Python DynamoDB Lock Documentation

Release 0.9.3

Mohan Kishore

Jul 14, 2020

Contents:

CHAPTER 1

Python DynamoDB Lock

This is a fork of the currently unmaintained (2 years) of Python DynamoDB Lock project. In the spirit of open-source whatnick is maintaining this while there is some time. Any enhancements targeting this project can be sent here.

This is a general purpose distributed locking library built on top of DynamoDB. It is heavily "inspired" by the java-based AmazonDynamoDBLockClient library, and supports both coarse-grained and fine-grained locking.

- Free software: Apache Software License 2.0
- Documentation: https://python-dynamodb-lock-whatnick.readthedocs.io
- Source Code: https://github.com/whatnick/python_dynamodb_lock

1.1 Features

- Acquire named locks - with configurable retry semantics
- Periodic heartbeat/update for the locks to keep them alive
- Auto-release the locks if there is no heartbeat for a configurable lease-duration
- Notify an app-callback function if the lock is stolen, or gets too close to lease expiry
- Store arbitrary application data along with the locks
- Uses monotonically increasing clock to avoid issues due to clock skew and/or DST etc.
- Auto-delete the database entries after a configurable expiry-period

1.2 Consistency Notes

Note that while the lock itself can offer fairly strong consistency guarantees, it does NOT participate in any kind of distributed transaction.

For example, you may wish to acquire a lock for some customer-id "xyz", and then make some changes to the corresponding database entry for this customer-id, and then release the lock - thereby guaranteeing that only one process changes any given customer-id at a time.

While the happy path looks okay, consider a case where the application changes take a long time, and some errors/gc-pauses prevent the heartbeat from updating the lock. Then, some other client can assume the lock to be abandoned, and start processing the same customer in parallel. The original lock-client will recognize that its lock has been "stolen" and will let the app know through a callback event, but the app may have already committed its changes to the database. This can only be solved by having the application changes and the lock-release be part of a single distributed transaction - which, as indicated earlier, is NOT supported.

That said, in most cases, where the heartbeat is not expected to get delayed beyond the lock's lease duration, the implementation should work just fine.

Refer to an excellent post by Martin Kleppmann on this subject: https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

1.3 Credits

- AmazonDynamoDBLockClient: https://github.com/awslabs/dynamodb-lock-client
- Cookiecutter: https://github.com/audreyr/cookiecutter
- Cookiecutter Python: https://github.com/audreyr/cookiecutter-pypackage

CHAPTER 2

Installation

2.1 Stable release

To install Python DynamoDB Lock, run this command in your terminal:

```
$ pip install python_dynamodb_lock
```

This is the preferred method to install Python DynamoDB Lock, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Python DynamoDB Lock can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/whatnick/python_dynamodb_lock
```

Or download the tarball:

```
$ curl  -OL https://github.com/whatnick/python_dynamodb_lock/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Python DynamoDB Lock in a project:

```
from python_dynamodb_lock.python_dynamodb_lock import *
```

3.1 Basic Usage

You would typically create (and shutdown) the DynamoDBLockClient at the application startup and shutdown:

```
# get a reference to the DynamoDB resource
dynamodb_resource = boto3.resource('dynamodb')

# create the lock-client
lock_client = DynamoDBLockClient(dynamodb_resource)

...

# close the lock_client
lock_client.close()
```

Then, you would wrap the lock acquisition and release around the code-block that needs to be protected by a mutex:

```
# acquire the lock
lock = lock_client.acquire_lock('my_key')

# ... app logic that requires the lock ...

# release the lock after you are done
lock.release()
```

Both the lock_client constructor and the acquire_lock method support numerous arguments to help control/customize the behavior. Please look at the *API documentation* for more details.

3.2 Context Management

The DynamoDBLock class implements the context-management interface and you can auto-release the lock by doing something like this:

```
with lock_client.acquire_lock('my_key'):
    # ... app logic that requires the lock ...
```

3.3 Table Creation

The DynamoDBLockClient provides a helper class-method to create the table in DynamoDB:

```
# get a reference to the DynamoDB client
ddb_client = boto3.client('dynamodb')

# create the table
DynamoDBLockClient.create_dynamodb_table(ddb_client)
```

The above code snippet will create a table with the default name, partition/sort-key column-names, read/write throughput, but the method supports optional parameters to configure all of these.

That said, you can always create the table offline (e.g. using the AWS console) and use whatever table and column names you wish. Please do remember to setup the TTL attribute to enable auto-deleting of old/abandoned locks.

3.4 Error-Handling

There are a lot of things that can go wrong when dealing with distributed systems - the library tries to strike the right balance between hiding these errors, and allowing the library to handle specific kinds of errors as needed. Let's go through the different use-cases one at a time.

3.4.1 Lock Acquisition

This is a synchronous use-case where the caller is waiting till it receives a lock. In this case, most of the errors are wrapped inside a DynamoDBError and raised up to the caller. The key error scenarios are the following:

- **Some other client holds the lock**
  - This is not treated as real error scenario. This client would just wait for a configurable retry_period, and then try to acquire the lock again.
- **Race-condition amongst multiple lock-clients waiting to acquire lock**
  - Whenever the "old" lock is released (or expires), there may be multiple "new" clients trying to grab the lock - in which case, one of those would succeed, and the rest of them would get a DynamoDB's ConditionalUpdateException. This is also not treated as a real error scenario, and the client would just wait for the retry_period and then try again.
- **This client goes over the configurable retry_timeout period**
  - After repeated retry attempts, this client might eventually go over the retry_timeout period (as provided by the caller) - then, a DynamoDBLockError with code == ACQUIRE_TIMEOUT will be thrown.
- **Any other error/exception**

  - Any other error would be wrapped inside a DynamoDBLockError with code == UNKNOWN_ERROR and raised to the caller.

3.4.2 Lock Release

While this is also a synchronous use-case, in most cases, by the time this method is called, the caller would have already committed his application-data changes, and would not have real rollback options. Therefore, this method defaults to the best_effort mode, where it will try to release the lock properly, but will log and swallow any exceptions encountered in the process. But, for the callers that are interested in being notified of the errors, they can pass in best_effort=False and have all the errors wrapped inside a DynamoDBLockError and raised up to them. The specific error scenarios could be one of the below:

- **This client does not own the lock**
  - This can happen if the caller tries to use this client to release a lock owned by some other client. The client will raise a DynamoDBLockError with code == LOCK_NOT_OWNED.
- **The lock was stolen by some other client**
  - This should typically not happen unless someone messes with the back-end DynamoDB table directly. The client will raise a DynamoDBLockError with code == LOCK_STOLEN.
- **Any other error/exception**
  - Any other error would be wrapped inside a DynamoDBLockError with code == UNKNOWN_ERROR and raised to the caller.

3.4.3 Lock Heartbeat

This is an asynchronous use-case, where the caller is not directly available to handle any errors. To handle any error scenarios encountered while sending a heartbeat for a given lock, the client allows the caller to pass in an app_callback function at the time of acquiring the lock.

- **The lock was stolen by some other client**
  - This should typically not happen unless someone messes with the back-end DynamoDB table directly. The client will call the app_callback with code == LOCK_STOLEN. The callback is expected to terminate the related application processing and rollback any changes made under this lock's protection.
- **The lock has entered the danger zone**
  - If the send_heartbeat call for a given lock fails multiple times, the lock could go over the configurable safe_period. The client will call the app_callback with code == LOCK_IN_DANGER. The callback is expected to complete/terminate the related application processing, and call the lock.release() as soon as possible.

Note: it is worth noting that the client spins up two separate threads - one to send out the heartbeats, and another one to check the lock-statuses. For whatever reason, if the send_heartbeat calls start hanging or taking too long, the other thread will allow the client to notify the app about the locks getting into the danger-zone. The actual app_callbacks are executed on a dedicated ThreadPoolExecutor.

3.4.4 Client Close

By default, the lock_client.close() will NOT release all the locks - as releasing the locks prematurely while the application is still making changes assuming that it has the lock can be dangerous. As soon as a lock is released by this client, some other client may pick it up, and the associated app may start processing the underlying business entity in parallel.

It is highly recommended that the application manage its shutdown-lifecycle such that all the worker threads operating under these locks are first terminated (committed or rolled-back), the corresponding locks released (one at a time - by each worker thread), and then the lock_client.close() method is called. Alternatively, consider letting the process die without releasing all the locks - they will be auto-released when their lease runs out after a while.

That said, if the caller does wish to release all locks when closing the lock_client, it can pass in release_locks=True argument when invoking the close() method. Please note that all the locks are released in the best_effort mode - i.e. all the errors will be logged and swallowed.

3.4.5 Process Termination

A sudden process termination would leave the locks frozen with the values as of their last heartbeat. These locks will go through one of the following scenarios:

- **Eventual expiry - as per the TTL attribute**
  - Each lock has a TTL attribute (named 'expiry_time' by default) - which stores the timestamp (as epoch) after which it is eligible for auto-deletion by DynamoDB. This deletion does not have a fixed SLA - but will likley happen over the next 24 hours after the lock expires.
- **Some other client tries to acquire the lock**
  - The client will treat the lock as an active lock - and will wait for a period equal to its lease_duration from the point it first sees the lock. This does need the acquire_lock call to be made with a retry_period larger than the lease_duration of the lock - otherwise, the acquire_lock call will timeout before the lease expires.

3.5 Throughput Provisioning

Whenever using DynamoDB, you have to think about how much read and write throughput you need to provision for your table. The DynamoDBLockClient makes the following calls to DynamoDB:

- **acquire_lock**
  - `get_item`: at least once per lock, and more often if there is lock contention and the lock_client needs to retry multiple times before acquiring the lock.
  - `put_item`: typically once per lock - whenever the lock becomes available.
  - `update_item`: should be fairly rare - only needed when this client needs to take over an abandoned lock.
  - So, the write throughput should be directly proportional to the applications need to acquire locks, but the read throughput is a little harder to predict - it can be more sensitive to the lock contention at runtime.
- **release_lock**
  - `delete_item`: once per lock
  - So, assuming that every lock that is acquired will be released, this is also directly proportional to the application's lock acquition TPS.
- **send_heartbeat**
  - `update_item`: the lock client supports a deterministic model where the caller can pass in a TPS value, and the client will honor the same when making the heartbeat calls. Alternatively, the client also supports an "adaptive" mode (the default), where it will take all the active locks at the beginning of each heartbeat_period and spread their individual heartbeat calls evenly across the whole period.

3.6 Differences from Java implementation

As indicated before, this library derives most of its design from the dynamo-db-lock (Java) module. This section goes over few details where this library goes a slightly different way:

- **Added suport for DynadmoDB TTL attribute**
  - Since Feb 2017, DynamoDB supports having the tables designate one of the attributes as a TTL attribute - containing an epoch timestamp value. Once the current time goes past that value, that row becomes eligible for automated deletion by DynamoDB. These deletes do not incur any additional costs and help keep the table clean of old/stale entries.
- **Dropped support for lock retention after release**
  - The java library supports an additional lock-attribute called "deleteOnRelease" - which allows the caller to control whether the lock, on its release, should be deleted or just marked as released. This python module drops that flexibility, and always deletes the lock on release. The idea is to not try and treat the lock table as a general purpose data-store, and treat it as a persistent representation of the "currently active locks".
- **Dropped support for BLOB data field**
  - The java library supports a byte[] field called 'data' in addition to supporting arbitrary named fields to be stored along with any lock. This python module drops that additional data field - with the understanding that any additional data that the app wishes to store, can be passed in as part of the additional_attributes map/dict that is already supported.
- **Separate lock classes to represent local vs remote locks**
  - The java library uses the same LockItem class to represent both the locks created/acquired by this client as well as the locks loaded from the database (currently held by other clients). This results in confusing overloading of fields e.g. the "lookupTime" is overloaded to store the "lastUpdatedTime" for the locks owned by this client, and the "lastLookupTime" for the locks owned by other clients.
- **Added support for explicit and adaptive heartbeat TPS**
  - The java library would fire off the heartbeat updates for all the active locks one-after-another - as fast as it can, and then wait till the end ot the heartbeat_period, and then do the same thing over. This can result in significant write TPS is the application has a lot (say ~100) active locks. This python module allows the caller to specific an explicit TPS value, or use an adaptive mode - where the heartbeats are evenly spread over the whole heatbeat_period.
- **Different callback model**
  - The java library creates a different thread for each lock that wishes to support "session-monitors". This python module uses a single thread (separate from the one used to send heartbeats) to periodically check that the locks are being "heartbeat"-ed and if needed, use a ThreadPoolExecutor to invoke the app_callbacks.
- **Uses retry_period/retry_timeout arguments instead of refreshPeriod/additionalTimeToWait**
  - Though the logic is pretty much the same, the names are a little clearer about the intent - the "retry_period" controls how long the client waits before retrying a previously failed lock acquisition, and "retry_timeout" controls how long the client keeps retrying before giving up and raising an error.
- **Simplified sort-key handling**
  - The java library goes to great lengths to support the caller's ability to use a simple hash-partitioned table as well as a hash-and-range partitioned table. This python module drops the support for hash-partitioned tables, and instead chooses to use a default sort-key of '-' to simplify the implementation.

- **Lock release best_effort mode**
  - The java library defaults to best_effort == False, whereas this python module defaults to True. i.e. trying to release a lock without choosing an explicit "best_effort" setting, could result in Exceptions being thrown in Java, but would be silently logged+swallowed in Python.
- **Releasing all locks on client code**
  - The java library will always try to release all locks when closing the lock_client. This python module will default to NOT releasing the locks on lock_client closure - but does support an optional argument called "release_locks" that will allow the caller to request lock releases. The idea behind this is that it is not a safe operation to release the locks without considering the application threads that could continue to process under the assumption that they hold a lock on the underlying business entity. Making the caller request the lock-release explicitly is meant to encourage them to try and wind up the application processing first and release the locks first, before trying to close the lock_client.
- **Dropped/Missing support for AWS RequestMetricCollector**
  - The java library has pervasive support for collecting the AWS request metrics. This python module does not (yet) support this capability.

CHAPTER 4

python_dynamodb_lock package

The package contains a single module - with the same name i.e. python_dynamodb_lock

4.1 python_dynamodb_lock module

This is a general purpose distributed locking library built on top of DynamoDB. It is heavily "inspired" by the java-based AmazonDynamoDBLockClient library, and supports both coarse-grained and fine-grained locking.

class DynamoDBLockClient(*dynamodb_resource*, *table_name='DynamoDBLockTable'*, *partition_key_name='lock_key'*, *sort_key_name='sort_key'*, *ttl_attribute_name='expiry_time'*, *owner_name=None*, *heartbeat_period=datetime.timedelta(0, 5)*, *safe_period=datetime.timedelta(0, 20)*, *lease_duration=datetime.timedelta(0, 30)*, *expiry_period=datetime.timedelta(0, 3600)*, *heartbeat_tps=-1*, *app_callback_executor=None*)

Bases: `object`

Provides distributed locks using DynamoDB's support for conditional reads/writes.

Parameters

- **dynamodb_resource** (*boto3.ServiceResource*) – mandatory argument
- **table_name** (*str*) – defaults to 'DynamoDBLockTable'
- **partition_key_name** (*str*) – defaults to 'lock_key'
- **sort_key_name** (*str*) – defaults to 'sort_key'
- **ttl_attribute_name** (*str*) – defaults to 'expiry_time'
- **owner_name** (*str*) – defaults to hostname + _uuid
- **heartbeat_period** (*datetime.timedelta*) – How often to update DynamoDB to note that the instance is still running. It is recommended to make this at least 4 times smaller than the leaseDuration. Defaults to 5 seconds.

- **safe_period** (*datetime.timedelta*) – How long is it okay to go without a heartbeat before considering a lock to be in "danger". Defaults to 20 seconds.
- **lease_duration** (*datetime.timedelta*) – The length of time that the lease for the lock will be granted for. i.e. if there is no heartbeat for this period of time, then the lock will be considered as expired. Defaults to 30 seconds.
- **expiry_period** (*datetime.timedelta*) – The fallback expiry timestamp to allow DynamoDB to cleanup old locks after a server crash. This value should be significantly larger than the _lease_duration to ensure that clock-skew etc. are not an issue. Defaults to 1 hour.
- **heartbeat_tps** (*int*) – The number of heartbeats to execute per second (per node) - this will have direct correlation to DynamoDB provisioned throughput for writes. If set to -1, the client will distribute the heartbeat calls evenly over the _heartbeat_period - which uses lower throughput for smaller number of locks. However, if you want a more deterministic heartbeat-call-rate, then specify an explicit TPS value. Defaults to -1.
- **app_callback_executor** (*ThreadPoolExecutor*) – The executor to be used for invoking the app_callbacks in case of un-expected errors. Defaults to a ThreadPoolExecutor with a maximum of 5 threads.

acquire_lock(*partition_key*, *sort_key='-'*, *retry_period=None*, *retry_timeout=None*, *additional_attributes=None*, *app_callback=None*, *raise_context_exception=False*)

Acquires a distributed DynaomDBLock for the given key(s).

If the lock is currently held by a different client, then this client will keep retrying on a periodic basis. In that case, a few different things can happen:

1) **The other client releases the lock - basically deleting it from the database** Which would allow this client to try and insert its own record instead.
2) **The other client dies, and the lock stops getting updated by the heartbeat thread.** While waiting for a lock, this client keeps track of the local-time whenever it sees the lock's record-version-number change. From that point-in-time, it needs to wait for a period of time equal to the lock's lease duration before concluding that the lock has been abandoned and try to overwrite the database entry with its own lock.
3) **This client goes over the max-retry-timeout-period** While waiting for the other client to release the lock (or for the lock's lease to expire), this client may go over the retry_timeout period (as provided by the caller) - in which case, a DynamoDBLockError with code == ACQUIRE_TIMEOUT will be thrown.
4) **Race-condition amongst multiple lock-clients waiting to acquire lock** Whenever the "old" lock is released (or expires), there may be multiple "new" clients trying to grab the lock - in which case, one of those would succeed, and the rest of them would get a "conditional-update-exception". This is just logged and swallowed internally - and the client moves on to another sleep-retry cycle.
5) **Any other error/exception** Would be wrapped inside a DynamoDBLockError and raised to the caller.

Parameters

- **partition_key** (*str*) – The primary lock identifier
- **sort_key** (*str*) – Forms a "composite identifier" along with the partition_key. Defaults to '-'
- **retry_period** (*datetime.timedelta*) – If the lock is not immediately available, how long should we wait between retries? Defaults to heartbeat_period.

- **retry_timeout** (*datetime.timedelta*) – If the lock is not available for an extended period, how long should we keep trying before giving up and timing out? This value should be set higher than the lease_duration to ensure that other clients can pick up locks abandoned by one client. Defaults to lease_duration + heartbeat_period.
- **additional_attributes** (*dict*) – Arbitrary application metadata to be stored with the lock
- **app_callback** (*Callable*) – Callback function that can be used to notify the app of lock entering the danger period, or an unexpected release
- **raise_context_exception** (*bool*) – Allow exception in the context to be raised

Return type *DynamoDBLock*

Returns A distributed lock instance

release_lock(*lock*, *best_effort=True*)

Releases the given lock - by deleting it from the database.

It allows the caller app to indicate whether it wishes to be informed of all errors/exceptions, or just have the lock-client swallow all of them. A typical usage pattern would include acquiring the lock, making app changes, and releasing the lock. By the time the app is releasing the lock, it would generally be too late to respond to any errors encountered during the release phase - but, the app may still wish to get informed and log it somewhere of offline re-conciliation/follow-up.

Parameters

- **lock** (DynamoDBLock) – The lock instance that needs to be released
- **best_effort** (*bool*) – If True, any exception when calling DynamoDB will be ignored and the clean up steps will continue, hence the lock item in DynamoDb might not be updated / deleted but will eventually expire. Defaults to True.

close(*release_locks=False*)

Shuts down the background thread - and releases all locks if so asked.

By default, this method will NOT release all the locks - as releasing the locks while the application is still making changes assuming that it has the lock can be dangerous. As soon as a lock is released by this client, some other client may pick it up, and the associated app may start processing the underlying business entity in parallel.

It is recommended that the application manage its shutdown-lifecycle such that all the worker threads operating under these locks are first terminated (committed or rolled-back), the corresponding locks released (one at a time - by each worker thread), and then the lock_client.close() method is called. Alternatively, consider letting the process die without releasing all the locks - they will be auto-released when their lease runs out after a while.

Parameters **release_locks** (*bool*) – if True, releases all the locks. Defaults to False.

classmethod create_dynamodb_table(*dynamodb_client*, *table_name='DynamoDBLockTable'*, *partition_key_name='lock_key'*, *sort_key_name='sort_key'*, *ttl_attribute_name='expiry_time'*, *read_capacity=5*, *write_capacity=5*)

Helper method to create the DynamoDB table

Parameters

- **dynamodb_client** (*boto3.DynamoDB.Client*) – mandatory argument
- **table_name** (*str*) – defaults to 'DynamoDBLockTable'

- **partition_key_name** (*str*) – defaults to 'lock_key'
- **sort_key_name** (*str*) – defaults to 'sort_key'
- **ttl_attribute_name** (*str*) – defaults to 'expiry_time'
- **read_capacity** (*int*) – the max TPS for strongly-consistent reads; defaults to 5
- **write_capacity** (*int*) – the max TPS for write operations; defaults to 5

class BaseDynamoDBLock(*partition_key*, *sort_key*, *owner_name*, *lease_duration*, *record_version_number*, *expiry_time*, *additional_attributes*)

Bases: object

Represents a distributed lock - as stored in DynamoDB.

Typically used within the code to represent a lock held by some other lock-client.

Parameters

- **partition_key** (*str*) – The primary lock identifier
- **sort_key** (*str*) – If present, forms a "composite identifier" along with the partition_key
- **owner_name** (*str*) – The owner name - typically from the lock_client
- **lease_duration** (*float*) – The lease duration in seconds - typically from the lock_client
- **record_version_number** (*str*) – A "liveness" indicating GUID - changes with every heartbeat
- **expiry_time** (*int*) – Epoch timestamp in seconds after which DynamoDB will auto-delete the record
- **additional_attributes** (*dict*) – Arbitrary application metadata to be stored with the lock

class DynamoDBLock(*partition_key*, *sort_key*, *owner_name*, *lease_duration*, *record_version_number*, *expiry_time*, *additional_attributes*, *app_callback*, *lock_client*, *raise_context_exception*)

Bases: *python_dynamodb_lock.python_dynamodb_lock.BaseDynamoDBLock*

Represents a lock that is owned by a local DynamoDBLockClient instance.

Parameters

- **partition_key** (*str*) – The primary lock identifier
- **sort_key** (*str*) – If present, forms a "composite identifier" along with the partition_key
- **owner_name** (*str*) – The owner name - typically from the lock_client
- **lease_duration** (*float*) – The lease duration - typically from the lock_client
- **record_version_number** (*str*) – Changes with every heartbeat - the "liveness" indicator
- **expiry_time** (*int*) – Epoch timestamp in seconds after which DynamoDB will auto-delete the record
- **additional_attributes** (*dict*) – Arbitrary application metadata to be stored with the lock
- **app_callback** (*Callable*) – Callback function that can be used to notify the app of lock entering the danger period, or an unexpected release
- **lock_client** (DynamoDBLockClient) – The client that "owns" this lock

- **raise_context_exception** (*bool*) – Allow exception in the context to be raised

PENDING = 'PENDING'

LOCKED = 'LOCKED'

RELEASED = 'RELEASED'

IN_DANGER = 'IN_DANGER'

INVALID = 'INVALID'

release(*best_effort=True*)

Calls the lock_client.release_lock(self, True) method

Parameters **best_effort** (*bool*) – If True, any exception when calling DynamoDB will be ignored and the clean up steps will continue, hence the lock item in DynamoDb might not be updated / deleted but will eventually expire. Defaults to True.

exception DynamoDBLockError(*code='UNKNOWN'*, *message='Unknown error'*)

Bases: Exception

Wrapper for all kinds of errors that might occur during the acquire and release calls.

CLIENT_SHUTDOWN = 'CLIENT_SHUTDOWN'

ACQUIRE_TIMEOUT = 'ACQUIRE_TIMEOUT'

LOCK_NOT_OWNED = 'LOCK_NOT_OWNED'

LOCK_STOLEN = 'LOCK_STOLEN'

LOCK_IN_DANGER = 'LOCK_IN_DANGER'

UNKNOWN = 'UNKNOWN'

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/mohankishore/python_dynamodb_lock/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Python DynamoDB Lock could always use more documentation, whether as part of the official Python DynamoDB Lock docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mohankishore/python_dynamodb_lock/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *python_dynamodb_lock* for local development.

1. Fork the *python_dynamodb_lock* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/python_dynamodb_lock.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv python_dynamodb_lock
$ cd python_dynamodb_lock/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 python_dynamodb_lock tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/whatnick/python_dynamodb_lock/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_python_dynamodb_lock
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Development Lead

- Mohan Kishore <mohankishore@yahoo.com>

6.2 Contributors/Maintainers

- Tisham Dhar <whatnickd@gmail.com>

CHAPTER 7

History

7.1 0.9.3 (2020-07-14)

- Forked Release from whatnick via CI

7.2 0.9.2 (2020-07-13)

- Forked Release from whatnick manual

7.3 0.9.1 (2019-10-29)

- Main repository second release

7.4 0.9.0 (2018-10-28)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index