OGitM Documentation

Release 0.0.1

Jonathan Frere

April 11, 2015

Contents:

CHAPTER 1

Installing OGitM

Todo

Actually write this page

CHAPTER 2

The Tutorial

2.1 Using The OGitM Model

Starting with OGitM is usually as simple as writing out the model declaration.

```
>>> import tempfile; db_directory = tempfile.TemporaryDirectory()
>>> import ogitm

>>> class Person(ogitm.Model, db=db_directory.name):
...
...     name = ogitm.fields.String()
...     age = ogitm.fields.Integer(min=0)
...     hobby = ogitm.fields.Choice(["football", "karate", "knitting"],
...                                 default="karate")
```

Note that the db parameter is mandatory - it specifies the place that the git repository will be stored. Currently, this system uses bare repositories. Multiple models stored in the same database location will be stored in individual tables by default. (The current implementation uses the inflection library's `tableize` method (based on RoR's tableize method).)

Here, we've used a string as the path to the directory. However, we can also separately instantiate a `ogitm.gitdb.GitDB` instance, and use that instead. Note that opening two databases with the same directory means that the two databases will point to the same place.

```
>>> db = ogitm.gitdb.GitDB(db_directory.name)
>>>
>>> class AlternatePerson(ogitm.Model, db=db):
...     pass
```

The next thing to do is to start inserting documents into the database. That's exactly as simple as it should be.

```
>>> bob = Person(name="bob", age=32, hobby="football")
>>> geoff = Person(name="geoff", age=18, hobby="knitting")
>>> roberta = Person(name="roberta", age=42, hobby="football")
>>> print(bob.age)
32
>>> print(geoff.hobby)
knitting
>>> print(roberta.name)
roberta

>>> bob.age = 33  # Ah, how time changes us all
>>> bob.hobby = "knitting"
>>> bob.save() == bob.id
True
```

The limitations on the fields will also stop you doing anything stupid by raising errors all over the place. They'll also automatically insert default values.

```
>>> roberta.age = -3
Traceback (most recent call last):
    ...
ValueError: ...

>>> roberta.hobby = "this is not a recognised hobby"
>>> # Note lack of error message here
>>> print(roberta.hobby)  # defaults to "karate" as specified in model
karate
```

More useful than just storing data is being able to retrieve it later. The easiest way to do that is by searching for it.

```
>>> Person.find(name="bob").first() == bob
True
>>> Person.find(age=19).all()  # No people aged 19
[]
>>> Person.find(hobby="knitting").all() == [bob, geoff]
True
```

Note that this also works for more complex queries. We can also chain queries together.

```
>>> len(Person.find(age={'gt': 2}))  # Matches all current documents
3
>>> len(Person.find(age={'gt': 2}, hobby={'startswith': 'kn'}))
2
>>> # same as
>>> len(Person.find(age={'gt': 2}).find(hobby={'startswith': 'kn'}))
2
>>> # complex queries may contain more than one operator at a time
>>> len(Person.find(age={'gt': 2, 'lt': 40}))
2
```

2.2 Using GitDB Directly

Note that OGitM is essentially a wrapper around GitDB. If you need access to GitDB as a simple document store, this is possible using the `ogitm.gitdb` module.

```
>>> import tempfile; db_directory = tempfile.TemporaryDirectory()
>>> from ogitm import gitdb
>>> db = gitdb.GitDB(db_directory.name)
```

A GitDB database is split up into tables. GitDB automatically creates the `__defaulttable__` table, and passes any methods called on it straight to an internal copy of that table. This allows for very simple usage of GitDB. However, it is more likely that a user would want to split up their data into multiple tables.

```
>>> db.default_table.name
'__defaulttable__'
>>> db.table('Table Name')
<ogitm.gitdb.Table object at ...>
>>> table = db.table('Table Name')
>>> doc = table.insert({'my doc': 'your doc'})
>>> table.get(doc)
{'my doc': 'your doc'}
>>> table.find_items({'my doc': 'your doc'})  # Using simple query
[{'my doc': 'your doc'}]
>>> table.find_items({'my doc': {'exists': True}})  # Using advanced query
[{'my doc': 'your doc'}]
```

CHAPTER 3

Searching In GitDB

Similarly to MongoDB, GitDB queries can either be really simple checks of equality, or more complex tests of a value using Python's extensive set of methods and functions. This page will describe how to use the search functions, for information on writing your own, see `gitdb.search_functions`. There is also an extensive testing suite for searching in the `tests/test_gitdb.py` file.

We will use a model that looks like this:

```
>>> import tempfile; db_directory = tempfile.TemporaryDirectory()
>>>
>>> import ogitm
>>> class MyModel(ogitm.Model, db=db_directory.name):
...     name = ogitm.fields.String()
...     age = ogitm.fields.Integer()
...     has_hair = ogitm.fields.Boolean()
>>> bob = MyModel(name="Bob", age=93, has_hair=False)
>>> bert = MyModel(name="Bert", age=23, has_hair=True)
>>> bex = MyModel(name="Bex", age=32, has_hair=True)
>>> bub = MyModel(name="Bubba", age=3243, has_hair=False)
```

3.1 Scalar Searches

Checking if something equals something else is the easiest check of all.

```
>>> MyModel.find(name="Bex").first() == bex
True
>>> MyModel.find(age=34).all()
[]
>>> len(MyModel.find(has_hair=True))
2
```

3.2 Comparison

The comparison operators (>, <, >=, <=, and ==) are supported with aliases. Generally, the query `a={'>': b}` will return all values of `a` such that `a > b`.

Operator	Shorthand	Longhand
>	'gt'	'greater-than'
<	'lt'	'less-than'
>=	'gte'	'greater-than-equal'
<=	'lte'	'less-than-equal'
==	'eq'	'equal'

```
>>> len(MyModel.find(age={'lt': 30}))
1
>>> len(MyModel.find(age={'gte': 32}))
3

>>> # Note that this also works for any
>>> # other type with a total ordering
>>> len(MyModel.find(name={'lt': 'Bf'}))
2
>>> # 'eq' will work for any two equivalent items
>>> MyModel.find(name={'eq': 'Bert'}) == MyModel.find(name='Bert')
True
```

3.3 String Checks

String types can be checked using the various `is*()` string methods, as well as `startswith()` and `endswith()`. These are hardcoded, but delegate to the string's natural methods. If you can think of some way of automatically selecting all string methods that return a boolean, please let me know!

```
>>> len(MyModel.find(name={'startswith': 'B'}))
4
>>> len(MyModel.find(name={'isalpha': True}))
4
```

3.4 Existence

Testing for existence isn't usually necessary when using models, as (assuming that you only use the model to insert documents), you know that the only fields that will exist will be the fields you inserted. It is more useful when using arbitrary documents with the raw GitDB instance. However, the syntax of the check is the same in both cases.

```
>>> len(MyModel.find(name={'exists': True}))
4
>>> len(MyModel.find(name={'exists': False}))
0
```

OGitM API Documentation

Contents:

4.1 API Docs: OGitM

class `ogitm.`**`Model`**`(`**args*, ***kwargs*`)`

Base class for models

Subclass this class to declare a model. `Model` provides a default initialiser, an equivalence function, and the `save()` and `find()` methods.

Parameters

- **model_id** (*int*) – If this is provided, the model will be initialised with the attributes specified by the document with that id in the database. This is generally for internal use (i.e, creating the object after a search has been completed) but it may be useful.
- **kwargs** (*mixed*) – This is the more usual way of initialising the model - that is, by passing in key=val pairs describing the values passed to the fields specified by the model. The default initialiser will then go through all of the arguments, check that they are known fields, and assign them.

Attribute id The instance will always have this attribute referring to the id that it refers to in the database. It will be None if the instance has not been inserted into the database (this should never happen, though!)

classmethod **`find`**`(`***kwargs*`)`

Finds documents in the database.

Given keyword arguments (which have the same format as the arguments given to `gitdb.GitDB.find()`), this method returns a `ReturnSet` containing all of the matching documents.

Parameters **kwargs** (*mixed*) – See `gitdb.GitDB.find()` for the full finding syntax.

Returns `ReturnSet` of all of the matching documents.

classmethod **`get_table`**`()`

Returns the table associated with this model.

`save``()`

Saves this instance to the database.

If this instance has been saved before, this will update the database document corresponding to the current id. Otherwise, it will insert a new document into the database, storing the document id.

class `ogitm.`**`ReturnSet`**(*ids*, *cls*)

A class representing the documents returned by a particular query.

This class can be used to further narrow down the search (`find()`), or return initialised instances of the model that found them (`first()`, `all()`, `__getitem__()`). It can also tell you how many items the set currently contains (`__len__()`)

The documents are returned sorted in order of the ids. This ensures that further operations on a set will preserve order, but should not be relied on, as the specifics of document ids is not part of the public interface.

`all`()

Returns a list of all of the documents.

`find`(***kwargs*)

Refine the terms of the original search

Using the model that created this set, find which documents match the new queres, then update this set to point to the intersection of both the old and new queries.

Parameters **kwargs** (*mixed*) – See `Model.find()`.

Returns This set, to allow for chaining method calls.

`first`()

Returns the first document, or None

class `ogitm.`**`MetaModel`**(*name*, *bases*, *dct*, ***kwargs*)

Metatype for OGitM Models

Generally, a user should subclass `Model` instead of touching this class. However, it is useful to be aware of the functions that MetaModel can perform. Firstly, upon instantiation it removes all class attributes that extend `BaseField`, and assigns them to an internal dict. It also ensures that any class that overrides `__init__()` calls the super method. This makes sure that the Model should always be in a useable state.

It also provides the `get_attributes()` class method which can be used to get the data for any particular class.

classmethod **`get_attributes`**(*instance*)

Get fields for a Model class or instance

Parameters **instance** (*type or instance*) – An instance or class that has MetaModel as a metatype.

Returns Dictionary of key -> field pairs

Raises KeyError if the type or instance is not recognised

4.2 API Docs: Fields

class `ogitm.fields.`**`BaseField`**(***kwargs*)

Abstract Base Class for field types.

Cannot be instantiated, but should be inherited to provide all the useful information that a field might need.

Parameters

- **default** (*any*) – A default value to provide if the input is ever None. If not provided, and nullable is False, a field will not accept None as an argument.
- **nullable** (*bool*) – True if this field can be None/null, False otherwise. Defaults to True.

- **coerce** – A function that can coerce any input into input of a valid type. If it cannot coerce, it should either return "False" or raise a ValueError. Defaults to a no-op.

  Example: `coerce=int` would convert values to int where possible.

check(*val*)

Base case method to check if a value is allowed by this field.

Must be overriden. Currently only returns True, but may do its own checking in future, and so should probably be checked before any overriden method.

Parameters **val** (*any*) – Value to check

Returns Whether that value is allowed by the parameters given to this field.

coerce(*val*)

Attempt to coerce a value using the pre-defined function.

If no function was passed in, the default operation is to return the value straight through. If the function fails to coerce (i.e. raises ValueError), the value is returned unchanged. (*type_check* should therefore always be used to check the type of a coerced value.)

Parameters **val** (*any*) – Value to coerce

Returns Coerced value

type_check(*val*, *typ=None*)

Check if value is of a certain type (using nullability).

If this field instance can be nulled, checks if the val is either of type `typ` or of the None type. Otherwise, it just checks if the val is of type `typ`. Note that `typ` is passed straight through to `isinstance`, so it can be any value allowed by the second parameter of `isinstance`.

Parameters

- **val** (*any*) – Value to check
- **typ** – Type(s) to check against

Returns Whether val is of type `typ`.

class `ogitm.fields.`**String**(***kwargs*)

A field representing string types.

Parameters

- **regex** (*str or regex*) – Regular expression that this string must match. If not present, any string will match. Can be either a regular expression object, or a string.
- **maxlen** (*int*) – Maximum length of the string. 'None' (default) for no length restrictions.

class `ogitm.fields.`**Number**(***kwargs*)

A field representing real numeric types.

Parameters

- **min** (*numeric*) – The minimum (inclusive) value that this field can contain. If not specified, there is no minimum.
- **max** (*numeric*) – The maximum (inclusive) value that this field can contain. If not specified, there is no maximum.

class `ogitm.fields.`**Float**(***kwargs*)

A field representing floating point numbers.

Parameters

- **min** (*numeric*) – See `Number`
- **max** (*numeric*) – See `Number`

class `ogitm.fields.`**`Integer`**`(`*`**kwargs`*`)`

A field representing integers.

Parameters

- **min** (*numeric*) – See `Number`
- **max** (*numeric*) – See `Number`

class `ogitm.fields.`**`Boolean`**`(`*`**kwargs`*`)`

A field representing boolean values

See `coerce_boolean()` for a useful coercion function for this field.

`ogitm.fields.`**`coerce_boolean`**`(`*`val`*`)`

A useful function for coercing various types to boolean.

Unlike the usual Python `bool()` function which simply tests if a value is empty, this matches boolean `True`, strings in the set `{'yes', 'y', 'true', 't', 'on'}` and the integer `1` for `True`, or boolean `False`, strings in the set `{'no', 'n', 'false', 'f', 'off'}` and the integer `0` for `False`.

This is done in a case-insensitive manner. If the value is a string not in the described sets, a number that doesn't equal `1` or `0`, or any other type (excepting `boolean` of course), this function will raise `ValueError`.

class `ogitm.fields.`**`Choice`**`(`*`choices=None`*, *`**kwargs`*`)`

A field representing a single item from a set of items.

Parameters **choices** (*collection*) – A required collection of items. The check method will then ensure that the value must be in this collection.

4.3 API Docs: GitDB

class `ogitm.gitdb.`**`GitDB`**`(`*`location`*`)`

The raw database class.

This class constructs a database instance in the location described. This is automatically created under the covers by `ogitm.Model`, but it can also be created and used outside the confines of Object-Model mappings. Total freedom!

Any methods called on GitDB that can't be found will be passed to the default `Table` instance, so this class could be used as a simple one-table document store without worrying about tables at all. This isn't recommended, however.

Parameters **location** (*str*) – The path of the database

`drop``(`*`table_name`*, *`force=False`*`)`

Completely and irevocably destroy a table.

Parameters

- **table_name** (*str*) – The name of the table to destroy
- **force** (*bool*) – If true, no errors will be raised if the table does not exist

Raises `ValueError` – if the table is reserved, or could not be deleted for other reasons

`table``(`*`table_name`*`)`

Create a new table.

This creates a new table in the current database. You can also use the form `gitdb['table name']`, which delegates to this method. If a table exists, this method will return a new instance of Table pointing to the same table. (Note that two tables pointing to the same location will always return equal.)

Parameters **table_name** (*str*) – The name this table will take

Raises `ValueError` – if the name is a reserved table name

class `ogitm.gitdb.`**`Table`**`(`*name*, *location*`)`

A class to represent an individual table in a database

This class should only really be created by a `GitDB` instance, although instantiating it manually won't actually change the way this class operates.

Parameters

- **name** (*str*) – The name of the table
- **path** (*str*) – The path of the table (Note that this is the path to this particular table's location, not the root path of the database.)

`begin_transaction``()`

Opens a new transaction.

Raises `ValueError` – if a transaction is already open

See also:

`transaction()` a context manager that automatically handles most of the details of a transaction

`commit()` **and** **`rollback()`** methods for closing the transaction created here

`commit``()`

Commits all work performed during a transaction.

Raises `ValueError` – if this method is called inside the `transaction()` context manager, or if there is no open transaction when this method is called

See also:

`transaction()` a context manager that automatically handles most of the details of a transaction

`begin_transaction()` opens up a transaction

`rollback()` rolls back instead of committing

`find``(`*where*`)`

Finds the documents that match a given query.

For details on searching, see *Searching In GitDB*. Searches in the raw `GitDB` should be documents, rather than keyword arguments, but otherwise searches are the same.

This method returns (id, document) pairs. There are also the convenience methods `find_ids()` and `find_items()`, which just return the ids and documents respectively.

Parameters **where** (*dict*) – Search definition

Returns *list[(int, dict)]* – A list of matching documents

`find_ids``(`*where*`)`

Find the ids that match a given query.

This method is the same as `find()`, but returns the ids rather than (id, doc) pairs.

Parameters **where** (*dict*) – Search definition (see `find()`)

Returns *list[int]* – A list of matching document ids

`find_items`(*where*)
Find the documents that match a given query.

This method is the same as `find()`, but returns the documents rather than (id, doc) pairs.

Parameters **where** (*dict*) – Search definition (see `find()`)

Returns *list[dict]* – A list of matching documents

`find_one`(*where*)
Finds one document

This method functions the same as `find()`, but returns just one element, or None if no element found.

Parameters **where** (*dict*) – Search definition (see `find()`)

Returns *(int, document)* or *None*

`get`(*doc_id*)
Gets a document given it's document id.

This is the simplest but least useful way of getting information out of the database. It returns the document.

Parameters **doc_id** (*int*) – The document ID to fetch

Returns *dict* – The document

`insert`(*document*)
Inserts a document into this database.

Documents are key-value python dicts. Nested documents are not currently tested, and will probably break everything. Documents also can't be scalar objects, although again that is untested and behaviour is therefore undefined in that area as well. Those should probably be tested and defined more rigorously.

Oh, and also the only allowed keys and values are the standard primitives (str, int, bool, float, etc), not other objects or collections.

If a transaction is not open, this method will commit all changes into the database.

Parameters **document** (*dict*) – A key-val single-level dictionary

Returns *int* – Document ID

`revert_steps`(*steps*, *doc_id=None*)
Reverts the whole database a number of steps.

Parameters

- **steps** (*int*) – The number of steps to revert
- **doc_id** (*int*) – Not implemented yet

See also:

`revert_to_state()` Another way of reverting changes to the database

`revert_to_state`(*state*, *doc_id=None*)
Reverts the whole database to a previously stored state.

Parameters

- **state** (*oid*) – The state to return to
- **doc_id** (*int*) – Not implemented yet

See also:

`revert_steps()` Another way of reverting changes to the database

`save_state()` A method that allows saving the state of the database

rollback()

Rolls back all work performed during a transaction.

Raises `ValueError` – if this method is called inside the `transaction()` context manager, or if there is no open transaction when this method is called.

See also:

`transaction()` a context manager that automatically handles most of the details of a transaction

`begin_transaction()` opens up a transaction

`commit()` commits instead of rolling back

save(*msg=''*)

Commits all current unsaved changes

Normally, this will be automatically called by any methods that make changes, or by the transaction methods. This shouldn't be called otherwise, unless in exceptional circumstances (in which case, file an issue because something's probably gone wrong.)

Parameters **msg** (*str*) – This will become git's commit message

save_state()

Returns a marker that can be used later to revert to the same state.

Because the database is built on top of git, all states are saved, and can be checked out. This method returns a marker to the particular commit that refers to the current database. Note that if the database is reverted to a position before this marker, the database can still be "for-verted" back to the marker position.

Returns A save state marker of arbitrary type

See also:

`revert_to_state()` Reverts to states saved by this method

transaction()

A context manager for transactions.

Sometimes it's more convenient to use with-blocks for transactions. This is a context manager to allow that. When entering the context, it calls `begin_transaction()`. When leaving the context due to normal execution, it will commit all changes. When leaving the context due to an error or exception being raised, it will revert all changes, and pass the error on up.

See also:

`begin_transaction()`, `commit()`, `rollback()` Methods for manually managing a transaction

transaction_open

Returns whether there is currently a transaction open.

Read-only

update(*d_id*, *document*)

Updates the document at *d_id* with a new document

This method replaces the document at d_id with a new document, completely deleting the old document to replace it with the new version. This is not very efficient.

See the documentation for `insert()` for a discussion on what actually counts as a document.

Parameters

- **d_id** (*int*) – A previously-saved document id
- **document** (*dict*) – The document to replace with

Returns *int* – Document ID

Raises `ValueError` – if the document id does not exist

4.4 API Docs: Search Functions

class `ogitm.gitdb.search_functions.`**`SearchFunction`**

A list of all search functions.

To add a function, use the `SearchFunction.add()` decorator. This passes the function itself through (so multiple invocations of the add decorator can be called on the same item), and appends it to the internal store of search functions.

To get a function, use the `SearchFunction.get()` classmethod. This returns the function described by the given name, or raises KeyError if no function is available.

This class should basically be used as a singleton instance - all methods are class methods, and operate on a shared store of data. This probably isn't best practice, but it works for now.

See `SearchFunction.add()` for details on what a search function should actually look like.

classmethod add(**funcnames*)

Add a function to the current list of functions.

The function should have the following signature:

Parameters

- **key** (*any*) – The key for which a value should be found. This is usually not important - the function is also passed the index that pertains to this particular key.
- **operator** (*str*) – The operator/name that this function has been called under. Sometimes it is simpler if different operators all map to the same function (for example, all string methods map to one function that dynamically calls the method on a string instance). This can be then used to work out the specific operation.
- **argument** (*any*) – The argument passed to this particular operator.
- **index** (*dict[any: list[id]]*) – The index related to the key being searched against. This is basically a dict mapping every value that has been assigned to this key to a list of the ids of the documents where this key-value mapping exists.
- **all** (*set[id]*) – The set of all ids that are currently stored. This is useful in the case where you want to search for, say, non-existance of a key, in which case the set of ids that should be returned is the set of all ids that aren't in the index that the function has been passed.

CHAPTER 5

Indices and tables

- *genindex*

Python Module Index