Sherlock Documentation

Release 0.3.0

Vaidik Kapoor

May 05, 2015

Contents

sherlock is a library that provides easy-to-use distributed inter-process locks and also allows you to choose a backend of your choice for lock synchronization.

CHAPTER 1

Overview

When you are working with resources which are accessed by multiple services or distributed services, more than often you need some kind of locking mechanism to make it possible to access some resources at a time.

Distributed Locks or Mutexes can help you with this. `sherlock` provides the exact same facility, with some extra goodies. It provides an easy-to-use API that resembles standard library's *threading.Lock* semantics.

Apart from this, `sherlock` gives you the flexibilty of using a backend of your choice for managing locks.

`sherlock` also makes it simple for you to extend `sherlock` to use backends that are not supported.

1.1 Features

- API similar to standard library's *threading.Lock*.
- Support for With statement, to cleanly acquire and release locks.
- Backend agnostic: supports Redis, Memcached and Etcd as choice of backends.
- Extendable: can be easily extended to work with any other of backend of choice by extending base lock class. Read *Extending*.

1.2 Supported Backends and Client Libraries

Following client libraries are supported for every supported backend:

- Redis: redis-py
- Memcached: pylibmc
- Etcd: python-etcd

As of now, only the above mentioned libraries are supported. Although `sherlock` takes custom client objects so that you can easily provide settings that you want to use for that backend store, but `sherlock` also checks if the provided client object is an instance of the supported clients and accepts client objects which pass this check, even if the APIs are the same. `sherlock` might get rid of this issue later, if need be and if there is a demand for that.

CHAPTER 2

Installation

Installation is simple.

```
pip install sherlock
```

Note: *sherlock* will install all the client libraries for all the supported backends.

CHAPTER 3

Basic Usage

sherlock is simple to use as at the API and semantics level, it tries to conform to standard library's `threading.Lock` APIs.

```
import sherlock
from sherlock import Lock

# Configure :mod:`sherlock`'s locks to use Redis as the backend,
# never expire locks and retry acquiring an acquired lock after an
# interval of 0.1 second.
sherlock.configure(backend=sherlock.backends.REDIS,
                   expire=None,
                   retry_interval=0.1)

# Note: configuring sherlock to use a backend does not limit you
# another backend at the same time. You can import backend specific locks
# like RedisLock, MCLock and EtcdLock and use them just the same way you
# use a generic lock (see below). In fact, the generic Lock provided by
# sherlock is just a proxy that uses these specific locks under the hood.

# acquire a lock called my_lock
lock = Lock('my_lock')

# acquire a blocking lock
lock.acquire()

# check if the lock has been acquired or not
lock.locked() == True

# release the lock
lock.release()
```

3.1 Support for `with` statement

```
# using with statement
with Lock('my_lock'):
    # do something constructive with your locked resource here
    pass
```

3.2 Blocking and Non-blocking API

```
# acquire non-blocking lock
lock1 = Lock('my_lock')
lock2 = Lock('my_lock')

# successfully acquire lock1
lock1.acquire()

# try to acquire lock in a non-blocking way
lock2.acquire(False) == True # returns False

# try to acquire lock in a blocking way
lock2.acquire() # blocks until lock is acquired to timeout happens
```

3.3 Using two backends at the same time

Configuring *sherlock* to use a backend does not limit you from using another backend at the same time. You can import backend specific locks like RedisLock, MCLock and EtcdLock and use them just the same way you use a generic lock (see below). In fact, the generic Lock provided by *sherlock* is just a proxy that uses these specific locks under the hood.

```
import sherlock
from sherlock import Lock

# Configure :mod:`sherlock`'s locks to use Redis as the backend
sherlock.configure(backend=sherlock.backends.REDIS)

# Acquire a lock called my_lock, this lock uses Redis
lock = Lock('my_lock')

# Now acquire locks in Memcached
from sherlock import MCLock
mclock = MCLock('my_mc_lock')
mclock.acquire()
```

CHAPTER 4

Tests

To run all the tests (including integration), you have to make sure that all the databases are running. Make sure all the services are running:

```
# memcached
memcached

# redis-server
redis-server

# etcd (etcd is probably not available as package, here is the simplest way
# to run it).
wget https://github.com/coreos/etcd/releases/download/<version>/etcd-<version>-<platform>.tar.gz
tar -zxvf etcd-<version>-<platform>.gz
./etcd-<version>-<platform>/etcd
```

Run tests like so:

```
python setup.py test
```

CHAPTER 5

Documentation

Available here.

CHAPTER 6

Roadmap

- Support for Zookeeper as backend.
- Support for Gevent, Multithreading and Multiprocessing.

CHAPTER 7

License

See LICENSE.

In short: This is an open-source project and exists in the public domain for anyone to modify and use it. Just be nice and attribute the credits wherever you can. :)

CHAPTER 8

Distributed Locking in Other Languages

- NodeJS - https://github.com/thedeveloper/warlock

CHAPTER 9

Contents

9.1 API Documentation

9.1.1 Configuration

sherlock can be globally configured to set a lot of defaults using the *sherlock.configure()* function. The configuration set using *sherlock.configure()* will be used as the default for all the lock objects.

sherlock.**configure**(***kwargs*)

Set basic global configuration for *sherlock*.

Parameters

- **backend** – global choice of backend. This backend will be used for managing locks by *sherlock.Lock* class objects.
- **client** – global client object to use to connect with backend store. This client object will be used to connect to the backend store by *sherlock.Lock* class instances. The client object must be a valid object of the client library. If the backend has been configured using the *backend* parameter, the custom client object must belong to the same library that is supported for that backend. If the backend has not been set, then the custom client object must be an instance of a valid supported client. In that case, *sherlock* will set the backend by introspecting the type of provided client object.
- **namespace** (*str*) – provide global namespace
- **expire** (*float*) – provide global expiration time. If expicitly set to *None*, lock will not expire.
- **timeout** (*float*) – provide global timeout period
- **retry_interval** (*float*) – provide global retry interval

Basic Usage:

```
>>> import sherlock
>>> from sherlock import Lock
>>>
>>> # Configure sherlock to use Redis as the backend and the timeout for
>>> # acquiring locks equal to 20 seconds.
>>> sherlock.configure(timeout=20, backend=sherlock.backends.REDIS)
>>>
>>> import redis
>>> redis_client = redis.StrictRedis(host='X.X.X.X', port=6379, db=1)
>>> sherlock.configure(client=redis_client)
```

Understanding the configuration parameters

sherlock.configure() accepts certain configuration patterns which are individually explained in this section. Some of these configurations are global configurations and cannot be overriden while others can be overriden at individual lock levels.

backend

The *backend* parameter allows you to set which backend you would like to use with the *sherlock.Lock* class. When set to a particular backend, instances of *sherlock.Lock* will use this backend for lock synchronization.

Basic Usage:

```
>>> import sherlock
>>> sherlock.configure(backend=sherlock.backends.REDIS)
```

Note: this configuration cannot be overriden at the time of creating a class object.

Available Backends To set the *backend* global configuration, you would have to choose one from the defined backends. The defined backends are:

- Etcd: `sherlock.backends.ETCD`
- Memcache: `sherlock.backends.MEMCACHE`
- Redis: `sherlock.backends.REDIS`

client

The *client* parameter allows you to set a custom clien object which *sherlock* can use for connecting to the backend store. This gives you the flexibility to connect to the backend store from the client the way you want. The provided custom client object must be a valid client object of the supported client libraries. If the global *backend* has been set, then the provided custom client object must be an instance of the client library supported by that backend. If the backend has not been set, then the custom client object must be an instance of a valid supported client. In this case, *sherlock* will set the backend by instrospecting the type of the provided client object.

The global default client object set using the *client* parameter will be used only by *sherlock.Lock* instances. Other *Backend Specific Locks* will either use the provided client object at the time of instantiating the lock object of their types or will default to creating a simple client object by themselves for their backend store, which will assume that their backend store is running on localhost.

Note: this configuration cannot be overriden at the time of creating a class object.

Example:

```
>>> import redis
>>> import sherlock
>>>
>>> # Configure just the backend
>>> sherlock.configure(backend=sherlock.backends.REDIS)
>>>
```

```
>>> # Configure the global client object. This sets the client for all the
>>> # locks.
>>> sherlock.configure(client=redis.StrictRedis())
```

And when the provided client object does not match the supported library for the set backend:

```
>>> import etcd
>>> import sherlock
>>>
>>> sherlock.configure(backend=sherlock.backends.REDIS)
>>> sherlock.configure(client=etcd.Client())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 148, in configure
    _configuration.update(**kwargs)
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 250, in update
    setattr(self, key, val)
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 214, in client
    self.backend['name']))
ValueError: Only a client of the redis library can be used when using REDIS as the backend store opti
```

And when the backend is not configured:

```
>>> import redis
>>> import sherlock
>>>
>>> # Congiure just the client, this will configure the backend to
>>> # sherlock.backends.REDIS automatically.
>>> sherlock.configure(client=redis.StrictRedis())
```

And when the client object passed as argument for client parameter is not a valid client object at all:

```
>>> import sherlock
>>> sherlock.configure(client=object())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 148, in configure
    _configuration.update(**kwargs)
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 250, in update
    setattr(self, key, val)
  File "/Users/vkapoor/Development/sherlock/sherlock/__init__.py", line 221, in client
    raise ValueError('The provided object is not a valid client'
ValueError: The provided object is not a valid client object. Client objects can only be instances of
```

namespace

The `namespace` parameter allows you to configure *sherlock* to set keys for synchronizing locks with a namespace so that if you are using the same datastore for something else, the keys set by locks don't conflict with your other keys set by your application.

In case of Redis and Memcached, the name of your locks are prepended with `NAMESPACE_` (where `NAMESPACE` is the namespace set by you). In case of Etcd, a directory with the name same as the `NAMESPACE` you provided is created and the locks are created in that directory.

By default, *sherlock* does not namespace the keys set for locks.

Note: this configuration can be overriden at the time of creating a class object.

---

expire

This parameter can be used to set the expiry of locks. When set to `None`, the locks will never expire.

This parameter's value defaults to `60 seconds`.

Note: this configuration can be overriden at the time of creating a class object.

Example:

```
>>> import sherlock
>>> import time
>>>
>>> # Configure locks to expire after 2 seconds
>>> sherlock.configure(expire=2)
>>>
>>> lock = sherlock.Lock('my_lock')
>>>
>>> # Acquire the lock
>>> lock.acquire()
True
>>>
>>> # Sleep for 2 seconds to let the lock expire
>>> time.sleep(2)
>>>
>>> # Acquire the lock
>>> lock.acquire()
True
```

timeout

This parameter can be used to set after how much time should *sherlock* stop trying to acquire an already acquired lock.

This parameter's value defaults to `10 seconds`.

Note: this configuration can be overriden at the time of creating a class object.

Example:

```
>>> import sherlock
>>>
>>> # Configure locks to timeout after 2 seconds while trying to acquire an
>>> # already acquired lock
>>> sherlock.configure(timeout=2, expire=10)
>>>
>>> lock = sherlock.Lock('my_lock')
>>>
>>> # Acquire the lock
>>> lock.acquire()
True
>>>
>>> # Acquire the lock again and let the timeout elapse
>>> lock.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sherlock/lock.py", line 170, in acquire
```

```
    'lock.' % self.timeout)
sherlock.lock.LockTimeoutException: Timeout elapsed after 2 seconds while trying to acquiring lock.
```

retry_interval

This parameter can be used to set after how much time should *sherlock* try to acquire lock after failing to acquire it. For example, a log has already been acquired, then when another lock object tries to acquire the same lock, it fails. This parameter sets the time interval for which we should sleep before retrying to acquire the lock.

This parameter can be set to 0 to continuously try acquiring the lock. But that will also mean that you are bombarding your datastore with requests one after another.

This parameter's value defaults to `0.1 seconds (100 milliseconds)`.

Note: this configuration can be overriden at the time of creating a class object.

9.1.2 Generic Locks

sherlock provides generic locks that can be globally configured to use a specific backend, so that most of your application code does not have to care about which backend you are using for lock synchronization and makes it easy to change backend without changing a ton of code.

class `sherlock.`**Lock**(*lock_name*, ***kwargs*)

A general lock that inherits global coniguration and provides locks with the configured backend.

Note: to use `Lock` class, you must configure the global backend to use a particular backend. If the global backend is not set, calling any method on instances of `Lock` will throw exceptions.

Basic Usage:

```
>>> import sherlock
>>> from sherlock import Lock
>>>
>>> sherlock.configure(sherlock.backends.REDIS)
>>>
>>> # Create a lock instance
>>> lock = Lock('my_lock')
>>>
>>> # Acquire a lock in Redis running on localhost
>>> lock.acquire()
True
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
True
>>>
>>> # Release the acquired lock
>>> lock.release()
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
False
>>>
>>> import redis
>>> redis_client = redis.StrictRedis(host='X.X.X.X', port=6379, db=2)
```

```
>>> sherlock.configure(client=redis_client)
>>>
>>> # Acquire a lock in Redis running on X.X.X.X:6379
>>> lock.acquire()
>>>
>>> lock.locked()
True
>>>
>>> # Acquire a lock using the with_statement
>>> with Lock('my_lock') as lock:
...     # do some stuff with your acquired resource
...     pass
```

Parameters

- **lock_name** (*str*) – name of the lock to uniquely identify the lock between processes.
- **namespace** (*str*) – Optional namespace to namespace lock keys for your application in order to avoid conflicts.
- **expire** (*float*) – set lock expiry time. If explicitly set to *None*, lock will not expire.
- **timeout** (*float*) – set timeout to acquire lock
- **retry_interval** (*float*) – set interval for trying acquiring lock after the timeout interval has elapsed.

Note: this Lock object does not accept a custom lock backend store client object. It instead uses the global custom client object.

---

acquire(*blocking=True*)
Acquire a lock, blocking or non-blocking.

Parameters **blocking** (*bool*) – acquire a lock in a blocking or non-blocking fashion. Defaults to True.

Returns if the lock was successfully acquired or not

Return type bool

locked()
Return if the lock has been acquired or not.

Returns True indicating that a lock has been acquired ot a shared resource is locked.

Return type bool

release()
Release a lock.

9.1.3 Backend Specific Locks

sherlock provides backend specific Lock classes as well which can be optionally used to use different backend than a globally configured backend. These locks have the same interface and semantics as *sherlock.Lock*.

Redis based Locks

class sherlock.**RedisLock**(*lock_name*, ***kwargs*)

Implementation of lock with Redis as the backend for synchronization.

Basic Usage:

```
>>> import redis
>>> import sherlock
>>> from sherlock import RedisLock
>>>
>>> # Global configuration of defaults
>>> sherlock.configure(expire=120, timeout=20)
>>>
>>> # Create a lock instance
>>> lock = RedisLock('my_lock')
>>>
>>> # Acquire a lock in Redis, global backend and client configuration need
>>> # not be configured since we are using a backend specific lock.
>>> lock.acquire()
True
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
True
>>>
>>> # Release the acquired lock
>>> lock.release()
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
False
>>>
>>> # Use this client object
>>> client = redis.StrictRedis()
>>>
>>> # Create a lock instance with custom client object
>>> lock = RedisLock('my_lock', client=client)
>>>
>>> # To override the defaults, just past the configurations as parameters
>>> lock = RedisLock('my_lock', client=client, expire=1, timeout=5)
>>>
>>> # Acquire a lock using the with_statement
>>> with RedisLock('my_lock') as lock:
...     # do some stuff with your acquired resource
...     pass
```

Parameters

- **lock_name** (*str*) – name of the lock to uniquely identify the lock between processes.
- **namespace** (*str*) – Optional namespace to namespace lock keys for your application in order to avoid conflicts.
- **expire** (*float*) – set lock expiry time. If explicitly set to *None*, lock will not expire.
- **timeout** (*float*) – set timeout to acquire lock
- **retry_interval** (*float*) – set interval for trying acquiring lock after the timeout interval has elapsed.

- **client** – supported client object for the backend of your choice.

acquire(*blocking=True*)
Acquire a lock, blocking or non-blocking.

Parameters **blocking** (*bool*) – acquire a lock in a blocking or non-blocking fashion. Defaults to True.

Returns if the lock was successfully acquired or not

Return type bool

locked()
Return if the lock has been acquired or not.

Returns True indicating that a lock has been acquired ot a shared resource is locked.

Return type bool

release()
Release a lock.

Etcd based Locks

class sherlock.**EtcdLock**(*lock_name*, ***kwargs*)
Implementation of lock with Etcd as the backend for synchronization.

Basic Usage:

```
>>> import etcd
>>> import sherlock
>>> from sherlock import EtcdLock
>>>
>>> # Global configuration of defaults
>>> sherlock.configure(expire=120, timeout=20)
>>>
>>> # Create a lock instance
>>> lock = EtcdLock('my_lock')
>>>
>>> # Acquire a lock in Etcd, global backend and client configuration need
>>> # not be configured since we are using a backend specific lock.
>>> lock.acquire()
True
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
True
>>>
>>> # Release the acquired lock
>>> lock.release()
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
False
>>>
>>> # Use this client object
>>> client = etcd.Client()
>>>
>>> # Create a lock instance with custom client object
>>> lock = EtcdLock('my_lock', client=client)
```

```
>>>
>>> # To override the defaults, just past the configurations as parameters
>>> lock = EtcdLock('my_lock', client=client, expire=1, timeout=5)
>>>
>>> # Acquire a lock using the with_statement
>>> with EtcdLock('my_lock') as lock:
...     # do some stuff with your acquired resource
...     pass
```

Parameters

- **lock_name** (*str*) – name of the lock to uniquely identify the lock between processes.
- **namespace** (*str*) – Optional namespace to namespace lock keys for your application in order to avoid conflicts.
- **expire** (*float*) – set lock expiry time. If explicitly set to *None*, lock will not expire.
- **timeout** (*float*) – set timeout to acquire lock
- **retry_interval** (*float*) – set interval for trying acquiring lock after the timeout interval has elapsed.
- **client** – supported client object for the backend of your choice.

acquire(*blocking=True*)
Acquire a lock, blocking or non-blocking.

Parameters **blocking** (*bool*) – acquire a lock in a blocking or non-blocking fashion. Defaults to True.

Returns if the lock was successfully acquired or not

Return type bool

locked()
Return if the lock has been acquired or not.

Returns True indicating that a lock has been acquired ot a shared resource is locked.

Return type bool

release()
Release a lock.

Memcached based Locks

class sherlock.**MCLock**(*lock_name*, ***kwargs*)
Implementation of lock with Memcached as the backend for synchronization.

Basic Usage:

```
>>> import pylibmc
>>> import sherlock
>>> from sherlock import MCLock
>>>
>>> # Global configuration of defaults
>>> sherlock.configure(expire=120, timeout=20)
>>>
>>> # Create a lock instance
>>> lock = MCLock('my_lock')
```

```
>>>
>>> # Acquire a lock in Memcached, global backend and client configuration
>>> # need not be configured since we are using a backend specific lock.
>>> lock.acquire()
True
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
True
>>>
>>> # Release the acquired lock
>>> lock.release()
>>>
>>> # Check if the lock has been acquired
>>> lock.locked()
False
>>>
>>> # Use this client object
>>> client = pylibmc.Client(['X.X.X.X'], binary=True)
>>>
>>> # Create a lock instance with custom client object
>>> lock = MCLock('my_lock', client=client)
>>>
>>> # To override the defaults, just past the configurations as parameters
>>> lock = MCLock('my_lock', client=client, expire=1, timeout=5)
>>>
>>> # Acquire a lock using the with_statement
>>> with MCLock('my_lock') as lock:
...     # do some stuff with your acquired resource
...     pass
```

Parameters

- **lock_name** (*str*) – name of the lock to uniquely identify the lock between processes.
- **namespace** (*str*) – Optional namespace to namespace lock keys for your application in order to avoid conflicts.
- **expire** (*float*) – set lock expiry time. If explicitly set to *None*, lock will not expire.
- **timeout** (*float*) – set timeout to acquire lock
- **retry_interval** (*float*) – set interval for trying acquiring lock after the timeout interval has elapsed.
- **client** – supported client object for the backend of your choice.

acquire(*blocking=True*)

Acquire a lock, blocking or non-blocking.

Parameters **blocking** (*bool*) – acquire a lock in a blocking or non-blocking fashion. Defaults to True.

Returns if the lock was successfully acquired or not

Return type bool

locked()

Return if the lock has been acquired or not.

Returns True indicating that a lock has been acquired ot a shared resource is locked.

Return type bool

release()
Release a lock.

9.2 Indices and tables

- genindex
- modindex
- search

9.3 Extending

sherlock can be easily extended to work with any backend. You just have to register your lock's implementation with *sherlock* and you will be able to use your lock with the backend of your choice in your project.

9.3.1 Registration

Custom locks can be registered using the following API:

backends.**register**(*name*, *lock_class*, *library*, *client_class*, *default_args=()*, *default_kwargs={}*)
Register a custom backend.

Parameters

- **name** (*str*) – Name of the backend by which you would want to refer this backend in your code.
- **lock_class** (*class*) – the sub-class of sherlock.lock.BaseLock that you have implemented. The reference to your implemented lock class will be used by *sherlock.Lock* proxy to use your implemented class when you globally set that the choice of backend is the one that has been implemented by you.
- **library** (*str*) – dependent client library that this implementation makes use of.
- **client_class** – the client class or valid type which you use to connect the datastore. This is used by the configure() function to validate that the object provided for the *client* parameter is actually an instance of this class.
- **default_args** (*tuple*) – default arguments that need to passed to create an instance of the callable passed to *client_class* parameter.
- **default_kwargs** (*dict*) – default keyword arguments that need to passed to create an instance of the callable passed to *client_class* parameter.

Usage:

```
>>> import some_db_client
>>> class MyLock(sherlock.lock.BaseLock):
...     # your implementation comes here
...     pass
>>>
>>> sherlock.configure(name='Mylock',
...                    lock_class=MyLock,
```

```
    ...                     library='some_db_client',
    ...                     client_class=some_db_client.Client,
    ...                     default_args=('localhost:1234'),
    ...                     default_kwargs=dict(connection_pool=6))
```

9.3.2 Example

Here is an example of implementing a custom lock that uses Elasticsearch as backend.

Note: You may distributed your custom lock implementation as package if you please. Just make sure that you add *sherlock* as a dependency.

The following code goes in a module called `sherlock_es.py`.

```
import elasticsearch
import sherlock
import uuid

from elasticsearch import Elasticsearch
from sherlock import LockException


class ESLock(sherlock.lock.BaseLock):
    def __init__(self, lock_name, **kwargs):
        super(ESLock, self).__init__(lock_name, **kwargs)

        if self.client is None:
            self.client = Elasticsearch(hosts=['localhost:9200'])

        self._owner = None

    def _acquire(self):
        owner = uuid.uuid4().hex

        try:
            self.client.get(index='sherlock', doc_type='locks',
                            id=self.lock_name)
        except elasticsearch.NotFoundError, err:
            self.client.index(index='sherlock', doc_type='locks',
                              id=self.lock_name, body=dict(owner=owner))
            self._owner = owner
            return True
        else:
            return False

    def _release(self):
        if self._owner is None:
            raise LockException('Lock was not set by this process.')

        try:
            resp = self.client.get(index='sherlock', doc_type='locks',
                                   id=self.lock_name)
            if resp['_source']['owner'] == self._owner:
                self.client.delete(index='sherlock', doc_type='locks',
                                   id=self.lock_name)
```

```
            else:
                raise LockException('Lock could not be released because it '
                                    'was not acquired by this process.')
        except elasticsearch.NotFoundError, err:
            raise LockException('Lock could not be released as it has not '
                                'been acquired.')

    @property
    def _locked(self):
        try:
            self.client.get(index='sherlock', doc_type='locks',
                            id=self.lock_name)
            return True
        except elasticsearch.NotFoundError, err:
            return False


# Register the custom lock with sherlock
sherlock.backends.register(name='ES',
                           lock_class=ESLock,
                           library='elasticsearch',
                           client_class=Elasticsearch,
                           default_args=(),
                           default_kwargs={
                               'hosts': ['localhost:9200'],
                           })
```

Our module can be used like so:

```
import sherlock
import sherlock_es

# Notice that ES is available as backend now
sherlock.configure(backend=sherlock.backends.ES)

lock1 = sherlock.Lock('test1')
lock1.acquire() # True

lock2 = sherlock_es.ESLock('test2')
lock2.acquire() # True
```

9.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

S

A

C

E

L

M

R

S