dict[key]
```

1.1.4 Embedding Documents

Attribute mapping to embedded documents is done via the `Embed` class. Because a document is also a dictionary, using Embed is totally optional, but helps keep your code more readable.

An embedded document can be assigned mapped attributes, just like a document.

Mapped embedded document attributes that aren't assigned return an empty dictionary when accessed.

When accessed via the class, an embedded document attribute returns the full dot-notation key name. If you want just the key name of the attribute, it is available as the attribute *key*.

Example: Embedded documents and nested documents

```
from humbledb import Document, Embed

class Example(Document):
    config_database = 'humble'
    config_collection = 'embed'

    # Any mapped attribute can be used as an embedded document
    my_attribute = 'my_attr'

    # This maps embedded_doc to embed_key
    embedded_doc = Embed('embed_key')

    # This maps embedded_doc.my_attribute to embed_key.my_key
    embedded_doc.my_attribute = 'my_key'

    # This is a nested embedded document
    embedded_doc.nested_doc = Embed('nested_key')
    embedded_doc.nested_doc.value = 'val'


empty_doc = Example()

# Empty or missing embedded documents are returned as an empty dictionary
empty_doc.embedded_doc # {}

# Missing attributes are returned as {} so you can have unmapped subdocs
empty_doc.embedded_doc.my_attribute # {}

doc = Example()

# You can use attributes as unmapped embedded documents
doc.my_attribute['embedded_key'] = 'Hello'
doc.my_attribute # {'embedded_key': 'Hello'}
doc # {'my_attr': {'embedded_key': 'Hello'}}

# Attribute assignment works like normal
if not doc.embedded_doc:
   doc.embedded_doc.my_attribute = "A Fish"

doc.embedded_doc.nested_doc.value = 42

# Class attributes return the dot-notation key
Example.embedded_doc                  # 'embed_key'
Example.embedded_doc.my_attribute     # 'embed_key.my_key'
```

```
Example.embedded_doc.nested_doc       # 'embed_key.nested_key'
Example.embedded_doc.nested_doc.value # 'embed_key.nested_key.val'

# The subdocument key is available via .key
Example.embedded_doc.my_attribute.key     # 'my_key'
Example.embedded_doc.nested_doc.key       # 'nested_key'
Example.embedded_Doc.nested_doc.value.key # 'val'

# Instances return the value
doc.embedded_doc.my_attribute     # "A Fish"
doc.embedded_doc.nested_doc.value # 42
doc.embedded_doc.nested_doc       # {'nested_key': {'val': 42}}
doc.embedded_doc                  # {'embed_key': {'my_key': "A Fish",
                                  #     'nested_key': {'val: 42}}}
```

Example: A BlogPost class with embedded document

```
from humbledb import Document, Embed

class BlogPost(Document):
    config_database = 'humble'
    config_collection = 'posts'

    meta = Embed('m')
    meta.timestamp = 'ts'
    meta.author = 'a'
    meta.tags = 't'

    title = 't'
    preview = 'p'
    content = 'c'
```

As you can see using embedded documents here lets you keep your keys short, and your code clear and understandable.

Embedded Document Lists

Sometimes your documents will have list of embedded documents in them, and for your convenience, HumbleDB allows you to use attribute mapping on those documents as well. Because attribute mapping is not just useful for retrieval, but also for creation, HumbleDB provides a special `new()` method for creating new embedded documents within lists.

HumbleDB doesn't treat embedded lists specially unless they actually have a list value. This is because HumbleDB's philosophy is to not validate data types based on their keys, just like MongoDB's.

You can also embed lists within documents within lists, etc., to your heart's delight and mapped attributes will work as you would expect.

Creating embedded documents within a list

The easiest way to create embedded documents within a list is to use the `new()` helper. Of course, you can always do it "manually" by building and appending dictionaries, but who wants to do that?

```
# An example student roster
class Roster(Document):
    config_database = 'humble'
    config_collection = 'lists'

    # Embedded lists are declared the same way as embedded documents
    students = Embed('s')
    students.name = 'n'
    students.grade = 'g'

# Create a new roster instance
roster = Roster()

# You must assign a list to it first
roster.students = []

# You can use the new() convenience method which creates and appends an
# empty embedded document to your list
student = roster.students.new()
student.name = "Lisa Simpson"
student.grade = "A"

# Note: We don't have to add it to our list - it is already appended
# roster.students.append(student) # DON'T DO THIS: it will create duplicates

# Everything else works the same
with Mongo:
   Roster.insert(roster)
```

Retrieving embedded list data

Upon retrieval, HumbleDB knows if an *Embed* attribute has a list assigned to it, and lets you use your mapped attributes normally.

```
# Contining our example from above
with Mongo:
   roster = Roster.find_one()

# You can iterate over it like any list
for student in roster.students:
   # You get attributes mapped to your embedded document values
   print student.name, student.grade

   # You can modify attributes of the embedded list items
   student.grade = "A" # Everybody gets As!

# Once modified, you can save your changes
with Mongo:
   Roster.save(roster)
```

Querying within lists

Because HumbleDB gives you dot-notation keys for embedded attribute mappings, querying for list values is straightforward.

```
# Find a roster containing a given student
with Mongo:
   roster = Roster.find_one({Roster.students.name: "Bart Simpson"})

# Find all rosters where at least one student has an F
with Mongo:
   rosters = Roster.find({Roster.students.grade: "F"})
```

1.1.5 Querying, Updating and Deleting

All the standard pymongo find/update/remove, etc., are mapped onto Document subclasses, however these are only available within a *Mongo* context. If you attempt to access the `collection` attribute of a document outside a *Mongo* context, a `RuntimeError` will be raised.

Document methods

For your convenience, all the methods of the `Document.collection` are mapped onto the document class.

Here's a listing of all those methods as of `pymongo` 2.4:

aggregate()	find_and_modify()	options()
count()	find_one()	reindex()
create_index()	get_lasterror_options()	remove()
distinct()	group()	rename()
drop()	index_information()	save()
drop_index()	inline_map_reduce()	set_lasterror_options()
drop_indexes()	insert()	unset_lasterror_options()
ensure_index()	map_reduce()	update()
find()		

Example: A blog post document

This class is used for all the examples in this section.

```
# A basic blog post class for illustration
class BlogPost(Document):
   config_database = 'humble'
   config_collection = 'posts'
   config_indexes = [Index('meta.url', unique=True)]

   meta = Embed('m')
   meta.tags = 't'
   meta.published = 'p'
   meta.url = 's'

   author = 'a'
   title = 't'
   body = 'b'
```

Best practices

Reference keys via their attributes when building query, update, or removal dictionaries. For example, use `BlogPost.meta.tags` rather than `'m.t'`. This helps keep your code clean, readable, and avoids typos in string keys.

```
# GOOD
# Clear, typo-proof and highly readlable
with Mongo:
   BlogPost.find({BlogPost.author: 'Humble'}).sort(BlogPost.meta.published)

# BAD
# Hard to read, prone to typos, and obfuscated
with Mongo:
   BlogPost.find({'a': 'Humble'}).sort('m.p')
```

Creating, inserting, and updating documents

If you're familiar with how Pymongo does inserting and updating, using HumbleDB will be much the same. The main difference is that HumbleDB lets you use attributes to reference the document keys, rather than using string keys.

```
# Creating a new post
post = BlogPost()
post.meta.tags = ['python', 'humbledb']
post.meta.url = 'using-humbledb'
post.author = "Humble"
post.title = "How to Use HumbleDB"
post.body = "Lorem ipsum, etc."

# Inserting a new post
with Mongo:
   post_id = BlogPost.insert(post)

# Updating a post atomically
with Mongo:
   BlogPost.update({BlogPost._id: post_id},
      {'$set': {BlogPost.meta.published: datetime.now()}})

# Updating a post by retrieval
with Mongo:
   post = BlogPost.find_one({BlogPost.meta.url: 'using-humbledb'})
   post.meta.published = False
   BlogPost.save(post)
```

Querying for documents

Querying, like inserting and updating, works just like raw Pymongo, but with the convenience and readability of using attributes instead of string keys.

```
# Get all posts by an author
with Mongo:
   posts = BlogPost.find({BlogPost.author: "Humble"})

# Get 10 most recent posts by an author
```

```
with Mongo:
   posts = BlogPost.find({BlogPost.author: "Humble"})
   posts = posts.sort(BlogPost.meta.published, humbledb.DESC)
   posts = posts.limit(10)

# Get all unpublished posts
with Mongo:
   posts = BlogPost.find({BlogPost.meta.published: {'$exists': False}})

# Get an individual post according to its URL
with Mongo:
   post = BlogPost.find_one({BlogPost.meta.url: 'using-humbledb'})

# Unpublish a post and retrieve it
with Mongo:
   post = BlogPost.find_and_modify({BlogPost.meta.url: 'using-humbledb'},
          {'$unset': {BlogPost.meta.published: 1}}, new=True)

# Find all posts with a Python tag
with Mongo:
   posts = BlogPost.find({BlogPost.meta.tags: 'python'})
```

Removing documents

Removing works just like removing in Pymongo, but with the convenience of using attributes rather than string keys. It's strongly recommended that you only use the `_id` key when removing items to prevent accidental removal.

```
# Remove an individual post
with Mongo:
   BlogPost.remove({BlogPost._id: post_id})
```

1.1.6 Specifying Indexes

Indexes are specified using the `config_indexes` attribute. This attribute should be a list of attribute names to index. These names will be automatically mapped to their key names when the index call is made. More complicated indexes can be made using the *Index* class, which takes the same areguements as `ensure_index()`.

HumbleDB uses an `ensure_index()` call with a default `cache_for=` of 24 hours, and `background=True`. This will be called before any `find()`, `find_one()`, or `find_and_modify()` operation.

New in version 2.2: *Index* class for index creation customization.

New in version 3.0: Support for compound indexes.

Example: Indexes on a BlogPost class

```
class BlogPost(Document):
   config_database = 'humble'
   config_collection = 'posts'
   config_indexes = [
           # Basic indexes
           'author',
           'timestamp',
```

```
            # Indexes with additional creation arguments
            Index('tags', sparse=True),
            # Directional indexes with additional creation arguments
            Index([('slug', humbledb.DESC)], unique=True),
            # Embedded indexes
            Index('meta.url')
            # Compound indexes
            Index([('author', humbledb.ASC), ('timestamp', humbledb.DESC)]),
        ]

    timestamp = 'ts'
    author = 'a'
    tags = 'g'
    title = 't'
    slug = 's'
    content = 'c'
    meta = Embed('m')
    meta.url = 'u'
```

1.1.7 Configuring Connections

The *Mongo* class provides a default connection for you, but what do you do if you need to connect to a different host, port, or a replica set? You can subclass Mongo to change your settings to whatever you need.

Mongo subclasses are used as context managers, just like Mongo. Different Mongo subclasses can be nested within one another, should your code require it, however you cannot nest a connection within itself (this will raise a `RuntimeError`).

Connection Settings

- **config_host** (`str`) - Hostname to connect to.
- **config_port** (`int`) - Port to connect to.
- **config_replica** (`str`, optional) - Name of the replica set.
- **config_ssl** (`bool`, optional) - If True, use SSL for this connection.

If `config_replica` is present on the class, then HumbleDB will automatically use a `ReplicaSetConnection` for you. (Requires `pymongo >= 2.1`.)

Global Connection Settings

These settings are available globally through Pyconfig configuration keys. Use either `Pyconfig.set()` (i.e. `pyconfig.set('humbledb.connection_pool', 20)` or create a Pyconfig plugin to change these.

- **humbledb.connection_pool** (`int`, default: `300`) - Size of the connection pool to use.
- **humbledb.allow_explicit_request** (`bool`, default: `True`) - Whether or not *`start()`* can be used to define a request, without using Mongo as a context manager.
- **humbledb.auto_start_request** (`bool`, default: `True`) - Whether to use `auto_start_request` with the `Connection` instance.
- **humbledb.use_greenlets** (`bool`, default: `False`) - Whether to use `use_greenlets` with the `Connection` instance. (This is only needed if you intend on using threading and greenlets at the same time.)

- **humbledb.ssl** (`bool`, default: `False`) - Whether to use `ssl` with the `Connection` instance. The mongod or mongos you are connecting to must have SSL enabled.

More configuration settings are going to be added in the near future, so you can customize your `Connection` to completely suit your needs.

Example: Using different connection settings

```
from humbledb import Mongo

# A basic class which connects to a different host and port
class MyDB(Mongo):
    config_host = 'mydb.example.com'
    config_port = 3001

# A replica set class which will use a ReplicaSetConnection
class MyReplica(Mongo):
    config_host = 'replica.example.com'
    config_port = 3002
    config_replica = 'RS1'

# Use your custom subclasses as context managers
with MyDB:
    docs = MyDoc.find({MyDoc.value: {'$gt': 3}})

# You can nest different connections when you need to
# (But you cannot nest the same connection)
with MyReplica:
    values = MyGroup.find({MyGroup.tags: 'example'})
    value = sum(doc['value'] for doc in values)

    # HumbleDB allows you to nest different connections when you need
    # consistency
    with MyDoc:
        doc = MyDoc()
        doc.value = value
        MyDoc.insert(doc)

    MyGroup.update({MyGroup.tags: 'example'},
            {'$push': {MyGroup.related: MyDoc._id},
            multi=True)
```

1.1.8 Pre-aggregated Reports

HumbleDB provides a framework for creating pre-aggregated reports based on the ideas laid out here.

These reports are ideal for gathering metrics on a relatively low number of unique events that happen with a regular frequency. For example, hits to a certain webpage, or offer signups.

In cases where the event data is sparse, diverse, or has many parameters, other aggregation approaches may work better.

> **note** The documentation on reports is incomplete, and a work in progress. Please see the *API Documentation* for more information. If you have questions or issues, please contact me via github issues.

Changed in version 5.0: Reports got a full rewrite in version 5 of HumbleDB. They are not backwards compatible, but much more useful and efficient. It's highly recommended that you migrate old reports to the new classes.

Example:

```
from humbledb.report import Report, MONTH, HOUR

class DailyPageHits(Report):
    """
    This is an example of a class used to record hits to pages.

    This class creates one document per page per month

    This class records the total hits per page per month, as well as the
    hits per page per hour for each hour in the month.

    """
    config_database = 'reports'
    config_collection = 'page_hits'
    config_period = MONTH  # This is the document timeframe
    config_intervals = [MONTH, HOUR]  # Timeframes that data is recorded

# Record a hit to the /about page
page = '/about'
with Mongo:
   DailyPageHits.record(page)

# Get the last 24 hours worth of hits for /about as a list
with Mongo:
   hits = DailyPageHits.hourly(page)[-24:]
```

1.1.9 Paginated Arrays

One of the limitations of HumbleDB is the performance of very large arrays within documents, particulary arrays which have indexing on keys within embedded documents for that array.

There are two issues which arise; first, that each append to the array increases the array size, which leads to document moves, decreasing the efficiency of the database, and second, that with each move of a very large array, the indexing on the whole array must be updated, which can be very slow for arrays over 10,000 elements.

The *humbledb.array.Array* class addresses both of these situations by first allowing for transparent preallocation of the array document to ensure that only a minimal number of document moves happen, and second, by transparently paginating the arrays into multiple documents, to limit the maximum array size for any single document.

> **note** Due to time constrants, the documentation on arrays is incomplete. I hope to fix this as soon as possible.

Example:

```
from humbledb import Mongo
from humbledb.array import Array

class Comments(Array):
    config_database = 'humble'
    config_collection = 'comments'
    config_padding = 10000  # Number of bytes to pad with
    config_max_size = 100  # Number of entries per page
```

```
with Mongo:
    post = BlogPost.find_one(...)  # This is the post associated with
                                   # comments for this example

comments = Comments(post._id)  # The argument is used to construct the array
                               # id, which uniquely identifies pages in this
                               # array. This can be any unique string value

with Mongo:
    # Appending adds to the Array, creating a new page if necessary, and
    # paginating when an array is too large.
    # The current number of pages is returned
    page_count = comments.append({
            'user_name': "example_user",
            'comment': "I really like arrays.",
            'timestamp': datetime.now(),
            })

    comments.pages()  # Return the current number of pages

    comments.length()  # Return the current number of entries

    entries = comments.all()  # Return all the entries as a list

    first_page = comments[0]  # Return a given page of entries as a list
    pages = comments[0:2]  # Slices work, but negative indexes *do not*

    spec = {'user_name': "example_user"}
    comments.remove(spec)  # Remove all entries matching `spec`

    comments.clear()  # Remove all entries
```

1.2 API Documentation

1.2.1 Documents

class humbledb.document.**Document**

This is the base class for a HumbleDB document. It should not be used directly, but rather configured via

subclassing.

Example subclass:

```
class BlogPost(Document):
    config_database = 'db'
    config_collection = 'example'

    meta = Embed('m')
    meta.tags = 't'
    meta.slug = 's'
    meta.published = 'p'

    author = 'a'
    title = 't'
    body = 'b'
```

See *Working with Documents* for more information.

mapped_keys()
Return a list of the mapped keys.

mapped_attributes()
Return a list of the mapped attributes.

for_json()
Return this document as a dictionary, with short key names mapped to long names. This method is used by `pytools.json.as_json()`.

1.2.2 Embedded Documents

class humbledb.document.**Embed**
This class is used to map attribute names on embedded subdocuments.

Example usage:

```
class MyDoc(Document):
    config_database = 'db'
    config_collection = 'example'

    embed = Embed('e')
    embed.val = 'v'
    embed.time = 't'
```

See *Embedding Documents* for more information.

as_name_map(*base_name*)
Return this object mapped onto `NameMap` objects.

as_reverse_name_map(*base_name*)
Return this object mapped onto reverse-lookup `NameMap` objects.

1.2.3 Indexes

class humbledb.index.**Index**(*index*, *cache_for=86400*, *background=True*, ***kwargs*)
This class is used to create more complex indices. Takes the same arguments and keyword arguments as `ensure_index()`.

Example:

```
class MyDoc(Document):
    config_database = 'db'
    config_collection = 'example'
    config_indexes = [Index('value', sparse=True)]

    value = 'v'
```

New in version 2.2.

See *Specifying Indexes* for more information.

ensure(*cls*)
Does an ensure_index call for this index with the given *cls*.

Parameters **cls** – A Document subclass

1.2.4 MongoDB Connections

class humbledb.mongo.**Mongo**
Singleton context manager class for managing a single `pymongo.connection.Connection` instance. It is necessary that there only be one connection instance for pymongo to work efficiently with gevent or threading by using its built in connection pooling.

This class also manages connection scope, so that we can prevent *Document* instances from accessing the connection outside the context manager scope. This is so that we always ensure that `end_request()` is always called to release the socket back into the connection pool, and to restrict the scope where a socket is in use from the pool to the absolute minimum necessary.

This class is made to be thread safe.

Example subclass:

```
class MyConnection(Mongo):
    config_host = 'cluster1.mongo.mydomain.com'
    config_port = 27017
```

Example usage:

```
with MyConnection:
    doc = MyDoc.find_one()
```

See *Configuring Connections* for more information.

start()
Public function for manually starting a session/context. Use carefully!

end()
Public function for manually closing a session/context. Should be idempotent. This must always be called after *Mongo.start()* to ensure the socket is returned to the connection pool.

config_uri
alias of `UNSET`

1.2.5 Reports

The report module contains the HumbleDB reporting framework.

class humbledb.report.**Report**
A report document.

classmethod record(*event*, *stamp=None*, *safe=False*, *count=1*)
Record an instance of *event* that happened at *stamp*.

If *safe* is `True`, then this method will wait for write acknowledgement from the server. The *safe* keyword has no effect for *pymongo >= 3.0.0*.

Parameters

- **event** (*str*) – Event identifier string
- **stamp** (*datetime.datetime*) – Datetime stamp for this event (default: now)
- **safe** (*bool*) – Safe write option passed to pymongo
- **count** (*int*) – Number to increment

classmethod record_id(*event*, *stamp*)
Return a string suitable for use as the document id.

Parameters

- **event** (*str*) – A event identifier
- **stamp** (*datetime.datetime*) – Datetime for this event

Periods/Intervals

report.**YEAR = 5**

report.**MONTH = 4**

report.**DAY = 3**

report.**HOUR = 2**

report.**MINUTE = 1**

1.2.6 Arrays

class humbledb.array.**Array**(*_id*, *page_count=UNSET*)
HumbleDB Array object. This helps manage paginated array documents in MongoDB. This class is designed to be inherited from, and not instantiated directly.

If you know the *page_count* for this array ahead of time, passing it in to the constructor will save an extra query on the first append for a given instance.

Parameters

- **_id** (*str*) – Sets the array's shared id
- **page_count** (*int*) – Total number of pages that already exist (optional)

all()
Return all entries in this array.

append(*entry*)
Append an entry to this array and return the page count.

Parameters **entry** (*dict*) – New entry

Returns Total number of pages

clear()
Remove all documents in this array.

length()
Return the total number of items in this array.

new_page(*page_number*)
Creates a new page document.

Parameters **page_number** (*int*) – The page number to create

page_id(*page_number=None*)
Return the document ID for *page_number*. If page number is not specified the `Array.page_count` is used.

Parameters **page_number** (*int*) – A page number (optional)

pages()
Return the total number of pages in this array.

class humbledb.array.**Page**
Document class used by *Array*.

1.2.7 Helpers

humbledb.helpers.**auto_increment**(*database*, *collection*, *_id*, *field='value'*, *increment=1*)
Factory method for creating a stored default value which is auto-incremented.

This uses a sidecar document to keep the increment counter sync'd atomically. See the MongoDB documentation for more information about how this works.

Note: If a *Document* subclass is inherited and has an auto_increment helper, it will share the counter unless it's overriden in the inheriting *Document*.

Example: using auto_increment fields

```
from humbledb.helpers import auto_increment

class MyDoc(Document):
    config_database = 'humbledb'
    config_collection = 'examples'

    # The auto_increment helper needs arguments:
    #     - database name: Database to store the sidecar document
    #     - collection name: Collection name that stores the sidecar
    #     - Id: A unique identifier for this document and field
    auto_id = auto_increment('humbledb', 'counters', 'MyDoc_auto_id')

    # The auto_increment helper can take an increment argument
    big_auto = auto_increment('humbledb', 'counters', 'MyDoc_big_auto',
            increment=10)
```

Parameters

- **database** (*str*) – Database name

- **collection** (*str*) – Collection name
- **_id** (*str*) – Unique identifier for auto increment field
- **field** (*str*) – Sidecar document field name (default: `"value"`)
- **increment** (*int*) – Amount to increment counter by (default: 1)

1.3 Changelog

1.3.1 Changes by version

This section contains all the changes that I can remember, by version.

6.0.0

- Drops support for pymongo < 2.9.
- Adds support for Python > 3.5.
- Fixes some minor test cases.
- Deprecates support for pymongo < 3.
- Deprecates support for Python < 3.

Released March 6, 2018.

5.6.1

- Makes HumbleDB handle the `safe=` keyword for `update()`, `insert()`, and `save()` when using Pymongo 3.0 or greater. If you specified `safe=False`, then HumbleDB will use `w=0` (no write concern), and otherwise will fall back to the configured write concern level, which defaults to `w=1`.

Released July 8, 2015

5.6.0

- Adds support for Pymongo version 3.0.0 and higher which made some backwards incompatible changes.

Released June 14, 2015

5.5.1

- Add `humbledb.ensure_indexes` pyconfig setting to allow for disabling ensure index calls.

5.5.0

- Adds support for SSL. Thanks to paulnues.

5.4.1

- Fix bug where *auto_increment()* helper didn't respect `increment` argument. Thanks to paulnues for the fix.

5.4.0

- Add helpers module and *auto_increment()* default value helper.

5.3.0

- Add *config_uri* configuration option for declaring default databases and databases with authentication.
- If a *Mongo* subclass specifies a *config_uri* which includes a database, and a *Document* is used which does not match the database, a `DatabaseMismatch` error will be raised.
- Fix a bug where declaring *Mongo* subclasses late (at runtime) would not correctly instantiate the connection instance.
- Fix a bug with Pymongo 2.1.x connection instances.

5.2.0

- *Document* declarations can now include default values. See *Giving Documents Default Values* for more details.
- *Array* regexes now escape periods to prevent name collisions.

5.1.4

- Patch from paulnues to fix brittle tests.

5.1.3

- Bump the default for `config_max_pool_size` up to 300, since in PyMongo 2.6, they changed the behavior of connection pools to make that a blocking limit, rather than a minimum size.

5.1.2

- Fix a bug where a *Report* would raise a ValueError on querying months with 30 days.

5.1.1

- Fix a bug where a *Array* may not have its page created before an append call attempts to modify it by adding write concern to the insert.

5.1.0

- Add *count* keyword argument to *humbledb.report.Report.record()* to allow recording multiple events instead of always incrementing one.

5.0.1

- Fix a bug with summing report intervals where too many or too few values could be returned, sometimes with the wrong timestamp.

5.0.0

- This release may break backwards compatibility.
- Total rewrite of the **:module:'humbledb.report'** module to make it much more useful. Sorry, but I'm fairly sure nobody was using it before anyway.

4.0.1

- Fix bug with `array.Array.remove()` in sharded environments.

4.0.0

- This release may break backwards compatibility.
- Restrict `from humbledb import *` to only basic document classes (*Mongo*, *Document*, *Embed*, *Index*).
- Create new `humbledb.errors` module, which contains shortcuts to Pymongo specific errors, as well as the new exceptions: `NoConnection`, `NestedConnection`, and `MissingConfig`.
- *Document* will now raise `MissingConfig` and `NoConnection`. The previous behavior was to raise just a `RuntimeError`.
- *Mongo* subclasses add the new configuration option `config_write_concern`. This now defaults to `1`, which may break backwards compatibility. The previous behavior depended on which version of Pymongo you were using.
- *Mongo* will now raise `NestedConnection`.
- *Document* instances which do not map attributes for embedded documents will no longer wrap the accessed embedded documents in `DictMap` instances. This should improve performance substantially for very large documents with many unmapped, embedded documents.
- The *Array* class has been refactored to no longer need the `array_id` and `number` fields, or the index on them. It now leverages regex queries against the `_id` field instead.
- The *Array* class now has shortcut properties for accessing the following attributes on the *Page* class: find, update, remove, entries, size. The find, update, and remove attributes require a *Mongo* (or a subclass) connection context.
- The `page_count` parameter to *Array* is not longer required. If omitted, the number of pages will be queried for before the first append operation.
- `remove()` now only removes the first matching element found. The previous behavior was to remove all matching elements, but this meant that the `Array.length()` could get out of sync with the actual size.

3.3.1

- Now depends on Pytool >= 3.0.1.

3.3.0

- Implement `for_json()` hook on *Document*, `DictMap` and `ListMap`.
- Implement version checking for `ttl` vs. `cache_for` keyword to `ensure_index()`.
- Fix `config_replica` handling when config_replica is set to a descriptor class (i.e. a `pyconfig.setting()` instance).
- Removed `humbledb.document.Document._asdict()`. Use *for_json()* instead.

3.2.0

- Add the `humbledb.array` module and *humbledb.array.Array* class for easily working with very large paginated arrays in MongoDB.

3.1.0

- Add support for `MongoClient` and `MongoReplicaSetClient`.

3.0.3

- Fix bug in deleting embedded document keys via attributes.

3.0.2

- Fix bug with DocumentMeta accidentally getting extra `name` attribute, which in turn became available on Document, and would override mapping behavior.

3.0.1

- Fix bug with checking config_resolution on the MonthlyReport.

3.0.0

- Major internal refactoring of module layout.
- Add support for compound indexes.
- Add Cursor subclass to do document type coercion rather than use as_class argument to pymongo methods.
- Change return value of unset attributes from `None` to `{}`.
- Add aliases `humbledb.DESC` and `humbledb.ASC` for `pymongo.DESCENDING` and `pymongo.ASCENDING` respectively.
- Add embedded document list attribute mapping.
- Lots of test coverage.

2.3.1

- Change `humbledb.report.DailyReport` to use 0-59 for minute range, rather than 0-1439.

2.3.0

- Add support for resolving dot-notation indexes.
- Add reporting framework.

2.2.1

- Fix bug when old version by using pkg_resources.parse_version to check pymongo version.

2.2.0

- Add *Index* class.
- Make HumbleDB compatible with `pymongo >= 2.0.1`.

2.1.1

- Fix bug when find_one or find_and_modify return None.

2.1.0

- Add *Document.mapped_keys()* and *Document.mapped_attributes()* methods.

2.0.2

- Fix bug where find_and_modify returned dict instead of Document subclass.

2.0.1

- Updated documentation.

2.0.0

- First release fit for public consumption.

CHAPTER 2

License

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index