Privex Django Lock Manager Documentation

Privex Inc., Chris (Someguy123)

Nov 22, 2019

MAIN:

Welcome to the documentation for Privex's Django Lock Manager - a small, open source Python 3 package for Django, designed to provide simple, frustration free locks in your Django application, without requiring any additional services like Redis / Memcached.

This documentation is automatically kept up to date by ReadTheDocs, as it is automatically re-built each time a new commit is pushed to the Github Project

Contents

CHAPTER

ONE

QUICK INSTALL

Installing with Pipenv (recommended)

```
pipenv install django-lockmgr
```

Installing with standard `pip3`

```
pip3 install django-lockmgr
```

Add *lockmgr* to your *INSTALLED_APPS*

```
INSTALLED_APPS = [
    'django.contrib.admin.apps.SimpleAdminConfig',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    # ...
    'lockmgr'
]
```

Run the migrations

```
./manage.py migrate lockmgr
```

CHAPTER

TWO

ALL DOCUMENTATION

2.1 Installing Django Lock Manager

2.1.1 Download and install from PyPi using pipenv / pip (recommended)

Installing with Pipenv **(recommended)**

```
pipenv install django-lockmgr
```

Installing with standard `pip3`

```
pip3 install django-lockmgr
```

2.1.2 (Alternative) Manual install from Git

You may wish to use the alternative installation methods if:

- You need a feature / fix from the Git repo which hasn't yet released as a versioned PyPi package
- You need to install django-lockmgr on a system which has no network connection
- You don't trust / can't access PyPi
- For some reason you can't use `pip` or `pipenv`

Option 1 - Use pip to install straight from Github

```
pip3 install git+https://github.com/Privex/django-lockmgr
```

Option 2 - Clone and install manually

```
# Clone the repository from Github
git clone https://github.com/Privex/django-lockmgr
cd django-lockmgr

# RECOMMENDED MANUAL INSTALL METHOD
# Use pip to install the source code
pip3 install .

# ALTERNATIVE MANUAL INSTALL METHOD
# If you don't have pip, or have issues with installing using it, then you can use␣
↪setuptools instead.
python3 setup.py install
```

2.1.3 Post-installation

Django Lock Manager requires very little configuration after installation. Simply add it to your `INSTALLED_APPS`, and run `./manage.py migrate lockmgr` to create the database tables.

Add `lockmgr` to your `INSTALLED_APPS`

```
INSTALLED_APPS = [
    'django.contrib.admin.apps.SimpleAdminConfig',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    # ...
    'lockmgr'
]
```

Run the migrations

```
./manage.py migrate lockmgr
```

lockmgr.lockmgr	This is the main module file for **'Django Lock Manager'_** (django-lockmgr) and contains lock management functions/classes.
lockmgr.models	
lockmgr.management.commands	

2.2 Using Django Lock Manager

This is the main module file for Django Lock Manager (django-lockmgr) and contains lock management functions/classes.

There are **two ways** you can use Django Lock Manager:

- The first (and recommended) way, is to use the context manager class *LockMgr*.
- The second (lower level) way, is to use the lock functions directly, such as *get_lock()*, *unlock()*, and *set_lock()*.

2.2.1 Using the context manager LockMgr (recommended)

LockMgr is a wrapper class for the various locking functions in this module, e.g. *get_lock()*, and is designed to be used as a **context manager**, i.e. using a `with` statement.

It's strongly recommended to use django-lockmgr via the *LockMgr* context manager unless you have a specific need for manual lock management, as it greatly reduces the risk of "stuck locks" due to human error, or incorrect exception handling.

By using django-lockmgr via this context manager, it ensures you don't forget to release any locks after you've finished with the resources you were using.

Not only that, but it also ensures in the event of an exception, or an unexpected crash of your application, that your locks will usually be safely released by *LockMgr.__exit__()*.

```
>>> from lockmgr.lockmgr import LockMgr
>>> try:
```

(continues on next page)

(continued from previous page)

```
...     with LockMgr('mylock', 60) as l:
...         print('Doing stuff with mylock locked.')
...         # Obtain an additional lock for 'otherlock' - will use the same expiry as␣
↪mylock
...         # Since ``ret`` is set to True, it will return a bool instead of raising␣
↪Lock
...         if l.lock('otherlock', ret=True):
...             print('Now otherlock is locked...')
...             l.unlock('otherlock')
...         else:
...             print('Not doing stuff because otherlock is already locked...')
...         # If you're getting close to your lock's expiry (timeout), you can call '.
↪renew()' to add an extra
...         # 2 minutes to your expiry time. Or manually specify the expiry with
↪'expires=120'
...         sleep(50)
...         l.renew(expires=30) # Add an extra 30 seconds to the expiration of 'mylock
↪'
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
```

2.2.2 Using the raw module lock management functions

In some cases, it might not be suitable to use context management due to a complex application flow, such as the use of threading / multiprocessing, sharing the locks across other applications, etc.

If you need to, you can access the lower level lock management functions by importing this module, or the individual functions.

Here's some examples:

First, let's get a lock using *get_lock()* that expires in 10 seconds, and wait a few seconds.

```
>>> from lockmgr import lockmgr
>>> lk = lockmgr.get_lock('my_app:somelock', expires=10)
>>> sleep(5)
```

Since our lock is going to expire soon, we'll use *renew_lock()* to reset the expiration time to 20 seconds from now.

```
>>> lk = lockmgr.renew_lock(lk, 20)    # Change the expiry time to 20 seconds from now
>>> sleep(15)
```

Using *is_locked()*, we can confirm that the lock ‘‘my_app:somelock‘ is still locked:

```
>>> lockmgr.is_locked('my_app:somelock') # 15 seconds later, the lock is still locked
True
```

Finally, we use *unlock()* to release the lock. You can pass either a string lock name such as `my_app:somelock`, or you can also pass a *Lock* database object i.e. the result from *get_lock()*. Use whichever parameter type you prefer, it doesn't make a difference.

```
>>> lockmgr.unlock(lk)
```

2.2.3 Extra documentation

This is not the end of the documentation, this is only the beginning! :)

You'll find detailed documentation on the pages for each function / class / method. Most things are documented using **PyDoc**, which means you can view usage information straight from most Python IDEs (e.g. PyCharm and VS Code), as well as via the `help()` function inside of the Python REPL.

Browsable HTML API docs

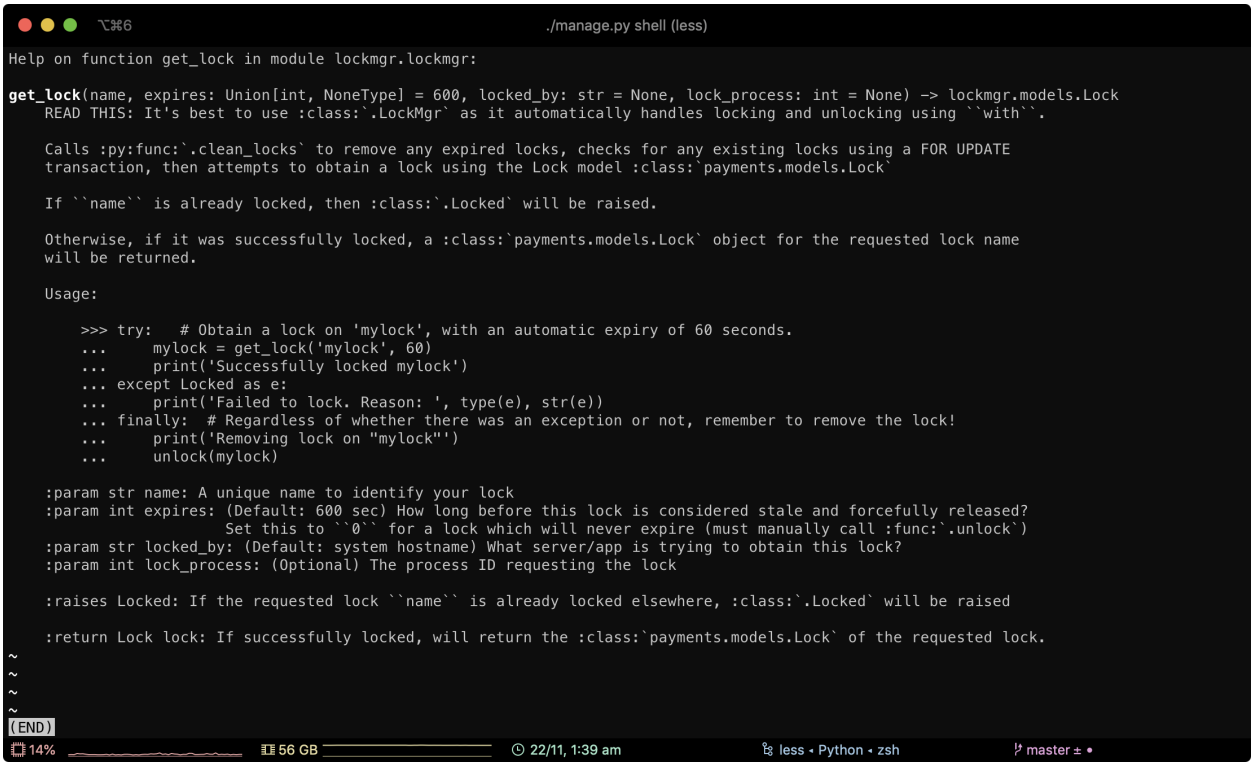
We have online documentation for this module, which shows the usage information for each individual function and class method in this module.

Python REPL help

Using the `help()` function, you can view help on modules, classes, functions and more straight from the REPL:

```
$ ./manage.py shell
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from lockmgr import lockmgr
>>> help(lockmgr.get_lock)
```

Below is a screenshot showing the REPL help page for *get_lock()*



exception lockmgr.lockmgr.**LockFail**(**args*, *lock: lockmgr.models.Lock = None*)

Raised when locks were requested, with failure/rollback if any already existed.

class lockmgr.lockmgr.**LockMgr**(*name*, *expires: Optional[int] = 600*, *locked_by=None*, *lock_process=None*, *wait: int = None*)

LockMgr is a wrapper class for the various locking functions in this module, e.g. *get_lock()*, and is designed

to be used as a **context manager**, i.e. using a `with` statement.

By using django-lockmgr via this context manager, it ensures you don't forget to release any locks after you've finished with the resources you were using.

Not only that, but it also ensures in the event of an exception, or an unexpected crash of your application, that your locks will usually be safely released by *__exit__()*.

Usage:

Using a `with` statement, create a LockMgr for `mylock` with automatic expiration if held for more than 60 seconds. After the `with` statement is completed, all locks created will be removed.

```
>>> try:
...     with LockMgr('mylock', 60) as l:
...         print('Doing stuff with mylock locked.')
...         # Obtain an additional lock for 'otherlock' - will use the␣
↪same expiry as mylock
...         # Since ``ret`` is set to True, it will return a bool instead␣
↪of raising Lock
...         if l.lock('otherlock', ret=True):
...             print('Now otherlock is locked...')
...             l.unlock('otherlock')
...         else:
...             print('Not doing stuff because otherlock is already␣
↪locked...')
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
```

You can also use *renew()* to request more time / re-create the lock if you're close to, or have already exceeded the lock expiration time (defaults to 10 mins).

```
>>> try:
...     with LockMgr('mylock', 60) as l:
...         print('Doing stuff with mylock locked.')
...         sleep(50)
...         l.renew(expires=30)     # Add an additional 30 seconds of time␣
↪to the lock expiration
...         sleep(50)               # It's now been 100 seconds. 'mylock'␣
↪should be expired.
...         # We can still renew an expired lock when using LockMgr. It␣
↪will simply re-create the lock.
...         l.renew()               # Add an additional 120 seconds␣
↪(default) of time to the lock expiration
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
```

expires = None
The user supplied expiration time in seconds

lock(*name*, *expires: int = None*, *ret: bool = False*, *wait: int = None*)
Obtains a lock using *get_lock()* and appends it to `_locks` if successful.

If the argument `ret` is `False` (default), it will raise *Locked* if the lock couldn't be obtained.

Otherwise, if `ret` is `True`, it will simply return `False` if the requested lock name is already locked.

Parameters

- **name** (*str*) – A unique name to identify your lock

- **expires** (*int*) – (Default: 600 sec) How long before this lock is considered stale and forcefully released?
- **ret** (*bool*) – (Default: False) Return `False` if locked, instead of raising `Locked`.
- **wait** (*int*) – (Optional) Retry obtaining the lock for this many seconds. MUST be divisible by 5. If not empty, will retry obtaining the lock every 5 seconds until `wait` seconds

Raises ***Locked*** – If the requested lock `name` is already locked elsewhere, *Locked* will be raised

Return bool success `True` if successful. If `ret` is true then will also return False on failure.

lock_process = None
Usually None, but sometimes may represent the process ID this lock belongs to

locked_by = None
Who/what created this lock - usually the hostname unless manually specified

main_lock = None
The *Lock* object created at the start of a `with LockManager('xyz')` statement

name = None
The lock name (from the constructor)

renew(*lock: Union[str, lockmgr.models.Lock] = None*, *expires: int = 120*, *add_time: bool = True*, ***kwargs*) → lockmgr.models.Lock
Add `expires` seconds to the lock expiry time of `lock`. If `lock` isn't specified, will default to the class instance's original lock *main_lock*

Alias for *renew_lock()* - but with `add_time` and `create` set to `True` by default, instead of `False`.

With no arguments specified, this method will renew the main lock of the class *main_lock* for an additional 2 minutes (or if the lock is already expired, will re-create it with 2 min expiry).

Example usage:

```
>>> with LockMgr('mylock', expires=30) as l:
...     sleep(10)
...     l.renew(expires=60)                  # Add 60 seconds more time to
↪'mylock' expiration
...     l.main_lock.refresh_from_db()
...     print(l.main_lock.expires_seconds)   # Output: 79
...     l.renew('lockx', expires=60)         # Add 60 seconds more time to
↪'lockx' expiration
```

Parameters

- **lock** (`Lock`) – Name of the lock to renew
- **lock** – A *Lock* object to renew
- **expires** (*int*) – (Default: 120) If not add_time, then this is the new expiration time in seconds from now. If add_time, then this many seconds will be added to the expiration time of the lock.
- **add_time** (*bool*) – (Default: `True`) If True, then `expires` seconds will be added to the existing lock expiration time, instead of setting the expiration time to `now + expires`

Extra Keyword Arguments

Key bool create (Default: `True`) If True, then create a new lock if it doesn't exist / already expired

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int lock_process (Optional) The process ID requesting the lock

Exceptions

Raises ***LockNotFound*** – Raised if the requested `lock` doesn't exist / is already expired and `create` is False.

Return Lock lock The *Lock* object which was renewed

unlock(*lock: Union[lockmgr.models.Lock, str] = None*)
Alias for *unlock()*

wait = None
How long to wait for a lock before giving up. If this is `None` then waiting will be disabled

exception lockmgr.lockmgr.**LockNotFound**
Raised when a requested lock doesn't exist

class lockmgr.lockmgr.**LockSetResult**(***kwargs*)

class lockmgr.lockmgr.**LockSetStatus**(***kwargs*)

exception lockmgr.lockmgr.**Locked**
Raised when a lock already exists with the given name

lockmgr.lockmgr.**clean_locks**()
Deletes expired *Lock* objects.

lockmgr.lockmgr.**get_lock**(*name*, *expires: Optional[int] = 600*, *locked_by: str = None*, *lock_process: int = None*) → lockmgr.models.Lock
READ THIS: It's best to use *LockMgr* as it automatically handles locking and unlocking using `with`.

Calls *clean_locks()* to remove any expired locks, checks for any existing locks using a FOR UPDATE transaction, then attempts to obtain a lock using the Lock model `payments.models.Lock`

If `name` is already locked, then *Locked* will be raised.

Otherwise, if it was successfully locked, a `payments.models.Lock` object for the requested lock name will be returned.

Usage:

```
>>> try:   # Obtain a lock on 'mylock', with an automatic expiry of 60 seconds.
...     mylock = get_lock('mylock', 60)
...     print('Successfully locked mylock')
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
... finally:  # Regardless of whether there was an exception or not, remember to␣
↪remove the lock!
...     print('Removing lock on "mylock"')
...     unlock(mylock)
```

Parameters

- **name** (*str*) – A unique name to identify your lock
- **expires** (*int*) – (Default: 600 sec) How long before this lock is considered stale and forcefully released? Set this to `0` for a lock which will never expire (must manually call *unlock()*)
- **locked_by** (*str*) – (Default: system hostname) What server/app is trying to obtain this lock?
- **lock_process** (*int*) – (Optional) The process ID requesting the lock

Raises ***Locked*** – If the requested lock `name` is already locked elsewhere, *Locked* will be raised

Return Lock lock If successfully locked, will return the `payments.models.Lock` of the requested lock.

`lockmgr.lockmgr.`**`is_locked`**(*name: Union[lockmgr.models.Lock, str]*) → bool

Cleans expired locks, then returns `True` if the given lock key `name` exists, otherwise `False`

`lockmgr.lockmgr.`**`renew_lock`**(*lock: Union[str, lockmgr.models.Lock], expires: int = 600, add_time: bool = False, **kwargs*) → lockmgr.models.Lock

Renew an existing lock for more expiry time.

Note: This function will NOT reduce a lock's expiry time, only lengthen. If `add_time` is `False`, and the new expiration time `expires` is shorter than the lock's existing expiration time, then the lock's expiry time will be left untouched.

Example - Renew an existing lock:

```
>>> lk = get_lock('my_app:somelock', expires=10)
>>> sleep(5)
>>> lk = renew_lock(lk, 20)    # Change the expiry time to 20 seconds from now
>>> sleep(15)
>>> is_locked('my_app:somelock') # 15 seconds later, the lock is still locked
True
```

Example - Try to renew, but get a new lock if it's already been released:

```
>>> lk = get_lock('my_app:somelock', expires=5)
>>> sleep(10)
>>> lk = renew_lock(lk, 20, create=True)    # If the lock is expired/non-existant,␣
↪make a new lock
>>> sleep(15)
>>> is_locked('my_app:somelock') # 15 seconds later, the lock is still locked
True
```

Parameters

- **lock** (`Lock`) – Name of the lock to renew
- **lock** – A *Lock* object to renew
- **expires** (*int*) – (Default: 600) If not add_time, then this is the new expiration time in seconds from now. If add_time, then this many seconds will be added to the expiration time of the lock.
- **add_time** (*bool*) – (Default: `False`) If True, then `expires` seconds will be added to the existing lock expiration time, instead of setting the expiration time to `now + expires`

Key bool create (Default: `False`) If True, then create a new lock if it doesn't exist / already expired.

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int lock_process (Optional) The process ID requesting the lock

Raises ***LockNotFound*** – Raised if the requested `lock` doesn't exist / is already expired and `create` is False.

Return Lock lock The *Lock* object which was renewed

`lockmgr.lockmgr.`**`set_lock`**(**locks*, *timeout=600*, *fail=False*, *renew=True*, *create=True*, ***options*) → lockmgr.lockmgr.LockSetResult

This function is for advanced users, offering multiple lock creation, renewing, along with "all or nothing" locking with database rollback via the argument `fail`.

Unlike other lock management functions, set_lock returns a *LockSetResult* object, which is designed to allow you to see clearly as to what locks were created, renewed, or skipped.

Example Usage

Let's set two locks, `hello` and `world`.

```
>>> res = set_lock('hello', 'world')
>>> res['locks']
[<Lock name='hello' locked_by='example.org' locked_until='2019-11-22 02:01:55.
↪439390+00:00'>,
 <Lock name='world' locked_by='example.org' locked_until='2019-11-22 02:01:55.
↪442734+00:00'>]
>>> res['counts']
{'created': 2, 'renewed': 0, 'skip_create': 0, 'skip_renew': 0}
```

If we run `set_lock` again with the same arguments, we'll still get the locks list, but we'll see the counts show that they were renewed instead of created.

```
>>> x = set_lock('hello', 'world')
>>> x['locks']
[<Lock name='hello' locked_by='example.org' locked_until='2019-11-22 02:03:06.
↪762620+00:00'>,
 <Lock name='world' locked_by='example.org' locked_until='2019-11-22 02:03:06.
↪766804+00:00'>]
>>> x['counts']
{'created': 0, 'renewed': 2, 'skip_create': 0, 'skip_renew': 0}
```

Since the result is an object, you can also access attributes via dot notation, as well as dict-like notation.

We can see inside of the `statuses` list - the action that was taken on each lock we specified, so we can see what locks were created, renewed, or skipped etc.

```
>>> x.statuses[0]
('hello', {'was_locked': True, 'status': 'extend', 'locked': True})
>>> x.statuses[1]
('world', {'was_locked': True, 'status': 'extend', 'locked': True})
```

Parameters

- **`locks`** (*str*) – One or more lock names, as positional arguments, to create or renew.
- **`timeout`** (*int*) – On existing locks, update locked_until to `now + timeout` (seconds)
- **`fail`** (*bool*) – (Default: False) If `True`, all lock creations will be rolled back if an existing lock is encountered, and *LockFail* will be raised.

- **renew** (*bool*) – (Default: True) If `True`, any existing locks in `locks` will be renewed to `now + timeout` (seconds). If False, existing locks will just be skipped.
- **create** (*bool*) – (Default: True) If `True`, any names in `locks` which aren't yet locked, will have a lock created for them, with their expiry set to `timeout` seconds from now.

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int process_id (Optional) The process ID requesting the lock

Return LockSetResult results A *LockSetResult* object containing the results of the set_lock operation.

`lockmgr.lockmgr.`**`unlock`**(*lock: Union[lockmgr.models.Lock, str]*)

Releases a given lock - either specified as a string name, or as a `payments.models.Lock` object.

Usage:

```
>>> mylock = get_lock('mylock', expires=60)
>>> unlock('mylock') # Delete the lock by name
>>> unlock(mylock)   # Or by Lock object.
```

Parameters

- **lock** (`Lock`) – The name of the lock to release
- **lock** – A *Lock* object to release

2.3 API Docs (lockmgr.lockmgr)

Functions

clean_locks()	Deletes expired *Lock* objects.
get_lock(name[, expires, locked_by, ...])	READ THIS: It's best to use *LockMgr* as it automatically handles locking and unlocking using `with`.
is_locked(name)	Cleans expired locks, then returns `True` if the given lock key `name` exists, otherwise `False`
renew_lock(lock[, expires, add_time])	Renew an existing lock for more expiry time.
set_lock(*locks[, timeout, fail, renew, create])	This function is for advanced users, offering multiple lock creation, renewing, along with "all or nothing" locking with database rollback via the argument `fail`.
unlock(lock)	Releases a given lock - either specified as a string name, or as a `payments.models.Lock` object.

2.3.1 clean_locks

`lockmgr.lockmgr.`**`clean_locks`**()

Deletes expired *Lock* objects.

2.3.2 get_lock

lockmgr.lockmgr.**get_lock**(*name*, *expires: Optional[int] = 600*, *locked_by: str = None*, *lock_process: int = None*) → lockmgr.models.Lock

READ THIS: It's best to use *LockMgr* as it automatically handles locking and unlocking using `with`.

Calls *clean_locks()* to remove any expired locks, checks for any existing locks using a FOR UPDATE transaction, then attempts to obtain a lock using the Lock model `payments.models.Lock`

If `name` is already locked, then *Locked* will be raised.

Otherwise, if it was successfully locked, a `payments.models.Lock` object for the requested lock name will be returned.

Usage:

```
>>> try:   # Obtain a lock on 'mylock', with an automatic expiry of 60␣
↪seconds.
...     mylock = get_lock('mylock', 60)
...     print('Successfully locked mylock')
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
... finally:  # Regardless of whether there was an exception or not,␣
↪remember to remove the lock!
...     print('Removing lock on "mylock"')
...     unlock(mylock)
```

Parameters

- **name** (*str*) – A unique name to identify your lock
- **expires** (*int*) – (Default: 600 sec) How long before this lock is considered stale and forcefully released? Set this to `0` for a lock which will never expire (must manually call *unlock()*)
- **locked_by** (*str*) – (Default: system hostname) What server/app is trying to obtain this lock?
- **lock_process** (*int*) – (Optional) The process ID requesting the lock

Raises ***Locked*** – If the requested lock `name` is already locked elsewhere, *Locked* will be raised

Return Lock lock If successfully locked, will return the `payments.models.Lock` of the requested lock.

2.3.3 is_locked

lockmgr.lockmgr.**is_locked**(*name: Union[lockmgr.models.Lock, str]*) → bool

Cleans expired locks, then returns `True` if the given lock key `name` exists, otherwise `False`

2.3.4 renew_lock

lockmgr.lockmgr.**renew_lock**(*lock: Union[str, lockmgr.models.Lock]*, *expires: int = 600*, *add_time: bool = False*, ***kwargs*) → lockmgr.models.Lock

Renew an existing lock for more expiry time.

Note: This function will NOT reduce a lock's expiry time, only lengthen. If `add_time` is `False`, and the new expiration time `expires` is shorter than the lock's existing expiration time, then the lock's expiry time will be left untouched.

Example - Renew an existing lock:

```
>>> lk = get_lock('my_app:somelock', expires=10)
>>> sleep(5)
>>> lk = renew_lock(lk, 20)    # Change the expiry time to 20 seconds␣
↪from now
>>> sleep(15)
>>> is_locked('my_app:somelock') # 15 seconds later, the lock is still␣
↪locked
True
```

Example - Try to renew, but get a new lock if it's already been released:

```
>>> lk = get_lock('my_app:somelock', expires=5)
>>> sleep(10)
>>> lk = renew_lock(lk, 20, create=True)    # If the lock is expired/non-
↪existant, make a new lock
>>> sleep(15)
>>> is_locked('my_app:somelock') # 15 seconds later, the lock is still␣
↪locked
True
```

Parameters

- **lock** (*Lock*) – Name of the lock to renew
- **lock** – A *Lock* object to renew
- **expires** (*int*) – (Default: 600) If not add_time, then this is the new expiration time in seconds from now. If add_time, then this many seconds will be added to the expiration time of the lock.
- **add_time** (*bool*) – (Default: `False`) If True, then `expires` seconds will be added to the existing lock expiration time, instead of setting the expiration time to `now + expires`

Key bool create (Default: `False`) If True, then create a new lock if it doesn't exist / already expired.

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int lock_process (Optional) The process ID requesting the lock

Raises ***LockNotFound*** – Raised if the requested `lock` doesn't exist / is already expired and `create` is False.

Return Lock lock The *Lock* object which was renewed

2.3.5 set_lock

lockmgr.lockmgr.**set_lock**(**locks*, *timeout=600*, *fail=False*, *renew=True*, *create=True*, ***options*) → lockmgr.lockmgr.LockSetResult

This function is for advanced users, offering multiple lock creation, renewing, along with "all or nothing" locking with database rollback via the argument `fail`.

Unlike other lock management functions, set_lock returns a *LockSetResult* object, which is designed to allow you to see clearly as to what locks were created, renewed, or skipped.

Example Usage

Let's set two locks, `hello` and `world`.

```
>>> res = set_lock('hello', 'world')
>>> res['locks']
[<Lock name='hello' locked_by='example.org' locked_until='2019-11-22␣
↪02:01:55.439390+00:00'>,
 <Lock name='world' locked_by='example.org' locked_until='2019-11-22␣
↪02:01:55.442734+00:00'>]
>>> res['counts']
{'created': 2, 'renewed': 0, 'skip_create': 0, 'skip_renew': 0}
```

If we run `set_lock` again with the same arguments, we'll still get the locks list, but we'll see the counts show that they were renewed instead of created.

```
>>> x = set_lock('hello', 'world')
>>> x['locks']
[<Lock name='hello' locked_by='example.org' locked_until='2019-11-22␣
↪02:03:06.762620+00:00'>,
 <Lock name='world' locked_by='example.org' locked_until='2019-11-22␣
↪02:03:06.766804+00:00'>]
>>> x['counts']
{'created': 0, 'renewed': 2, 'skip_create': 0, 'skip_renew': 0}
```

Since the result is an object, you can also access attributes via dot notation, as well as dict-like notation.

We can see inside of the `statuses` list - the action that was taken on each lock we specified, so we can see what locks were created, renewed, or skipped etc.

```
>>> x.statuses[0]
('hello', {'was_locked': True, 'status': 'extend', 'locked': True})
>>> x.statuses[1]
('world', {'was_locked': True, 'status': 'extend', 'locked': True})
```

Parameters

- **locks** (*str*) – One or more lock names, as positional arguments, to create or renew.
- **timeout** (*int*) – On existing locks, update locked_until to `now + timeout` (seconds)
- **fail** (*bool*) – (Default: False) If `True`, all lock creations will be rolled back if an existing lock is encountered, and *LockFail* will be raised.
- **renew** (*bool*) – (Default: True) If `True`, any existing locks in `locks` will be renewed to `now + timeout` (seconds). If False, existing locks will just be skipped.

- **create** (*bool*) – (Default: True) If `True`, any names in `locks` which aren't yet locked, will have a lock created for them, with their expiry set to `timeout` seconds from now.

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int process_id (Optional) The process ID requesting the lock

Return LockSetResult results A *LockSetResult* object containing the results of the set_lock operation.

2.3.6 unlock

`lockmgr.lockmgr.`**`unlock`**(*lock: Union[lockmgr.models.Lock, str]*)

Releases a given lock - either specified as a string name, or as a `payments.models.Lock` object.

Usage:

```
>>> mylock = get_lock('mylock', expires=60)
>>> unlock('mylock') # Delete the lock by name
>>> unlock(mylock)   # Or by Lock object.
```

Parameters

- **lock** (*Lock*) – The name of the lock to release
- **lock** – A *Lock* object to release

Classes

LockMgr(name[, expires, locked_by, ...])	LockMgr is a wrapper class for the various locking functions in this module, e.g.

2.3.7 LockMgr

class `lockmgr.lockmgr.`**`LockMgr`**(*name*, *expires: Optional[int] = 600*, *locked_by=None*, *lock_process=None*, *wait: int = None*)

LockMgr is a wrapper class for the various locking functions in this module, e.g. *get_lock()*, and is designed to be used as a **context manager**, i.e. using a `with` statement.

By using django-lockmgr via this context manager, it ensures you don't forget to release any locks after you've finished with the resources you were using.

Not only that, but it also ensures in the event of an exception, or an unexpected crash of your application, that your locks will usually be safely released by *__exit__()*.

Usage:

Using a `with` statement, create a LockMgr for `mylock` with automatic expiration if held for more than 60 seconds. After the `with` statement is completed, all locks created will be removed.

```
>>> try:
...     with LockMgr('mylock', 60) as l:
...         print('Doing stuff with mylock locked.')
```

(continues on next page)

(continued from previous page)

```
...         # Obtain an additional lock for 'otherlock' - will␣
↪use the same expiry as mylock
...         # Since ``ret`` is set to True, it will return a bool␣
↪instead of raising Lock
...         if l.lock('otherlock', ret=True):
...             print('Now otherlock is locked...')
...             l.unlock('otherlock')
...         else:
...             print('Not doing stuff because otherlock is␣
↪already locked...')
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
```

You can also use *renew()* to request more time / re-create the lock if you're close to, or have already exceeded the lock expiration time (defaults to 10 mins).

```
>>> try:
...     with LockMgr('mylock', 60) as l:
...         print('Doing stuff with mylock locked.')
...         sleep(50)
...         l.renew(expires=30)     # Add an additional 30 seconds␣
↪of time to the lock expiration
...         sleep(50)               # It's now been 100 seconds.
↪'mylock' should be expired.
...         # We can still renew an expired lock when using␣
↪LockMgr. It will simply re-create the lock.
...         l.renew()               # Add an additional 120␣
↪seconds (default) of time to the lock expiration
... except Locked as e:
...     print('Failed to lock. Reason: ', type(e), str(e))
```

__init__(*name*, *expires: Optional[int] = 600*, *locked_by=None*, *lock_process=None*, *wait: int = None*)

Create an instance of *LockMgr*. This class is primarily intended to be used as a context manager (i.e. `with LockMgr('mylock') as l:`), see the main PyDoc block for *LockMgr* for more info.

Parameters

- **name** (*str*) – The lock name to create (when using as a context manager)
- **expires** (*int*) – How many seconds before this lock is considered stale and forcefully released?
- **locked_by** (*str*) – (Optional) Who/what is using this lock. Defaults to system hostname.
- **lock_process** (*int*) – (Optional) The process ID of the app using this lock
- **wait** (*int*) – (Optional) Wait this many seconds for a lock to be released before giving up. If this is `None` then waiting will be disabled

expires = None

The user supplied expiration time in seconds

lock(*name*, *expires: int = None*, *ret: bool = False*, *wait: int = None*)

Obtains a lock using *get_lock()* and appends it to `_locks` if successful.

If the argument `ret` is `False` (default), it will raise *Locked* if the lock couldn't be obtained.

Otherwise, if `ret` is `True`, it will simply return `False` if the requested lock name is already locked.

Parameters

- **name** (*str*) – A unique name to identify your lock
- **expires** (*int*) – (Default: 600 sec) How long before this lock is considered stale and forcefully released?
- **ret** (*bool*) – (Default: False) Return `False` if locked, instead of raising `Locked`.
- **wait** (*int*) – (Optional) Retry obtaining the lock for this many seconds. MUST be divisible by 5. If not empty, will retry obtaining the lock every 5 seconds until `wait` seconds

Raises ***Locked*** – If the requested lock `name` is already locked elsewhere, *Locked* will be raised

Return bool success `True` if successful. If `ret` is true then will also return False on failure.

lock_process = None

Usually None, but sometimes may represent the process ID this lock belongs to

locked_by = None

Who/what created this lock - usually the hostname unless manually specified

main_lock = None

The *Lock* object created at the start of a `with LockManager('xyz')` statement

name = None

The lock name (from the constructor)

renew(*lock: Union[str, lockmgr.models.Lock] = None*, *expires: int = 120*, *add_time: bool = True*, ***kwargs*) → lockmgr.models.Lock

Add `expires` seconds to the lock expiry time of `lock`. If `lock` isn't specified, will default to the class instance's original lock *main_lock*

Alias for *renew_lock()* - but with `add_time` and `create` set to `True` by default, instead of `False`.

With no arguments specified, this method will renew the main lock of the class *main_lock* for an additional 2 minutes (or if the lock is already expired, will re-create it with 2 min expiry).

Example usage:

```
>>> with LockMgr('mylock', expires=30) as l:
...     sleep(10)
...     l.renew(expires=60)                  # Add 60 seconds more␣
↪time to 'mylock' expiration
...     l.main_lock.refresh_from_db()
...     print(l.main_lock.expires_seconds)   # Output: 79
...     l.renew('lockx', expires=60)         # Add 60 seconds more␣
↪time to 'lockx' expiration
```

Parameters

- **lock** (*Lock*) – Name of the lock to renew
- **lock** – A *Lock* object to renew
- **expires** (*int*) – (Default: 120) If not add_time, then this is the new expiration time in seconds from now. If add_time, then this many seconds will be added to the expiration time of the lock.
- **add_time** (*bool*) – (Default: `True`) If True, then `expires` seconds will be added to the existing lock expiration time, instead of setting the expiration time to `now + expires`

Extra Keyword Arguments

Key bool create (Default: `True`) If True, then create a new lock if it doesn't exist / already expired

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int lock_process (Optional) The process ID requesting the lock

Exceptions

Raises ***LockNotFound*** – Raised if the requested `lock` doesn't exist / is already expired and `create` is False.

Return Lock lock The *Lock* object which was renewed

unlock(*lock: Union[lockmgr.models.Lock, str] = None*)

Alias for *unlock()*

wait = None

How long to wait for a lock before giving up. If this is `None` then waiting will be disabled

2.3.7.1 Methods

Methods

__init__(name[, expires, locked_by, ...])	Create an instance of *LockMgr*.
lock(name[, expires, ret, wait])	Obtains a lock using *get_lock()* and appends it to `_locks` if successful.
renew([lock, expires, add_time])	Add `expires` seconds to the lock expiry time of `lock`.
unlock([lock])	Alias for *unlock()*
__enter__()	When *LockMgr* is used as a context manager, i.e.
__exit__(exc_type, exc_val, exc_tb)	When the context manager is finished or an exception occurs, we unlock all locks that were created during the context manager session.

2.3.7.1.1 __init__

LockMgr.**__init__**(*name*, *expires: Optional[int] = 600*, *locked_by=None*, *lock_process=None*, *wait: int = None*)

Create an instance of *LockMgr*. This class is primarily intended to be used as a context manager (i.e. `with LockMgr('mylock') as l:`), see the main PyDoc block for *LockMgr* for more info.

Parameters

- **name** (*str*) – The lock name to create (when using as a context manager)
- **expires** (*int*) – How many seconds before this lock is considered stale and forcefully released?
- **locked_by** (*str*) – (Optional) Who/what is using this lock. Defaults to system hostname.

- **lock_process** (*int*) – (Optional) The process ID of the app using this lock
- **wait** (*int*) – (Optional) Wait this many seconds for a lock to be released before giving up. If this is `None` then waiting will be disabled

2.3.7.1.2 lock

`LockMgr.`**`lock`**(*name*, *expires: int = None*, *ret: bool = False*, *wait: int = None*)

Obtains a lock using *`get_lock()`* and appends it to `_locks` if successful.

If the argument `ret` is `False` (default), it will raise *`Locked`* if the lock couldn't be obtained.

Otherwise, if `ret` is `True`, it will simply return `False` if the requested lock name is already locked.

Parameters

- **name** (*str*) – A unique name to identify your lock
- **expires** (*int*) – (Default: 600 sec) How long before this lock is considered stale and forcefully released?
- **ret** (*bool*) – (Default: False) Return `False` if locked, instead of raising `Locked`.
- **wait** (*int*) – (Optional) Retry obtaining the lock for this many seconds. MUST be divisible by 5. If not empty, will retry obtaining the lock every 5 seconds until `wait` seconds

Raises ***Locked*** – If the requested lock `name` is already locked elsewhere, *`Locked`* will be raised

Return bool success `True` if successful. If `ret` is true then will also return False on failure.

2.3.7.1.3 renew

`LockMgr.`**`renew`**(*lock: Union[str, lockmgr.models.Lock] = None*, *expires: int = 120*, *add_time: bool = True*, ***kwargs*) → lockmgr.models.Lock

Add `expires` seconds to the lock expiry time of `lock`. If `lock` isn't specified, will default to the class instance's original lock *`main_lock`*

Alias for *`renew_lock()`* - but with `add_time` and `create` set to `True` by default, instead of `False`.

With no arguments specified, this method will renew the main lock of the class *`main_lock`* for an additional 2 minutes (or if the lock is already expired, will re-create it with 2 min expiry).

Example usage:

```
>>> with LockMgr('mylock', expires=30) as l:
...     sleep(10)
...     l.renew(expires=60)                  # Add 60 seconds more time␣
↪to 'mylock' expiration
...     l.main_lock.refresh_from_db()
...     print(l.main_lock.expires_seconds)   # Output: 79
...     l.renew('lockx', expires=60)         # Add 60 seconds more time␣
↪to 'lockx' expiration
```

Parameters

- **lock** (Lock) – Name of the lock to renew
- **lock** – A *Lock* object to renew
- **expires** (*int*) – (Default: 120) If not add_time, then this is the new expiration time in seconds from now. If add_time, then this many seconds will be added to the expiration time of the lock.
- **add_time** (*bool*) – (Default: `True`) If True, then `expires` seconds will be added to the existing lock expiration time, instead of setting the expiration time to `now + expires`

Extra Keyword Arguments

Key bool create (Default: `True`) If True, then create a new lock if it doesn't exist / already expired

Key str locked_by (Default: system hostname) What server/app is trying to obtain this lock?

Key int lock_process (Optional) The process ID requesting the lock

Exceptions

Raises ***LockNotFound*** – Raised if the requested `lock` doesn't exist / is already expired and `create` is False.

Return Lock lock The *Lock* object which was renewed

2.3.7.1.4 unlock

LockMgr.**unlock**(*lock: Union[lockmgr.models.Lock, str] = None*)

Alias for *unlock()*

2.3.7.1.5 __enter__

LockMgr.**__enter__**()

When *LockMgr* is used as a context manager, i.e. `with LockManager('xyz') as l:` - this method is called to setup the context manager and return the object used for the `with` statement.

This function simply creates the lock specified by the user to *__init__()* - then when the context manager is finished, or an exception occurs, *__exit__()* is called.

2.3.7.1.6 __exit__

LockMgr.**__exit__**(*exc_type*, *exc_val*, *exc_tb*)

When the context manager is finished or an exception occurs, we unlock all locks that were created during the context manager session.

Exceptions

LockNotFound	Raised when a requested lock doesn't exist
Locked	Raised when a lock already exists with the given name

2.4 Database Models (lockmgr.models)

Functions

default_lock_expiry()

2.4.1 default_lock_expiry

lockmgr.models.**default_lock_expiry**()

Classes

Lock(name, locked_by, lock_process, ...)

2.4.2 Lock

class lockmgr.models.**Lock**(*name*, *locked_by*, *lock_process*, *locked_until*, *created_at*, *updated_at*)

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

exception DoesNotExist

exception MultipleObjectsReturned

property expired
Property - returns `True` if the Lock is past expiry (*locked_until*), otherwise False

property expires_in
The amount of time until this lock expires, as a `timedelta` - or `None` if it doesn't expire

property expires_seconds
The amount of seconds until this lock expires as integer seconds - or `None` if it doesn't expire

locked_by
Name of the node / app which created this lock

locked_until
Locks have an expiration time, to help avoid the issue of stuck locks, either due to forgetting to add cleanup code, or simply due to the app/server crashing before it can release the lock.

After a lock has expired, it's assumed that the lock is stale and needs to be removed, and the affected resources are safe to use.

name
Unique name of the lock, referring to what specific resource(s) is locked

2.4.2.1 Methods

Methods

get_next_by_created_at(*[, field, is_next])	
get_next_by_updated_at(*[, field, is_next])	
get_previous_by_created_at(*[, field, is_next])	
get_previous_by_updated_at(*[, field, is_next])	

2.4.2.1.1 get_next_by_created_at

`Lock.`**`get_next_by_created_at`**`(*, field=<django.db.models.fields.DateTimeField: created_at>, is_next=True, **kwargs)`

2.4.2.1.2 get_next_by_updated_at

`Lock.`**`get_next_by_updated_at`**`(*, field=<django.db.models.fields.DateTimeField: updated_at>, is_next=True, **kwargs)`

2.4.2.1.3 get_previous_by_created_at

`Lock.`**`get_previous_by_created_at`**`(*, field=<django.db.models.fields.DateTimeField: created_at>, is_next=False, **kwargs)`

2.4.2.1.4 get_previous_by_updated_at

`Lock.`**`get_previous_by_updated_at`**`(*, field=<django.db.models.fields.DateTimeField: updated_at>, is_next=False, **kwargs)`

2.4.2.2 Attributes

Attributes

created_at	A wrapper for a deferred-loading field.
expired	Property - returns `True` if the Lock is past expiry (*locked_until*), otherwise False
expires_in	The amount of time until this lock expires, as a `timedelta` - or `None` if it doesn't expire
expires_seconds	The amount of seconds until this lock expires as integer seconds - or `None` if it doesn't expire
lock_process	A wrapper for a deferred-loading field.
locked_by	Name of the node / app which created this lock

Continued on next page

Table 9 – continued from previous page

locked_until	Locks have an expiration time, to help avoid the issue of stuck locks, either due to forgetting to add cleanup code, or simply due to the app/server crashing before it can release the lock.
name	Unique name of the lock, referring to what specific resource(s) is locked
objects	
updated_at	A wrapper for a deferred-loading field.

2.4.2.2.1 created_at

`Lock.`**`created_at`**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

2.4.2.2.2 expired

property `Lock.`**`expired`**
Property - returns `True` if the Lock is past expiry (*locked_until*), otherwise False

2.4.2.2.3 expires_in

property `Lock.`**`expires_in`**
The amount of time until this lock expires, as a `timedelta` - or `None` if it doesn't expire

2.4.2.2.4 expires_seconds

property `Lock.`**`expires_seconds`**
The amount of seconds until this lock expires as integer seconds - or `None` if it doesn't expire

2.4.2.2.5 lock_process

`Lock.`**`lock_process`**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

2.4.2.2.6 locked_by

`Lock.`**`locked_by`**
Name of the node / app which created this lock

2.4.2.2.7 locked_until

`Lock.`**`locked_until`**
Locks have an expiration time, to help avoid the issue of stuck locks, either due to forgetting to add cleanup code, or simply due to the app/server crashing before it can release the lock.

After a lock has expired, it's assumed that the lock is stale and needs to be removed, and the affected resources are safe to use.

2.4.2.2.8 name

`Lock`**.name**
Unique name of the lock, referring to what specific resource(s) is locked

2.4.2.2.9 objects

`Lock.objects = <django.db.models.manager.Manager object>`

2.4.2.2.10 updated_at

`Lock`**.updated_at**
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class `lockmgr.models.`**Lock**(*name*, *locked_by*, *lock_process*, *locked_until*, *created_at*, *updated_at*)

exception DoesNotExist

exception MultipleObjectsReturned

property expired
Property - returns `True` if the Lock is past expiry (*`locked_until`*), otherwise False

property expires_in
The amount of time until this lock expires, as a `timedelta` - or `None` if it doesn't expire

property expires_seconds
The amount of seconds until this lock expires as integer seconds - or `None` if it doesn't expire

locked_by
Name of the node / app which created this lock

locked_until
Locks have an expiration time, to help avoid the issue of stuck locks, either due to forgetting to add cleanup code, or simply due to the app/server crashing before it can release the lock.

After a lock has expired, it's assumed that the lock is stale and needs to be removed, and the affected resources are safe to use.

name
Unique name of the lock, referring to what specific resource(s) is locked

2.5 Django Management Commands

While Django Lock Manager is primarily designed to be used programmatically within your Django application via the Python functions and methods - sometimes it can be useful to have administration commands to help when troubleshooting or experimenting with the lock manager.

Once you've *installed django-lockmgr*, including adding the app to your `INSTALLED_APPS` and ran the migrations, you'll be able to use the below commands via your Django application's `./manage.py` script.

See the below module links for documentation about each command.

lockmgr.management.commands.clear_lock	The `clear_lock` management command allows you to **delete one or more locks**, which may be useful for troubleshooting if you have stagnant locks.
lockmgr.management.commands.list_locks	The `list_locks` management command allows you to **view all current locks**, which may be useful for troubleshooting, e.g.
lockmgr.management.commands.reset_locks	The `reset_locks` management command allows you to **delete ALL LOCKS** set by django-lockmgr in your application's database, regardless of their expiration time, name, or who/what created them.
lockmgr.management.commands.set_lock	The `set_lock` management command allows you to **create / renew locks** using django-lockmgr from the command line.

2.5.1 clear_lock

The `clear_lock` management command allows you to **delete one or more locks**, which may be useful for troubleshooting if you have stagnant locks.

You may encounter stagnant locks if you're using locking functions such as *get_lock()*, instead of using the context manager *LockMgr* (or in rare events where your application exits unexpectedly, without time to cleanup locks).

Below is an excerpt from the manage.py help `./manage.py clear_lock --help`:

```
Releases one or more specified locks set using Privex's django-lockmgr package

positional arguments:
  locks                 One or more lockmgr lock names (as positional args) to␣
↪release the locks for
```

Example usage

```
# Create the two locks 'hello' and 'world'
./manage.py set_lock hello world

    Finished creating / renewing 2 locks.


# Delete the locks 'hello', 'world' and 'test' (it doesn't matter if some of the␣
↪passed locks don't exist)
./manage.py clear_lock hello world test

    Releasing lock hello from LockMgr...
    Lock hello has been removed (if it exists).

    Releasing lock world from LockMgr...
    Lock world has been removed (if it exists).

    Releasing lock test from LockMgr...
    Lock test has been removed (if it exists).
```

Classes

Command()

2.5.1.1 Command

class lockmgr.management.commands.clear_lock.**Command**

__init__()
Initialize self. See help(type(self)) for accurate signature.

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.1.1.1 Methods

Methods

__init__()	Initialize self.
add_arguments(parser)	Entry point for subclassed commands to add custom arguments.
handle(*args, **options)	The actual logic of the command.

2.5.1.1.1.1 __init__

Command.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

2.5.1.1.1.2 add_arguments

Command.**add_arguments**(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

2.5.1.1.1.3 handle

Command.**handle**(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.1.1.2 Attributes

Attributes

help

2.5.1.1.2.1 help

Command.**help = "Releases one or more specified locks set using Privex's django-lockmgr pacl**

class lockmgr.management.commands.clear_lock.**Command**

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.2 list_locks

The `list_locks` management command allows you to **view all current locks**, which may be useful for troubleshooting, e.g. checking whether or not a lock name really is locked, and what locked it

Below is an excerpt from the manage.py help `./manage.py list_locks --help`:

```
List all locks that were set using Privex's django-lockmgr package
```

There are no arguments nor switches available for this command.

Example usage

```
# Create the two locks 'hello' and 'world'
./manage.py set_lock hello world

    Finished creating / renewing 2 locks.


./manage.py list_locks

    There are currently 2 active locks using Privex Django-LockMgr

    ========================================================

    <Lock name='hello' locked_by='example.org' lock_process='None' locked_until='2019-
↪11-22 00:49:02.264729+00:00'>
    <Lock name='world' locked_by='example.org' lock_process='None' locked_until='2019-
↪11-22 00:49:02.267728+00:00'>

    ========================================================
```

Classes

Command()

2.5.2.1 Command

class lockmgr.management.commands.list_locks.**Command**

__init__()

Initialize self. See help(type(self)) for accurate signature.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.2.1.1 Methods

Methods

__init__()	Initialize self.
handle(*args, **options)	The actual logic of the command.

2.5.2.1.1.1 __init__

Command.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

2.5.2.1.1.2 handle

Command.**handle**(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.2.1.2 Attributes

Attributes

help

2.5.2.1.2.1 help

Command.**help = "List all locks that were set using Privex's django-lockmgr package"**

class lockmgr.management.commands.list_locks.**Command**

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.3 reset_locks

The `reset_locks` management command allows you to **delete ALL LOCKS** set by django-lockmgr in your application's database, regardless of their expiration time, name, or who/what created them.

You may encounter stagnant locks if you're using locking functions such as *`get_lock()`*, instead of using the context manager *`LockMgr`* (or in rare events where your application exits unexpectedly, without time to cleanup locks).

Below is an excerpt from the manage.py help `./manage.py reset_locks --help`:

```
Clears ALL locks that were set using Privex's django-lockmgr package

optional arguments:
  -h, --help            show this help message and exit
  -f, --force           Do not show warning / ask if you are sure before deleting ALL␣
↪locks
```

Example usage

First let's create two locks using *lockmgr.management.commands.set_lock*

```
# Create the two locks 'hello' and 'world'
./manage.py set_lock hello world

    Finished creating / renewing 2 locks.
```

Now we'll run `reset_locks` without any arguments. You can see it requires confirmation, since it can be dangerous to clear all locks if there are any applications running (or scheduled on a cron) that depend on the locks to avoid conflicts.

```
./manage.py reset_locks

    WARNING: You are about to clear ALL locks set using Privex LockMgr.
    You should only do this if you know what you're doing, and have made sure to stop␣
↪any running
    instances of your application, to ensure no conflicts are caused by removing ALL␣
↪LOCKS.


    The following 2 locks would be removed:

    ============================================================

    <Lock name='hello' locked_by='example.org' locked_until='2019-11-22 00:49:02.
↪264729+00:00'>
    <Lock name='world' locked_by='example.org' locked_until='2019-11-22 00:49:02.
↪267728+00:00'>

    ============================================================

    Are you SURE you want to clear all locks?
    Type YES in all capitals if you are sure > YES

    ============================================================

    Please wait... Removing all locks regardless of their status or expiration.

    A total of 2 lock rows were deleted. All locks should now be removed.

    ============================================================

    Finished clearing locks.

    ============================================================
```

Example 2 - Using the FORCE argument to skip the prompt

Let's re-create those locks, and now run `reset_locks` with `-f` (force).

```
# Create the two locks 'hello' and 'world'
./manage.py set_lock hello world

    Finished creating / renewing 2 locks.

# Run 'reset_locks' with option '-f' (force / do not ask for confirmation)
./manage.py reset_locks -f

    The following 2 locks would be removed:

    ========================================================

    <Lock name='hello' locked_by='example.org' locked_until='2019-11-22 00:58:00.
↪042322+00:00'>
    <Lock name='world' locked_by='example.org' locked_until='2019-11-22 00:58:00.
↪045513+00:00'>

    ========================================================

    Option 'force' (-f / --force) was specified. Skipping confirmation prompt.
    Please wait... Removing all locks regardless of their status or expiration.

    A total of 2 lock rows were deleted. All locks should now be removed.

    ========================================================

    Finished clearing locks.

    ========================================================
```

Classes

Command()

2.5.3.1 Command

class lockmgr.management.commands.reset_locks.**Command**

__init__()
Initialize self. See help(type(self)) for accurate signature.

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.3.1.1 Methods

Methods

__init__()	Initialize self.
add_arguments(parser)	Entry point for subclassed commands to add custom arguments.
handle(*args, **options)	The actual logic of the command.

2.5.3.1.1.1 __init__

Command.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

2.5.3.1.1.2 add_arguments

Command.**add_arguments**(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

2.5.3.1.1.3 handle

Command.**handle**(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.3.1.2 Attributes

Attributes

help

2.5.3.1.2.1 help

Command.**help = "Clears ALL locks that were set using Privex's django-lockmgr package"**

class lockmgr.management.commands.reset_locks.**Command**

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.4 set_lock

The `set_lock` management command allows you to **create / renew locks** using django-lockmgr from the command line.

If you don't specify any behaviour switches such as `--no-renew` or `--fail`, then set_lock will create any locks which aren't already locked, and renew any locks which are already locked.

Below is an excerpt from `./manage.py set_lock --help` (added at 21 Nov @ 9 PM UTC):

```
-h, --help            show this help message and exit
-n, --no-renew        Do not renew any locks which already exist
-r, --only-renew      Only renew existing locks, do not create new ones.
-k, --no-timeout      Never expire these locks (--timeout will be ignored). They must␣
↪be manually unlocked.
-t TIMEOUT, --timeout TIMEOUT     Lock timeout in seconds (default 600)
-e, --fail            Return an error (exit code 2) if one or more locks already␣
↪exist. (will not create/renew ANY
passed locks if one of the passed lock names exists)
```

The `-e` or `--fail` option can be a useful option when you're wanting to set multiple locks in unison, but you need to be sure that all of the locks are set at the same time - and if any of the locks already exist, any of the locks specified should not be created / be rolled back.

Below is an example of this special feature in use:

```
user@host ~ $ ./manage.py set_lock example
 > Lock example did not yet exist. Successfully locked 'example' - expiry: 2019-11-21␣
↪15:30:03.857412


user@host ~ $ ./manage.py set_lock -e hello world example

 > Lock hello did not yet exist. Successfully locked 'hello' - expiry: 2019-11-21␣
↪15:30:27.706378

 > Lock world did not yet exist. Successfully locked 'world' - expiry: 2019-11-21␣
↪15:30:27.709321

 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
↪!!!!!
 !!! An existing lock was found:
 !!!     <Lock name='example' locked_by='Chriss-iMac-Pro.local' locked_until='2019-11-
↪21 15:30:03.857412'>
 !!! As you have specified -e / --fail, any locks created during this session will␣
↪now be
 !!! rolled back for your safety.
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
↪!!!!!
 !!! Any locks created during this session should now have been removed.
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
↪!!!!!


user@host ~ $ ./manage.py list_locks

There are currently 1 active locks using Privex Django-LockMgr

==========================================================

<Lock name='example' locked_by='example.org' lock_process='None' locked_until='2019-
↪11-21 15:30:03.857412'>

==========================================================
```

Classes

Command()

2.5.4.1 Command

class lockmgr.management.commands.set_lock.**Command**

__init__()
Initialize self. See help(type(self)) for accurate signature.

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.4.1.1 Methods

Methods

__init__()	Initialize self.
add_arguments(parser)	Entry point for subclassed commands to add custom arguments.
handle(*args, **options)	The actual logic of the command.

2.5.4.1.1.1 __init__

Command.**__init__**()
Initialize self. See help(type(self)) for accurate signature.

2.5.4.1.1.2 add_arguments

Command.**add_arguments**(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

2.5.4.1.1.3 handle

Command.**handle**(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.5.4.1.2 Attributes

Attributes

help

2.5.4.1.2.1 help

`Command.help = "Add and/or renew locks using Privex's django-lockmgr package"`

class lockmgr.management.commands.set_lock.**Command**

add_arguments(*parser: django.core.management.base.CommandParser*)
Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)
The actual logic of the command. Subclasses must implement this method.

2.6 How to use the unit tests

This module contains test cases for Privex's Django Lock Manager (django-lockmgr).

2.6.1 Testing pre-requisites

- Install all core and development requirements listed in requirements.txt
- Either PostgreSQL or MySQL is recommended, however the default SQLite3 may or may not work.
- Python 3.7 or 3.8 is recommended at the time of writing this. See README.md in-case this has changed.

```
pip3 install -r requirements.txt
```

If you're using MySQL / Postgres, create a `.env` file in the root of the project, and enter the database connection details:

```
# If not specified, DB_USER and DB_NAME both default to 'lockmgr'
DB_USER=root
DB_NAME=lockmgr
# If not specified, then the DB user password defaults to blank
DB_PASS=
# If not specified, the DB host defaults to localhost, and the port as blank␣
↪(automatic depending on backend)
DB_HOST=localhost
DB_PORT=5432
# If not specified, the DB backend defaults to SQLite3 (stored in db.sqlite3 in root␣
↪of project)
# If you're using PostgreSQL:
DB_BACKEND=postgresql
# If you're using MySQL / MariaDB:
DB_BACKEND=mysql
```

2.6.2 Running the tests via Django Test Runner / Django-Nose

After installing the packages listed in `requirements.txt`, you should now be able to run the tests using Django's manage.py:

```
# Ensure you have all development requirements installed
user@host: ~/django-lockmgr $ pip3 install -r requirements.txt

# Then run the tests using manage.py
user@host: ~/django-lockmgr $ ./manage.py test

nosetests --verbosity=1
Creating test database for alias 'default'...
.............................
----------------------------------------------------------------------
Ran 28 tests in 20.291s

OK
Destroying test database for alias 'default'...
```

For more verbosity, simply add `--verbose` to the end of the command:

```
$ ./manage.py test --verbose

    nosetests --verbose --verbosity=2

    Creating test database for alias 'default' ('test_lockmgr')...
    Operations to perform:
      Synchronize unmigrated apps: django_nose
      Apply all migrations: lockmgr
    Synchronizing apps without migrations:
      Creating tables...
        Running deferred SQL...
    Running migrations:
      Applying lockmgr.0001_initial... OK

    Locking test_getlock then checking if Lock is raised when calling it again. ... ok
    Locking test_unlock, unlocking it, then lock/unlock again to confirm it was freed.
↪ ... ok
    Test that expired locks are correctly removed ... ok
    Test that LockMgr runs code with 'wait for lock expiry' when lock expires within␣
↪wait period ... ok
    Test that LockMgr raises Locked with 'wait for lock expiry' when lock still␣
↪locked after waiting period ... ok
    Locking test_lockmgr and test_lockmgr2 using LockMgr, then verifying the lock was␣
↪created ... ok
    Testing that LockMgr correctly removes Locks after an exception ... ok
    Renew an existing lock by lock name and confirm locked_until was increased ... ok
    Renew an existing lock by lock name with add_time=True and confirm locked_until␣
↪was increased ... ok
    Renew an existing lock by Lock object with add_time=True and confirm locked_until␣
↪was increased ... ok
    Renew an existing lock by Lock object and confirm locked_until was increased ...␣
↪ok
    Renew a non-existent lock by lock name and confirm LockNotFound is raised ... ok
    Renew a non-existent lock by lock name with create=True and confirm new lock is␣
↪created ... ok

    ----------------------------------------------------------------------
    Ran 13 tests in 10.106s

    OK
    Destroying test database for alias 'default' ('test_lockmgr')...
```

Copyright:

2.7 Unit Test List / Overview

test_lockmgr
test_lockmgr_class
test_renew

2.7.1 tests.test_lockmgr

Classes

TestLockMgrModule([methodName])	Tests which are related to the module-level functions in *lockmgr.lockmgr*

2.7.1.1 TestLockMgrModule

class tests.test_lockmgr.**TestLockMgrModule**(*methodName='runTest'*)

Tests which are related to the module-level functions in *lockmgr.lockmgr*

Tests related to the manager class *lockmgr.lockmgr.LockMgr* can be found in *tests.test_lockmgr_class*

__init__(*methodName='runTest'*)

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_getlock_clean()

Locking test_getlock then checking if Locked is raised when calling it again.

test_getlock_unlock()

Locking test_unlock, unlocking it, then lock/unlock again to confirm it was freed.

test_is_locked()

Locking test_is_locked then testing is_locked returns True for existing locks and False for non-existent.

test_lock_expiry()

Test that expired locks are correctly removed

test_lock_no_expiry()
Test that locks with None timeout aren't removed by clean_locks

test_lock_zero_expiry()
Test that locks with 0 timeout aren't removed by clean_locks

2.7.1.1.1 Methods

Methods

test_getlock_clean()	Locking test_getlock then checking if Locked is raised when calling it again.
test_getlock_unlock()	Locking test_unlock, unlocking it, then lock/unlock again to confirm it was freed.
test_is_locked()	Locking test_is_locked then testing is_locked returns True for existing locks and False for non-existent.
test_lock_expiry()	Test that expired locks are correctly removed
test_lock_no_expiry()	Test that locks with None timeout aren't removed by clean_locks
test_lock_zero_expiry()	Test that locks with 0 timeout aren't removed by clean_locks

2.7.1.1.1.1 test_getlock_clean

TestLockMgrModule.**test_getlock_clean**()
Locking test_getlock then checking if Locked is raised when calling it again.

2.7.1.1.1.2 test_getlock_unlock

TestLockMgrModule.**test_getlock_unlock**()
Locking test_unlock, unlocking it, then lock/unlock again to confirm it was freed.

2.7.1.1.1.3 test_is_locked

TestLockMgrModule.**test_is_locked**()
Locking test_is_locked then testing is_locked returns True for existing locks and False for non-existent.

2.7.1.1.1.4 test_lock_expiry

TestLockMgrModule.**test_lock_expiry**()
Test that expired locks are correctly removed

2.7.1.1.1.5 test_lock_no_expiry

TestLockMgrModule.**test_lock_no_expiry**()
Test that locks with None timeout aren't removed by clean_locks

2.7.1.1.1.6 test_lock_zero_expiry

TestLockMgrModule.**test_lock_zero_expiry**()
Test that locks with 0 timeout aren't removed by clean_locks

2.7.1.1.2 Attributes

Attributes

—

class tests.test_lockmgr.**TestLockMgrModule**(*methodName='runTest'*)
Tests which are related to the module-level functions in *lockmgr.lockmgr*

Tests related to the manager class *lockmgr.lockmgr.LockMgr* can be found in *tests.test_lockmgr_class*

test_getlock_clean()
Locking test_getlock then checking if Locked is raised when calling it again.

test_getlock_unlock()
Locking test_unlock, unlocking it, then lock/unlock again to confirm it was freed.

test_is_locked()
Locking test_is_locked then testing is_locked returns True for existing locks and False for non-existent.

test_lock_expiry()
Test that expired locks are correctly removed

test_lock_no_expiry()
Test that locks with None timeout aren't removed by clean_locks

test_lock_zero_expiry()
Test that locks with 0 timeout aren't removed by clean_locks

2.7.2 tests.test_lockmgr_class

Classes

TestLockMgrClass([methodName])	Tests which are related to the manager class *lockmgr.lockmgr.LockMgr*

2.7.2.1 TestLockMgrClass

class tests.test_lockmgr_class.**TestLockMgrClass**(*methodName='runTest'*)
Tests which are related to the manager class *lockmgr.lockmgr.LockMgr*

Tests related to the module-level functions in *lockmgr.lockmgr* can be found in *tests.test_lockmgr*

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_lock_wait()
Test that LockMgr runs code with 'wait for lock expiry' when lock expires within wait period

test_lock_wait_timeout()
Test that LockMgr raises Locked with 'wait for lock expiry' when lock still locked after waiting period

test_lockmgr()
Locking test_lockmgr and test_lockmgr2 using LockMgr, then verifying the lock was created

test_lockmgr_except()
Testing that LockMgr correctly removes Locks after an exception

2.7.2.1.1 Methods

Methods

test_lock_wait()	Test that LockMgr runs code with 'wait for lock expiry' when lock expires within wait period
test_lock_wait_timeout()	Test that LockMgr raises Locked with 'wait for lock expiry' when lock still locked after waiting period
test_lockmgr()	Locking test_lockmgr and test_lockmgr2 using LockMgr, then verifying the lock was created
test_lockmgr_except()	Testing that LockMgr correctly removes Locks after an exception

2.7.2.1.1.1 test_lock_wait

TestLockMgrClass.**test_lock_wait**()
Test that LockMgr runs code with 'wait for lock expiry' when lock expires within wait period

2.7.2.1.1.2 test_lock_wait_timeout

TestLockMgrClass.**test_lock_wait_timeout**()
Test that LockMgr raises Locked with 'wait for lock expiry' when lock still locked after waiting period

2.7.2.1.1.3 test_lockmgr

TestLockMgrClass.**test_lockmgr**()
Locking test_lockmgr and test_lockmgr2 using LockMgr, then verifying the lock was created

2.7.2.1.1.4 test_lockmgr_except

TestLockMgrClass.**test_lockmgr_except**()
Testing that LockMgr correctly removes Locks after an exception

2.7.2.1.2 Attributes

Attributes

—

class tests.test_lockmgr_class.**TestLockMgrClass**(*methodName='runTest'*)
Tests which are related to the manager class *lockmgr.lockmgr.LockMgr*

Tests related to the module-level functions in *lockmgr.lockmgr* can be found in *tests.test_lockmgr*

test_lock_wait()
Test that LockMgr runs code with 'wait for lock expiry' when lock expires within wait period

test_lock_wait_timeout()
Test that LockMgr raises Locked with 'wait for lock expiry' when lock still locked after waiting period

test_lockmgr()
Locking test_lockmgr and test_lockmgr2 using LockMgr, then verifying the lock was created

test_lockmgr_except()
Testing that LockMgr correctly removes Locks after an exception

2.7.3 tests.test_renew

Classes

TestLockRenew([methodName])

2.7.3.1 TestLockRenew

class tests.test_renew.**TestLockRenew**(*methodName='runTest'*)

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_lockmgr_renew_expired()
Renew an expired main lock within a LockMgr 'with' statement, confirm time was added to the lock expiry

test_lockmgr_renew_main()
Renew the main lock within a LockMgr 'with' statement, confirm appropriate time was added to the lock

test_renew_existing_name()
Renew an existing lock by lock name and confirm locked_until was increased

test_renew_existing_name_add_time()
Renew an existing lock by lock name with add_time=True and confirm locked_until was increased

test_renew_existing_object_add_time()
Renew an existing lock by Lock object with add_time=True and confirm locked_until was increased

test_renew_lock_object()
Renew an existing lock by Lock object and confirm locked_until was increased

test_renew_non_existing_name()
Renew a non-existent lock by lock name and confirm LockNotFound is raised

test_renew_non_existing_name_create()
Renew a non-existent lock by lock name with create=True and confirm new lock is created

test_renew_shorter_expiration()
Renew a lock with a shorter expiration time than it already has. Test the expiration time doesn't drop.

test_renew_shorter_expiration_add_time()
Renew a lock with a shorter expiration seconds (but with add_time=True). Test expiration time increases.

2.7.3.1.1 Methods

Methods

test_lockmgr_renew_expired()	Renew an expired main lock within a LockMgr 'with' statement, confirm time was added to the lock expiry
test_lockmgr_renew_main()	Renew the main lock within a LockMgr 'with' statement, confirm appropriate time was added to the lock
test_renew_existing_name()	Renew an existing lock by lock name and confirm locked_until was increased
test_renew_existing_name_add_time()	Renew an existing lock by lock name with add_time=True and confirm locked_until was increased
test_renew_existing_object_add_time()	Renew an existing lock by Lock object with add_time=True and confirm locked_until was increased
test_renew_lock_object()	Renew an existing lock by Lock object and confirm locked_until was increased
test_renew_non_existing_name()	Renew a non-existent lock by lock name and confirm LockNotFound is raised
test_renew_non_existing_name_create()	Renew a non-existent lock by lock name with create=True and confirm new lock is created
test_renew_shorter_expiration()	Renew a lock with a shorter expiration time than it already has.
test_renew_shorter_expiration_add_time()	Renew a lock with a shorter expiration seconds (but with add_time=True).

2.7.3.1.1.1 test_lockmgr_renew_expired

TestLockRenew.**test_lockmgr_renew_expired**()
Renew an expired main lock within a LockMgr 'with' statement, confirm time was added to the lock expiry

2.7.3.1.1.2 test_lockmgr_renew_main

TestLockRenew.**test_lockmgr_renew_main**()
Renew the main lock within a LockMgr 'with' statement, confirm appropriate time was added to the lock

2.7.3.1.1.3 test_renew_existing_name

TestLockRenew.**test_renew_existing_name**()
Renew an existing lock by lock name and confirm locked_until was increased

2.7.3.1.1.4 test_renew_existing_name_add_time

TestLockRenew.**test_renew_existing_name_add_time**()
Renew an existing lock by lock name with add_time=True and confirm locked_until was increased

2.7.3.1.1.5 test_renew_existing_object_add_time

TestLockRenew.**test_renew_existing_object_add_time**()
Renew an existing lock by Lock object with add_time=True and confirm locked_until was increased

2.7.3.1.1.6 test_renew_lock_object

TestLockRenew.**test_renew_lock_object**()
Renew an existing lock by Lock object and confirm locked_until was increased

2.7.3.1.1.7 test_renew_non_existing_name

TestLockRenew.**test_renew_non_existing_name**()
Renew a non-existent lock by lock name and confirm LockNotFound is raised

2.7.3.1.1.8 test_renew_non_existing_name_create

TestLockRenew.**test_renew_non_existing_name_create**()
Renew a non-existent lock by lock name with create=True and confirm new lock is created

2.7.3.1.1.9 test_renew_shorter_expiration

TestLockRenew.**test_renew_shorter_expiration**()
Renew a lock with a shorter expiration time than it already has. Test the expiration time doesn't drop.

2.7.3.1.1.10 test_renew_shorter_expiration_add_time

TestLockRenew.**test_renew_shorter_expiration_add_time**()
Renew a lock with a shorter expiration seconds (but with add_time=True). Test expiration time increases.

2.7.3.1.2 Attributes

Attributes

---

class tests.test_renew.**TestLockRenew**(*methodName='runTest'*)

test_lockmgr_renew_expired()
Renew an expired main lock within a LockMgr 'with' statement, confirm time was added to the lock expiry

test_lockmgr_renew_main()

Renew the main lock within a LockMgr 'with' statement, confirm appropriate time was added to the lock

test_renew_existing_name()
Renew an existing lock by lock name and confirm locked_until was increased

test_renew_existing_name_add_time()
Renew an existing lock by lock name with add_time=True and confirm locked_until was increased

test_renew_existing_object_add_time()
Renew an existing lock by Lock object with add_time=True and confirm locked_until was increased

test_renew_lock_object()
Renew an existing lock by Lock object and confirm locked_until was increased

test_renew_non_existing_name()
Renew a non-existent lock by lock name and confirm LockNotFound is raised

test_renew_non_existing_name_create()
Renew a non-existent lock by lock name with create=True and confirm new lock is created

test_renew_shorter_expiration()
Renew a lock with a shorter expiration time than it already has. Test the expiration time doesn't drop.

test_renew_shorter_expiration_add_time()
Renew a lock with a shorter expiration seconds (but with add_time=True). Test expiration time increases.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

t

INDEX

Symbols

A

C

D

E

G

H

I

L

M

N

O

R

S

T

U

W