dynamongo Documentation

Release 0.2

Musyoka Morris

Oct 06, 2018

Contents

Release v0.2

dynamongo is Python ORM/framework-agnostic library for DynamoDB. It is highly inspired by the PyMongo project. This documentation attempts to explain everything you need to know to use dynamongo.

```
import datetime
from dynamongo import Model, Connection
from dynamongo import IntField, StringField, ListField, EmailField, DateTimeField

# This only need be called once.
# Alternatively, it can be set using env variables
Connection.set_config(
    access_key_id='<KEY>',
    secret_access_key='<SECRET>',
    table_prefix='test-'
)


class User(Model):
    __table__ = 'users'
    __hash_key__ = 'email'

    email = EmailField(required=True)
    name = StringField(required=True)
    year_of_birth = IntField(max_value=2018, min_value=1900)
    cities_visited = ListField(StringField)
    created_at = DateTimeField(default=datetime.datetime.now)


# store data to DynamoDB
john = User.save_one({
    'email': 'johndoe@gmail.com',
    'name': 'John Doe',
    'year_of_birth': 1990,
    'cities_visited': ['Nairobi', 'New York']
})

# year_of_birth, cities_visited & created_at are all optional
jane = User.save_one({
    'email': 'jane@gmail.com',
    'name': 'Jane Doe'
})

# Access attribute values
print(john.name)

# Fetch data from dynamoDB
user = User.get_one(User.email == 'johndoe@gmail.com')
print(user.to_dict())
```

In short, dynamongo models can be used to easily:

- **validate** input data
- **save** serialized data to DynamoDB
- **read** and deserialize data from DynamoDB
- **delete** items from DynamoDB
- **update** data in DynamoDB

CHAPTER 1

Get It Now

```
$ pip install dynamongo
```

CHAPTER 2

Documentation

Full documentation is available at http://dynamongo.readthedocs.io/ .

CHAPTER 3

Requirements

- Python >= 3.5

CHAPTER 4

Guide

4.1 Installation

dynamongo requires Python >= 3.5.

4.1.1 Installing/Upgrading from the PyPI

To install the latest stable version from the PyPI:

```
$ pip install -U dynamongo
```

To install the latest pre-release version from the PyPI:

```
$ pip install -U dynamongo --pre
```

4.1.2 Install from source

If you'd rather install directly from the source (i.e. to stay on the bleeding edge), to get the latest development version of dynamongo, run

```
$ pip install -U git+https://github.com/musyoka-morris/dynamongo.git@dev
```

4.2 Quickstart

This guide will walk you through the basics of working with *DynamoDB* and *dynamongo*

4.2.1 Prerequisites

Before we start, make sure that you have an AWS access key id & AWS secret access key. If you don't have these keys yet, you can create them from the AWS Management Console by following this documentation.

4.2.2 Connection

Before making any calls, dynamongo needs to have access to AWS dynamoDB. Additionally, it is recommended each repository using this library should have a unique prefix for table names. AWS connection credentials and the table name prefix can be set in either of two ways:

1. **ENVIRONMENT VARIABLES** This is the recommended way of setting dynamongo connection. The env variables are

    - `AWS_ACCESS_KEY_ID` : Required
    - `AWS_SECRET_ACCESS_KEY` : Required
    - `AWS_REGION_NAME` : Optional, defaults to `us-east-2`
    - `AWS_TABLE_PREFIX` : Optional, defaults to `None`

2. **USING CONNECTION CLASS**

```
from dynamongo import Connection

Connection.set_config(
    access_key_id='<your aws access key id>',
    secret_access_key='<your aws secret access key>',
    region='<aws region name>',
    table_prefix='<table prefix of your choice>'
)
```

Any values set using this method override environment variables. This only need be called once, but it must be called before any attempt to make calls to DynamoDB.

Note: The `table_prefix` is more of a good practice than a feature. In DynamoDB, each customer is allocated a single database. It is highly recommended to prefix your tables with a name of the form `application-specific-name` to avoid table name collisions with other projects.

4.2.3 Declaring Models

Lets start with a basic user 'model'

```
import datetime
from dynamongo import Model
from dynamongo import IntField, StringField, ListField, EmailField, DateTimeField

class User(Model):
    __table__ = 'users'
    __hash_key__ = 'email'

    email = EmailField(required=True)
    name = StringField(required=True)
    year_of_birth = IntField(max_value=2018, min_value=1900)
```

(continues on next page)

(continued from previous page)

```
    cities_visited = ListField(StringField)
    created_at = DateTimeField(default=datetime.datetime.now)
```

Every model must declare the following attributes:

```
__table__: The name of the table

__hash_key__: Hash key for the table
```

and at least one field for the Hash key. See *Model* for detailed documentation on the allowed Model attributes

4.2.4 Creating the table

Unlike other NoSQL engines like MongoDB, tables must be created and managed explicitly. At the moment, dynamongo abstracts only the initial table creation. Other lifecycle management operations may be done directly via Boto3.

To create the table, use *create_table()*. The throughput provisioned for this table is determined by the attributes `__read_units__` & `__write_units__`. These are optional and they default to `8`.

Note: Unlike most databases, table creation may take up to 1 minute.

For more information, please see Amazon's official documentation.

4.2.5 Saving data

Saving single item

Saving a single item can be done by calling *save_one()* method. item to be saved is passed as a `dict` or an instance of *Model*.

By default, if an item that has the same primary key as the new item already exists, the new item completely replaces the existing item.

You can override this behaviour by passing `overwrite=False`. In this case, if an item that has the same primary key as the new item already exists, a *ConditionalCheckFailedException* exception is raised. Otherwise, the item is saved.

Example using a *dict* object

```
john = User.save_one({
    'email': 'johndoe@gmail.com',
    'name': 'John Doe',
    'year_of_birth': 1990,
    'cities_visited': ['Nairobi', 'New York']
})
```

Example using a *Model* instance

```
user = User(
    email='johndoe@gmail.com',
    name='John Doe',
    cities_visited=[]
```

(continues on next page)

(continued from previous page)

```
)
user.year_of_birth = 1990
user.cities_visited = ['Nairobi', 'New York']
user = User.save_one(user)
```

Saving multiple items

Multiple items can be saved by calling *save_many()* method. This method takes as input a `list` of:

- `dict` objects, or
- *Model* instances, or
- mixture of both `dict` objects and *Model* instances

This method returns an *BatchResult* instance.

By default, existing items are completely replaced by new items. passing `overwrite=False` changes the default behaviour, and items which could not be created since an item already exists with the same primary key, are considered `failed`.

```
user_list = [
    # first user. defined as a dict
    {
        'email': 'johndoe@gmail.com',
        'name': 'John Doe',
        'year_of_birth': 1990,
        'cities_visited': ['Nairobi', 'New York']
    },

    # second user. User instance
    User(
        email='johndoe@gmail.com',
        name='John Doe',
        cities_visited=[]
    )
]

result = User.save_many(user_list, overwrite=False)
print(result.fail_count)
```

4.2.6 Deleting Data

Just as with saving data, you can delete a single item or many items at once.

Deleting a single item

Deleting a single item can be done by calling *delete_one()* method. If an item by the given strategy exists, it is deleted and the deleted item is returned. Otherwise `None` is returned.

This method takes in `strategy` as input. `strategy` can be either of the following:

1. **The primary key value**.

If a model has a *hash_key* only, this is passed in as a scalar. Otherwise, if the model has both *hash_key* and *range_key*, the value is passed as a `(hash_key, range_key)` tuple.

```
user = User.delete_one('johndoe@gmail.com')
```

2. **Dict object**

The dict should contain all primary key values. i.e, if the model has both *hash_key* and *range_key*, both should be included in the dict. Otherwise only a dict with the `hash_key` is required.

Non primary key items in the dict are ignored.

```
user = User.delete_one({'email': 'johndoe@gmail.com'})
```

3. **Model instance**

The primary fields attributes must have valid values. Item is deleted by the primary keys.

```
user = User.delete_one(User(email='johndoe@gmail.com'))
```

4. **Key condition**

In its simplest form, if the model does not have a *range_key*, this should be an equality condition on the hash_key field.

if the model has both *hash_key* and *range_key*, this should be two equality conditions on both key fields *ANDed* together.

```
user = User.delete_one(User.email == 'johndoe@gmail.com')
```

5. **Key condition + additional checks**

This allows one to delete an item based on the primary key, but with an additional check.

Example #1. Suppose we want to delete a user whose primary key `email=johndoe@gmail.com`, but only if the user was born on or before the year `2000`

```
user = User.delete_one(
    (User.email == 'johndoe@gmail.com') & (User.year_of_birth <= 2000)
)
```

Example #2. Delete a user whose `email=johndoe@gmail.com` if the user has already visited `Nairobi` city

```
user = User.delete_one(
    (User.email == 'johndoe@gmail.com') & User.cities_visited.contains('Nairobi')
)
```

Example #3. This can become even more complex. Delete a user whose `email=johndoe@gmail.com` AND the user was born after `2000` or the user has already visited `Nairobi` city

```
user = User.delete_one(
    (User.email == 'johndoe@gmail.com') &
    ((User.year_of_birth > 2000) | User.cities_visited.contains('Nairobi'))
)
```

In all cases, equality conditions for the primary keys **must** be present in the condition. All other conditional checks **must** be *ANDed* to the primary key conditions. This rule is strictly enforced by both *dynamongo* and *DynamoDB*. For example, the following strategy would fail:

```
# This raises an ExpressionError. The condition is ORed instead of being ANDed
user = User.delete_one(
    (User.email == 'johndoe@gmail.com') | (User.year_of_birth > 2000))
)
```

Deleting multiple items

Multiple items can be deleted by calling Model.delete_many method. This method takes in `strategy` as input. `strategy` can be either of the following:

1. **List**

Each entry in this list must be a valid object that can be passed to the `delete_one()` method as described above.

Examples

```
result = User.delete_many([
    'johndoe@gmail.com',
    'email1@gmail.com',
    {'email': 'email2@gmail.com'},
    User(email='email3@gmail.com'),
    User.email == 'email4@gmail.com',
    (User.email == 'email5@gmail.com') & (User.year_of_birth <= 2000)
])
```

2. **Condition**

Here you can pass any valid condition. Suppose we have list of user emails:

```
emails = ['johndoe@gmail.com', 'email2@abc.io', 'anotherone@xyz.com']
```

Example #1. Delete those users unconditionally. It can be achieved in either of the following ways

```
# simply passing in the list of emails
result = User.delete_many(emails)
```

```
# more control. We know exactly what emails is
result = User.delete_many(User.email.in_(emails))
```

```
# Useful when using composite primary keys
result = User.delete_many(User.keys_in(emails))
```

Example #2. Only delete users in the list, but only if the user was born on or before the year `2000`

```
result = User.delete_many(
            (User.email.in_(emails)) &
            (User.year_of_birth > 2000)
        )
```

Example #3. Delete all users who have ever visited `Nairobi` city

```
result = User.delete_many(User.cities_visited.contains('Nairobi'))
```

Example #4. Delete any user who was born before `1990` and has never visited `Nairobi`. *(we do not need boring people in our system)*

```
result = User.delete_many(
            (User.year_of_birth < 1990) &
            (not User.cities_visited.contains('Nairobi'))
        )
```

4.2.7 Accessing data

dynamongo supports retrieval of a single item or many items at once.

Getting a single item

Getting a single item can be done by calling *get_one()* method. This method raises `Exception` if an item by the given strategy does not exists.

This method takes in `strategy` as input. `strategy` can be either of the following:

1. **The primary key value**.

If a model has a *hash_key* only, this is passed in as a scalar. Otherwise, if the model has both *hash_key* and *range_key*, the value is passed as a `(hash_key, range_key)` tuple.

```
user = User.get_one('johndoe@gmail.com')
```

2. **Dict object**

The dict should contain all primary key values. i.e, if the model has both *hash_key* and *range_key*, both should be included in the dict. Otherwise only a dict with the `hash_key` is required.

Non primary key items in the dict are ignored.

```
user = User.get_one({'email': 'johndoe@gmail.com'})
```

3. **Model instance**

The primary fields attributes must have valid values. Item is selected by the primary keys.

```
user = User.get_one(User(email='johndoe@gmail.com'))
```

4. **Key condition**

In its simplest form, if the model does not have a *range_key*, this should be an equality condition on the hash_key field.

if the model has both *hash_key* and *range_key*, this should be two equality conditions on both key fields *ANDed* together.

```
user = User.get_one(User.email == 'johndoe@gmail.com')
```

Getting multiple items

Multiple items can be fetched by calling *get_many()* method. This method takes in `strategy` as input. `strategy` can be either of the following:

1. **List**

Each entry in this list must be a valid object that can be passed to the *get_one()* method as described above.

Examples

```
users = User.get_many([
    'johndoe@gmail.com',
    'email1@gmail.com',
    {'email': 'email2@gmail.com'},
```

(continues on next page)

(continued from previous page)

```
    User(email='email3@gmail.com'),
    User.email == 'email4@gmail.com'
])
```

2. **Condition**

Here you can pass any valid condition. Suppose we have list of user emails:

```
emails = ['johndoe@gmail.com', 'email2@abc.io', 'anotherone@xyz.com']
```

Example #1. Finding users by their email address, can be achieved in either of the following ways

```
# simply passing in the list of emails
users = User.get_many(emails)
```

```
# more control. We know exactly what emails is
users = User.get_many(User.email.in_(emails))
```

```
# Useful when using composite primary keys
users = User.get_many(User.keys_in(emails))
```

Example #2. Only get users in the list, but only if the user was born on or before the year `2000`

```
users = User.get_many(
            (User.email.in_(emails)) &
            (User.year_of_birth > 2000)
        )
```

Example #3. Get all users who have ever visited `Nairobi` city

```
users = User.get_many(User.cities_visited.contains('Nairobi'))
```

Example #4. Get all user who were born before `1990` and have never visited `Nairobi`.

```
users = User.get_many(
            (User.year_of_birth < 1990) &
            (not User.cities_visited.contains('Nairobi'))
        )
```

CHAPTER 5

API Reference

5.1 API Reference

5.1.1 Utility Methods

dynamongo.utils.**is_empty**(*value*)

Determine if a value is empty.

A value is considered empty if it is `None` or empty string `""`

dynamongo.utils.**non_empty_values**(*d*)

Return a dict with empty values removed recursively

dynamongo.utils.**merge_deep**(*destination*, *source*)

Merge dict objects recursively

dynamongo.utils.**is_subclass**(*value*, *class_*)

Check if value is a sub class of **class_**

dynamongo.utils.**key_proto**(*attr*)

Return associated DynamoDB attribute type

5.1.2 Connection

Connection borg

class dynamongo.connection.**Connection**(*access_key_id=None*, *secret_access_key=None*, *region=None*, *table_prefix=None*)

Borg that handles access to DynamoDB.

You should never make any explicit/direct `boto3.dynamodb` calls by yourself except for table maintenance operations

Before making any calls, aws credentials must be set by either:

1. calling `set_config()`, or
2. setting environment variables
   - `AWS_ACCESS_KEY_ID`
   - `AWS_SECRET_ACCESS_KEY`
   - `AWS_REGION_NAME`
   - `AWS_TABLE_PREFIX`

classmethod from_env()
Read config from the env

client()
Return the DynamoDB client

resource()
Return DynamoDB Resource

get_table(*name*)
Return DynamoDB Table object

5.1.3 Model

class dynamongo.model.**Model**(***kwargs*)
Base model class with which to define custom models.

Example usage:

```python
from dynamongo import Model
from dynamongo import IntField, StringField, EmailField

class User(Model):
    __table__ = 'users'
    __hash_key__ = 'email'

    # fields
    email = EmailField(required=True)
    name = StringField(required=True)
    year_of_birth = IntField(max_value=2018, min_value=1900)
```

Each custom model can declare the following class meta data variables:

__table__ *(required)*

The name of table to be associated with this model. This is usually prefixed with the table prefix as set in *Connection*. i.e, in dynamodb, the table name will appear as `<table_prefix><table_name>`

__hash_key__ *(required)*

The name of the field to be used as the Hash key for the table. **NOTE**: A field for the hash key **MUST** be declared and it must be of primitive type `str|numeric`

__range_key__ *(optional)*

The name of the field to be used as the Range key for the table. **NOTE**: This is Optional. However, if declared, a corresponding field **MUST** be declared and it must be of primitive type `str|numeric`

__read_units__ *(optional)*

The number of read units to provision for this table (default `8`)

__write_units__ *(optional)*

The number of write units to provision for this table (default `8`)

See Amazon's developer guide for more information about provisioned throughput Capacity for Reads and Writes

classmethod keys_in(*values*)

Convenient method to generate `CompoundKeyCondition`

This is useful when working with a model that has a composite primary key i.e, both `hash_key` and `range_key`

Example usage:

```
import datetime
from dynamongo import Model
from dynamongo import EmailField, UUIDField, DateTimeField


class Contacts(Model):
    __table__ = 'user-contacts'
    __hash_key__ = 'user_id'
    __range_key__ = 'email'

    # fields
    user_id = UUIDField(required=True)
    email = EmailField(required=True)
    created_at = DateTimeField(default=datetime.datetime.now)


# select multiple contacts for different users when you have a
# list of (user_id, email) tuples
keys = [('user_id_1', 'john@gmail.com'), ('user_id_2', 'doe@gmail.com')]
contacts = Contacts.get_many(
    Contacts.keys_in(keys)
)
```

classmethod table_name()

Get prefixed table name

classmethod table()

Get a dynamoDB Table instance for this model

classmethod create_table()

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

This operation should be called before any table read or write operation is undertaken

classmethod get_one(*strategy*)

Retrieve a single item from DynamoDB according to strategy.

See *Getting a single item*

Returns Instance of `cls` - The fetched item

classmethod get_many(*strategy*, *descending=False*, *limit=None*)

Retrieve a multiple items from DynamoDB according to strategy.

Performs either a BatchGet, Query, or Scan depending on strategy

See *Getting multiple items*

Parameters

- **strategy** – See *Getting multiple items*
- **descending** (*bool*) – Sort order. Items are sorted by the hash key. Items with the same hash key value are sorted by range key
- **limit** (*int*) – The maximum number of items to get (not necessarily the number of items returned)

Returns list of `cls`

classmethod delete_one(*strategy*)

Deletes a single item in a table. You can perform a conditional delete operation that deletes the item if it exists, or if it has an expected attribute value.

see *Deleting a single item*

Returns The deleted item

classmethod delete_many(*strategy*)

Deletes multiple items in a table.

see *Deleting multiple items*

Returns *BatchResult*

classmethod save_one(*item*, *overwrite=True*)

Creates a new item, or replaces an old item with a new item. If an item that has the same primary key as the new item already exists in the specified table, the new item completely replaces the existing item `overwrite` specifies under what circumstances should we overwrite an existing item.

If `overwrite = True`, an existing item with the same primary key is replaced by the new item unconditionally. This is the default behaviour.

If `overwrite = False`, a *ConditionalCheckFailedException* is raised if there is an existing item with the same primary key

If `overwrite` is a conditional expression, an existing item with the same primary key is replaced by the new item if and only if the condition is met. otherwise *ConditionalCheckFailedException* is raised.

see *Saving single item*

Parameters

- **item** – the item to save. either a `dict` or `cls`
- **overwrite** – This can be a `bool` or a condition. it defaults to `True`

Raises *ConditionalCheckFailedException*

Returns cls

classmethod save_many(*items*, *overwrite=True*)

Creates or replaces multiple items. If an item that has the same primary key as the new item already exists in the specified table, the new item completely replaces the existing item `overwrite` specifies under what circumstances should we overwrite an existing item.

If `overwrite = True`, an existing item with the same primary key is replaced by the new item unconditionally. This is the default behaviour.

If `overwrite = False` and there is an existing item with the same primary key, the item is added on `BatchResult.fail` list

If `overwrite` is a conditional expression and an existing item with the same primary key does not meet the condition specified, then the item is added on `BatchResult.fail` list.

see *Saving multiple items*

Parameters

- **`items`** – a list of items to save. each item can be either a `dict` or `cls`
- **`overwrite`** – `bool` or a condition. it defaults to `True`

Returns *`BatchResult`*

`classmethod update_from_dict`(*item*)

Updates an item if and only if it exists in the db

item primary keys must be provided.

Parameters **`item`** (*`dict`*) –

Returns updated item

`classmethod update_one`(*strategy*, *updates*)

Update all items in the db that satisfy condition

updates are: 'ADD'|'PUT'|'DELETE'

Parameters

- **`strategy`** – Single item selection strategy
- **`updates`** – list[Update]

Returns List of updated items

`class` `dynamongo.model.`**`BatchResult`**(*fail=[]*, *success=[]*)

Batch result class

5.1.4 Fields

Field classes for various types of data.

`class` `dynamongo.fields.`**`Field`**

Basic field from which other fields should extend. It applies no formatting by default, and should only be used in cases where data does not need to be serialized or deserialized.

Supported primitive conditions are ==, !=, <, <=, >, and >=

`set_name`(*name*, *parent=None*)

Set name

schema names should start with a alphabetic character

`in_`(*value*)

Creates a condition where the attribute is in the value,

Parameters **`value`** (*`list`*) – The list of values that the attribute is in.

`contains`(*value*)

Creates a condition where the attribute contains the value.

Parameters **`value`** – The value the attribute contains.

`begins_with`(*value*)

Creates a condition where the attribute begins with the value.

Parameters **value** – The value that the attribute begins with.

exists()
Creates a condition where the attribute exists.

not_exists()
Creates a condition where the attribute does not exist.

between(*low*, *high*)
Creates a condition where the attribute is greater than or equal to the low value and less than or equal to the high value.

Parameters

- **low** – The value that the attribute is greater than or equal to.
- **high** – The value that the attribute is less than or equal to.

set(*value*)
Set field to the given value if it does not exist otherwise update

set_if_not_exists(*value*)
Set field to the given value if it does not exist otherwise do nothing

remove()
Remove field

default
Get the default value

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to store in DynamoDB.

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

class dynamongo.fields.**IntField**(***kwargs*)
A field that validates input as an Integer

See Schematics IntType

class dynamongo.fields.**FloatField**(***kwargs*)
A field that validates input as a Float

See Schematics FloatType

class dynamongo.fields.**BooleanField**(***kwargs*)
A boolean field

See Schematics BooleanType

class dynamongo.fields.**StringField**(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A Unicode string field.

See Schematics StringType

class dynamongo.fields.**EmailField**(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that validates input as an E-Mail-Address

See Schematics EmailType

class dynamongo.fields.**URLField**(*fqdn=True*, *verify_exists=False*, ***kwargs*)
A field that validates the input as a URL.

See Schematics URLType

class `dynamongo.fields.`**`UUIDField`**(***kwargs*)

A field that stores a valid UUID value.

See Schematics UUIDType

class `dynamongo.fields.`**`IPAddressField`**(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)

A field that stores a valid IPv4 or IPv6 address.

See Schematics IPAddressType

class `dynamongo.fields.`**`DateTimeField`**(*formats=None*, *serialized_format=None*, *parser=None*, *tzd='allow'*, *convert_tz=False*, *drop_tzinfo=False*, ***kwargs*)

A field that holds a combined date and time value.

See Schematics DateTimeType

class `dynamongo.fields.`**`DateField`**(*formats=None*, ***kwargs*)

A field that stores and validates date values.

See Schematics DateType

class `dynamongo.fields.`**`TimedeltaField`**(*precision='seconds'*, ***kwargs*)

A field that stores and validates timedelta value

See Schematics TimedeltaType

class `dynamongo.fields.`**`ListField`**(*field*, *default=[]*, ***kwargs*)

A field for storing a list of items, all of which must conform to the type specified by the `field` parameter.

See Schematics ListType

Note: This field cannot be set to `None`

`append`(**values*)

Append one or more values at the end of the list

`prepend`(**values*)

Prepend one or more values at the start of the list

class `dynamongo.fields.`**`DictField`**(***fields*)

A field that stores dict values.

Accepts named parameters which must be instances of *`Field`*

`primitive_type`

alias of `builtins.dict`

`native_type`

alias of `builtins.dict`

`set_name`(*name*, *parent=None*)

Set name

schema names should start with a alphabetic character

`default`

Get the default value

`to_native`(*value*, *context=None*)

Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to store in DynamoDB.

5.1.5 Exceptions

exception dynamongo.exceptions.**ValidationError**(**args*, ***kwargs*)
Exception raised when invalid data is encountered.

exception dynamongo.exceptions.**ConditionalCheckFailedException**
Raised when saving a Model instance would overwrite something in the database and we’ve forbidden that

exception dynamongo.exceptions.**ExpressionError**(*msg*, *expression*)
raised if some expression rules are violated

exception dynamongo.exceptions.**SchemaError**(*msg=”*, *name=None*, *value=None*)
SchemaError exception is raised when a schema consistency check fails.

Common consistency failure includes:

- lacks of `__table__` or `__hash_key__` definitions
- lack of corresponding field definitions for the primary keys
- When an invalid field type is used in `DictField` or `ListField`

5.1.6 Conditional Expressions

Note: These classes should never be instantiated directly by the user

class dynamongo.conditions.**OP**

class dynamongo.conditions.**BaseCondition**
Base class for all expressions

class dynamongo.conditions.**JoinCondition**(*left*, *right*)
Base class for joiner expressions

class dynamongo.conditions.**AndCondition**(*left*, *right*)
Initialized by ANDing two expressions i.e, BaseCondition & BaseCondition

class dynamongo.conditions.**OrCondition**(*left*, *right*)
Initialized by ORing two expressions i.e, BaseCondition | BaseCondition

class dynamongo.conditions.**PrimitiveCondition**(*attr*, *op*, *value=None*)
Primitive expression

5.1.7 Update Expressions

Note: These classes should never be instantiated directly by the user

class dynamongo.updates.**UpdateBuilder**(*type_*, *expression*, *values*)
Update expression builder

classmethod create(*updates*)
Prepares update-expression & expression-attribute-values

Parameters **updates** (*list[Update]|tuple(Update)*) – list of updates to be performed

Returns a tuple (update-expression, expression-attribute-values)

class dynamongo.updates.**Update**(*field*, *value=None*)
Base abstract class for update expressions

value
validated value

static placeholder()
Generate a unique placeholder string

class dynamongo.updates.**RemoveUpdate**(*field*)
Update to remove attributes from the db

class dynamongo.updates.**SetUpdate**(*field*, *value*, *if_not_exists=False*)
Update to set an attribute to the given value

class dynamongo.updates.**ListExtendUpdate**(*field*, *value*, *append=True*)
Update to append or prepend values to a list

class dynamongo.updates.**AddUpdate**(*field*, *value=None*)
Update to perform an addition to a numeric value

Python Module Index

Index