flywheel

Release 0.4.6

July 14, 2016

Contents

Flywheel is a library for mapping python objects to DynamoDB tables. It uses a SQLAlchemy-like syntax for queries.

Code lives here: https://github.com/mathcamp/flywheel

CHAPTER 1

User Guide

1.1 Getting Started

Flywheel can be installed with pip

```
pip install flywheel
```

Here are the steps to set up a simple example model with flywheel:

```
# Take care of some imports
from datetime import datetime
from flywheel import Model, Field, Engine

# Set up our data model
class Tweet(Model):
    userid = Field(hash_key=True)
    id = Field(range_key=True)
    ts = Field(data_type=datetime, index='ts-index')
    text = Field()

    def __init__(self, userid, id, ts, text):
        self.userid = userid
        self.id = id
        self.ts = ts
        self.text = text

# Create an engine and connect to an AWS region
engine = Engine()
engine.connect_to_region('us-east-1')

# Register our model with the engine so it can create the Dynamo table
engine.register(Tweet)

# Create the dynamo table for our registered model
engine.create_schema()
```

Now that you have your model, your engine, and the Dynamo table, you can begin adding tweets:

```
tweet = Tweet('myuser', '1234', datetime.utcnow(), text='@awscloud hey '
              'I found this cool new python library for AWS...')
engine.save(tweet)
```

To get data back out, query it using the engine:

```
# Get the 10 most recent tweets by 'myuser'
recent = engine.query(Tweet)\
        .filter(Tweet.ts <= datetime.utcnow(), userid='myuser')\
        .limit(10).all(desc=True)

# Get a specific tweet by a user
tweet = engine.query(Tweet).filter(userid='myuser', id='1234').first()
```

Since DynamoDB has no schema, you can set arbitrary fields on the tweets:

```
tweet = Tweet('myuser', '1234', datetime.utcnow(), text='super rad')
tweet.link = 'http://drmcninja.com'
tweet.retweets = 0
engine.save(tweet)
```

If you want to change a field, just make the change and sync it:

```
tweet.link = 'http://www.smbc-comics.com'
tweet.sync()
```

That's enough to give you a taste. The rest of the docs have more information on *creating models*, *writing queries*, or *how updates work*.

1.2 Models

1.2.1 Model Basics

This is what a model looks like:

```
class Tweet(Model):
    userid = Field(hash_key=True)
    id = Field(range_key=True)
    ts = Field(data_type=datetime, index='ts-index')
    text = Field()
```

The model declares the fields an object has, their *data types*, and the *schema* of the table.

Since DynamoDB is a NoSQL database, you can attach arbitrary additional fields (undeclared fields) to the model, and they will be stored appropriately. For example, this tweet doesn't declare a `retweets` field, but you could assign it anyway:

```
tweet.retweets = 7
tweet.sync()
```

Undeclared fields will **not** be saved if they begin or end with an underscore. This is intentional behavior so you can set local-only variables on your models.

```
tweet.retweets = 7  # this is saved to Dynamo
tweet._last_updated = datetime.utcnow()  # this is NOT saved to Dynamo
```

Since models define the schema of a table, you can use them to create or delete tables. Every model has a `meta_` field attached to it which contains metadata about the model. This metadata object has the `create` and `delete` methods.

```
from dynamo3 import DynamoDBConnection

connection = DynamoDBConnection.connect_to_region('us-east-1')
```

```
Tweet.meta_.create_dynamo_schema(connection)
Tweet.meta_.delete_dynamo_schema(connection)
```

You can also register your models with the engine and create all the tables at once:

```
engine.register(User, Tweet, Message)
engine.create_schema()
```

1.2.2 Data Types

DynamoDB supports three different data types: STRING, NUMBER, and BINARY. It also supports sets of these types: STRING_SET, NUMBER_SET, BINARY_SET.

You can use these values directly for the model declarations, though they require an import:

```
from flywheel import Model, Field, STRING, NUMBER

class Tweet(Model):
    userid = Field(data_type=STRING, hash_key=True)
    id = Field(data_type=STRING, range_key=True)
    ts = Field(data_type=NUMBER, index='ts-index')
    text = Field(data_type=STRING)
```

There are other settings for data_type that are represented by python primitives. Some of them (like `unicode`) are functionally equivalent to the DynamoDB option (`STRING`). Others, like `int`, enforce an additional application-level constraint on the data. Each option works transparently, so a `datetime` field would be set with `datetime` objects and you could query against it using other `datetime`'s.

Below is a table of python types, how they are stored in DynamoDB, and any special notes. For more information, the code for data types is located in *types*.

PY2 Type	PY3 Type	Dynamo Type	Notes
unicode	str	STRING	Basic STRING type. This is the default for fields
str	bytes	BINARY	Binary data, (serialized objects, compressed data, etc)
int/long	int	NUMBER	Enforces integer constraint on data
float		NUMBER	
Decimal		NUMBER	
set		*_SET	This will use the appropriate type of DynamoDB set
bool		BOOL	
datetime		NUMBER	Stored with UTC timezone. See *DateTimeType* for more.
date		NUMBER	
dict		MAP	
list		LIST	

If you attempt to set a field with a type that doesn't match, it will raise a `TypeError`. If a field was created with `coerce=True` it will first attempt to convert the value to the correct type. This means you could set an `int` field with the value `"123"` and it would perform the conversion for you.

Note: Certain fields will auto-coerce specific data types. For example, a `bytes` field will auto-encode a `unicode` to utf-8 even if `coerce=False`. Similarly, a `unicode` field will auto-decode a `bytes` value to a unicode string.

Warning: If an `int` field is set to coerce values, it will still refuse to drop floating point data. This has the following effect:

```
>>> class Game(Model):
...     title = Field(hash_key=True)
...     points = Field(data_type=int, coerce=True)

>>> mygame = Game()
>>> mygame.points = 1.8
ValueError: Field 'points' refusing to convert 1.8 to int! Results in data loss!
```

Set types

If you define a `set` field with no additional parameters `Field(data_type=set)`, flywheel will ensure that the field is a set, but will perform no type checking on the items within the set. This should work fine for basic uses when you are storing a number or string, but sets are able to contain any data type listed in the table above (and any *custom type* you declare). All you have to do is specify it in the `data_type` like so:

```
from flywheel import Model, Field, set_
from datetime import date

class Location(Model):
    name = Field(hash_key=True)
    events = Field(data_type=set_(date))
```

If you don't want to import `set_`, you can use an equivalent expression with the python `frozenset` builtin:

```
events = Field(data_type=frozenset([date]))
```

Field Validation

You can apply one or more validators to a field. These are functions that enforce some constraint on the field value beyond the type. Unlike the type checking done above, the validation checks are only run when saving to the database. An example:

```
class Widget(Model):
    id = Field(data_type=int, check=lambda x: x > 0)
```

To apply multiple validation checks, pass them in as a list or tuple:

```
def is_odd(x):
    return x % 2 == 1

def is_natural(x):
    return x >= 0

class Widget(Model):
    odd_natural_num = Field(data_type=int, check=(is_odd, is_natural))
```

There is a special case for enforcing that a field is non-null, since it is a common case:

```
username = Field(nullable=False)
```

The `nullable=False` will generate an additional check to make sure the value is non-null.

Custom Types

You can define your own custom data types and make them available across all of your models. All you need to do is create a subclass of *`TypeDefinition`*. Let's make a type that will store any python object in pickled format.

```
from flywheel.fields.types import TypeDefinition, BINARY, Binary
import cPickle as pickle

class PickleType(TypeDefinition):
    data_type = pickle #  name you use to reference this type
    aliases = ['pickle'] # alternate names that reference this type
    ddb_data_type = BINARY # data type of the field in dynamo

    def coerce(self, value, force):
        # Perform no type checking because we can pickle ANYTHING
        return value

    def ddb_dump(self, value):
        # Pickle and convert to a Binary object
        return Binary(pickle.dumps(value))

    def ddb_load(self, value):
        # Convert from a Binary object and unpickle
        return pickle.loads(value.value)
```

Now that you have your type definition, you can either use it directly in your code:

```
class MyModel(Model):
    myobj = Field(data_type=PickleType())
```

Or you can register it globally and reference it by its `data_type` or any `aliases` that were defined.

```
from flywheel.fields.types import register_type

register_type(PickleType)

class MyModel(Model):
    myobj = Field(data_type='pickle')
```

1.2.3 Schema

There are four main key concepts to understanding a DynamoDB table.

Hash key: This field will be sharded. Pick something with relatively random access (e.g. userid is good, timestamp is bad)

Range key: Optional. This field will be indexed, so you can query against it (within a specific hash key).

The hash key and range key together make the **Primary key**, which is the unique identifier for each object.

Local Secondary Indexes: Optional, up to 5. You may only use these if your table has a range key. These fields are indexed in a similar fashion as the range key. You may also query against them within a specific hash key. You can think of these as range keys with no uniqueness requirements.

Global Secondary Indexes: Optional, up to 5. These indexes have a hash key and optional range key, and can be put on any declared field. This allows you to shard your tables by more than one value.

For additional information on table design, read the AWS docs on best practices

Example declaration of hash and range key:

```
class Tweet(Model):
    userid = Field(hash_key=True)
    ts = Field(data_type=datetime, range_key=True)
```

For this version of a Tweet, each `(userid, ts)` pair is a unique value. The Dynamo table will be sharded across userids.

Local Secondary Indexes

Indexes also have a Projection Type. Creating an index requires duplicating some amount of data in the storage, and the projection type allows you to optimize how much additional storage is used. The projection types are:

All: All fields are projected into the index

Keys only: Only the primary key and indexed keys are projected into the index

Include: Like the "keys only" projection, but allows you to specify additional fields to project into the index

This is how they it looks in the model declaration:

```
class Tweet(Model):
    userid = Field(hash_key=True)
    id = Field(range_key=True)
    ts = Field(data_type=datetime).all_index('ts-index')
    retweets = Field(data_type=int).keys_index('rt-index')
    likes = Field(data_type=int).include_index('like-index', ['text'])
    text = Field()
```

The default index projection is "All", so you could replace the `ts` field above with:

```
ts = Field(data_type=datetime, index='ts-index')
```

Global Secondary Indexes

Like their Local counterparts, Global Secondary Indexes can specify a projection type. Unlike their Local counterparts, Global Secondary Indexes are provisioned with a *separate* read/write throughput from the base table. This can be specified in the model declaration. Here are some examples below:

```
class Tweet(Model):
    __metadata__ = {
        'global_indexes': [
            GlobalIndex.all('ts-index', 'city', 'ts').throughput(read=10, write=2),
            GlobalIndex.keys('rt-index', 'city', 'retweets')\
                    .throughput(read=10, write=2),
            GlobalIndex.include('like-index', 'city', 'likes',
                                includes=['text']).throughput(read=10, write=2),
        ],
    }
    userid = Field(hash_key=True)
    city = Field()
    id = Field(range_key=True)
    ts = Field(data_type=datetime)
    retweets = Field(data_type=int)
    likes = Field(data_type=int)
    text = Field()
```

If you want more on indexes, check out the AWS docs on indexes.

1.2.4 Composite Fields

Composite fields allow you to create fields that are combinations of multiple other fields. Suppose you're creating a table where you plan to store a collection of social media items (tweets, facebook posts, instagram pics, etc). If you make the hash key the id of the item, there is the remote possiblity that a tweet id will collide with a facebook id. Here is the solution:

```
class SocialMediaItem(Model):
    userid = Field(hash_key=True)
    type = Field()
    id = Field()
    uid = Composite('type', 'id', range_key=True)
```

This will automatically generate a `uid` field from the values of `type` and `id`. For example:

```
>>> item = SocialMediaItem(type='facebook', id='12345')
>>> print item.uid
facebook:12345
```

Note that setting a Composite field just doesn't work:

```
>>> item.uid = 'ILikeThisIDBetter'
>>> print item.uid
facebook:12345
```

By default, a Composite field simply joins its subfields with a `':'`. You can change that behavior for fancier applications:

```
def score_merge(likes, replies, deleted):
    if deleted:
        return None
    return likes + 5 * replies

class Post(Model):
    userid = Field(hash_key=True)
    id = Field(range_key=True)
    likes = Field(data_type=int)
    replies = Field(data_type=int)
    deleted = Field(data_type=bool)
    score = Composite('likes', 'replies', 'deleted', data_type=int,
                      merge=score_merge, index='score-index')
```

So now you can update the `likes` or `replies` count, and the score will automatically change. Which will re-arrange it in the index that you created. Then, if you mark the post as "deleted", it will remove the score field which removes it from the index.

Whoooooaaahh...

The last neat little thing about Composite fields is how you can query them. For numeric Composite fields you probably want to query directly on the score like any other field. But if you're merging strings like with SocialMediaItem, it can be cleaner to refer to the component fields themselves:

```
>>> fb_post = engine.query(SocialMediaItem).filter(userid='abc123',
...     type='facebook', id='12345').first()
```

The engine will automatically detect that you're trying to query on the range key, and construct the `uid` from the pieces you provided.

1.2.5 Metadata

Part of the model declaration is the `__metadata__` attribute, which is a dict that configures the `Model.meta_` object. Models will inherit and merge the __metadata__ fields from their ancestors. Keys that begin with an underscore will not be merged. For example:

```
class Vehicle(Model):
    __metadata__ = {
        '_name': 'all-vehicles',
        'throughput': {
            'read': 10,
            'write': 2,
        }
    }

class Car(Vehicle):
    pass
```

```
>>> print Car.__metadata__
{'throughput': {'read': 10, 'write': 2}}
```

Below is a list of all the values that may be set in the `__metadata__` attribute of a model.

Key	Type	Description
_name	str	The name of the DynamoDB table (defaults to class name)
_abstract	bool	If True, no DynamoDB table will be created for this model (useful if you just want a class to inherit from)
throughput	dict	The table read/write throughput (defaults to {'read': 5, 'write': 5})
global_indexes	list	A list of GlobalIndex objects

1.3 Table Queries

The query syntax is heavily inspired by SQLAlchemy. In DynamoDB, queries must use one of the table's indexes. Queries are constrained to a single hash key value. This means that for a query there will always be at least one call to `filter` which will, at a minimum, set the hash key to search on.

```
# Fetch all tweets made by a user
engine.query(Tweet).filter(Tweet.userid == 'abc123').all()
```

You may also use inequality filters on range keys and secondary indexes

```
# Fetch all tweets made by a user in the past day
earlyts = datetime.utcnow() - timedelta(days=1)
engine.query(Tweet).filter(Tweet.userid == 'abc123',
                           Tweet.ts >= earlyts).all()
```

There are two finalizing statements that will return all results: `all()` and `gen()`. Calling `all()` will return a list of results. Calling `gen()` will return a generator. If your query will return a large number of results, using `gen()` can help you avoid storing them all in memory at the same time.

```
# Count how many retweets a user has in total
retweets = 0
all_tweets = engine.query(Tweet).filter(Tweet.userid == 'abc123').gen()
for tweet in all_tweets:
    retweets += tweet.retweets
```

There are two finalizing statements that retrieve a single item: `first()` and `one()`. Calling `first()` will return the first element of the results, or None if there are no results. Calling `one()` will return the first element of the results *only if* there is *exactly* one result. If there are no results or more results it will raise a `ValueError`.

```
# Get a single tweet by a user
tweet = engine.query(Tweet).filter(Tweet.userid == 'abc123').first()

# Get a specific tweet and fail if missing
tweet = engine.query(Tweet).filter(Tweet.userid == 'abc123',
                                   Tweet.id == '1234').one()
```

There is one more finalizing statement: `count()`. This will return the number of results that matched the query, instead of returning the results themselves.

```
# Get the number of tweets made by user abc123
num = engine.query(Tweet).filter(Tweet.userid == 'abc123').count()
```

You can set a `limit()` on a query to limit the number of results it returns:

```
# Get the first 10 tweets by a user after a timestamp
afterts = datetime.utcnow() - timedelta(hours=1)
tweets = engine.query(Tweet).filter(Tweet.userid == 'abc123',
                                    Tweet.ts >= afterts).limit(10).all()
```

One way to delete items from a table is with a query. Calling `delete()` will delete all items that match a query:

```
# Delete all of a user's tweets older than 1 year
oldts = datetime.utcnow() - timedelta(days=365)
engine.query(Tweet).filter(Tweet.userid == 'abc123',
                           Tweet.ts < oldts).delete()
```

Most of the time the query engine will be able to automatically detect which local or global secondary index you intend to use. If the index is ambiguous, you can manually specify the index. This can also be useful if you want the results to be sorted by a particular index when only querying the hash key.

```
# This is the schema for the following example
class Tweet(Model):
    userid = Field(hash_key=True)
    id = Field(range_key=True)
    ts = Field(data_type=datetime, index='ts-index')
    retweets = Field(data_type=int, index='rt-index')

# This returns 10 tweets in id order (more-or-less random)
ten_tweets = engine.query(Tweet).filter(userid='abc123').limit(10).all()

# Get the 10 most retweeted tweets for a user
top_ten = engine.query(Tweet).filter(userid='abc123').index('rt-index')\
        .limit(10).all(desc=True)

# Get The 10 most recent tweets for a user
ten_recent = engine.query(Tweet).filter(userid='abc123').index('ts-index')\
        .limit(10).all(desc=True)
```

New in 0.2.1

Queries can filter on fields that are not the hash or range key. Filtering this way will strip out the results server-side, but it will not use an index. When filtering on these extra fields, you may use the additional filter operations that are listed under *Table Scans*.

1.3.1 Shorthand

If you want to avoid typing 'query' everywhere, you can simply call the engine:

```
# Long form query
engine.query(Tweet).filter(Tweet.userid == 'abc123').all()

# Abbreviated query
engine(Tweet).filter(Tweet.userid == 'abc123').all()
```

Filter constraints with == can be instead passed in as keyword arguments:

```
# Abbreviated filter
engine(Tweet).filter(userid='abc123').all()

engine(Tweet).filter(userid='abc123', id='1234').first()
```

You can still pass in other constraints as positional arguments to the same filter:

```
# Multiple filters in same statement
engine(Tweet).filter(Tweet.ts <= earlyts, userid='abc123').all()
```

1.3.2 Table Scans

Table scans are similar to table queries, but they do not use an index. This means they have to read every item in the table. This is EXTREMELY SLOW. The benefit is that they do not have to filter based on the hash key, and they have a few additional filter arguments that may be used.

```
# Fetch all tweets ever
alltweets = engine.scan(Tweet).gen()

# Fetch all tweets that tag awscloud
tagged = engine.scan(Tweet).filter(Tweet.tags.contains_('awscloud')).all()

# Fetch all tweets with annoying, predictable text
annoying = set(['first post', 'hey guys', 'LOOK AT MY CAT'])
first = engine.scan(Tweets).filter(Tweet.text.in_(annoying)).all()

# Fetch all tweets with a link
linked = engine.scan(Tweet).filter(Tweet.link != None).all()
```

Since table scans don't use indexes, you can filter on fields that are not declared in the model. Here are some examples:

```
# Fetch all tweets that link to wikipedia
educational = engine.scan(Tweet)\
        .filter(Tweet.field_('link').beginswith_('http://wikipedia')).all()

# You can also use the keyword arguments to filter
best_tweets = engine.scan(Tweet)\
        .filter(link='http://en.wikipedia.org/wiki/Morgan_freeman').all()
```

1.4 CRUD

This section covers the operations you can do to save, read, update, and delete items from the database. All of these methods exist on the *Engine* object and can be called on one or many items. After being saved-to or loaded-from Dynamo, the items themselves will have these methods attached to them as well. For example, these are both valid:

```
>>> engine.sync(tweet)
>>> tweet.sync()
```

1.4.1 Save

Save the item to Dynamo. This is intended for new items that were just created and need to be added to the database. If you `save()` an item that already exists in Dynamo, it will raise an exception. You may optionally use `save(overwrite=True)` to instead clobber existing data and write your version of the item to Dynamo.

```
>>> tweet = Tweet()
>>> engine.save(tweet)
>>> tweet.text = "Let's replace the whole item"
>>> tweet.save(overwrite=True)
```

1.4.2 Refresh

Query dynamo to get the most up-to-date version of a model. Clobbers any existing data on the item. To force a consistent read use `refresh(consistent=True)`.

This call is very useful if you query indexes that use an incomplete projection type. The results won't have all of the item's fields, so you can call `refresh()` to get any attributes that weren't projected onto the index.

```
>>> tweet = engine.query(Tweet).filter(userid='abc123')\
...          .index('ts-index').first(desc=True)
>>> tweet.refresh()
```

1.4.3 Get

Fetch an item from its primary key fields. This will be faster than a query, but requires you to know the primary keys of all items you want fetched.

```
>>> my_tweet = engine.get(Tweet, userid='abc123', id='1')
```

You can also fetch many at a time:

```
>>> key1 = {'userid': 'abc123', 'id': '1'}
>>> key2 = {'userid': 'abc123', 'id': '2'}
>>> key3 = {'userid': 'abc123', 'id': '3'}
>>> some_tweets = engine.get(Tweet, [key1, key2, key3])
```

1.4.4 Delete

Deletes an item. You may pass in `delete(raise_on_conflict=True)`, which will only delete the item if none of the values have changed since it was read.

```
>>> tweet = engine.query(Tweet).filter(userid='abc123', id='123').first()
>>> tweet.delete()
```

You may also delete an item from a primary key specification:

```
>>> engine.delete_key(Tweet, userid='abc123', id='1')
```

And you may delete many at once:

```
>>> key1 = {'userid': 'abc123', 'id': '1'}
>>> key2 = {'userid': 'abc123', 'id': '2'}
>>> key3 = {'userid': 'abc123', 'id': '3'}
>>> engine.delete_key(Tweet, [key1, key2, key3])
```

1.4.5 Sync

Save any fields that have been changed on an item. This will update changed fields in Dynamo and ensure that all fields exactly reflect the item in the database. This is usually used for updates, but it can be used to create new items as well.

```
>>> tweet = Tweet()
>>> engine.sync(tweet)
>>> tweet.text = "Update just this field"
>>> tweet.sync()
```

Models will automatically detect changes to mutable fields, such as `dict`, `list`, and `set`.

```
>>> tweet.tags.add('awscloud')
>>> tweet.sync()
```

Since sync does a partial update, it can tolerate concurrent writes of different fields.

```
>>> tweet = engine.query(Tweet).filter(userid='abc123', id='1234').first()
>>> tweet2 = engine.query(Tweet).filter(userid='abc123', id='1234').first()
>>> tweet.author = "The Pope"
>>> tweet.sync()
>>> tweet2.text = "Mo' money mo' problems"
>>> tweet2.sync() #  it works!
>>> print tweet2.author
The Pope
```

This "merge" behavior is also what happens when you `sync()` items to create them. If the item to create already exists in Dynamo, that's fine as long as there are no conflicting fields. Note that this behavior is distinctly different from `save()`, so make sure you pick the right call for your use case.

If you call `sync()` on an object that has not been changed, it is equivalent to calling `refresh()`.

Safe Sync

If you use `sync(raise_on_conflict=True)`, the sync operation will check that the fields that you're updating have not been changed since you last read them. This is very useful for preventing concurrent writes.

Note: If you change a key that is part of a *composite field*, flywheel will **force** the sync to raise on conflict. This avoids the risk of corrupting the value of the composite field.

Atomic Increment

DynamoDB supports truly atomic increment/decrement of NUMBER fields. To use this functionality, there is a special call you need to make:

```
>>> # Increment the number of retweets by 1
>>> tweet.incr_(retweets=1)
>>> tweet.sync()
```

BOOM.

Note: Incrementing a field that is part of a composite field will also force the sync to raise on conflict.

Atomic Add/Remove

DynamoDB also supports truly atomic add/remove to SET fields. To use this functionality, there is another special call:

```
>>> # Add two users to the set of tagged users
>>> tweet.add_(tags=set(['stevearc', 'dsa']))
>>> tweet.sync()
```

And to delete:

```
>>> tweet.remove_(tags='stevearc')
>>> tweet.sync()
```

Note than you can pass in a single value or a set of values to both `add_` and `remove_`.

Sync-if-Constraints

New in 0.2.1

You may pass in a list of constraints to check upon sync. If any of the constraints fail, then the sync will not complete. This should be used with `raise_on_conflict=True`. For example:

```
>>> account = engine.get(Account, username='dsa')
>>> account.incr_(moneys=-200)
>>> # atomically remove $200 from DSA's account, iff there is at least $200 to remove.
>>> account.sync(constraints=[Account.moneys >= 200])
```

1.4.6 Default Conflict Behavior

You can configure the default behavior for each of these endpoints using *`default_conflict`*. The default setting will cause `sync()` to check for conflicts, `delete()` not to check for conflicts, and `save()` to overwrite existing values. Check the attribute docs for more options. You can, of course, pass in the argument to the calls directly to override this behavior on a case-by-case basis.

1.5 Developing

To get started developing flywheel, run the following command:

```
wget https://raw.github.com/mathcamp/devbox/0.1.0/devbox/unbox.py && \
python unbox.py git@github.com:mathcamp/flywheel
```

This will clone the repository and install the package into a virtualenv

1.5.1 Running Tests

The command to run tests is `python setup.py nosetests`, or `tox`. Most of these tests require DynamoDB Local. There is a nose plugin that will download and run the DynamoDB Local service during the tests. It requires the java 6/7 runtime, so make sure you have that installed.

1.6 Changelog

1.6.1 0.4.6

- Pass exclusive_start_key through to dynamo3

1.6.2 0.4.5

- Bug fix: Calling refresh() could sometimes crash from unordered results.

1.6.3 0.4.4

- Bug fix: Mutable field defaults are no longer shared among model instances

1.6.4 0.4.3

- Bug fix: Incorrect `ConditionalCheckFailedException` when syncing changes to a Composite field.
- Allow `DateTimeType` to be stored as a naive datetime.

1.6.5 0.4.2

- Make the `dict`, `list`, and `bool` types backwards-compatible with the old json-serialized format (**:pr:'24'**)
- Allow queries to use `in`, `not null`, and a few other constraints that were missing (commit 8b8854d)
- Models are smarter about marking fields as dirty for sync (issue 26)
- Stopped using deprecated `expected` syntax for dynamo3

1.6.6 0.4.1

- **Warning**: Stored datetime objects will now be timezone-aware (commit a7c253d)
- **Warning**: Stored datetime objects will now keep their microseconds (commit fffe92c)

1.6.7 0.4.0

- **Breakage**: Dropping support for python 3.2 due to lack of botocore support
- **Breakage**: Changing the `list`, `dict`, and `bool` data types to use native DynamoDB types instead of JSON serializing

- **Breakage** and bug fix: Fixing serialization of `datetime` and `date` objects (for more info see the commit) (commit df049af)
- Feature: Can now do 'contains' filters on lists
- Feature: Fields support multiple validation checks
- Feature: Fields have an easy way to enforce non-null values (`nullable=False`)

Data type changes are due to an update in the DynamoDB API

1.6.8 0.3.0

- **Breakage**: Engine namespace is slightly different. If you pass in a string it will be used as the table name prefix with no additional '-' added.

1.6.9 0.2.1

- **Breakage**: Certain queries may now require you to specify an index where it was auto-detected before
- Feature: Queries can now filter on non-indexed fields
- Feature: More powerful "sync-if" constraints
- Feature: Can OR together filter constraints in queries

All changes are due to an update in the DynamoDB API

1.6.10 0.2.0

- **Breakage**: Engine no longer accepts boto connections (using dynamo3 instead)
- **Breakage**: Removing S3Type (no longer have boto as dependency)
- Feature: Support Python 3.2 and 3.3
- Feature: `.count()` terminator for queries (commit bf3261c)
- Feature: Can override throughputs in `Engine.create_schema()` (commit 4d1abe0)
- Bug fix: Engine `namespace` is truly isolated (commit 3b4fad7)

1.6.11 0.1.3

- Bug fix: Some queries fail when global index has no range key (issue 9, commit edce6e2)

1.6.12 0.1.2

- Bug fix: Field names can begin with an underscore (commit 637f1ee, issue 7)
- Feature: Models have a nice default __init__ method (commit 40068c2)

1.6.13 0.1.1

- Bug fix: Can call `incr_()` on models that have not been saved yet (commit 0a1990f)
- Bug fix: Model comparison with `None` (commit 374dda1)

1.6.14 0.1.0

- First public release

CHAPTER 2

API Reference

2.1 flywheel package

2.1.1 Subpackages

flywheel.fields package

Submodules

flywheel.fields.conditions module Query constraints

class `flywheel.fields.conditions.`**`Condition`**
Bases: `object`

A constraint that will be applied to a query or scan

Attributes

eq_fields	(dict) Mapping of field name to field value
fields	(dict) Mapping of field name to (operator, value) tuples
limit	(int) Maximum number of results
index_name	(str) Name of index to use for a query

classmethod `construct` (*field*, *op*, *other*)
Create a Condition on a field

Parameters **field** : str

Name of the field to constrain

op : str

Operator, such as 'eq', 'lt', or 'contains'

other : object

The value to constrain the field with

Returns **condition** : *`Condition`*

classmethod `construct_index` (*name*)
Force the query to use a certain index

Parameters **name** : str

Returns **condition** : *Condition*

classmethod **construct_limit**(*count*)
Create a condition that will limit the results to a count

Parameters **count** : int

Returns **condition** : *Condition*

query_kwargs(*model*)
Get the kwargs for doing a table query

scan_kwargs()
Get the kwargs for doing a table scan

flywheel.fields.indexes module Index definitions

class flywheel.fields.indexes.**GlobalIndex**(*name*, *hash_key*, *range_key=None*)
Bases: object

A global index for DynamoDB

Parameters **name** : str

The name of the index

hash_key : str

The name of the field that is the hash key for the index

range_key : str, optional

The name of the field that is the range key for the index

throughput : dict, optional

The read/write throughput of this global index. Used when creating a table. Dict has a 'read' and a 'write' key. (Default 5, 5)

classmethod **all**(*name*, *hash_key*, *range_key=None*)
Project all attributes into the index

get_ddb_index(*fields*)
Get the dynamo index class for this GlobalIndex

classmethod **include**(*name*, *hash_key*, *range_key=None*, *includes=None*)
Select which attributes to project into the index

classmethod **keys**(*name*, *hash_key*, *range_key=None*)
Project key attributes into the index

throughput(*read=5*, *write=5*)
Set the index throughput

Parameters **read** : int, optional

Amount of read throughput (default 5)

write : int, optional

Amount of write throughput (default 5)

Notes

This is meant to be used as a chain:

```
class MyModel(Model):
    __metadata__ = {
        'global_indexes': [
            GlobalIndex('myindex', 'hkey', 'rkey').throughput(5, 2)
        ]
    }
```

flywheel.fields.types module Field type definitions

class flywheel.fields.types.**BinaryType**
Bases: *flywheel.fields.types.TypeDefinition*

Binary strings, stored as a str/bytes

aliases = ['B', <class 'dynamo3.types.Binary'>]

coerce(*value*, *force*)

data_type
alias of str

ddb_data_type = 'B'

ddb_dump(*value*)

ddb_load(*value*)

class flywheel.fields.types.**BoolType**
Bases: *flywheel.fields.types.TypeDefinition*

Boolean type

coerce(*value*, *force*)

data_type
alias of bool

ddb_data_type = 'BOOL'

class flywheel.fields.types.**DateTimeType**(*naive=False*)
Bases: *flywheel.fields.types.TypeDefinition*

Datetimes, stored as a unix timestamp

Parameters **naive** : bool, optional

If True, will load values from Dynamo with no timezone. If False, will add a UTC timezone. (Default False).

Notes

If you want to use naive datetimes, you will need to reference the type class directly instead of going through an alias. For example:

```
from flywheel.fields.types import DateTimeType

field = Field(data_type=DateTimeType(naive=True))
```

data_type
alias of datetime

ddb_data_type = 'N'

ddb_dump(*value*)

ddb_load(*value*)

class flywheel.fields.types.**DateType**
Bases: *flywheel.fields.types.TypeDefinition*

Dates, stored as timestamps

data_type
alias of date

ddb_data_type = 'N'

ddb_dump(*value*)

ddb_load(*value*)

class flywheel.fields.types.**DecimalType**
Bases: *flywheel.fields.types.TypeDefinition*

Numerical values that use Decimal in the application layer.

This should be used if you want to work with floats but need the additional precision of the Decimal type.

coerce(*value*, *force*)

data_type
alias of Decimal

ddb_data_type = 'N'

class flywheel.fields.types.**DictType**
Bases: *flywheel.fields.types.TypeDefinition*

Dict type, stored as a map

coerce(*value*, *force*)

data_type
alias of dict

ddb_data_type = 'M'

mutable = True

class flywheel.fields.types.**FloatType**
Bases: *flywheel.fields.types.TypeDefinition*

Float values

coerce(*value*, *force*)

data_type
alias of float

ddb_data_type = 'N'

ddb_load(*value*)

class flywheel.fields.types.**IntType**
Bases: *flywheel.fields.types.TypeDefinition*

Integer values (includes longs)

aliases = [<type 'int'>, <type 'long'>]

coerce(*value*, *force*)

data_type
alias of `int`

ddb_data_type = 'N'

ddb_load(*value*)

class `flywheel.fields.types.`**ListType**
Bases: *flywheel.fields.types.TypeDefinition*

List type

coerce(*value*, *force*)

data_type
alias of `list`

ddb_data_type = 'L'

mutable = True

class `flywheel.fields.types.`**NumberType**
Bases: *flywheel.fields.types.TypeDefinition*

Any kind of numerical value

coerce(*value*, *force*)

data_type = 'N'

ddb_data_type = 'N'

ddb_load(*value*)

class `flywheel.fields.types.`**SetType**(*item_type=None*, *type_class=None*)
Bases: *flywheel.fields.types.TypeDefinition*

Set types

classmethod bind(*item_type*)
Create a set factory that will contain a specific data type

coerce(*value*, *force*)

data_type
alias of `set`

ddb_dump(*value*)

ddb_dump_inner(*value*)
We need to expose this for 'contains' and 'ncontains'

ddb_load(*value*)

mutable = True

class `flywheel.fields.types.`**StringType**
Bases: *flywheel.fields.types.TypeDefinition*

String values, stored as unicode

aliases = ['S']

coerce(*value*, *force*)

data_type
alias of `unicode`

ddb_data_type = 'S'

class `flywheel.fields.types.`**TypeDefinition**
Bases: *flywheel.compat.UnicodeMixin*

Base class for all Field types

Attributes

data_type	(object) The value you wish to pass in to Field as the data_type.
aliases	(list) Other values that will reference this type if passed to Field
ddb_data_type	(STRING, BINARY, NUMBER, STRING_SET, BINARY_SET, NUMBER_SET, BOOL, LIST, MAP}) The DynamoDB data type that backs this type
mutable	(bool) If True, flywheel will track updates to this field automatically when making calls to sync()
allowed_filters	(set) The set of filters that can be used on this field type

aliases = []

coerce(*value*, *force*)
Check the type of a value and possible convert it

Parameters **value** : object

The value to check

force : bool

If True, always attempt to convert a bad type to the correct type

Returns **value** : object

A variable of the correct type

Raises **exc** : TypeError or ValueError

If the value is the incorrect type and could not be converted

data_type = None

ddb_data_type = None

ddb_dump(*value*)
Dump a value to a form that can be stored in DynamoDB

ddb_dump_inner(*value*)
If this is a set type, dump a value to the type contained in the set

ddb_load(*value*)
Turn a value into this type from a DynamoDB value

mutable = False

class `flywheel.fields.types.`**UTCTimezone**
Bases: `datetime.tzinfo`

UTC

dst(*dt*)

tzname(*dt*)

utcoffset(*dt*)

flywheel.fields.types.**register_type**(*type_class*, *allow_in_set=True*)
Register a type class for use with Fields

flywheel.fields.types.**set_**(*data_type*)
Create an alias for a SetType that contains this data type

Module contents

Field declarations for models

class flywheel.fields.**Composite**(**args*, ***kwargs*)
Bases: *flywheel.fields.Field*

A field that is composed of multiple other fields

Parameters ***fields** : list

List of names of fields that compose this composite field

hash_key : bool, optional

This key is a DynamoDB hash key (default False)

range_key : bool, optional

This key is a DynamoDB range key (default False)

index : str, optional

If present, create a local secondary index on this field with this as the name.

data_type : str, optional

The dynamo data type. Valid values are (NUMBER, STRING, BINARY, NUMBER_SET, STRING_SET, BINARY_SET, dict, list, bool, str, unicode, int, float, set, datetime, date, Decimal) (default unicode)

coerce : bool, optional

Attempt to coerce the value if it's the incorrect type (default False)

check : callable, optional

A function that takes the value and returns True if the value is valid (default None)

merge : callable, optional

The function that merges the subfields together. By default it simply joins them with a ':'.

get_cached_value(*obj*)

resolve(*obj=None*, *scope=None*)
Resolve a field value from an object or scope dict

class flywheel.fields.**Field**(*hash_key=False*, *range_key=False*, *index=None*, *data_type=<type 'unicode'>*, *coerce=False*, *check=None*, *nullable=True*, *default=<object object>*)
Bases: object

Declarative way to specify model fields

Parameters **hash_key** : bool, optional

This key is a DynamoDB hash key (default False)

range_key : bool, optional

This key is a DynamoDB range key (default False)

index : str, optional

If present, create a local secondary index on this field with this as the name.

data_type : object, optional

The field data type. You may use int, unicode, set, etc. or you may pass in an instance of *TypeDefinition* (default unicode)

coerce : bool, optional

Attempt to coerce the value if it's the incorrect type (default False)

check : callable or list, optional

A function that takes the value and returns True if the value is valid. May also be a list of such functions. (default None)

nullable : bool, optional

If false, will add a check (above) to ensure the value is not null (default True).

default : object, optional

The default value for this field that will be set when creating a model (default None, except for `set` data types which default to set())

Notes

```
Field(index='my-index')
```

Is shorthand for:

```
Field().all_index('my-index')
```

Attributes

name	(str) The name of the attribute on the model
model	(class) The *Model* this field is attached to
composite	(bool) True if this is a composite field

all_index(*name*)
Index this field and project all attributes

Parameters **name** : str

The name of the index

beginswith_(*other*)
Create a query condition that this field must begin with a string

between_(*low*, *high*)
Create a query condition that this field must be between two values (inclusive)

betwixt_(*low*, *high*)
Poetic version of *between_()*

can_resolve(*fields*)
Check if the provided fields are enough to fully resolve this field

Parameters **fields** : list or set

Returns **needed** : set

Set of the subfields needed to resolve this field. If empty, then it cannot be resolved.

coerce(*value*, *force_coerce=None*)
Coerce the value to the field's data type

contains_(*other*)
Create a query condition that this field must contain a value

ddb_data_type
Get the native DynamoDB data type

ddb_dump(*value*)
Dump a value to its Dynamo format

ddb_dump_for_query(*value*)
Dump a value to format for use in a Dynamo query

classmethod ddb_dump_overflow(*val*)
Dump an overflow value to its Dynamo format

ddb_load(*val*)
Decode a value retrieved from Dynamo

classmethod ddb_load_overflow(*val*)
Decode a value of an overflow field

default
Get a shallow copy of the default value

get_cached_value(*obj*)
Get the cached value of a field before any local modifications

get_ddb_index()
Construct a dynamo local index object

in_(*other*)
Create a query condition that this field must be within a set of values

include_index(*name*, *includes=None*)
Index this field and project selected attributes

Parameters **name** : str

The name of the index

includes : list, optional

List of non-key attributes to project into this index

is_mutable
Return True if the data type is mutable

classmethod is_overflow_mutable(*val*)
Check if an overflow field is mutable

is_set
Return True if data type is a set

keys_index(*name*)
Index this field and project all key attributes

Parameters **name** : str

The name of the index

ncontains_(*other*)
Create a query condition that this field cannot contain a value

resolve(*obj=None*, *scope=None*)
Resolve a field value from an object or scope dict

validate(*obj*)
Run the validation checks for this field on a model object.

Parameters **obj** : *Model*

Raises **err** : ValueError

Raised if any of the checks fail.

2.1.2 Submodules

flywheel.compat module

Utilities for Python 2/3 compatibility

class flywheel.compat.**UnicodeMixin**
Bases: object

Mixin that handles __str__ and __bytes__. Just define __unicode__.

flywheel.engine module

Query engine

class flywheel.engine.**Engine**(*dynamo=None*, *namespace=()*, *default_conflict='update'*)
Bases: object

Query engine for models

Parameters **dynamo** : dynamodb3.DynamoDBConnection, optional

namespace : list or str, optional

String prefix or list of component parts of a prefix for models. All table names will be prefixed by this string or strings (joined by '-').

default_conflict : {'update', 'overwrite', 'raise'}, optional

Default setting for delete(), save(), and sync() (default 'update')

Notes

The engine is used to save, sync, delete, and query DynamoDB. Here is a basic example of saving items:

```
item1 = MyModel()
engine.save(item1)
item1.foobar = 'baz'
item2 = MyModel()
engine.save([item1, item2], overwrite=True)
```

You can also use the engine to query tables:

```
user = engine.query(User).filter(User.id == 'abcdef).first()

# calling engine() is a shortcut for engine.query()
user = engine(User).filter(User.id == 'abcdef).first()

d_users = engine(User).filter(User.school == 'MIT',
                              User.name.beginswith_('D')).all()

# You can pass in equality constraints as keyword args
user = engine(User).filter(id='abcdef').first()
```

Scans are like queries, except that they don't use an index. Scans iterate over the ENTIRE TABLE so they are REALLY SLOW. Scans have access to additional filter conditions such as "contains" and "in".

```
# This is suuuuuper slow!
user = engine.scan(User).filter(id='abcdef').first()

# If you're doing an extremely large scan, you should tell it to return
# a generator
all_users = engine.scan(User).gen()

# to filter a field not specified in the model declaration:
prince = engine.scan(User).filter(User.field_('bio').beginswith_(
           'Now this is a story all about how')).first()
```

connect(**args*, ***kwargs*)
Connect to a specific host

connect_to_host(***kwargs*)
Connect to a specific host

connect_to_region(*region*, ***kwargs*)
Connect to an AWS region

create_schema(*test=False*, *throughput=None*)
Create the DynamoDB tables required by the registered models

Parameters **test** : bool, optional

If True, perform a dry run (default False)

throughput : dict, optional

If provided, override the throughputs of the Models when creating them. Details below.

Returns **names** : list

List of table names that were created

Examples

The `throughput` argument is a mapping of table names to their throughputs. The throughput is a dict with a 'read' and 'write' value. It may also include the names of global indexes that map to their own dicts with a 'read' and 'write' value.

```
engine.create_schema(throughput={
    'table1': {
        'read': 4,
        'write': 10,
        'gindex-1': {
            'read': 6,
            'write': 3,
        }
    }
})
```

default_conflict
Get the default_conflict value

Notes

The `default_conflict` setting configures the default behavior of *save()*, *sync()*, and *delete()*. Below is an explanation of the different values of `default_conflict`.

default_conflict	method	default
'update'		
	save	overwrite=True
	sync	raise_on_conflict=True
	delete	raise_on_conflict=False
'overwrite'		
	save	overwrite=True
	sync	raise_on_conflict=False
	delete	raise_on_conflict=False
'raise'		
	save	overwrite=False
	sync	raise_on_conflict=True
	delete	raise_on_conflict=True

delete(*items*, *raise_on_conflict=None*)
Delete items from dynamo

Parameters **items** : list or `Model`

List of *Model* objects to delete

raise_on_conflict : bool, optional

If True, raise exception if the object was changed concurrently in the database (default set by *default_conflict*)

Raises **exc** : `dynamo3.ConditionalCheckFailedException`

If overwrite is False and an item already exists in the database

Notes

Due to the structure of the AWS API, deleting with raise_on_conflict=False is much faster because the requests can be batched.

delete_key(*model*, *pkeys=None*, ***kwargs*)

Delete one or more items from dynamo as specified by primary keys

Parameters **model** : *Model*

pkeys : list, optional

List of primary key dicts

****kwargs** : dict

If pkeys is None, delete only a single item and use kwargs as the primary key dict

Returns **count** : int

The number of deleted items

Notes

If the model being deleted has no range key, you may use strings instead of primary key dicts. ex:

```
>>> class Item(Model):
...     id = Field(hash_key=True)
...
>>> items = engine.delete_key(Item, ['abc', 'def', '123', '456'])
```

delete_keys(*model*, *pkeys=None*, ***kwargs*)

Delete one or more items from dynamo as specified by primary keys

Parameters **model** : *Model*

pkeys : list, optional

List of primary key dicts

****kwargs** : dict

If pkeys is None, delete only a single item and use kwargs as the primary key dict

Returns **count** : int

The number of deleted items

Notes

If the model being deleted has no range key, you may use strings instead of primary key dicts. ex:

```
>>> class Item(Model):
...     id = Field(hash_key=True)
...
>>> items = engine.delete_key(Item, ['abc', 'def', '123', '456'])
```

delete_schema(*test=False*)

Drop the DynamoDB tables for all registered models

Parameters **test** : bool, optional

If True, perform a dry run (default False)

Returns **names** : list

List of table names that were deleted

get(*model*, *pkeys=None*, *consistent=False*, ***kwargs*)
Fetch one or more items from dynamo from the primary keys

Parameters **model** : *Model*

pkeys : list, optional

List of primary key dicts

consistent : bool, optional

Perform a consistent read from dynamo (default False)

****kwargs** : dict

If pkeys is None, fetch only a single item and use kwargs as the primary key dict.

Returns **items** : list or object

If pkeys is a list of key dicts, this will be a list of items. If pkeys is None and **kwargs is used, this will be a single object.

Notes

If the model being fetched has no range key, you may use strings instead of primary key dicts. ex:

```
>>> class Item(Model):
...     id = Field(hash_key=True)
...
>>> items = engine.get(Item, ['abc', 'def', '123', '456'])
```

get_schema()
Get the schema for the registered models

query(*model*)
Create a table query for a specific model

Returns **query** : *Query*

refresh(*items*, *consistent=False*)
Overwrite model data with freshest from database

Parameters **items** : list or *Model*

Models to sync

consistent : bool, optional

If True, force a consistent read from the db. (default False)

register(**models*)
Register one or more models with the engine

Registering is required for schema creation or deletion

save(*items*, *overwrite=None*)
Save models to dynamo

Parameters **items** : list or *Model*

overwrite : bool, optional

If False, raise exception if item already exists (default set by *default_conflict*)

Raises **exc** : dynamo3.ConditionalCheckFailedException

If overwrite is False and an item already exists in the database

Notes

Overwrite will replace the *entire* item with the new one, not just different fields. After calling save(overwrite=True) you are guaranteed that the item in the database is exactly the item you saved.

Due to the structure of the AWS API, saving with overwrite=True is much faster because the requests can be batched.

scan(*model*)
Create a table scan for a specific model

Returns **scan** : *Scan*

sync(*items*, *raise_on_conflict=None*, *consistent=False*, *constraints=None*)
Sync model changes back to database

This will push any updates to the database, and ensure that all of the synced items have the most up-to-date data.

Parameters **items** : list or *Model*

Models to sync

raise_on_conflict : bool, optional

If True, raise exception if any of the fields that are being updated were concurrently changed in the database (default set by *default_conflict*)

consistent : bool, optional

If True, force a consistent read from the db. This will only take effect if the sync is only performing a read. (default False)

constraints : list, optional

List of more complex constraints that must pass for the update to complete. Must be used with raise_on_conflict=True. Format is the same as query filters (e.g. Model.fieldname > 5)

Raises **exc** : dynamo3.CheckFailed

If raise_on_conflict=True and the model changed underneath us

flywheel.model_meta module

Model metadata and metaclass objects

class flywheel.model_meta.**ModelMetaclass**(*name*, *bases*, *dct*)
Bases: type

Metaclass for Model objects

Merges model metadata, sets the meta_ attribute, and performs validation checks.

class flywheel.model_meta.**ModelMetadata**(*model*)

Bases: object

Container for model metadata

Parameters **model** : *Model*

Attributes

abstract	Getter for abstract

name	(str) The unique name of the model. This is set by the '_name' field in __metadata__. Defaults to the name of the model class.
global_indexes	(list) List of global indexes (hash_key, [range_key]) pairs.
related_fields	(dict) Mapping of field names to set of fields that change when that field changes (usually just that field name, but can be more if composite fields use it)
orderings	(list) List of *Ordering*
throughput	(dict) Mapping of 'read' and 'write' to the table throughput (default 5, 5)

abstract

Getter for abstract

create_dynamo_schema(*connection*, *tablenames=None*, *test=False*, *wait=False*, *throughput=None*, *namespace=()*)

Create all Dynamo tables for this model

Parameters **connection** : DynamoDBConnection

tablenames : list, optional

List of tables that already exist. Will call 'describe' if not provided.

test : bool, optional

If True, don't actually create the table (default False)

wait : bool, optional

If True, block until table has been created (default False)

throughput : dict, optional

The throughput of the table and global indexes. Has the keys 'read' and 'write'. To specify throughput for global indexes, add the name of the index as a key and another 'read', 'write' dict as the value.

namespace : str or tuple, optional

The namespace of the table

Returns **table** : str

Table name that was created, or None if nothing created

ddb_tablename(*namespace=()*)

The name of the DynamoDB table

Parameters **namespace** : list or str, optional

String prefix or list of component parts of a prefix for the table name. The prefix will be this string or strings (joined by '-').

delete_dynamo_schema(*connection*, *tablenames=None*, *test=False*, *wait=False*, *namespace=()*)
Drop all Dynamo tables for this model

Parameters **connection** : `DynamoDBConnection`

tablenames : list, optional

List of tables that already exist. Will call 'describe' if not provided.

test : bool, optional

If True, don't actually delete the table (default False)

wait : bool, optional

If True, block until table has been deleted (default False)

namespace : str or tuple, optional

The namespace of the table

Returns **table** : str

Table name that was deleted, or None if nothing deleted

get_ordering_from_fields(*eq_fields*, *fields*)
Get a unique ordering from constraint fields.

This does a best-effort guess of which index is being queried. It prioritizes indexes that have a constraint on the range key. It prioritizes the primary key over local and global indexes.

Parameters **eq_fields** : list

List of field names that are constrained with '='.

fields : list

List of field names that are constrained with inequality operators ('>', '<', 'beginswith', etc)

Returns **ordering** : *Ordering*

Raises **exc** : `TypeError`

If more than one possible Ordering is found

get_ordering_from_index(*index*)
Get the ordering with matching index name

hk(*obj=None*, *scope=None*)
Construct the primary key value

pk_dict(*obj=None*, *scope=None*, *ddb_dump=False*)
Get the dynamo primary key dict for an item

pk_tuple(*obj=None*, *scope=None*, *ddb_dump=False*, *ddb_load=False*)
Get a tuple that represents the primary key for an item

post_create()
Create the orderings

post_validate()
Build the dict of related fields

rk(*obj=None*, *scope=None*)
Construct the range key value

validate_model()
Perform validation checks on the model declaration

class flywheel.model_meta.**Ordering**(*meta*, *hash_key*, *range_key=None*, *index_name=None*)
Bases: object

A way that the models are ordered

This will be a combination of a hash key and a range key. It may be the primary key, a local secondary index, or a global secondary index.

query_kwargs(*eq_fields*, *fields*)
Get the query and filter kwargs for querying against this index

exception flywheel.model_meta.**ValidationError**
Bases: exceptions.Exception

Model inconsistency

flywheel.model_meta.**merge_metadata**(*cls*)
Merge all the __metadata__ dicts in a class's hierarchy

keys that do not begin with '_' will be inherited.

keys that begin with '_' will only apply to the object that defines them.

flywheel.models module

Model code

class flywheel.models.**Model**(**args*, ***kwargs*)
Bases: object

Base class for all tube models

For documentation on the metadata fields, check the attributes on the *ModelMetadata* class.

Attributes

__metadata_class__	alias of ModelMetadata

__metadata__	(dict) For details see *Metadata*
meta_	(*ModelMetadata*) The metadata for the model
__engine__	(*Engine*) Cached copy of the Engine that was used to save/load the model. This will be set after saving or loading a model.
__dirty__	(set) The set of all immutable fields that have been changed since the last save operation.
__cache__	(dict) The last seen value that was stored in the database. This is used to construct the `expects` dict when making updates that raise on conflict.
__incrs__	(dict) Mapping of fields to atomic add/delete operations for numbers and sets.

add_(***kwargs*)
Atomically add to a set

cached_(*name*, *default=None*)
Get the cached (server) value of a field

construct_ddb_expects_(*fields=None*)
Construct a dynamo "expects" mapping based on the cached fields

ddb_dump_()
Return a dict for inserting into DynamoDB

ddb_dump_cached_(*name*)
Dump a cached field to a Dynamo-friendly value

ddb_dump_field_(*name*)
Dump a field to a Dynamo-friendly value

classmethod ddb_load_(*engine*, *data*)
Load a model from DynamoDB data

delete(*raise_on_conflict=None*)
Delete the model from the database

classmethod field_(*name*)
Construct a placeholder Field for an undeclared field

This is used for creating scan filter constraints on fields that were not declared in the model

get_(*name*, *default=None*)
Dict-style getter for overflow attrs

hk_
The value of the hash key

incr_(***kwargs*)
Atomically increment a number value

keys_()
All declared fields and any additional fields

loading_(**args*, ***kwds*)
Context manager to speed up object load process

mark_dirty_(*name*)
Mark that a field is dirty

meta_ = <flywheel.model_meta.ModelMetadata object>

mutate_(*action*, ***kwargs*)
Atomically mutate a set

persisted_
True if the model exists in DynamoDB, False otherwise

pk_dict_
The primary key dict, encoded for dynamo

pk_tuple_
The primary key dict, encoded for dynamo

post_load_(*engine*)
Called after model loaded from database

post_save_()
Called after item is saved to database

pre_save_(*engine*)
Called before saving items

refresh(*consistent=False*)
Overwrite model data with freshest from database

remove_(***kwargs*)
Atomically remove from a set

rk_
The value of the range key

set_ddb_val_(*key*, *val*)
Decode and set a value retrieved from Dynamo

sync(*raise_on_conflict=None*, *constraints=None*)
Sync model changes back to database

class flywheel.models.**SetDelta**
Bases: object

Wrapper for an atomic change to a Dynamo set

Used to track the changes when using *add_()* and *remove_()*

add(*action*, *value*)
Add another update to the delta

Parameters **action** : {'ADD', 'DELETE'}

value : object

The value to add or remove

merge(*other*)
Merge the delta with a set

Parameters **other** : set

The original set to merge the changes with

flywheel.query module

Query and Scan builders

exception flywheel.query.**DuplicateEntityException**
Bases: exceptions.ValueError

Raised when too many results are found.

exception flywheel.query.**EntityNotFoundException**
Bases: exceptions.ValueError

Raised when results are expected and not found.

class flywheel.query.**Query**(*engine*, *model*)
Bases: object

An object used to query dynamo tables

See the *Engine* for query examples

Parameters **engine** : *Engine*

model : class

Subclass of *Model*

all (*desc=False*, *consistent=False*, *attributes=None*, *filter_or=False*, *exclusive_start_key=None*)
Return the query results as a list

Parameters **desc** : bool, optional

Return results in descending order (default False)

consistent : bool, optional

Force a consistent read of the data (default False)

attributes : list, optional

List of fields to retrieve from dynamo. If supplied, returns dicts instead of model objects.

filter_or : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

exclusive_start_key : dict, optional

The ExclusiveStartKey to resume a previous query

Returns **results** : list

count (*filter_or=False*)
Find the number of elements the match this query

Parameters **filter_or** : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

Returns **count** : int

delete (*filter_or=False*)
Delete all items that match the query

Parameters **filter_or** : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

dynamo
Shortcut to access DynamoDBConnection

filter (**conditions*, ***kwargs*)
Add a Condition to constrain the query

Notes

The conditions may be passed in as positional arguments:

```
engine.query(User).filter(User.id == 12345)
```

Or they may be passed in as keyword arguments:

```
engine.query(User).filter(firstname='Monty', lastname='Python')
```

The limitations of the keyword method is that you may only create equality conditions. You may use both types in a single filter:

```
engine.query(User).filter(User.num_friends > 10, name='Monty')
```

first (*desc=False*, *consistent=False*, *attributes=None*, *filter_or=False*)
Return the first result of the query, or None if no results

Parameters **desc** : bool, optional

Return results in descending order (default False)

consistent : bool, optional

Force a consistent read of the data (default False)

attributes : list, optional

List of fields to retrieve from dynamo. If supplied, returns dicts instead of model objects.

filter_or : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

Returns **result** : *Model* or None

gen(*desc=False*, *consistent=False*, *attributes=None*, *filter_or=False*, *exclusive_start_key=None*)
Return the query results as a generator

Parameters **desc** : bool, optional

Return results in descending order (default False)

consistent : bool, optional

Force a consistent read of the data (default False)

attributes : list, optional

List of fields to retrieve from dynamo. If supplied, gen() will iterate over dicts instead of model objects.

filter_or : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

exclusive_start_key : dict, optional

The ExclusiveStartKey to resume a previous query

Returns **results** : generator

index(*name*)
Use a specific local or global index for the query

limit(*count*)
Limit the number of query results

one(*consistent=False*, *attributes=None*, *filter_or=False*)
Return the result of the query. If there is not exactly one result, raise a ValueError

Parameters **consistent** : bool, optional

Force a consistent read of the data (default False)

attributes : list, optional

List of fields to retrieve from dynamo. If supplied, returns dicts instead of model objects.

filter_or : bool, optional

If True, multiple filter() constraints will be joined with an OR (default AND).

Returns **result** : *Model*

Raises **e1** : *EntityNotFoundException*

If no entity is found. Subclasses `ValueError`.

e2 : *DuplicateEntityException*

If more than one entity is found. Subclasses `ValueError`.

tablename
Shortcut to access dynamo table name

class `flywheel.query.`**Scan**(*engine*, *model*)
Bases: *flywheel.query.Query*

An object used to scan dynamo tables

scans are like Queries except they don't use indexes. This means they iterate over all data in the table and are SLOW

Parameters **engine** : *Engine*

model : class

Subclass of *Model*

count(*filter_or=False*)

gen(*attributes=None*, *desc=False*, *consistent=False*, *filter_or=False*, *exclusive_start_key=None*)

index(*name*)

flywheel.tests module

Unit and system tests for flywheel

class `flywheel.tests.`**DynamoSystemTest**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Base class for tests that need an *Engine*

dynamo = None

models = []

classmethod setUpClass()

tearDown()

classmethod tearDownClass()

2.1.3 Module contents

flywheel

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

f

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V