kvkit Documentation

Release 0.1

Shuhao Wu

April 18, 2014

Contents

Contents:

CHAPTER 1

Introduction to KVKit

1.1 Simple Tutorial

KVKit is a natural extension to JSON based document stores. It naturally serve as your model layer.

Using the API is simple. Here's a simple example with a blog:

```
>>> from kvkit import *
>>> from kvkit.backends import slow_memory
>>> class BlogPost(Document):
...     # _backend is required
...     _backend = slow_memory
...
...     title = StringProperty(required=True) # StringProperty auto converts all strings to unicode
...     content = StringProperty() # let's say content is not required.
...     some_cool_attribute = NumberProperty() # Just a random attribute for demo purpose
...
...     def __str__(self): # Totally optional..
...         return "%s:%s" % (self.title, self.content)
```

We can create and save a document:

```
>>> post = BlogPost(data={"title": "hi"})
>>> print post
hi:None
>>> post.save() # Note since we are using the slow_memory backend, this is not actually saved.
<__main__.BlogPost object at ...>
```

Modifying the document is easy too:

```
>>> post.title = "Hello"
>>> post.content = "mrrow"
>>> post.save()
<__main__.BlogPost object at ...>
>>> print post
Hello:mrrow
>>> key = post.key # The key.
```

Since title is required, trying to save an object without a title won't work:

```
>>> another_post = BlogPost(data={"content": "lolol"})
>>> another_post.save()
Traceback (most recent call last):
```

```
    ...
ValidationError: ...
```

We can also try getting some stuff from the db:

```
>>> same_post = BlogPost.get(key) # this is the key from before
>>> print same_post
Hello:mrrow
```

If your data is modified somehow, you need to reload:

```
>>> same_post.reload() # Obviously we haven't changed anything, but if we did, this would get those c
>>> print same_post.title
Hello
```

You can also use dictionary notation:

```
>>> print same_post.title
Hello
>>> print same_post["title"]
Hello
```

You can have attributes not in your schema:

```
>>> same_post.random_attr = 42
>>> same_post.save()
<__main__.BlogPost object at ...>
>>> print same_post.random_attr
42
```

Accessing non-existent attributes will fail:

```
>>> same_post.none_existent
Traceback (most recent call last):
  ...
AttributeError: ...
```

Accessing a non-existent attribute that's **in** your schema will return `None`:

```
>> print same_post.some_cool_attribute  # Remember? We never set this
None
```

You can also delete stuff:

```
>>> same_post.delete()
>>> BlogPost.get(key)
Traceback (most recent call last):
    ...
NotFoundError: ...
```

1.2 Indexes

Backends that support indexes can retrieve data not just via the key, but also some field and values. Here are some examples:

```
class BlogPost(Document):
  _backend = slow_memory

  title = StringProperty(required=True)
```

```
  tags = ListProperty(index=True)
  # Let's say we just store a lowercase username
  author = StringProperty(index=True)


post = BlogPost(data={"title": "hello world", "tags": ["hello", "world"], "author": "john"})
post.save()
```

That just created a post with index and saved the index in whatever the backend prefers. The slow_memory backend doesn't as index operation just iterates through. This is why you shouldn't be using it.

To query indexed data, there are two ways: getting an object, or just get the primary keys:

```
# Get the indexed object. Returns an iterator
for post in BlogPost.index("tags", "hello"):
  # prints hello world once.
  print post.title

# Get the keys only.
# Prints [post.key]
print BlogPost.index_keys_only("tags", "hello")
```

You can also query for a range:

```
# low <= value <= high
for post in BlogPost.index("tags", "h", "i"):
  # prints hello world once
  print post.title
```

1.3 Fancy Properties

JSON documents can really contain anything right? This is absolutely true, but it might still be helpful to use KVKit's properties to define your documents as different backends might have optimizations for different properties.

There are plenty of properties available from the built in that does a lot of different things. For more details you can take a look at the API docs' properties section. However, there are some patterns that we should follow.

By default, kvkit deserializes values into their corresponding python objects on load. Occasionally, this could be slow, such as in the case of a `ReferenceProperty`. When an object with a `ReferenceProperty` loads, by default, it will automatically load the referenced object, and cascade down the tree in a depth first fashion. This could be really bad.

To avoid this, all property initialization takes an optional argument called `load_on_demand`. This moves deserialization when the value is *accessed*.

To illustrate this:

```
class ReferencedDocument(Document):
  _backend = slow_memory

class SomeDocument(Document):
  _backend = slow_memory

  ref = ReferenceProperty(ReferencedDocument, load_on_demand=True)

refdoc = ReferencedDocument()
somedoc = SomeDocument(data={"ref": refdoc})
refdoc.save() # No magic here. You need to save this document first.
```

```
somedoc.save()

d = SomeDocument.get(somedoc.key)
print d.ref.key == refdoc.key # True
```

Note `d.ref` is not actually loaded when you perform the get operation. It is done when you access it and cached there. This is when you should be careful with race conditions as any changes with `refdoc` will not be immediately reflected by `d.ref` until the next reload!

1.4 Property Validators

KVKit provides a way to validate data on save, as well as some convenience methods for checking for validity or which field is invalid. In effect, it could in theory replace your forms library if you don't need things that are too complicated.

Here's an example:

```
class ValidatedDoc(Document):
    _backend = slow_memory

    required = StringProperty(required=True)
    integer = NumberProperty(validators=lambda x: abs(int(x) - x) < 0.00001)

vdoc = ValidatedDoc()

# Check if document is valid
print vdoc.is_valid() # false

# Check which fields are invalid
print vdoc.invalids() # ["required"]
```

Note that only `required` is considered to be invalid as `integer` is not required and accepted as valid.

If we are to do this:

```
vdoc.integer = 1.5
print vdoc.invalids() # ["integer", "required"]
```

Note that calling `invalids` is slightly less efficient than `is_valid` as `is_valid` quits on the first validation error.

You will not be allowed to save a document with validation error:

```
vdoc.save() # Raises ValidationError
```

There's another convenience feature for KVKit that disallows extra attributes for the document:

```
class OnlyDefined(Document):
    _backend = slow_memory
    DEFINED_PROPERTIES_ONLY = True

    a = StringProperty()

odoc = OnlyDefined()
odoc.b = "yay"
print odoc.invalids() # ["_extra_prop"]
```

Note the invalids returns a special `_extra_prop` as a hint saying that there is extra properties.

This is good if you doing something like `Document(data=request.forms)`.

1.5 Embedded Documents

CHAPTER 2

Installation

To install this, simply install from pip:

```
$ pip install kvkit
```

By default, this installation does not come with any backends other than the `slow_memory` backend, which you should not use.

If you need to use the riak backend, you need to install the riak pip package as well:

```
$ pip install riak
```

If you need to use the leveldb backend, you need to install.. TBD:

```
$ # TBD
```

For development of kvkit:

Compile leveldb from source (from http://code.google.com/p/leveldb/), and then install it to the system:

```
# cp --preserve=links libleveldb.* /usr/local/lib
# cp -r include/leveldb /usr/local/include/
# ldconfig
```

Now, start a new virtualenv and do:

```
$ pip install -r requirements.txt
```

CHAPTER 3

Backends

KVKit uses backends to support multiple backend databases. The current supported ones includes: Riak, LevelDB, and a very slow memory implementation for testing.

3.1 Using Backends

Each class has an attribute called `_backend`. This needs to be set to an appropriate backend.

To use a backend that's built in, import from `kvkit.backends`:

```
from kvkit.backends import riak

class YourDocument(Document):
    _backend = riak

    # ...
```

If you use the same backend all the time, you can define a base class:

```
from kvkit.backends import riak

class BaseDocument(Document):
    _backend = riak

class YourDocument(BaseDocument):
    # ... your attributes
```

If your have your own custom backend, you can use that. The current available backends are detailed below.

3.2 Riak Backend

`kvkit.backends.riak` This backend uses Riak to store data.

For more information about Riak, checkout https://basho.com/riak/.

3.3 LevelDB Backend

`kvkit.backends.leveldb`

3.4 Slow Memory Backend

`kvkit.backends.slow_memory` This backend uses the memory to store data.

This should **not** be used as it is very very slow for index operations.

Should also not really use as right now it does not segregate between classes.

A good one for unittests, however.

`kvkit.backends.slow_memory.`**`cleardb`**`()`
Clears the database.

3.5 Writing Backends

There are two approaches in writing a backend. You can either

- start with a module and fill in all the functions listed below, import the module and set it as the backend.
- inherit the class, and then fill in the appropriate functions, import the class, initialize it, and set it as the backend.

Make sure you stick to the format specified in the docs below or else it won't work. Feel free to `raise NotImplementedError` if your backend does not support the operations.

3.6 Base Backend API Documentations for Implementation

This is mostly for writing the backends. When you actually use kvkit, `Document` hides all these nonsense from you.

class `kvkit.backends.base.`**`BackendBase`**
This is a backend base that you can extend.

Used for a class based approach. All documentations are the same as above, except the first argument is now `self`, which refers to the class backend object.

`clear_document`(*doc*)

`delete`(*doc*, *key*, ***args*)

`get`(*cls*, *key*, ***args*)

`index`(*cls*, *field*, *start_value*, *end_value=None*, ***args*)

`index_keys_only`(*cls*, *field*, *start_value*, *end_value=None*, ***args*)

`init_class`(*cls*)

`init_document`(*doc*, ***args*)

`list_all`(*cls*, *start_value=None*, *end_value=None*, ***args*)

`list_all_keys`(*cls*, *start_value=None*, *end_value=None*, ***args*)

`save`(*cls*, *key*, *data*, ***args*)

`kvkit.backends.base.`**`clear_document`**(*self*, ***args*)
Called after the document gets cleared.

Used for clearing meta/indexes, or whatever. Also needs to handle the clearing of the _backend_obj

Args: self: The document

Returns: None

Raises: Nothing. If you raise `NotImplementedError` here, clearing wouldn't work!

`kvkit.backends.base.`**`delete`**(*cls*, *key*, *doc=None*, ***args*)
Deletes cls key from the db.

Args: cls: The class to delete from key: The key to delete doc: If the delete originated from a document, this will be pointing to

the document instance.

****args: additional arguments passed in from `Document.delete_key` or `doc.delete()`.**

Returns: None

Note: If the object does not exists in the backend, this will just return.

`kvkit.backends.base.`**`get`**(*cls*, *key*, ***args*)
Getting the json document from the backend.

Args: cls: The class of document to get from. key: The key to get from **args: Any additional arguments passed from `Document.get`

Returns: (JSON document, backend representation of the object (like RiakObject))

Note: This backend representation would be set as _backend_obj on the Document.

Raises: NotFoundError if this is not found

`kvkit.backends.base.`**`index`**(*cls*, *field*, *start_value*, *end_value=None*, ***args*)
Does an index operation and returns the documents matched.

Args: The same as `index_keys_only`.

Returns: An iterator for (key, json document, backend representation)

Raises: kvkit.exceptions.NotIndexed if not indexed.

Note: This could just be loading from index_keys_only. However, it should attempt to use backend-specific optimizations if available.

`kvkit.backends.base.`**`index_keys_only`**(*cls*, *field*, *start_value*, *end_value=None*, ***args*)
Does an index operation but only return the keys.

Args: cls: The class that the index operation is coming from. field: The field to query the index with. start_value: If `end_value` is None, this is the exact value to match.

Otherwise this is the start value to match from.

end_value: If not None, a range query is done. The range is: `start_value <= matched <= end_value`.

**args: Any additional keyword arguments passed in at `Document.index_keys_only`.

Returns: An iterator of keys only.

Raises: kvkit.exceptions.NotIndexed if not indexed. This is optional, however, as backends might be able to index anyway..

Note: This exists because some backends might have optimizations getting only keys as oppose to the document itself.

kvkit.backends.base.**init_class**(*cls*)
Called right after a class has been initialized.

Can be used to transform the class a little with backend specific metadata. Alternatively, you can use this to initialize your backend with information about the class rather than sticking the info right on the class.

It is encouraged that if and when you transform classes, you use a namespace like cls.<backend>_meta = {} and stick your variables in there.

Furthermore, this function should not fail when whatever variables you defined is not present because tests might set those options /after/ the class has already been initialized by the metaclass. The tests will set the variables and manually call `init_class` again.

It could also be the case that this is a parent base class.

Args: cls: The class that was just created.

Returns: None

Raises: Nothing. If you raise `NotImplementedError` here, nothing would work!

kvkit.backends.base.**init_document**(*self*, ***args*)
Called right after a document has been initialized.

Args: self: The document itself. ******args: The additional keyword arguments passed in at `Document.__init__`

Returns: None

Raises: Nothing. If you raise `NotImplementedError` here, nothing would work!

kvkit.backends.base.**list_all**(*cls*, *start_value=None*, *end_value=None*, ***args*)
List all the objects for this class.

Args: The same as list_all_keys

Returns: An iterator for (key, json document, backend_obj)

Note: This could just be loading from list_all_keys. However, it should attempt to use backend-specific optimizations if available.

kvkit.backends.base.**list_all_keys**(*cls*, *start_value=None*, *end_value=None*, ***args*)
Lists all the keys for this class.

Args: cls: The class to list from. start_value: Sometimes you only want a range of these keys. This is the

start value.

end_value: The end value of the range. The range is `start_value <= v <= end_value`.

****args: additional keyword arguments passed in from** `Document.list_all_keys`.

Returns: An iterator of all keys.

Note: This exists because some backends might have optimizations getting only keys as oppose to the document itself.

kvkit.backends.base.**post_deserialize**(*self*, *data*)
Runs after deserializing an object.

This is probably because the object has been loaded from the db.

Args: self: The document object after deserializing. data: The original data deserialized.

Returns: None

Raise: None or else deserializing won't work.

kvkit.backends.base.**save**(*self*, *key*, *data*, ***args*)

Saves a key and a json document into the backend.

Args: self: The document object to be saved. key: The key to save data: The json document to save. **args: The arguments passed from `doc.save()`

Returns: None

CHAPTER 4

API Documentations

Note that even though everything here says kvkit.xxxxx.xxxx, you can access all of the documented stuff here from just kvkit.xxx

like:

```
from kvkit import Document
```

4.1 Document

class kvkit.document.**Document**(*key=<function <lambda> at 0x3133050>, data={}, backend_obj=None, **args*)

Bases: kvkit.emdocument.EmDocument

Initializes a new document.

Args:

key: A key for the object. Or a function that returns a key. By default it generates an uuid1 hex.

data: The data to be merged in. backend_obj: The object representation for the backend. Stored as self._backend_obj. Should not be touching unless you're the backend.

**args: gets passed into EmDocument's __init__

DEFINED_PROPERTIES_ONLY = False

clear(*to_default=True*)

delete(***args*)

Deletes this object from the db.

Will also clear this document.

Usually more efficient to use YourDocument.delete_key. However, if you already have the object loaded...

Returns: self

classmethod delete_key(*key*, ***args*)

Deletes a document with a key from the database.

Args: key: The key to be deleted.

Note: This is usually more efficient than YourDocument(key).delete() as that involves a get operation.

deserialize(*data*)

classmethod **get**(*key*, ***args*)
Gets an object from the db given a key.

Args: key: The key

Returns: The document.

Raises: NotFoundError if not found.

classmethod **get_or_new**(*key*, ***args*)
Gets an object from the db given a key. If fails, create one.

Note that this method does not save the object, nor does it populate it.

Args: key: The key

Returns: The document. If it is not available from the db, a new one will be created.

classmethod **index**(*field*, *start_value*, *end_value=None*, ***args*)
Uses the index to find documents that matches.

Args: field: the property/field name.

start_value: The value that you're indexing for. If an end_value is not provided, it has to be an exact match (field == start_value).

end_value: the end value for a range query. If left to be None, it will match exact with start_value. Otherwise the range is start_value <= value <= end_value

Returns: An iterator of loaded documents. Loaded at each iteration to save time and space.

classmethod **index_keys_only**(*field*, *start_value*, *end_value=None*, ***args*)
Uses the index to find document keys.

Args: field: the property/field name.

start_value: The value that you're indexing for. If an end_value is not provided, it has to be an exact match (field == start_value).

end_value: the end value for a range query. If left to be None, it will match exact with start_value. Otherwise the range is start_value <= value <= end_value

Returns: A list/iterator of keys that matches the query in arbitrary order that depends on the backend.

invalids()
Get all the attributes' names that are invalid.

Returns: A list of attribute names that have invalid values.

is_valid()
Test if all the attributes pass validation.

Returns: True or False

classmethod **list_all**(*start_value=None*, *end_value=None*, ***args*)
List all the objects.

Args: start_value: if specified, it will be the start of a range. end_value: if specified, it will be the end of a range.

Returns: A generator with the following properties: allobjs[start_value:] if only start_value is given, allobjs[start_value:end_value+1] if end_value is given. The +1 is largely a metaphor, as it will return all the values that == end_value.

classmethod **list_all_keys**(*start_value=None*, *end_value=None*, ***args*)
List all the keys from the db.

Args: start_value: if specified, it will be the start of a range. end_value: if specified, it will be the end of a range.

Returns: allkeys[start_value:] if only start_value is given, allkeys[start_value:end_value+1] if end_value is given. The +1 is largely a metaphor, as it will return all the values that == end_value.

classmethod **load**(*data*)
A convenient method that creates an object and deserializes the data.

Args: data: The data to be deserialized

Returns: A document with the data deserialized.

merge(*data*, *merge_none=False*)
Merge the data from a non-db source.

This method treats all *None* values in data as if the key associated with that *None* value is not even present. This will cause us to automatically convert the value to that property's default value if available.

If None is indeed what you want to merge as oppose to the default value for the property, set *merge_none* to *True*.

Args:

data: The data dictionary, a json string, or a foreign document to merge into the object.

merge_none: Boolean. If set to True, None values will be merged as is instead of being converted into the default value of that property. Defaults to False.

Returns: self

reload(***args*)
Reloads an object from the database.

It will update inplace. Keyword arguments are passed to the backend.

Returns: self

Raises: NotFoundError

save(***args*)
Saves an object into the db.

Keyword arguments are passed to the backend.

Returns: self

Raises: ValidationError

serialize(*include_key=False*, ***args*)
Serializes a document.

Args: include_key: If true, the key will be available as the field "key"

4.2 EmDocument

class kvkit.emdocument.**EmDocument**(*data={}*)
Embedded document as a JSON object

Class Variables:

- ***DEFINED_PROPERTIES_ONLY*: A boolean value indicating that the only** properties allowed are the ones defined. Defaults to False. If a violation is found, an error will be raised on *serialize* (and *save* for Document) and False will be returned in *is_valid*. *invalids* will return *"_extra_props"* in the list.
- defined_properties: A list of defined properties. For read only.

Initializes a new EmDocument

Args:

data: A dictionary that's supposed to be initialized with the document as attributes.

DEFINED_PROPERTIES_ONLY = False

clear(*to_default=True*)
Clears the object. Set all attributes to default or nothing.

Args:

to_default: Boolean. If True, all properties defined will be set to its default value. Otherwise the document will be empty.

Returns: self

deserialize(*data*)
Deserializes the data. This uses the *from_db* method of all the properties. This differs from *merge* as this assumes that the data is from the database and will convert from db whereas merge assumes the the data is from input and will not do anything to it. (unless the property has *on_set*).

Args: data: The data dictionary from the database.

Returns: self, with its attributes populated.

invalids()
Get all the attributes' names that are invalid.

Returns: A list of attribute names that have invalid values.

is_valid()
Test if all the attributes pass validation.

Returns: True or False

classmethod load(*data*)
A convenient method that creates an object and deserializes the data.

Args: data: The data to be deserialized

Returns: A document with the data deserialized.

merge(*data*, *merge_none=False*)
Merge the data from a non-db source.

This method treats all *None* values in data as if the key associated with that *None* value is not even present. This will cause us to automatically convert the value to that property's default value if available.

If None is indeed what you want to merge as oppose to the default value for the property, set *merge_none* to *True*.

Args:

data: The data dictionary, a json string, or a foreign document to merge into the object.

merge_none: Boolean. If set to True, None values will be merged as is instead of being converted into the default value of that property. Defaults to False.

Returns: self

serialize(*dictionary=True*, *restricted=()*)
Serializes the object into a dictionary with all the proper conversions

Args:

dictionary: boolean. If True, this will return a dictionary, otherwise the dictionary will be dumped by json.

restricted: The properties to not output in the serialized output. Useful in scenarios where you need to hide info from something. An example would be you need to hide some sort of token from the user but you need to return the object to them, without the token. This defaults to tuple(), which means nothing is restricted.

Returns: A plain dictionary representation of the object after all the conversion to make it json friendly.

4.3 Exceptions

exception kvkit.exceptions.**DatabaseError**

exception kvkit.exceptions.**KVKitError**

exception kvkit.exceptions.**NotFoundError**

exception kvkit.exceptions.**NotIndexed**

exception kvkit.exceptions.**ValidationError**

4.4 Properties

class kvkit.properties.standard.**BaseProperty**(*required=False*, *default=<object object at 0x2f9b220>*, *validators=[]*, *load_on_demand=False*, *index=False*)
The base property that all other properties extends from.

Args:

required: If True and if a value is None, save will raise a ValidationError. Defaults to False.

default: The default value that document populates when it creates a new object. It could be a function that takes no arguments that returns a value to be filled as a default value. If this is not given, no default value is specified, the default value for that property will be None.

validators: A function or a list of functions that takes a value and returns True or False depending on if the value is valid or not. If a function returns False, save will fail with ValidationError and is_valid will return False. Defaults to []

load_on_demand: If True, the value will not be deserialized until the programmer accesses the value. Defaults to False, which means value will be deserialized when the object is loaded from the DB.

index: If True, the database backend will create an index for this entry allowing it to be queried via index. This is only valid for StringProperty, NumberProperty, ListProperty, and ReferenceProperty. Individual backends may have more but will not necessarily be portable.

default()
Returns the default value of the property.

It will first try to return the default you set (either function or value). If not it will return None.

Returns: The default value specified by you or the type. Or None.

from_db(*value*)
Converts the value from to_db back into its original form.

This should be the inverse function of to_db and it also shouldn't convert None into anything else other than itself.

Args: value: Value from the database

Returns: Whatever that type wants to be once its in application code rather than the db form.

Note: It is important if you want to implement this that it is actually and inverse function of to_db. Do not convert None unless you absolutely mean it.

If load_on_demand is True, this method will only be called the first time when the property is accessed.

to_db(*value*)
Converts a value to a database friendly format (serialization).

This should never convert None into anything else.

Args: value: The value to be converted

Returns: Whatever the property wants to do to that value. The BaseProperty version does nothing and just returns the value.

Note: If you want to implement this version yourself it the return value of this should be JSON friendly. Do not convert None unless you absolutely mean it.

validate(*value*)
Validates a value.

This invokes both the builtin validator as well as the custom validators. The BaseProperty version only validates with custom ones if available.

Args: value: The value to be validated.

Returns: True or False depending on if the value is valid or not.

Note: If you are writing your own property class, subclass this one and in the validate method, put the following first to validate the custom validators:

```
if not BaseProperty.validate(self, value):
  return False
```

class kvkit.properties.standard.**BooleanProperty**(*required=False*, *default=<object object at 0x2f9b220>*, *validators=[]*, *load_on_demand=False*, *index=False*)
Boolean property. Values will be converted to boolean upon save.

The base property that all other properties extends from.

Args:

required: If True and if a value is None, save will raise a ValidationError. Defaults to False.

default: The default value that document populates when it creates a new object. It could be a function that takes no arguments that returns a value to be filled as a default value. If this is not given, no default value is specified, the default value for that property will be None.

validators: A function or a list of functions that takes a value and returns True or False depending on if the value is valid or not. If a function returns False, save will fail with ValidationError and is_valid will return False. Defaults to []

load_on_demand: If True, the value will not be deserialized until the programmer accesses the value. Defaults to False, which means value will be deserialized when the object is loaded from the DB.

index: If True, the database backend will create an index for this entry allowing it to be queried via index. This is only valid for StringProperty, NumberProperty, ListProperty, and ReferenceProperty. Individual backends may have more but will not necessarily be portable.

to_db(*value*)

class kvkit.properties.standard.**DictProperty**(***args*)
Dictionary property. Or a JSON object when stored.

Value must be an instance of a dictionary (or subclass).

validate(*value*)

class kvkit.properties.standard.**EmDocumentProperty**(*emdocument_class*, ***args*)
Embedded document property.

Value must be an EmDocument or a dictionary

Initializes a new embedded document property.

Args: emdocument_class: The EmDocument child class. Everything else are inheritted from BaseProperty

from_db(*value*)

to_db(*value*)

validate(*value*)

class kvkit.properties.standard.**EmDocumentsListProperty**(*emdocument_class*, ***args*)
A list of embedded documents.

Probably shouldn't abuse this.

Initializes a new embedded document property.

Args: emdocument_class: The EmDocument child class. Everything else are inheritted from BaseProperty

from_db(*value*)

to_db(*value*)

validate(*value*)

class kvkit.properties.standard.**ListProperty**(***args*)
List property.

Value must be an instance of a list/tuple (or subclass).

validate(*value*)

class kvkit.properties.standard.**NumberProperty**(***kwargs*)
For storing real numbers.

This always converts to a floating point, unless integer is specified.

Number Property

Args:

integer: If true, this will be marked as an integer field instead of a floating point field. Defaults to false.

to_db(*value*)

validate(*value*)
Checks if value is a number by taking *float(value)*.

kvkit.properties.standard.**Property**
alias of BaseProperty

class kvkit.properties.standard.**ReferenceProperty**(*reference_class*, *strict=False*, ***kwargs*)
Reference property references another Document.

Stores the key of the other class in the db and retrieves on demand. Probably shouldn't even use this as index is better in most scenarios.

Initializes a new ReferenceProperty

Args:

reference_class: The Document child class that this property is referring to.

strict: If True and if the object is not found in the database loading it, it will raise a NotFoundError. If load_on_demand is False and it tries to load this property with a non-existent object, it will also raise a NotFoundError.

from_db(*value*)

to_db(*value*)

validate(*value*)

class kvkit.properties.standard.**StringProperty**(*required=False*, *default=<object object at 0x2f9b220>*, *validators=[]*, *load_on_demand=False*, *index=False*)
For storing strings.

Converts all value into unicode when serialized. Deserializes all to unicode as well.

The base property that all other properties extends from.

Args:

required: If True and if a value is None, save will raise a ValidationError. Defaults to False.

default: The default value that document populates when it creates a new object. It could be a function that takes no arguments that returns a value to be filled as a default value. If this is not given, no default value is specified, the default value for that property will be None.

validators: A function or a list of functions that takes a value and returns True or False depending on if the value is valid or not. If a function returns False, save will fail with ValidationError and is_valid will return False. Defaults to []

load_on_demand: If True, the value will not be deserialized until the programmer accesses the value. Defaults to False, which means value will be deserialized when the object is loaded from the DB.

index: If True, the database backend will create an index for this entry allowing it to be queried via index. This is only valid for StringProperty, NumberProperty, ListProperty, and ReferenceProperty. Individual backends may have more but will not necessarily be portable.

to_db(*value*)

class kvkit.properties.fancy.**DateTimeProperty**(***args*)
Stores the datetime in UTC seconds since the unix epoch.

Takes in a datetime.

from_db(*value*)

to_db(*value*)

validate(*value*)

class kvkit.properties.fancy.**EnumProperty**(*possible_values*, ***args*)
Stores a choice out of some possible values.

Stores in the database as 1, 2, 3 to save space. This is probably not really used as most of the time a string property is sufficient.

Initializes a new EnumProperty.

Args:

possible_values: A list of a values that are possible in strings. Only these values will be allowed to set to this after.

from_db(*value*)

to_db(*value*)

validate(*value*)

class kvkit.properties.fancy.**PasswordProperty**(*required=False*, *default=<object object at 0x2f9b220>*, *validators=[]*, *load_on_demand=False*, *index=False*)
Password property that's secure if bcrypt is installed.

MAKE SURE YOU ARE USING BCRYPT IN PRODUCTION IF YOU USE THIS.

When the value is set (i.e. document.password = "mypassword"), a salt will automatically be generated and document.password from now on will {"salt": salt, "hash": hash}. This is how it will be stored into the db.

The base property that all other properties extends from.

Args:

required: If True and if a value is None, save will raise a ValidationError. Defaults to False.

default: The default value that document populates when it creates a new object. It could be a function that takes no arguments that returns a value to be filled as a default value. If this is not given, no default value is specified, the default value for that property will be None.

validators: A function or a list of functions that takes a value and returns True or False depending on if the value is valid or not. If a function returns False, save will fail with ValidationError and is_valid will return False. Defaults to []

load_on_demand: If True, the value will not be deserialized until the programmer accesses the value. Defaults to False, which means value will be deserialized when the object is loaded from the DB.

index: If True, the database backend will create an index for this entry allowing it to be queried via index. This is only valid for StringProperty, NumberProperty, ListProperty, and ReferenceProperty. Individual backends may have more but will not necessarily be portable.

static check_password(*password*, *record*)
Checks if a password matches a record.

Args: password: the plain text password. record: The record in the db, which is in the format of {"salt": salt, "hash": hash}.

Returns: True or false depending of the password is the right one or not.

Note: Typically you would just do:

```
PasswordProperty.check_password(req.forms["password"],
                                user.password)
```

or something to that effect.

on_set(*value*)

kvkit.properties.fancy.**generate_salt**()
Generates a random salt.

If bcrypt is not installed, it uses os.urandom to select from a-Z0-9 with a length of 25.

With bcrypt installed, it uses bcrypt.gensalt().

MAKE SURE YOU ARE USING BCRYPT IN PRODUCTION IF YOU USE THIS.

Returns: A salt.

kvkit.properties.fancy.**hash_password**(*password*, *salt*)
Takes a plaintext password and a salt and generates a hash.

If bcrypt is installed, it uses bcrypt.hashpw.

Without bcrypt, it uses sha256(password + salt).

MAKE SURE YOU ARE USING BCRYPT IN PRODUCTION IF YOU USE THIS.

Args: password: the plain text password salt: the salt.

Returns: A hash.

CHAPTER 5

Indices and tables

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

k