dynamodb-mapper Documentation

Release 1.6.2

Max Noel

August 13, 2012

CONTENTS

CHAPTER

ONE

OVERVIEW

DynamoDB is a minimalistic NoSQL engine provided by Amazon as a part of their AWS product.

DynamoDB allows you to stores documents composed of unicode strings or numbers as well as sets of unicode strings and numbers. Each tables must define a hash key and may define a range key. All other fields are optional.

Dynamodb-mapper brings a tiny abstraction layer over DynamoDB to overcome some of the limitations with no performance compromise. It is highly inspired by the mature MoongoKit project

CHAPTER

TWO

DOCUMENTATION

2.1 User guide

2.1.1 Overview of Dynamodb-mapper

DynamoDB is a minimalistic NoSQL engine provided by Amazon as a part of their AWS product.

DynamoDB allows you to stores documents composed of unicode strings or numbers as well as sets of unicode strings and numbers. Each tables must define a hash key and may define a range key. All other fields are optional.

Dynamodb-mapper brings a tiny abstraction layer over DynamoDB to overcome some of the limitations with no performance compromise. It is highly inspired by the mature MoongoKit project

Requirements

The documentation currently assumes that you're running Boto 2.3.0 or later. If you're not, then the API for query and scan changes. You will have to supply raw condition dicts, as is done in boto itself.

Also note that Boto 2.3.1 or later is required for autoincrement_int hash keys. Earlier versions will fail.

Features

- Python <–> DynamoDB type mapping
- dict and lists serialization
- default values
- Multi-target transaction support with auto-retry (new in 1.6.0)
- Auto-inc hash_key
- Protection against the 'lost update' syndrom
- New table creation
- Framework agnostic
- Log all successful database access

Logging

Dynamodb-mapper uses 3 "logging" loggers:

- model
- model.database-access
- transactions

Known limitations

- Dates nested in a dict or set can not be saved as `datetime` does not support JSON serialization. (issue #7)

2.1.2 Getting started with Dynamodb-mapper

Setup Dynamodb-mapper

Installation

```
$ pip install dynamodb-mapper
```

Set you Amazon's API credential in `~/.boto`

```
[Credentials]
aws_access_key_id = <your access key>
aws_secret_access_key = <your secret key>
```

For advance configuration, please see the official Boto documentation.

Example data: DoomMap

We want a DoomMap to be part of an `episode`. In our schema, the `episodes` are identified by an integer ID, this is the `hash_key`. We also want our episodes to have multiple maps also identified by an integer. This `map` id is the `range_key`. `range_key` allows to logically group items that belongs to a same group.

Our maps also have an `name` and a set of `cheats` codes. In DynamoDB, all strings are stored as `unicode` hence the type. Lastly, we want each maps to recognize by `__defaults__` the famous "Konami" cheat code.

DoomMap Model

Start by defining the document structure.

```
from dynamodb_mapper.model import DynamoDBModel


class DoomMap(DynamoDBModel):
    __table__ = u"doom_map"
    __hash_key__ = u"episode"
    __range_key__ = u"map"
    __schema__ = {
        u"episode": int,
```

```
        u"map": int,
        u"name": unicode,
        u"cheats": set,
    }
    __defaults__ = {
        u"cheats": set([u"Konami"]),
    }
```

All class attributes of the form `__attr__` are used to configure the mapper. Note that they are defined on the class level. Any accidental override in the instances will be ignored.

- `__table__` Table name in DynamoDB
- `__hash_key__` Name of the the hash key field
- `__range_key__` Name of the (optional) range key field
- **`__schema__` Dict mapping of `{"field_name": type}`. Must at least contain** the keys
- `__defaults__` Define an optional default value for each field used by `__init__`

For more informations on the models and defaults, please see the *data models* section of this manual.

Initial Table creation

Unlike MongoDB, table creation must be done explicitly. At the moment `create_table()`, is the only case where you'd want to directly use the `ConnectionBorg` class.

```
conn = ConnectionBorg()
conn.create_table(DoomMap, 10, 10, wait_for_active=True)
```

When creating a table with, you must specify the model class and the desired R/W throughput that is to say the peek number of request per seconds you expect for you application. For more information, please see Amazon's official documentation.

Default behavior is to create the tables asynchronously but you may explicitly ask for synchronous creation with `wait_for_active=True`. Please note that only 10 tables may be in `CREATING` simultaneously.

Example Usage

First, create and `save()` new map in episode 1 and call it "Hangar". Let's also register a couple a cheats.

```
e1m1 = DoomMap()
e1m1.episode = 1
e1m1.map = 1
e1m1.name = u"Hangar"
e1m1.cheats = set([u"idkfa", u"iddqd", u"idclip"])
e1m1.save()
```

It is now possible to `get()` it from the database using a conpound index that is to say, both a `hash_key` and a `range_key`. By default, `get` uses "eventual consistence" for data access but it is possible to ask for strongly consistent data using `consistent_read=True`.

```
# Later on, retrieve that same object from the DB...
e1m1 = DoomMap.get(1, 1)
```

What if I want to get all the maps in a given episode? This is the purpose of the `query()` methode which also allows to filter the results based on the `range_key` value.

```
# query all maps of episode 1
e1_maps = DoomMap.query(hash_key=1)

# query all maps of episode 1 with 'map' hash_key > 5
from boto.dynamodb.condition import GT
e1_maps_after_5 = DoomMap.query(
    hash_key=1,
    range_key_condition=GT(5))
```

Dynamodb-mapper offers much more usage tools like `scan()` and `delete()`, `Transaction` support...

2.1.3 Data models

Models are formal Pythons objects telling the mapper how to map DynamoDB data to regular Python and vice versa.

Bare minimal model

A bare minimal model with only a `hash_key` needs only to define a `__table__` and a `hash_key`.

```
from dynamodb_mapper.model import DynamoDBModel

class MyModel(DynamoDBModel):
    __table__ = u"..."
    __hash_key__ = u"key"
    __schema__ = {
        u"key": int,
        #...
    }
```

The model can then be instanciated and used like any other Python class.

```
>>> data = MyModel()
>>> data.key = u"foo/bar"
```

About keys

While this is not stricly speaking related the mapper itself, it seems important to clarify this point as this is a key feature of Amazon's DynamoDB.

Amazon's DynamoDB has support for 1 or 2 keys per objects. They must be specified at table creation time and can not be altered. Neither renamed nor added or removed. It is not even possible to change their values whithout deleting and re-inserting the object in the table.

The first key is mandatory. It is called the `hash_key`. The `hash_key` is to access data and controls its replications among database partitions. To take advantage of all the provisioned R/W throughput, keys should be as random as possible. For more informations about `hash_key`, please see Amazon's developer guide

The second key is optional. It is called the `range_key`. The `range_key` is used to logically group data with a given `hash_key`. *More informations below*.

Data access relying either on the `hash_key` or both the `hash_key` and the `range_key` is fast and cheap. All other options are **very** expensive.

We intend to add migration tools to Dynamodb-mapper in a later revision but do not expect miracles in this area.

This is why correctly modeling your data is crucial with DynamoDB.

Creating the table

Unlike other NoSQL engines like MongoDB, tables must be created and managed explicitely. At the moment, dynamodb-mapper abstracts only the initial table creation. Other lifecycle managment operations may be done directly via Boto.

To create the table, use `create_table()` with the model class as first argument. When calling this method, you must specify how much throughput you want to provision for this table. Throughput is mesured as the number of atomic KB requested or sent per second. For more information, please see Amazon's official documentation.

```
from dynamodb_mapper.model import DynamoDBModel, ConnectionBorg

conn = ConnectionBorg()
conn.create_table(MyModel, read_units=10, write_units=10, wait_for_active=True)
```

Important note: Unlike most databases, table creation may take up to 1 minute. during this time, the table is *not* usable. Also, you can not have more than 10 tables in `CREATING` or `DELETING` state any given time for your whole Amazon account. This is an Amazon's DynamoDB limitation.

The connection manager automatically reads your credentials from either:

- `/etc/boto.cfg`
- `~/.boto`
- or `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables

If none of these places defines them or if you want to overload them, please use `set_credentials()` before calling `create_table`.

For more informations on the connection manager, pease see `ConnectionBorg`

Advanced usage

Namespacing the models

This is more an advice, than a feature. In DynamoDB, each customer is allocated a single database. It is highly recommended to namespace your tables with a name of the form <application>-<env>-<model>.

Using auto-incrementing index

For those comming from SQL-like world or even MongoDB with its UUIDs, adding an ID field or using the default one has become automatic but these environement are not limited to 2 indexes. Moreover, DynamoDB has no built-in support for it. Nonetheless, Dynamodb-mapper implements this feature at a higher level while. For more technical background on the *internal implementation*.

If the field value is left to its default value of 0, a new hash_key will automatically be generated when saving. Otherwise, the item is inserted at the specified `hash_key`.

Before using this feature, make sure you *really need it*. In most cases another field can be used in place. A good hint is "which field would I have marked UNIQUE in SQL ?".

- for users, `email` or `login` field shoud do it.
- for blogposts, `permalink` could to it too.
- for orders, `datetime` is a good choice.

In some applications, you need a combination of 2 fields to be unique. You may then consider using one as the `hash_key` and the other as the `range_key` or, if the `range_key` is needed for another purpose, combine try combining them.

At Ludia, this is a feature we do not use anymore in our games at the time of writing.

So, when to use it ? Some applications still need a ticket like approach and dates could be confusing for the end user. The best example for this is a bugtracking system.

Use case: Bugtracking System

```
from dynamodb_mapper.model import DynamoDBModel, autoincrement_int

class Ticket(DynamoDBModel):
    __table__ = u"bugtracker-dev-ticket"
    __hash_key__ = u"ticket_number"
    __schema__ = {
        u"ticket_number": autoincrement_int,
        u"title": unicode,
        u"description": unicode,
        u"tags": set, # target, version, priority, ..., order does not matter
        u"comments": list, # probably not the best because of the 64KB limitation...
        #...
    }

# Create a new ticket and auto-generate an ID
ticket = Ticket()
ticket.title = u"Chuck Norris is the reason why Waldo hides"
ticket.tags = set([u'priority:critical', u'version:yesterday'])
ticket.description = u"Ludia needs to create a new social game to help people all around the world f
ticket.comments.append(u"...")
ticket.save()
print ticket.ticket_number # A new id has been generated

# Create a new ticket and force the ID
ticket = Ticket()
ticket.ticket_number = 42
ticket.payload = u"foo/bar"
ticket.save() # create or replace item #42
print ticket.ticket_number # id has not changed
```

To prevent accidental data overwrite when saving to an arbitrary location, please see the detailed presentation of *Saving*.

Please note that `hash_key=-1` is currently reserved and nothing can be stored at this index.

You can not use `autoincrement_int` and a `range_key` at the same time. In the bug tracker example above, it also means that tickets number are distributed on the application scope, not on a per project scope.

This feature is only part of Dynamodb-mapper. When using another mapper or direct data access, you might *corrupt* the counter. Please see the reference documentation for implementation details and technical limitations.

Using a range_key

Models may define a second key index called `range_key`. While `hash_key` only allows dict like access, `range_key` allows to group multiple items under a single `hash_key` and to further filter them.

For example, let's say you have a customer and want to track all it's orders. The naive/SQL-like implementation would be:

```
from dynamodb_mapper.model import DynamoDBModel, autoincrement_int

class Customer(DynamoDBModel):
    __table__ = u"myapp-dev-customers"
    __hash_key__ = u"login"
    __schema__ = {
        u"login": unicode,
        u"order_ids": set,
        #...
    }

class Order(DynamoDBModel):
    __table__ = u"myapp-dev-orders"
    __hash_key__ = u"order_id"
    __schema__ = {
        u"order_id": autoincrement_int,
        #...
    }

# Get all orders for customer "John Doe"
customer = Customer(u"John Doe")
order_generator = Order.get_batch(customer.order_ids)
```

But this approach has many drawbacks.

- **It is expensive:**
  - An update to generate a new autoinc ID
  - An insertion for the new order item
  - An update to add the new order id to the customer
- **It is risky:**
  - Items are limited to 64KB but the `order_ids` set has no growth limit
- **To get all orders from a giver customer, you need to read the customer first** and use a `get_batch()` request

As a first enhancement and to spare a request, you can use `datetime` instead of `autoincrement_int` for the key `order_id` but with the power of range keys, you could to get all orders in a single request:

```
from dynamodb_mapper.model import DynamoDBModel
from datetime import datetime

class Customer(DynamoDBModel):
    __table__ = u"myapp-dev-customers"
    __hash_key__ = u"login"
    __schema__ = {
        u"login": unicode,
        #u"orders": set, => This field is not needed anymore
        #...
    }

class Order(DynamoDBModel):
    __table__ = u"myapp-dev-orders"
    __hash_key__ = u"login"
    __range_key__ = u"order_id"
```

```
    __schema__ = {
        u"order_id": datetime,
        #...
    }

# Get all orders for customer "John Doe"
Order.query(u"John Doe")
```

Not only is this approach better, it is also much more powerful. We could easily limit the result count, sort them in reverse order or filter them by creation date if needed. For more background on the querying system, please see the *accessing data* section of this manual.

Default values

When instanciating a model, all fields are initialised to "neutral" values. For containers (`dict`, `set`, `list`, ...) it is the empty container, for `unicode`, it's the empty string, for numbers, 0...

It is also possible to specify the values taken by the fields when instanciating either with a `__defaults__` dict or directly in `__init__`. The former applies to all new instances while the later is obviously on a per instance basis and has a higher precedence.

`__defaults__` is a `{u'keyname':default_value}`. `__init__` syntax follows the same logic: `Model(keyname=default_value, ...)`.

`default_value` can either be a scalar value or a callable with no argument returning a scalar value. The value must be of type matching the schema definition, otherwise, a `TypeError` exception is raised.

Example:

```
from dynamodb_mapper.model import DynamoDBModel, utc_tz
from datetime import datetime

# define a model with defaults
class PlayerStrength(DynamoDBModel):
    __table__ = u"player_strength"
    __hash_key__ = u"player_id"
    __schema__ = {
        u"player_id": int,
        u"strength": unicode,
        u"last_update": datetime,
    }
    __defaults__ = {
        u"strength": u'weak', # scalar default value
        u"last_update": lambda: datetime.now(utc_tz), # callable default value
    }

>>> player = PlayerStrength(strength=u"chuck norris") # overload one of the defaults
>>> print player.strength
chuck norris
>>> print player.lastUpdate
2012-12-21 13:37:00.00000
```

Related exceptions

SchemaError

class dynamodb_mapper.model.**SchemaError**
SchemaError exception is raised when a schema consistency check fails. Most of the checks are performed in `create_table()`.

Common consistency failure includes lacks of `__table__`, `__hash_key__`, `__schema__` definition or when an `autoincrement_int` `hash_key` is used with a `range_key`.

ThroughputError

class dynamodb_mapper.model.**ThroughputError**
Raised when requested throughput can not be allocated by `create_table()`. It probably means that either read or write is below Amazon's minimum of 5.

2.1.4 Accessing data

Amazon's DynamoDB offers 4 data access method. Dynamodb-mapper directly exposes them. They are documented here from the fastest to the slowest. It is interesting to note that, because of Amazon's throughput credit, the slowest is also the most expensive.

Strong vs eventual consistency

While this is not stricly speaking related the mapper itself, it seems important to clarify this point as this is a key feature of Amazon's DynamoDB.

Tables are spreaded among partitions for redundancy and performance purpose. When writing an item, it takes some time to replicate it on all partitions. Usually less than a second according to the technical specifications. Accessing an item right after writing it might get you an outdated version.

In most applications, this will not be an issue. In this case we say that data is 'eventually consistent'. If this matters, you may request 'strong consistency' thus asking for the most up to date version. 'strong consistency' is also more twice as expensive in terms of capacity units as 'eventual consistency' and a bit slower too. So that keeping this aspect in mind is important.

'Eventual consistency' is the default behavior in all requests. It also the only available option for `scan` and `get_batch`.

Querying

The 4 DynamoDB query methods are:

- `get()`
- `get_batch()`
- `query()`
- `scan()`

They all are `classmethods` returning instance(s) of the model. To get object(s):

```
>>> obj = MyModelClass.get(...)
```

Use `get` or `batch_get` to get one or more item by exact id. If you need more than one item, it is highly recommended to use `batch_get` instead of `get` in a loop as it avoids the cost of multiple network call. However, if strong consistency is required, `get` is the only option as DynamoDB does not support it in batch mode.

When objects are logically grouped using a *range_key* it is possible to get all of them in a simple query and fast query provided they all have the same known `hash_key`. `query()` also supports a couple of handy filters.

When querying, you pay only for the results you really get this is what makes filtering interesting. They work both for strings and for numbers. The `BEGINSWITH` filter is extremely handy for namespaced `range_key`. When using `EQ(x)` filter, it may be preferable for readability to rewrite it as a regular `get`. The cost in terms of read units is strictly speaking the same.

If needed `query()` support `strong consistency`, reversing scan order and limiting the results count.

The last function, `scan`, is like a generalised version of `query`. Any field can be filtered and more filters are available. There is a complete list on the Boto website. Nonetheless, `scan` results are *always* `eventually consistent`.

This said, `scan` is extremely expensive in terms of throughput and its use should be avoided as much as possible. It may even impact negatively pending regular requests causing them to repetively fail. Underlying Boto tries to gracefully handle this but you overall application's performance and user experience might suffer a lot. For more informations about `scan` impact, please see Amazon's developer guide

Use case: Get user `Chuck Norris`

This first example is pretty straight-forward.

```
from dynamodb_mapper.model import DynamoDBModel

# Example model
class MyUserModel(DynamoDBModel):
    __table__ = u"..."
    __hash_key__ = u"fullname"
    __schema__ = {
        # This is probably a good key in a real world application because of homonynes
        u"fullname": unicode,
        # [...]
    }

# Get the user
myuser = MyUserModel.get("Chuck Norris")

# Do some work
print "myuser({})".format(myuser.fullname)
```

Use case: Get only objects after `2012-12-21 13:37`

At the moment, filters only accepts strings and numbers. If you need to filter dates for time based applications. To workaround this limitation, you need to export the `datetime` object to the internal W3CDTF representation.

```
from datetime import datetime
from dynamodb_mapper.model import DynamoDBModel, utc_tz
from boto.dynamodb.condition import *

# Example model
```

```
class MyDataModel(DynamoDBModel):
    __table__ = u"..."
    __hash_key__ = u"h_key"
    __range_key__ = u"r_key"
    __schema__ = {
        u"h_key": int,
        u"r_key": datetime,
        # [...]
    }

# Build the date condition and export it to W3CDTF representation
date_obj = datetime.datetime(2012, 12, 21, 13, 31, 0, tzinfo=utc_tz),
date_str = date_obj.astimezone(utc_tz).strftime("%Y-%m-%dT%H:%M:%S.%f%z")

# Get the results generator
mydata_generator = MyDataModel.query(
    hash_key_value=42,
    range_key_condition=GT(date_str)
)

# Do some work
for data in mydata_generator:
    print "data({}, {})".format(data.h_key, data.r_key)
```

Use case: Query the most up to date revision of a blogpost

There is no builtin filter but this can easily be achieved using a conjunction of `limit` and `reverse` parameters. As `query` returns a generator, `limit` parameter could seem to be of no use. However, internaly DynamoDB sends results by batches of 1MB and you pay for all the results so... you'd beter use it.

```
from dynamodb_mapper.model import DynamoDBModel, utc_tz

# Example model
class MyBlogPosts(DynamoDBModel):
    __table__ = u"..."
    __hash_key__ = u"post_id"
    __range_key__ = u"revision"
    __schema__ = {
        u"post_id": int,
        u"revision": int,
        u"title": unicode,
        u"tags": set,
        u"content": unicode,
        # [...]
    }

# Get the results generator
mypost_last_revision_generator = MyBlogPosts.query(
    hash_key_value=42,
    limit=1,
    reverse=True
)

# Get the actual blog post to render
try:
    mypost = mypost_last_revision_generator.next()
except StopIteration:
```

```
    mypost = None # Not Found
```

This example could easily be adapted to get the first revision, the n first comments. You may also combine it with a condition to get pagination like behavior.

2.1.5 Data manipulation

Amazon's DynamoDB offers the ability to both update and insert data with a single `save()` method that is mostly exposed by Dynamodb-mapper.

Saving

As Dynamodb-mapper directly exposes items properties as python properties, manipulating data is as easy as manipulating any Python object. Once done, just call `save()` on your model instance.

`save()` has 2 optional parameters. When manually set to `False`, `allow_overwrite` will only allow the *insertion* of a new Item. this is done by setting a condition on the keys.

The second parameter `expected_values` will garantee that the Item is saved only if these values are present in the database. This `dict` is a bit tricky to use as it needs to be a raw DynamoDB mapping.

- It supports only string and numers.
- When a field is set to `False`, it will ensure that it does *not* exist.

Hopefully, DynamodbModel class offers an utility method to ease the mapping creation. `to_db_dict()` converts the current Item to a DynamoDB compatible representation.

Use case: Virtual coins

When a player purchases a virtual good in a game, virtual money is withdrawn from from its internal account. After the operation, the balance must be > 0. If multiple orders are being processed at the same time, we must prevent the *lost update* scenario:

- initial balance = 200
- purchase P1 150
- purchase P2 100
- read balance P1 -> 200
- read balance P2 -> 200
- update balance P1 -> 50
- update balance P1 -> 100

Indeed, when saving, you **expect** that the balance has not changed. This is what `expected_values` are for.

```
from dynamodb_mapper.model import DynamoDBModel, autoincrement_int

class NotEnoughCreditException(Exception):
    pass

class User(DynamoDBModel):
    __table__ = u"game-dev-users"
    __hash_key__ = u"login"
    __schema__ = {
```

```
        u"e-mail": unicode,
        u"firstname": unicode,
        u"lastname": unicode,
        u"e-mail": unicode,
        u"connexioncount": int,
        #...
        u"balance": int,
    }

user = User.get("waldo")
oldbalance = user.balance
if user.balance - 150 < 0:
    raise NotEnoughCreditException
user.balance -= 150

try:
    user.save(expected_values={"balance": oldbalance})
except ExpectedValueError:
    print "Ooops: Lost update syndrome caught!"
```

Note: In a real world application, this would most probably be wrapped in *Transactions*

Autoincrement technical background

When saving an Item with an `autoincrement_int` `hash_key`, the `save()` method will automatically add checks to prevent accidental overwrite of the "magic item". The magic item holds the last allocated ID and is saved at `hash_key=-1`. If `hash_key == 0` then a new ID is automatically and atomically allocated meaning that no collision can occure even if the database connection is lost. Additionaly, a check is performed to make sure no Item were manually inserted to this location. If applicable, a maximum of `MAX_RETRIES=100` attempts to allocate a new ID will be performed before raising `MaxRetriesExceededError`. In all other cases, the Item will be saved exactly where requested.

To make it short, Items involving an `autoincrement_int` `hash_key` will involve 2 write request on first save. It is important to keep it in mind when dimensioning an insert-intensive application.

*Know when to use it, when *not* to use it.*

Example:

```
>>> model = MyModel() # model with an autoincrement_int 'id' hash_key
>>> model.do_stuff()
>>> model.save()
>>> print model.id # An id field is automatically generated
7
```

About editing `hash_key` and/or `range_key` values

Dynamodb-mapper let you edit `hash_key` and/or `range_key` fields like any other. However, Amazon's DynamoDB has no support for changing their values. If they are edited, a new item will be saved in the table with these keys. If you indeed meant to change the keys, first delete the item and then save it again. Beware that any item pre-existing at this keys will be overwritten unless `allow_overwrite=True` in `save`.

Example:

```
>>> model = MyModel.get(24)
>>> model.delete() # Delete *first*
```

```
>>> model.id = 42   # Then change the key(s)
>>> model.save()    # Finally, save it
```

There is no plan to protect the key fields in any future release.

Logically group data manipulations

Some data manipulations requires a whole context to be consistent, status saving or whatever. If your application requires any of these features, please go to the *transactions section* of this guide.

Limitations

Some limitations over Amazon's DynamoDB currently applies to this mapper. `save()` has no support for :

- returning data after a transaction
- atomic increments

Please, let us know if this is a blocker to you!

Related exceptions

OverwriteError

class `dynamodb_mapper.model.`**`OverwriteError`**

Raised when saving a DynamoDBModel instance would overwrite something in the database and we've forbidden that because we believe we're creating a new one (see `DynamoDBModel.save()`).

ExpectedValueError

class `dynamodb_mapper.model.`**`ExpectedValueError`**

Conditional write failure. Raised when the expected_values argument to `DynamoDBModel.save()` doesn't match what is stored in the database (i.e. when somebody changed the DB's version of your object behind your back).

2.1.6 Transactions

The *save use case* demonstrates the use of `expected_values` argument. What it does is actually implement by hand a transaction. Amazon's DynamoDB has no "out of the box" transaction engines but provides this parameter as an elementary block for this purpose.

Transaction concepts

Transactions are a convenient way to logically group database operations while trying as much as possible to enforce consistency. In Dynamodb-mapper, transactions *are* plain `DynamoDBModel` thus allowing them to persist their state. Dynamodb-mapper provides 2 grouping level: Targets and sub-transactions.

Transactions operates on a list of 'targets'. For each target, it needs list of `transactors`. `transactors` are tuples of `(getter, setter)`. The getter is responsible of getting a fresh copy of the target from the target while setter performs the modifications. The call to save is handled by the engine itself.

For each target, the transaction engine will successively call `getter` and `setter` until `save()` succeeds. `save()` will succeed if and only if the target has not been altered by another thread in the mean time thus avoiding the lost update syndrome.

Optionally, transactions may define a method `_setup()` which will be called before any transactors.

Sub-transactions, if applicable, are ran after the main transactors if they all succeeded. Hence, `_setup()` and the `transactors` may dynamically append sub-transactions to the main transactions.

Unless the transaction is explicitely marked `transient`, its state will be persisted to a dedicated table. `Transaction` base class embeds a minimal schema that should suit most applications but may be overloaded as long as a `datetime range_key` is preserved along with a `unicode status` field.

Using the transaction engine

To use the transaction engine, all you have to do is to define *__table__* and overload `_get_transactors()`. Of course the transactors will themselves will need to be implemented. Optionally, you may overload the whole schema or set `transient=True`. A `_setup()` method may also be implemented.

During the transaction itself, please set `requester_id` field to any relevant interger unless the transaction is `transient`. `_setup()` is a good place to do it.

Note: `transient` flag may be toggled on a per instance basis. It may even be toggled in one of the transactors.

Use case: Bundle purchase

```
from dynamodb_mapper.transactions import Transaction, TargetNotFoundError

# define PlayerExperience, PlayerPowerUp, PlayerSkins, Players with user_id as hash_key

class InsufficientResourceError(Exception):
    pass

bundle = {
    u"cost": 150,
    u"items": [
        PlayerExperience,
        PlayerPowerUp,
        PlayerSkins
    ]
}

class BundleTransaction(Transaction):
    transient = False # Make it explicit. This is anyway the default.
    __table__ = u"mygame-dev-bundletransactions"

    def __init__(self, user_id, bundle):
        super(BundleTransaction, self).__init__()
        self.requester_id = user_id
        self.bundle = bundle

    # _setup() is not needed here

    def _get_transactors(self):
        transactors = [(
            lambda: Players.get(self.requester_id), # lambda
            self.user_payment # regular callback
```

```
        )]

        for Item in self.bundle.items:
            transactors.append((
                lambda: Item.get(self.requester_id),
                lambda item: item.do_stuff()
            ))

        return transactors

    def user_payment(self, player):
        if player.balance < self.bundle.cost:
            raise InsufficientResourceError()
        player.balance -= self.bundle.cost

# Run the transaction
try:
    transaction = BundleTransaction(42, bundle)
    transaction.commit()
except InsufficientResourceError:
    print "Ooops, user {} has not enough coins to proceed...".format(42)

#That's it !
```

This example has been kept simple on purpose. In a real world application, you certainly would *not* model your data this way ! You can notice the power of this approach that is compatible with `lambda` niceties as well as regular callbacks.

Use case: PowerUp purchase

This example is a bit more subtle than the previous one. The customer may purchase a '*surprise*' bundle of powerups. The database knows what is in the pack while the client application does not. As bundles may change from time to time, we want to log what exactly was purchased. Also, the actual `PowerUp` registration should not start until the `Coins` transaction has succeeded.

To reach this goal, we could

- pre-load the `Bundle`
- dynamically use the content in get_transactors
- save the detailed status in a specially overloaded Transaction's __`schema`__

But that's more hand work.

A much better way is to split the transaction into `PowerupTransaction` and `UserPowerupTransaction`. The former handles the coins and the registration of the sub-transaction while the later handles the PowerUo magic.

```
from dynamodb_mapper.transactions import Transaction, TargetNotFoundError

# define PlayerPowerUp, Players with user_id as hash_key

class InsufficientResourceError(Exception):
    pass

# Sub-Transaction of PowerupTransaction. Will have i's own status
class UserPowerupTransaction(transaction):
    __table__ = u"mygame-dev-userpoweruptransactions"
```

```
    def __init__(self, player, powerup):
        super(UserPowerupTransaction, self).__init__()
        self.requester_id = player.user_id
        self.powerup = powerup

    def _get_transactors(self):
        return [(
            lambda: PlayerPowerUp.get(self.requester_id, self.powerup),
            do_stuff()
        )]

# Main Transaction class. Will have it's own status
class PowerupTransaction(Transaction):
    __table__ = u"mygame-dev-poweruptransactions"

    cost = 150 # hard-coded cost for the demo
    powerups = ["..."] # hard-coded powerups for the demo

    def _get_transactors(self):
        return [(
            lambda: Players.get(self.requester_id),
            self.user_payment
        )]

    def user_payment(self, player):
        # Payment logic
        if player.balance < self.cost:
            raise InsufficientResourceError()
        player.balance -= self.cost

        # Register (overwrite) sub-transactions
        self.subtransactions = []
        for powerupName in self.powerups:
            self.subtransactions.append = (player, powerupName)


# Run the transaction
try:
    transaction = PowerupTransaction(requester_id=42)
    transaction.commit()
except InsufficientResourceError:
    print "Ooops, user {} has not enough coins to proceed...".format(42)

#That's it !
```

Note: In some special "real-World(tm)" situations, it may be necessary to modify the behavior of subtransactions. It is possible to overload the method `Transaction._apply_subtransactions()` for this purpose. Use case: sub-transactions have been automatically/randomly generated by the main transaction and the application needs to know wich one were generated or perform some other application specific tasks when applying.

Related exceptions

MaxRetriesExceededError

class `dynamodb_mapper.model.`**`MaxRetriesExceededError`**

Raised when a failed operation couldn't be completed after retrying `MAX_RETRIES` times (e.g. saving an autoincrementing hash_key).

Note: `MAX_RETRIES` is currently hardcoded to `100` in transactions module.

TargetNotFoundError

class `dynamodb_mapper.transactions.`**`TargetNotFoundError`**
Raised when attempting to commit a transaction on a target that doesn't exist.

2.1.7 Change log - Migration guide.

DynamoDBMapper 1.6.2

This section documents all user visible changes included between DynamoDBMapper versions 1.5.0 and versions 1.6.0

Additions

- transactions may generate a list of sub-transactions to run after the main one
- log all successful queries
- add parameter `limit` on `query` method defaulting to `None`
- extensive documentation

Upgrade

sub-transactions If `__init__()` is called in any of your transactions, make sure to call `super(MyTransactionClass, self).__init__(**kwargs)`

Known bugs - limitations

- #7 Can't save models where a datetime field is nested in a dict/list.
- Can't use `datetime` objects in `scan` and `query` filters.
- DynamoDBModel.from_dict() does not check types as opposed to `__init__()`

DynamoDBMapper 1.6.1

This section documents all user visible changes included between DynamoDBMapper version 1.6.0 and version 1.6.1

Changes

- fixed bug in scan

DynamoDBMapper 1.6.0

This section documents all user visible changes included between DynamoDBMapper versions 1.5.0 and versions 1.6.0

Additions

- support for default values in a `__defaults__` dict
- specify instances members via global `__init__ **kwargs`
- autogenerated API documentation

Changes

- transactions engine rewrite to support multiple targets
- transactions always persisted after first write attempt
- transactions engine now embeds its own minimal schema
- transactions can be set `transient` on a 'per instance basis' instead of class
- autoinc hash key now relies on `atomic add` to prevent risks of races
- autoinc magic element moved to -1 instead of 0 to prevent accidental overwrite
- autoinc magic element now hidden from scan results
- factorized default value code
- enforce batch size 100 limit
- full inline documentation
- fixed issue: All transactions fail if they have a bool field set to False
- 99% test coverage

Removal

(None)

Upgrade

autoinc For all tables relying on autoinc feature, manually move element at `'hash_key' = 0` to `'hash_key' = -1`.

transactions Should be retro-compatible but you are strongly advised to adopt the new API. - specify `targets` and `setters` via `Transactions._get_transactors` - avoid any use of `Transactions._get_target` and `Transactions._alter_target` - save is now called automatically as long as at least 1 write was attempted - `__schema__` might not be required anymore due to `Transaction` having a new one - `requester_id` hash key must be set by the user See these method's documentation for more informations

Known bugs

(None)

2.2 Api reference

2.2.1 Connection class

Class definition

class `dynamodb_mapper.model.`**`ConnectionBorg`**

Borg that handles access to DynamoDB.

You should never make any explicit/direct `boto.dynamodb` calls by yourself except for table maintenance operations :

- `boto.dynamodb.table.update_throughput()`
- `boto.dynamodb.table.delete()`

Remember to call `set_credentials()`, or to set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables before making any calls.

Initialisation

`ConnectionBorg.`**`set_credentials`**(*aws_access_key_id*, *aws_secret_access_key*)

Set the DynamoDB credentials. If boto is already configured on this machine, this step is optional. Access keys can be found in Amazon's console.

Parameters

- **aws_access_key_id** – AWS api access key ID
- **aws_secret_access_key** – AWS api access key

Create a table

`ConnectionBorg.`**`create_table`**(*cls*, *read_units*, *write_units*, *wait_for_active=False*)

Create a table that'll be used to store instances of cls.

See Amazon's developer guide for more information about provisioned throughput.

Parameters

- **cls** – The class whose instances will be stored in the table.
- **read_units** – The number of read units to provision for this table (minimum 5)
- **write_units** – The number of write units to provision for this table (minimum 5).
- **wait_for_active** – If True, create_table will wait for the table to become ACTIVE before returning (otherwise, it'll be CREATING). Note that this can take up to a minute. Defaults to False.

2.2.2 Model class

Class definition

class `dynamodb_mapper.model.`**`DynamoDBModel`**`(`***kwargs*`)`

Abstract base class for all models that use DynamoDB as their storage backend.

Each subclass must define the following attributes:

- `__table__`: the name of the table used for storage.
- `__hash_key__`: the name of the primary hash key.
- **`__range_key__`: (optional) if you're using a composite primary key,** the name of the range key.
- **`__schema__`: `{attribute_name: attribute_type}` mapping.** Supported attribute_types are: int, long, float, str, unicode, set. Default values are obtained by calling the type with no args (so 0 for numbers, "" for strings and empty sets).
- **`__defaults__`: (optional) `{attribute_name: defaulter}` mapping.** This dict allows to provide a default value for each attribute_name at object creation time. It will *never* be used when loading from the DB. It is fully optional. If no value is supplied the empty value corresponding to the type will be used. "defaulter" may either be a scalar value or a callable with no arguments.

To redefine serialization/deserialization semantics (e.g. to have more complex schemas, like auto-serialized JSON data structures), override the from_dict (deserialization) and to_db_dict (serialization) methods.

Important implementation note regarding sets: DynamoDB can't store empty sets/strings. Therefore, since we have schema information available to us, we're storing empty sets/strings as missing attributes in DynamoDB, and converting back and forth based on the schema.

So if your schema looks like the following:

```
{
    "id": unicode,
    "name": str,
    "cheats": set
}
```

then:

```
{
    "id": "e1m1",
    "name": "Hangar",
    "cheats": set([
        "idkfa",
        "iddqd"
    ])
}
```

will be stored exactly as is, but:

```
{
    "id": "e1m2",
    "name": "",
    "cheats": set()
}
```

will be stored as simply:

```
{
    "id": "e1m2"
}
```

Constructors

__init__

DynamoDBModel.**__init__**(***kwargs*)

Create an instance of the model. All fields defined in the schema are created. By order of prioritym its value will be loaded from:

- kwargs
- __defaults__
- mapper's default (0, empty string, empty set, ...)

We're supplying this method to avoid the need for extra checks in save and ease object initial creation.

from_dict

classmethod DynamoDBModel.**from_dict**(*d*)

Build an instance from a dict-like mapping, according to the class's schema.

Default values are used for anything that's missing from the dict (see DynamoDBModel class docstring).

Data access

get

classmethod DynamoDBModel.**get**(*hash_key_value*, *range_key_value=None*, *consistent_read=False*)

Retrieve a single object from DynamoDB according to its primary key.

Note that this is not a query method – it will only return the object matching the exact primary key provided. Meaning that if the table is using a composite primary key, you need to specify both the hash and range key values.

Parameters

- **hash_key_value** – The value of the requested item's hash_key.
- **range_key_value** – The value of the requested item's range_key, if the table has a composite key.
- **consistent_read** – If False (default), an eventually consistent read is performed. Set to True for strongly consistent reads.

get_batch

classmethod DynamoDBModel.**get_batch**(*keys*)

Retrieve multiple objects according to their primary keys.

Like get, this isn't a query method – you need to provide the exact primary key(s) for each object you want to retrieve:

- If the primary keys are hash keys, keys must be a list of their values (e.g. `[1, 2, 3, 4]`).
- If the primary keys are composite (hash + range), keys must be a list of `(hash_key, range_key)` values (e.g. `[("user1", 1), ("user1", 2), ("user1", 3)]`).

get_batch *always* performs eventually consistent reads.

Please not that a batch can *not* read more than 100 items at once.

Parameters keys – iterable of keys. ex `[(hash1, range1), (hash2, range2)]`

query

classmethod `DynamoDBModel.`**`query`**(*hash_key_value*, *range_key_condition=None*, *consistent_read=False*, *reverse=False*, *limit=None*)

Query DynamoDB for items matching the requested key criteria.

You need to supply an exact hash key value, and optionally, conditions on the range key. If no such conditions are supplied, all items matching the hash key value will be returned.

This method can only be used on tables with composite (hash + range) primary keys – since the exact hash key value is mandatory, on tables with hash-only primary keys, cls.get(k) does the same thing cls.query(k) would.

Parameters

- **hash_key_value** – The hash key's value for all requested items.
- **range_key_condition** – A condition instance from `boto.dynamodb.condition` – one of
  - EQ(x)
  - LE(x)
  - LT(x)
  - GE(x)
  - GT(x)
  - BEGINS_WITH(x)
  - BETWEEN(x, y)
- **consistent_read** – If False (default), an eventually consistent read is performed. Set to True for strongly consistent reads.
- **reverse** – Ask DynamoDB to scan the `range_key` in the reverse order. For example, if you use dates here, the more recent element will be returned first. Defaults to `False`.
- **limit** – Specify the maximum number of items to read from the table. Even though Boto returns a generator, it works by batchs of 1MB. using this option may help to spare some read credits. Defaults to `None`

Return type generator

scan

classmethod `DynamoDBModel.`**`scan`**(*scan_filter=None*)

Scan DynamoDB for items matching the requested criteria.

You can scan based on any attribute and any criteria (including multiple criteria on multiple attributes), not just the primary keys.

Scan is a very expensive operation – it doesn't use any indexes and will look through the entire table. As much as possible, you should avoid it.

Parameters scan_filter – A `{attribute_name: condition}` dict, where condition is a condition instance from `boto.dynamodb.condition`.

Return type generator

save

`DynamoDBModel.save(allow_overwrite=True, expected_values=None)`

Save the object to the database.

This method may be used both to insert a new object in the DB, or to update an existing one (iff `allow_overwrite == True`).

Parameters

- **allow_overwrite** – If False, the method will only succeed if this object's primary keys don't exist in the database (otherwise, `OverwriteError` is raised).
- **expected_values** – dict of expected attribute values – if any one of these values doesn't match what is in the database (i.e. someone went ahead and modified the object in the DB behind your back), the operation fails and raises `ExpectedValueError`.

delete

`DynamoDBModel.delete()`

Delete the current object from the database.

Data export

to_json_dict

`DynamoDBModel.to_json_dict()`

Return a dict representation of the object, suitable for JSON serialization.

This means the values must all be valid JSON object types (in particular, sets must be converted to lists), but types not suitable for DynamoDB (e.g. nested data structures) may be used.

Note that this method is never used for interaction with the database (`to_db_dict()` is).

to_db_dict

`DynamoDBModel.to_db_dict()`

Return a dict representation of the object according to the class's schema, suitable for direct storage in DynamoDB.

Auto-increment

class `dynamodb_mapper.model.autoincrement_int`

Dummy int subclass for use in your schemas.

If you're using this class as the type for your key in a hash_key-only table, new objects in your table will have an auto-incrementing primary key.

Note that you can still insert items with explicit values for your primary key – the autoincrementing scheme is only used for objects with unset hash_keys (or to be more precise, left set to the default value of 0).

Auto-incrementing int keys are implemented by storing a special "magic" item in the table with the following properties:

- `hash_key_value = -1`
- `__max_hash_key__ = N`

where N is the maximum used hash_key value.

Inserting a new item issues an atomic add on the '__max_hash_key__' value. Its new value is returned and used as the primary key for the new elem.

Note that hash_key_value is set to '-1' while `__max_hash_key__` initial value is 0. This will element at key '0' unused. It's actually a garbage item for cases where a value is manually added to an unitialized index.

2.2.3 Transactions class

Class definition

class `dynamodb_mapper.transactions.`**`Transaction`**(***kwargs*)

Abstract base class for transactions. A transaction may involve multiple targets and needs to be fully successful to be marked as "DONE".

This class gracefully handles concurrent modifications and auto-retries but embeds no tool to rollback.

Transactions may register `subtransactions`. This field is a list of `Transaction`. Sub-transactions are played after the main transactors

Transactions status may be persisted for tracability, further analysis... for this purpose, a minimal schema is embedded in this base class. When deriving, you MUST keep

- `datetime` field as rangekey
- `status` field

The hash key field may be changed to pick a ore relevant name or change its type. In any case, you are responsible of setting its value. For example, if collecting rewards for a player, you may wish to keep track of related transactions by user_id hence set requester_id to user_id

Deriving class **MUST** set field `__table__` and `requester_id` field

Public API

commit

`Transaction.`**`commit`**()

Run the transaction and, if needed, store its states to the database

- set up preconditions and parameters (`_setup()` – only called once no matter what).
- fetch all transaction steps (`_get_transactors()`).
- for each transaction :
  - fetch the target object from the DB.
  - modify the target object according to the transaction's parameters.
  - save the (modified) target to the DB
- run sub-transactions (if any)
- save the transaction to the DB

Each transation may be retried up to `MAX_RETRIES` times automatically. commit uses conditional writes to avoid overwriting data in the case of concurrent transactions on the same target (see `_retry()`).

save

`Transaction.`**`save`**(*allow_overwrite=True*, *expected_values=None*)

If the transaction is transient (`transient = True`), do nothing.

If the transaction is persistent (`transient = False`), save it to the DB, as `DynamoDBModel.save()`.

Note: this method is called automatically from `commit`. You may but do not need to call it explicitly.

Transactions interface

_setup

`Transaction._setup()`

Set up preconditions and parameters for the transaction.

This method is only run once, regardless of how many retries happen. You should override it to fetch all the *unchanging* information you need from the database to run the transaction (e.g. the cost of a Bingo card, or the contents of a reward).

_get_transactors

`Transaction._get_transactors()`

Fetch a list of targets (getter, setter) tuples. The transaction engine will walk the list. For each tuple, the getter and the setter are called successively until this step of the transaction succeed or exhaust the MAX_RETRIES.

- **getter: Fetch the object on which this transaction is supposed to operate** (e.g. a User instance for UserResourceTransactions) from the DB and return it. It is important that this method actually connect to the database and retrieve a clean, up-to-date version of the object – because it will be called repeatedly if conditional updates fail due to the target object having changed. The getter takes no argument and returns a DBModel instance
- **setter: Applyies the transaction to the target, modifying it in-place.** Does *not* attempt to save the target or the transaction to the DB. The setter takes a DBModel instance as argument. Its return value is ignored

The list is walked from 0 to len(transactors)-1. Depending on your application, Order may matter.

Raises TargetNotFoundError If the target doesn't exist in the DB.

_apply_subtransactions

`Transaction._apply_subtransactions()`

Run sub-transactions if applicable. This is called after the main transactors.

This code has been moved to its own method to ease overloading in real-world applications without re-implementing the whole `commit` logic.

This method should *not* be called directly. It may only be overloaded to handle special behaviors like callbacks.

2.3 Indices and tables

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

CHAPTER

THREE

CONTRIBUTE

Want to contribute, report a but of request a feature ? The development goes on at Ludia's BitBucket account:

- **Report bugs**: https://bitbucket.org/Ludia/dynamodb-mapper/issues
- **Fork the code**: https://bitbucket.org/Ludia/dynamodb-mapper/overview