# couchbasekit Documentation

***Release 0.2.3-dev***

**Roy Enjoy**

April 19, 2013

# CONTENTS

**couchbasekit** is *a wrapper around CouchBase Python driver for document validation and more*. It was inspired by MongoKit and was developed by *the project coming soon?*, which is also an open source project.

You can get detailed information about couchbase itself from http://www.couchbase.com/ and about its Python driver form http://www.couchbase.com/develop/python/next.

Documentation: https://couchbasekit.readthedocs.org/en/latest/

Source code: https://github.com/kirpit/couchbasekit

Contents:

CHAPTER

ONE

# INSTALLATION AND CONFIGURATION

It is strongly suggested that you should already know what is virtualenv, preferably virtualenvwrapper at this stage.

You can easily install couchbasekit via pip:

```
$ pip install couchbasekit
```

---

**Note:** couchbasekit has dependencies on:

- couchbase
- jsonpickle
- python-dateutil
- py-bcrypt (optional for `couchbasekit.fields.PasswordField`)

---

Then, the only configuration you have to do is couchbase authentication, somewhere at the beginning of your application (such as *settings.py* if you're using Django Web Framework for example):

```
from couchbasekit import Connection
Connection.auth('theusername', 'p@ssword')
```

or:

```
from couchbasekit import Connection
Connection.auth(
    username='theusername', password='p@ssword',
    server='localhost', port='8091', # default already
)
```

That's it. Now, you are ready for a crash course. See *Quick Start*.

CHAPTER

TWO

# QUICK START

Less talk, more code. Set your authentication details first:

```
from couchbasekit import Connection

# you should do this somewhere beginning such as settings.py:
Connection.auth('myusername', 'p@ssword')
```

Then define your model document.

**author.py**:

```
import datetime
from couchbasekit import Document, register_view
from couchbasekit.fields import EmailField, ChoiceField
from example.samples.publisher import Publisher
from example.samples.book import Book

class Gender(ChoiceField):
    CHOICES = {
        'M': 'Male',
        'F': 'Female',
    }


@register_view('dev_authors')
class Author(Document):
    __bucket_name__ = 'couchbasekit_samples'
    __key_field__ = 'slug' # optional
    doc_type = 'author'
    structure = {
        'slug': unicode,
        'first_name': unicode,
        'last_name': unicode,
        'gender': Gender,
        'email': EmailField,
        'publisher': Publisher, # kind of foreign key
        'books': [Book], # 1-to-many, or many-to-many? some-to-some.. :)
        'has_book': bool,
        'age': int,
        'birthday': datetime.date,
        'created_at': datetime.datetime,
    }
    default_values = { # optional
        'has_book': False,
```

```
        # don't worry about the timezone info!
        # it's auto assigned as to UTC, so all you have to do is:
        'created_at': datetime.datetime.utcnow,
    }
    required_fields = ( # optional
        'slug',
        'first_name',
        'last_name',
        'email',
    )
```

Then use it as such;

```
>>> from example.samples.author import Author, Gender
>>> from couchbasekit.fields import EmailField
>>>
>>> douglas = Author()
>>> douglas.is_new_record
True
>>> try:
...     douglas.validate()
... except Author.StructureError as why:
...     print why
...
Key field "slug" is defined but not provided.
>>>
>>> douglas.slug = u'douglas_adams'
>>> try:
...     douglas.validate()
... except Author.StructureError as why:
...     print why
...
Required field for "first_name" is missing.
>>>
>>> isinstance(douglas, dict)
True
>>> douglas.update({
...     'first_name': u'Douglas',
...     'last_name': u'Adams',
...     'email': EmailField('dna@example.com'),
... })
...
>>> douglas.validate()
True
>>> douglas.save()
14379837794698
>>> douglas.cas_value # CAS value (version) of the couchbase document
14379837794698
>>> douglas.id
u'douglas_adams'
>>> douglas.doc_id
u'author_douglas_adams'
>>> douglas.birthday is None
True
>>> douglas.non_exist_field
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "couchbasekit/document.py", line 68, in __getattr__
```

```
    return super(Document, self).__getattribute__(item)
AttributeError: 'Author' object has no attribute 'non_exist_field'
>>>
>>> dna = Author('douglas_adams')
>>> dna.is_new_record
False
>>> douglas==dna
True
>>> douglas.has_book = True
>>> douglas==dna
False
>>> # because we set @register_view decorator, here are the CouchBase views:
>>> douglas.view()
<couchbase.client.DesignDoc at 0x10d3ebe10>
>>> view = douglas.view('by_fullname')
>>> view
<couchbase.client.View at 0x10ce57410>
>>> view.results({'key': 'Douglas Adams'})
<couchbase.client.ViewResultsIterator at 0x10d40dad0>
>>> # please refer to CouchBase views documentation for further usage..
>>> # and the bucket itself for advanced folks:
>>> douglas.bucket
<couchbase.client.Bucket at 0x10fb0c2d0>
>>> print [m for m in dir(douglas.bucket) if not m.startswith('_')]
['add', 'append', 'cas', 'decr', 'delete', 'design_docs', 'flush', 'gat', 'get', 'getl', 'incr', 'in
>>> # nice!
```

CHAPTER

THREE

# MODEL DOCUMENT

`couchbasekit.document.Document` is the main class you will be extending to define your own model documents. There are 3 attributes must be set within your model:

- __bucket_name__
- doc_type
- structure

and the optional ones are:

- __key_field__
- default_values
- required_fields

## 3.1 __bucket_name__ (required)

The name of the couchbase bucket that the new records will be saved into and retrieved back from. This bucket should be already created manually.

## 3.2 doc_type (required)

The type of the document that will be used in various places and usually lowercase of your model name but not checked or forced.

The most important function of `doc_type` attribute is to create an auto-field named `doc_type` in every document. That means you can use it in your JavaScript views to check which type of documents you want to emit:

```
function (doc, meta) {
  if (doc.doc_type=='author') {
    emit(doc.slug, {first_name: doc.first_name, last_name: doc.last_name});
  }
}
```

Another function of `doc_type` is to prefix your document id, if you're using `__key_field__` optional attribute in order to create meaningful document IDs. Its format is; `{doc_type}_{key_value.lower()}`. If you don't choose to use `__key_field__` in your models, `doc_type` will not be used to prefix your document IDs either.

## 3.3 structure (required)

Structure definition dictionary is a wide topic, therefore explained in another section. See *Model Document Structure*.

## 3.4 __key_field__ (optional)

Key field is kind of simulating primary key feature in relational databases that gives you ability to retrieve a single document by its *key value* without doing a map-reduce in your buckets. It should be set to one of your root field in your structure and **it is your responsibility** to check if a document exist with the same *key value* before over-writing it.

Assuming the `username` field is the `__key_field__` in your structure:

```
userdata = {'username': u'kirpit', 'is_superuser': True}
try:
    user = User(userdata['username'])
except User.DoesNotExist:
    # good, username is not taken
    user = User(userdata)
    user.save()
else:
    print 'Sorry, this username is already taken.'
```

If you don't provide a `__key_field__` in your structure, first 12 characters of the hash key of your initial document will be used without prefixing with `doc_type` attribute. Hashing is done via `hashlib.sha1`.

## 3.5 default_values (optional)

As it explains itself, it sets the default values for specified fields before saving a document. Practically, you may assign a static value, a custom field, a model document to relate or any callable that gives a value for it.

It does NOT set the document value, if it was already provided (which is not surprising, is it?).

Refer to *Quick Start* for an example.

## 3.6 required_fields (optional)

Another self explanatory attribute that checks if its items was provided at the time of validation. It should be a `tuple()` (`list()` is ok too) and have all the names of the fields that are required.

Refer to *Quick Start* for an example.

## 3.7 Bonus: @register_view decorator

You may use this decorator to declare which design view your document instances will be using. This feature is used by `couchbasekit.document.Document.view()` that lets you to query your views:

```
from couchbasekit import Document, register_view

@register_view('dev_books')
```

```
class Book(Document):
    __bucket_name__ = 'mybucket'
    doc_type = 'book'
    structure = {
        # snip snip
    }
```

Then it becomes easier to get your view queries. Please refer to CouchBase views documentation for advanced query options:

```
>>> by_title = Book().view('by_title').results({
...     'startkey': 'A',
...     'endkey': 'C',
...     'stale': 'ok',
...     'limit': 1000,
... })
>>> for result in by_title.results:
...     print result['id'], result['key'], result['value']
```

An experimental tool `couchbasekit.viewsync.ViewSync` also uses this decorator to backup/restore your server-side map/reduce functions.

---

**Note:** `@register_view` decorator automatically attaches `'full_set': True` parameter to your development views by default, so you don't have do it programmatically. To disable it simply use as: `@register_view('dev_books', full_set=False)`. This feature doesn't affect production views at all.

---

CHAPTER

FOUR

# MODEL DOCUMENT STRUCTURE

Structure definition dictionary is the most important part of your model document. They are similar to table fields in relational SQL systems in a way. You simply need to define them as key-value that corresponds to field_name: field_type. Keep reading...

## 4.1 Allowed Types

To begin, you may simply define those field types as standard Python types, see `couchbasekit.schema.ALLOWED_TYPES` for the list of them.

## 4.2 Document Relations

You can define a field type as another model document (or even recursively) within your structure. This simulates kind of foreign key scenario in the relational systems but you must know that every related document will be fetched separately from couchbase server as the nature of the non-relational systems.

The good news is, these relations are lazy-loaded, fetched on-demand and *couchbasekit* caches them during the object's life time.

```
>>> lonely_galaxy = Publisher('lonely_galaxy')
>>> dna = Author('douglas_adams')
>>> dna.publisher = lonely_galaxy
>>> dna.save()
4535519295771
>>> dna = Author('douglas_adams') # retrieve the same doc
>>> dna.get('publisher')
u'publisher_lonely_galaxy'
>>> dna.publisher # or dna.load()
{u'doc_type': u'publisher', u'created_at': u'2012-11-18 16:24:16.784474+00:00', 'slug': u'lonely_gala
>>> dna.get('publisher') # no more raw, already cached
{u'doc_type': u'publisher', u'created_at': u'2012-11-18 16:24:16.784474+00:00', 'slug': u'lonely_gala
>>>
```

## 4.3 Custom Fields

With couchbasekit, of course you can have your specific field types and a few of them may be already defined in `couchbasekit.fields`. Creating your own custom field is quite easy, please refer to `couchbasekit.fields.CustomField`.

As an example, the password fields are salted randomly and encrypted on the fly, thus cannot be decrypted back:

```
>>> from couchbasekit.fields import PasswordField
>>> raw = '123456'
>>> PasswordField(raw)
'$2a$12$1nshsN7Nt8e3.dPd1ZcA7uJnesu2sg52nZl6CX1N0ETZwc2UYCGYS'
>>> PasswordField(raw)
'$2a$12$UyLdw0QwHJmONipuyQ3Mq.NA4YteHZ8NDwXFpaJP.xi9ZnUjmxvWa'
>>> hashed = PasswordField(raw)
>>> hashed.value
'$2a$12$r490Tn0zEaMYTf.dfjBNoe0I729Ej2Z18xTJLbwfqZyOXeabXZUky'
>>> hashed.check_password('incorrect')
False
>>> hashed.check_password('123456')
True
>>>
```

## 4.4 List (Multi Value) Fields

You can also define a `list()` of values. For example:

```
class Book(Document):
    __bucket_name__ = 'couchbasekit_samples'
    doc_type = 'book'
    structure = {
        'title': unicode,
        'published_at': datetime.date,
        'pictures': list,
        'tags': [unicode],
    }
```

Note that if you are sure what type of elements a *List Field* will have, you should explicitly specify it as an instance with a **single value** in it (i.e. `'tags': [unicode]`). Otherwise just let it be `list` then it can have any combination of values in it.

However, be careful if you define your field as a plain `list` (such as `'pictures': list`). You will always get a list of basic types (e.g. `unicode`, `int`, `float`, `bool` etc..) as *couchbasekit* doesn't know if they're any of the advanced ones (e.g. custom fields, document relations, datetime.date, etc..). For example, if you save a document relation in them, you will get its `couchbasekit.document.Document.doc_id()` as `unicode` but not the document itself, which actually would be useful for performance tuning.

## 4.5 Schemaless Fields

Some of your model documents may need complicated structure, such as pre-defined item types of a dictionary, deeply nested dictionary or totally schemaless sub-structures.

> **Warning:** One downside of such free dictionary models is that you can't use attribute access (a.k.a. dot notation), so you have to use dictionary-like item assignment and the same rule applies for retrieving of your data.

First and easiest example would be a total schemaless model document:

```
class FreeModel(Document):
    __bucket_name__ = 'couchbasekit_samples'
    doc_type = 'free'
```

```
    structure = {}

free = FreeModel()
# that does NOT work because 'somefield' wasn't defined in the structure
free.somefield = 'some value'
# but that will work:
free['somefield'] = 'some value'
# and those also will work as the Document class is a dictionary itself!
free = FreeModel(somefield='some value', listfield=['list', 'of', 'items'])
# or that's ok too:
data = {'somefield': 'some value', 'listfield': ['list', 'of', 'items']}
free = FreeModel(data)
```

If you want a semi schemaless structure on a specific field that means you know it will be dictionary and what type for its keys and values will be, you may define only types for its key-value pair:

```
class User(Document):
    __bucket_name__ = 'couchbasekit_samples'
    doc_type = 'user'
    structure = {
        'username': unicode,
        'email': EmailField,
        'password': PasswordField,
        'logins': {
            # datetime: ip
            datetime.datetime: unicode,
        },
    }
```

Finally, deeply nested dictionary fields:

```
class Book(Document):
    __bucket_name__ = 'couchbasekit_samples'
    doc_type = 'book'
    structure = {
        'title': unicode,
        'published_at': datetime.date,
        'pictures': list,
        'tags': [unicode],
        'category': {
            u'History': bool,
            u'Sci-Fiction': bool,
            u'Cooking': {
                u'Turkish': bool,
                u'Italian': bool,
                u'Fast Food': bool,
                u'Dessert': bool,
            },
        },
    }
```

---

**Note:** Please note that again; dot notation does **not** work for deeply nested dictionaries either. So you can't check or set of a book's *Dessert* category by dot notation:

```
>>> book = Book('ad45556b3ba4')
>>> book.category.Cooking.Dessert # wrong!
>>> book.category.Cooking[u'Dessert'] # wrong!
>>> book.category is None
```

```
True
>>> book.category['Cooking']['Dessert'] = False # wrong, as 'category' is not assigned yet
>>> book.category = {u'Cooking': {u'Dessert': True}} # correct
>>> book.category['Cooking']['Dessert'] = True # it was created, so it's ok now
>>> book['category']['Cooking']['Dessert'] = True # correct, same as above
>>> book.category['History'] # wrong, you'll get a KeyError
>>> 'History' in book.category # that's the way
False
>>> book.category[u'History'] = True # correct, only assigns the u'History'
>>> book['category'] = {u'History': True} # correct, but overwrites the 'category'
>>>
```

CHAPTER

FIVE

# API DOCUMENTATION

## 5.1 couchbasekit.connection

**website** http://github.com/kirpit/couchbasekit



**license** MIT, see LICENSE.txt for details.

**class** `couchbasekit.connection.`**`Connection`**

This is the singleton pattern for handling couchbase connections application-wide.

Simply set your authentication credentials at the beginning of your application (such as in "settings.py") by:

```
>>> from couchbasekit import Connection
>>> Connection.auth('theusername', 'p@ssword')
```

or

```
>>> Connection.auth(
...     username='theusername', password='p@ssword',
...     server='localhost', port='8091', # default already
... )
```

---

**Note:** This class is not intended to create instances, so don't try to do:

```
>>> conn = Connection() # wrong
```

or you will get a `RuntimeWarning`.

---

**classmethod `auth`** (*username*, *password*, *server='localhost'*, *port='8091'*)

Sets the couchbase connection credentials, globally.

**Parameters**

- **username** (*str*) – bucket username (or "Administrator" for working with multi buckets).
- **password** (*str*) – bucket password (or Administrator's password for working with multi buckets).
- **server** (*str*) – couchbase server to connect, defaults to "localhost".
- **port** (*str*) – couchbase server port, defaults to "8091".

**Returns** None

**classmethod bucket**(*bucket_name*)
Gives the bucket from couchbase server.

**Parameters** **bucket_name** (*str*) – Bucket name to fetch.

**Returns** couchbase driver's Bucket object.

**Return type** couchbase.client.Bucket

**Raises** RuntimeError If the credentials wasn't set.

**classmethod close**()
Closes the current connection, which would be useful to ensure that no orphan couchbase processes are left. Use it in, for example one of your Django middleware's process_response().

---

**Note:** The class will open a new connection if a bucket is requested even though its connection was closed already.

---

**Returns** None

## 5.2 couchbasekit.document

**website** http://github.com/kirpit/couchbasekit

**copyright** Copyright 2013, Roy Enjoy <kirpit *at* gmail.com>, see AUTHORS.txt.

**license** MIT, see LICENSE.txt for details.

**class** couchbasekit.document.**Document**(*key_or_map=None*, *get_lock=False*, ***kwargs*)
Couchbase document to be inherited by user-defined model documents that handles everything from validation to comparison with the help of couchbasekit.schema.SchemaDocument parent class.

**Parameters**

- **key_or_map** (*basestring or dict*) – Either the document id to be fetched or dictionary to initialize the first values of a new document.
- **get_lock** (*bool*) – True, if the document wanted to be locked for other processes, defaults to False.
- **kwargs** – key=value arguments to be passed to the dictionary.

**Raises** couchbasekit.errors.StructureError or couchbasekit.errors.DoesNotExist

**exception DoesNotExist**(*doc*)
Raised when a model class passed with an id to be fetched, but not found within couchbase.

You don't have to specifically import this error to check if does not exist because your model document has just the same error for convenience. For example:

```
try:
    mrnobody = Author('someone_doesnt_exist')
except Author.DoesNotExist:
    # some useful code here
    pass
```

Document.**bucket**
Returns the couchbase Bucket object for this instance, object property.

**Returns** See: couchbase.client.Bucket.

**Return type** couchbase.client.Bucket

Document.**delete**()
Deletes the current document explicitly with CAS value.

**Returns** Response from CouchbaseClient.

**Return type** unicode

**Raises** couchbasekit.errors.DoesNotExist or couchbase.exception.MemcachedError

Document.**doc_id**
Returns the couchbase document's id, object property.

**Returns** The document id (that is created from doc_type and __key_field__ value, or auto-hashed document id at first saving).

**Return type** unicode

Document.**id**
Returns the document's key field value (sort of primary key if you defined it in your model, which is optional), object property.

**Returns** The document key if __key_field__ was defined, or None.

**Return type** unicode or None

Document.**save**(*expiration=0*)
Saves the current instance after validating it.

**Parameters** **expiration** (*int*) – Expiration in seconds for the document to be removed by couchbase server, defaults to 0 - will never expire.

**Returns** couchbase document CAS value

**Return type** int

**Raises** couchbasekit.errors.StructureError, See couchbasekit.schema.SchemaDocument.validate().

Document.**touch**(*expiration*)
Updates the current document's expiration value.

**Parameters** **expiration** (*int*) – Expiration in seconds for the document to be removed by couchbase server, defaults to 0 - will never expire.

**Returns** Response from CouchbaseClient.

**Return type** unicode

**Raises** couchbasekit.errors.DoesNotExist or couchbase.exception.MemcachedError

Document.**view**(*view_name=None*)
Returns a couchbase view (or design document view with no view_name provided) if couchbasekit.viewsync.register_view() decorator was applied to model class.

**Parameters** **view_name** (*str*) – If provided returns the asked couchbase view object or design document otherwise.

**Returns** couchbase design document, couchbase view or None

**Return type** couchbase.client.View or couchbase.client.DesignDoc or None

## 5.3 couchbasekit.schema

**website** http://github.com/kirpit/couchbasekit

**copyright** Copyright 2013, Roy Enjoy <kirpit *at* gmail.com>, see AUTHORS.txt.

**license** MIT, see LICENSE.txt for details.

**class** couchbasekit.schema.**SchemaDocument**(*seq=None*, ***kwargs*)

Schema document class that handles validations and restoring raw couchbase documents into Python values as defined in model documents.

Under normal circumstances, you don't use or inherit this class at all, because it is only being used by couchbasekit.document.Document class.

**Parameters** **seq** (*dict*) – Document data to store at initialization, defaults to None.

**Raises** couchbasekit.errors.StructureError if the minimum structure requirements wasn't satisfied.

**exception** **StructureError**(*key=None*, *exp=None*, *given=None*, *msg=''*)

Raised when things go wrong about your model class structure or instance values. For example, you pass an int value to some field that should be str or some required field wasn't provided etc..

SchemaDocument.**load**()

Helper function to pre-load all the raw document values into Python ones, custom types and/or other document relations as they are defined in model document.

This is only useful when you need the instance to convert all its raw values into Python types, custom fields and/or other document relations *before* sending that object to somewhere else. For example, sending a User document to your framework's login(request, user) function.

If your code is the only one accessing its values such as; user.posts, you don't have to .load() it as they're auto-converted and cached on-demand.

Returns the instance itself (a.k.a. chaining) so you can do:

```
>>> book = Book('hhg2g').load()
```

**Returns** The Document instance itself on which was called from.

SchemaDocument.**validate**()

Validates the document object with current values, always called within couchbasekit.document.Document.save() method.

**Returns** Always True, or raises couchbasekit.errors.StructureError exception.

**Raises** couchbasekit.errors.StructureError if any validation problem occurs.

couchbasekit.schema.**ALLOWED_TYPES**

This is the constant that will be used to check your model structure definitions:

```
ALLOWED_TYPES = (
    bool,
    int,
    long,
    float,
    unicode,
    basestring,
    list,
    dict,
```

```
    datetime.datetime,
    datetime.date,
    datetime.time,
)
```

However, it doesn't include `str` type intentionally because couchbase will keep your values in `unicode` and you will have trouble re-saving a `str` field right after you fetch it. If you really have to pass a `str` value while you first creating a document record, you can simply define your field as `basestring` and both types will be accepted at the time of validation.

Besides these ones, you can also use `couchbasekit.document.Document` for document relations and `couchbasekit.fields.CustomField` subclasses for special field types. See `couchbasekit.fields`.

## 5.4 couchbasekit.fields

**website** http://github.com/kirpit/couchbasekit





- `couchbasekit.fields.CustomField`
- `couchbasekit.fields.ChoiceField`
- `couchbasekit.fields.EmailField`
- `couchbasekit.fields.PasswordField`

**class** `couchbasekit.fields.`**`ChoiceField`**(*choice*)

The custom field to be used for multi choice options such as gender, static category list etc. This class can't be used directly that has to be extended by your choice list class. Thankfully, it's just easy:

```
class Gender(ChoiceField):
    CHOICES = {
        'M': 'Male',
        'F': 'Female',
    }
```

and all you have to do is to pass the current value to create your choice object:

```
>>> choice = Gender('F')
>>> choice.value
'F'
>>> choice.text
'Female'
```

**Parameters** **choice** (*basestring*) – The choice value.

**`text`**

Returns the text of the current choice, object property.

**Return type** unicode

**class** `couchbasekit.fields.`**`CustomField`**

The abstract custom field to be extended by all other field classes.

---

**Note:** You can also create your own custom field types by implementing this class. All you have to do is to assign your final (that is calculated and ready to be saved) value to the `value` property. Please note that it should also accept unicode raw values, which are fetched and returned from couchbase server. See `PasswordField` source code as an example.

Please contribute back if you create a generic and useful custom field.

---

**value**
Property to be used when saving a custom field into `couchbasekit.document.Document` instance.

**Returns** The value to be saved for the field within `couchbasekit.document.Document` instances.

**Return type** mixed

**class** couchbasekit.fields.**EmailField**(*email*)
The custom field to be used for email addresses and intended to validate them as well.

**Parameters** **email** (*basestring*) – Email address to be saved.

**static is_valid**(*email*)
Email address validation method.

**Parameters** **email** (*basestring*) – Email address to be saved.

**Returns** True if email address is correct, False otherwise.

**Return type** bool

**class** couchbasekit.fields.**PasswordField**(*password*)
The custom field to be used for password types.

It encrypts the raw passwords on-the-fly and depends on *py-bcrypt* library for such encryption.

**Parameters** **password** (*unicode*) – Raw or encrypted password value.

**Raises** `ImportError` if *py-bcrypt* was not found.

**check_password**(*raw_password*)
Validates the given raw password against the intance's encrypted one.

**Parameters** **raw_password** (*unicode*) – Raw password to be checked against.

**Returns** True if comparison was successful, False otherwise.

**Return type** bool

**Raises** `ImportError` if *py-bcrypt* was not found.

**static get_bcrypt**()
Returns the *py-bcrypt* library for internal usage.

**Returns** *py-bcrypt* package.

**Raises** `ImportError` if *py-bcrypt* was not found.

## 5.5 couchbasekit.errors

**website** http://github.com/kirpit/couchbasekit



**license** MIT, see LICENSE.txt for details.

**exception** couchbasekit.errors.**CouchbasekitException**
Just to have some base exception class.

**exception** couchbasekit.errors.**DoesNotExist**(*doc*)
Raised when a model class passed with an id to be fetched, but not found within couchbase.

You don't have to specifically import this error to check if does not exist because your model document has just the same error for convenience. For example:

```
try:
    mrnobody = Author('someone_doesnt_exist')
except Author.DoesNotExist:
    # some useful code here
    pass
```

**exception** couchbasekit.errors.**StructureError**(*key=None*, *exp=None*, *given=None*, *msg=''*)
Raised when things go wrong about your model class structure or instance values. For example, you pass an `int` value to some field that should be `str` or some required field wasn't provided etc..

## 5.6 couchbasekit.middlewares

**website** http://github.com/kirpit/couchbasekit

**copyright** Copyright 2013, Roy Enjoy <kirpit *at* gmail.com>, see AUTHORS.txt.

**license** MIT, see LICENSE.txt for details.

**class** couchbasekit.middlewares.**CouchbasekitMiddleware**
A helper that can be used in Django Middlewares to close couchbase connection gracefully in order not leave any orphan subprocess behind.

## 5.7 couchbasekit.viewsync

**website** http://github.com/kirpit/couchbasekit

**copyright** Copyright 2013, Roy Enjoy <kirpit *at* gmail.com>, see AUTHORS.txt.

**license** MIT, see LICENSE.txt for details.

**class** couchbasekit.viewsync.**ViewSync**
This is an experimental helper to download, upload and synchronize your couchbase views (both map and reduce JavaScript functions) in an organized way.

Unfortunately, it's quite impossible to synchronize these views since couchbase doesn't provide any information about when a specific view was created and modified. So we can't know if previously downloaded .js file or the current one at couchbase server should be replaced..

This class also works in a singleton pattern so all its methods are `@classmethod` that you don't need to create an instance at all.

In order to use this tool, you have to set VIEW_PATH attribute of the class to the directory wherever you want to keep downloaded JavaScript files. It is better to keep that directory under version controlled folder, as they can also become your view backups:

```
ViewSync.VIEW_PATH = '/path/to/your/js/view/backups'
```

**classmethod download()**
Downloads all the views from server for the registered model documents into the defined VIEW_PATHS directory.

This method **removes** previous views directory if exist.

**classmethod sync()**
Not implemented yet.

**classmethod upload()**
Uploads all the local views from VIEW_PATHS directory to CouchBase server

This method **over-writes** all the server-side views with the same named ones coming from VIEW_PATHS folder.

couchbasekit.viewsync.**register_view**(*design_doc*, *full_set=True*)
Model document decorator to register its design document view:

```
@register_view('dev_books')
class Book(Document):
    __bucket_name__ = 'mybucket'
    doc_type = 'book'
    structure = {
        # snip snip
    }
```

**Parameters**

- **design_doc** (*basestring*) – The name of the design document.
- **full_set** (*bool*) – Attach full_set param to development views.

CHAPTER

SIX

# INDICES AND TABLES

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

# PYTHON MODULE INDEX

## C