pymesync Documentation

Release 0.0.1

OSU Open Source Lab

June 03, 2017

Contents

Contents:

CHAPTER 1

pymesync - Communicate with a TimeSync API

Contents

This module provides an interface to communicate with an implementation of the OSU Open Source Lab's TimeSync API. An implementation of the TimeSync API, built in Node.js, can be found at github.com/osuosl/timesync-node.

This module allows users to

- Authenticate a pymesync object with a TimeSync implementation (**authenticate()**)
- Send times, projects, activities, and users to TimeSync (**create_time()**, **create_project()**, **create_activity()**, **create_user()**),
- Update times, projects, activities, and users (**update_time()**, **update_project()**, **update_activity()**, **update_user()**)
- Get one or a list of times projects, activities, and users (**get_times()**, **get_projects()**, **get_activities()**, **get_users()**)
- Delete an object in the TimeSync database (**delete_time()**, **delete_project()**, **delete_activity()**, **delete_user()**)

Pymesync currently supports the following TimeSync API versions:

- v1

All of these methods return a python dictionary (or a list of zero or more python dictionaries in the case of the `get_*` methods).

- **authenticate(username, password, auth_type)** - Authenticates a pymesync object with a TimeSync implementation

- **create_time(time)** - Sends time to TimeSync baseurl set in constructor
- **create_project(project)** - Send new project to TimeSync

- **create_activity(activity)** - Send new activity to TimeSync
- **create_user(user)** - Send a new user to TimeSync

- **update_time(time, uuid)** - Update time entry specified by uuid
- **update_project(project, slug)** - Update project specified by slug
- **update_activity(activity, slug)** - Update activity specified by slug
- **update_user(user, username)** - Update user specified by username

- **get_times(query_parameters)** - Get times from TimeSync
- **get_projects(query_parameters)** - Get project information from TimeSync
- **get_activities(query_parameters)** - Get activity information from TimeSync
- **get_users(username=None)** - Get user information from TimeSync

- **delete_time(uuid)** - Delete time entry from TimeSync
- **delete_project(slug)** - Delete project record from TimeSync
- **delete_activity(slug)** - Delete activity record from TimeSync
- **delete_user(username)** - Delete user record from TimeSync

Install pymesync

Pymesync is on PyPi, so you can simply `pip install pymesync`. We recommend you use virtualenv, like so:

```
virtualenv venv
source venv/bin/activate
pip install pymesync
```

Initiate and Authenticate a TimeSync object

To access pymesync's public methods you must first initiate a TimeSync object

```
import pymesync

ts = pymesync.TimeSync(baseurl="http://ts.example.com/v1")
ts.authenticate(username="user", password="password", auth_type="password")
```

Where

- `baseurl` is a string containing the url of the TimeSync instance you will communicate with. This must include the version endpoint (example `"http://ts.example.com/v1"`)
- `user` is a string containing the username of the user communicating with TimeSync
- `password` is a string containing the user's password
- `auth_type` is a string containing the type of authentication your TimeSync implementation uses for login, such as `"password"`, or `"ldap"`.

You can also optionally include a token in the constructor like so:

```
import pymesync

ts = pymesync.TimeSync(baseurl="http://ts.example.com/v1", token="SOMETOKENYOUGOTEARLIER")
# ts.authenticate() is not required
```

This is handy when state is not kept between different parts of your system, but you don't want to have to re-authenticate your TimeSync objectfor every section of code.

Note: If you attempt to get, create, or update objects before authenticating, pymesync will return this error (get methods will return this error nested in a list):

```
{"pymesync error": "Not authenticated with TimeSync, call self.authenticate() first"}
```

Errors

Pymesync returns errors the same way it returns successes for whatever method is in use. This means that most of the time errors are returned as a Python dictionary, except in the case of get methods. If the error is a local pymesync error, the key for the error message will be `"pymesync error"`. If the error is from TimeSync, the dictionary will contain the same keys described in the TimeSync error documentation, but as a python dictionary.

If there is an error connecting with the TimeSync instance specified by the baseurl passed to the pymesync constructor, the error will also contain the status code of the response.

An example for a method that returns a dict within a list:

```
[{"pymesync error": "connection to TimeSync failed at baseurl http://ts.example.com/v1 - response sta
```

The same error returned from a method that returns a single dict:

```
{"pymesync error": "connection to TimeSync failed at baseurl http://ts.example.com/v1 - response stat
```

Public methods

These methods are available to general TimeSync users with applicable user roles on the projects they are submitting times to.

TimeSync.**authenticate(user, password, auth_type)**

> Authenticate a pymesync object with a TimeSync implementation. The authentication is subject to any time limits imposed by that implementation.
>
> `user` is a string containing the username of the user communicating with TimeSync
>
> `password` is a string containing the user's password

`auth_type` is a string containing the type of authentication your TimeSync implementation uses for login, such as `"password"`, or `"ldap"`.

authenticate() will return a python dictionary. If authentication was successful, the dictionary will look like this:

```
{"token": "SOMELONGTOKEN"}
```

If authentication was unsuccessful, the dict will contain an error message:

```
{"status": 401, "error": "Authentication failure", "text": "Invalid username or password"}
```

Example:

```
>>> ts.authenticate(username="example-user", password="example-password", auth_type="password")
{u'token': u'eyJ0eXAi...XSnv0ghQ=='}
>>>
```

TimeSync.**token_expiration_time()**

Returns a python datetime representing the expiration time of the current authentication token.

If an error occurs, the error is returned in a single python dict.

Example:

```
>>> ts.authenticate(username="username", password="user-pass", auth_type="password")
{u'token': u'eyJ0eXAiOiJKV1QiLCJhbGciOiJITUFDLVNIQTUxMiJ9.eyJpc3MiOiJvc3Vvc2wtdGltZXN5bmMtc3RhZ2
>>> ts.token_expiration_time()
datetime.datetime(2016, 1, 13, 11, 45, 34)
>>>
```

TimeSync.**project_users(project)**

Returns a dictionary containing the user field of the specified project.

`project` is a string containing the desired project slug.

Example:

```
>> ts.project_users(project="pyme")
{u'malcolm': [u'member', u'manager'], u'jayne': [u'member'], u'kaylee': [u'member'], u'zoe': [u'
>>>
```

TimeSync.**create_time(time)**

Send a time entry to the TimeSync instance at the baseurl provided when instantiating the TimeSync object. This method will return a single python dictionary containing the created entry if successful. The dictionary will contain error information if `create_time()` was unsuccessful.

`time` is a python dictionary containing the time information to send to TimeSync. The syntax is `"string_key": "string_value"` with the exception of the key `"duration"` which takes an integer value, and the key `"activities"`, which takes a list of strings containing activity slugs. `create_time()` accepts the following fields in `time`:

Required:

- `"duration"` - duration of time spent working on a project. May be entered as a positive integer (which will default to seconds) or a string. As a string duration, follow the format `<val>h<val>m`. An internal method will convert the duration to seconds.
- `"project"` - slug of project worked on
- `"user"` - username of user that did the work, must match `user` specified in instantiation

- "date_worked" - date worked for this time entry in the form "yyyy-mm-dd"

Optional:

- "notes" - optional notes about this time entry
- "issue_uri" - optional uri to issue worked on
- "activities" - list of slugs identifying the activies worked on for this time entry. If this is not provided and the project submitted has no default_activity defined by TimeSync, an error will be returned informing the user to include an activity.

Example usage:

```
>>> time = {
...     "duration": 1200,
...     "user": "example-2",
...     "project": "ganeti_web_manager",
...     "activities": ["docs"],
...     "notes": "Worked on documentation toward settings configuration.",
...     "issue_uri": "https://github.com/osuosl/ganeti_webmgr/issues",
...     "date_worked": "2014-04-17"
...}
>>> ts.create_time(time=time)
{u'activities': [u'docs'], u'deleted_at': None, u'date_worked': u'2014-04-17', u'uuid': u'838853
>>>
```

```
>>> time = {
...     "duration": "3h30m",
...     "user": "example-2",
...     "project": "ganeti_web_manager",
...     "activities": ["docs"],
...     "notes": "Worked on documentation toward settings configuration.",
...     "issue_uri": "https://github.com/osuosl/ganeti_webmgr/issues",
...     "date_worked": "2014-04-17"
...}
>>> ts.create_time(time=time)
{u'activities': [u'docs'], u'deleted_at': None, u'date_worked': u'2014-04-17', u'uuid': u'838853
>>>
```

TimeSync.**update_time(time, uuid)**

Update a time entry by uuid on the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the updated entry if successful. The dictionary will contain error information if update_time() was unsuccessful.

time is a python dictionary containing the time information to send to TimeSync. The syntax is "string_key": "string_value" with the exception of the key "duration" which takes an integer value, and the key "activities", which takes a list of strings containing activity slugs. You only need to send the fields that you want to update.

uuid is a string containing the uuid of the time to be updated.

update_time() accepts the following fields in time:

- "duration" - duration of time spent working on a project. May be entered as a positive integer (which will default to seconds) or a string. As a string duration, follow the format <val>h<val>m. An internal method will convert the duration to seconds.
- "project" - slug of project worked on

- `"user"` - username of user that did the work, must match `user` specified in `authenticate()`
- `"activities"` - list of slugs identifying the activies worked on for this time entry
- `"date_worked"` - date worked for this time entry in the form `"yyyy-mm-dd"`
- `"notes"` - optional notes about this time entry
- `"issue_uri"` - optional uri to issue worked on

Example usage:

```
>>> time = {
...     "duration": 1900,
...     "user": "red-leader",
...     "activities": ["hello", "world"],
...}
>>> ts.update_time(time=time, uuid="some-uuid")
{u'activities': [u'hello', u'world'], u'date_worked': u'2015-08-07', u'updated_at': u'2015-10-18

>>> time = {
...     "duration": "3h35m",
...     "user": "red-leader",
...     "activities": ["hello", "world"],
...}
>>> ts.update_time(time=time, uuid="some-uuid")
{u'activities': [u'hello', u'world'], u'date_worked': u'2015-08-07', u'updated_at': u'2015-10-18
```

TimeSync.**get_times(query_parameters=None)**

Request time entries from the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. The time entries are filtered by parameters passed in `query_parameters`. Returns a list of python dictionaries containing the time information returned by TimeSync or an error message if unsuccessful. This method always returns a list, even if the list contains zero or one time object.

`query_parameters` is a python dictionary containing the optional query parameters described in the TimeSync documentation. If `query_parameters` is missing, it defaults to `None`, in which case `get_times()` will return all times the current user is authorized to see. The syntax for each argument is `{"query": ["parameter1", "parameter2"]}` except for the `uuid` parameter which is `{"uuid": "uuid-as-string"}` and the `include_deleted` and `include_revisions` parameters which should be set to booleans.

Currently the valid queries allowed by pymesync are:

- `user` - filter time request by username
  - example: `{"user": ["username"]}`
- `project` - filter time request by project slug
  - example: `{"project": ["slug"]}`
- `activity` - filter time request by activity slug
  - example: `{"activity": ["slug"]}`
- `start` - filter time request by start date
  - example: `{"start": ["2014-07-23"]}`
- `end` - filter time request by end date
  - example: `{"end": ["2015-07-23"]}`

- `include_revisions` - either `True` or `False` to include revisions of times. Defaults to `False`
  - example: `{"include_revisions": True}`
- `include_deleted` - either `True` or `False` to include deleted times. Defaults to `False`
  - example: `{"include_deleted": True}`
- `uuid` - get specific time entry by time uuid
  - example: `{"uuid": "someuuid"}`

  To get a deleted time by `uuid`, also add the `include_deleted` parameter.

Example usage:

```
>>> ts.get_times()
[{u'activities': [u'docs', u'planning'], u'date_worked': u'2014-04-17', u'updated_at': None, u'u
>>> ts.get_times({"uuid": "c3706e79-1c9a-4765-8d7f-89b4544cad56"})
[{u'activities': [u'docs', u'planning'], u'date_worked': u'2014-04-17', u'updated_at': None, u'u
>>>
```

Warning: If the `uuid` parameter is passed all other parameters will be ignored except for `include_deleted` and `include_revisions`. For example, `ts.get_times({"uuid": "time-entry-uuid", "user": ["bob", "rob"]})` is equivalent to `ts.get_times({"uuid": "time-entry-uuid"})`.

TimeSync.**delete_time(uuid)**

Allows the currently authenticated user to delete their own time entry by uuid.

`uuid` is a string containing the uuid of the time entry to be deleted.

delete_time() returns a `{"status": 200}` if successful or an error message if unsuccessful.

Example usage:

```
>>> ts.delete_time(uuid="some-uuid")
{"status": 200}
>>>
```

TimeSync.**get_projects(query_parameters=None)**

Request project entries from the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. The project entries are filtered by parameters passed in `query_parameters`. Returns a list of python dictionaries containing the project information returned by TimeSync or an error message if unsuccessful. This method always returns a list, even if the list contains one project object.

`query_parameters` is a dict containing the optional query parameters described in the TimeSync documentation. If `query_parameters` is empty, `get_projects()` will return all projects in the database. The syntax for each argument is `{"query": "parameter"}` or `{"bool_query": <boolean>}`.

The optional parameters currently supported by the TimeSync API are:

- `slug` - filter project request by project slug
  - example: `{"slug": "gwm"}`
- `include_deleted` - tell TimeSync whether to include deleted projects in request. Default is `False` and cannot be combined with a `slug`.

  - example: `{"include_deleted": True}`

- `include_revisions` - tell TimeSync whether to include past revisions of projects in request. Default is `False`

  - example: `{"include_revisions": True}`

Example usage:

```
>>> ts.get_projects()
[{u'users': {u'tschuy': {u'member': true, u'spectator': false, u'manager': false}, u'mrsj': {u'm
>>> ts.get_projects({"slug": "gwm"})
[{u'users': {u'tschuy': {u'member': true, u'spectator': false, u'manager': false}, u'mrsj': {u'm
>>>
```

Warning: Does not accept a `slug` combined with `include_deleted`, but does accept any other combination.

TimeSync.**get_activities(query_parameters=None)**

Request activity entries from the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. The activity entries are filtered by parameters passed in `query_parameters`. Returns a list of python dictionaries containing the activity information returned by TimeSync or an error message if unsuccessful. This method always returns a list, even if the list contains one activity object.

`query_parameters` contains the optional query parameters described in the TimeSync documentation. If `query_parameters` is empty, `get_activities()` will return all activities in the database. The syntax for each argument is `{"query": "parameter"}` or `{"bool_query": <boolean>}`.

The optional parameters currently supported by the TimeSync API are:

- `slug` - filter activity request by activity slug

  - example: `{"slug": "code"}`

- `include_deleted` - tell TimeSync whether to include deleted activities in request. Default is `False` and cannot be combined with a `slug`.

  - example: `{"include_deleted": True}`

- `include_revisions` - tell TimeSync whether to include past revisions of activities in request. Default is `False`

  - example: `{"include_revisions": True}`

Example usage:

```
>>> ts.get_activities()
[{u'uuid': u'adf036f5-3d49-4a84-bef9-062b46380bbf', u'created_at': u'2014-04-17', u'updated_at':
>>> ts.get_activities({"slug": "docs"})
[{u'uuid': u'adf036f5-3d49-4a84-bef9-062b46380bbf', u'created_at': u'2014-04-17', u'updated_at':
>>>
```

Warning: Does not accept a `slug` combined with `include_deleted`, but does accept any other combination.

TimeSync.**get_users(username=None)**

Request user entities from the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. Returns a list of python dictionaries containing the user information returned by TimeSync or an error message if unsuccessful. This method always returns a list, even if the list contains one user object.

`username` is an optional parameter containing a string of the specific username to be retrieved. If `username` is not provided, a list containing all users will be returned. Defaults to `None`.

Example usage:

```
>>> ts.get_users()
[{u'username': u'userone', u'display_name': u'One Is The Loneliest Number', u'site_admin': False
>>> ts.get_users(username="userone")
[{u'username': u'userone', u'display_name': u'One Is The Loneliest Number', u'site_admin': False
>>>
```

Administrative methods

These methods are available to TimeSync users with administrative permissions.

TimeSync.**create_project(project)**

Create a project on the TimeSync instance at the baseurl provided when instantiating the TimeSync object. This method will return a single python dictionary containing the created project if successful. The dictionary will contain error information if `create_project()` was unsuccessful.

`project` is a python dictionary containing the project information to send to TimeSync. The syntax is `"key": "value"` except for the `"slugs"` field, which is `"slugs": ["slug1", "slug2", "slug3"]`. `project` requires the following fields:

- `"uri"`
- `"name"`
- `"slugs"` - this must be a list of strings

Optionally include a users field to add users to the project:

- **`"users"` - this must be a python dictionary containing individual user** permissions. See example below.

Example usage:

```
>>> project = {
...     "uri": "https://code.osuosl.org/projects/timesync",
...     "name": "TimeSync API",
...     "slugs": ["timesync", "time"],
...     "users": {"tschuy": {"member": True, "spectator": False, "manager": True},
...               "mrsj": {"member": True, "spectator": False, "manager": False},
...               "patcht": {"member": True, "spectator": False, "manager": True},
...               "oz": {"member": False, "spectator": True, "manager": False}
...              }
...}
>>>
>>> ts.create_project(project=project)
{u'users': {u'tschuy': {u'member': true, u'spectator': false, u'manager': true}, u'mrsj': {u'mem
>>>
```

TimeSync.**update_project(project, slug)**

Update an existing project by slug on the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the updated project if successful. The dictionary will contain error information if `update_project()` was unsuccessful.

`project` is a python dictionary containing the project information to send to TimeSync. The syntax is `"key": "value"` except for the `"slugs"` field, which is `"slugs": ["slug1", "slug2", "slug3"]`.

`slug` is a string containing the slug of the project to be updated.

If `"uri"`, `"name"`, or `"owner"` are set to `""` (empty string) or `"slugs"` is set to `[]` (empty array), the value will be set to the empty string/array.

You only need to pass the fields you want to update in `project`.

`project` accepts the following fields:

- `"uri"`
- `"name"`
- `"slugs"` - this must be a list of strings
- `"user"`

Example usage:

```
>>> project = {
...     "uri": "https://code.osuosl.org/projects/timesync",
...     "name": "pymesync",
...}
>>> ts.update_project(project=project, slug="ps")
{u'users': {u'tschuy': {u'member': True, u'spectator': True, u'manager': True}, u'patcht': {u'me
>>>
```

TimeSync.**delete_project(slug)**

Allows the currently authenticated admin user to delete a project record by slug.

`slug` is a string containing the slug of the project to be deleted.

delete_project() returns a `{"status": 200}` if successful or an error message if unsuccessful.

Example usage:

```
>>> ts.delete_project(slug="some-slug")
{u'status': 200}
>>>
```

TimeSync.**create_activity(activity)**

Create an activity on the TimeSync instance at the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the created activity if successful. The dictionary will contain error information if `create_activity()` was unsuccessful.

`activity` is a python dictionary containing the activity information to send to TimeSync. The syntax is `"key": "value"`. `activity` requires the following fields:

- "name"
- "slug"

Example usage:

```
>>> activity = {
...     "name": "Quality Assurance/Testing",
...     "slug": "qa"
...}
>>> ts.create_activity(activity=activity)
{u'uuid': u'cfa07a4f-d446-4078-8d73-2f77560c35c0', u'created_at': u'2013-07-27', u'updated_at':
>>>
```

TimeSync.**update_activity(activity, slug)**

Update an existing activity by slug on the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the updated activity if successful. The dictionary will contain error information if `update_activity()` was unsuccessful.

`activity` is a python dictionary containing the activity information to send to TimeSync. The syntax is `"key": "value"`.

`slug` is a string containing the slug of the activity to be updated.

If `"name"` or `"slug"` in `activity` are set to `""` (empty string), the value will be set to the empty string.

You only need to pass the fields you want to update in `activity`.

`activity` accepts the following fields to update an activity:

- "name"
- "slug"

Example usage:

```
>>> activity = {"name": "Code in the wild"}
>>> ts.update_activity(activity=activity, slug="ciw")
{u'uuid': u'3cf78d25-411c-4d1f-80c8-a09e5e12cae3', u'created_at': u'2014-04-16', u'updated_at':
>>>
```

TimeSync.**delete_activity(slug)**

Allows the currently authenticated admin user to delete an activity record by slug.

`slug` is a string containing the slug of the activity to be deleted.

delete_activity() returns a `{"status": 200}` if successful or an error message if unsuccessful.

Example usage:

```
>>> ts.delete_activity(slug="some-slug")
{u'status': 200}
>>>
```

TimeSync.**create_user(user)**

Create a user on the TimeSync instance at the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the created user if successful. The dictionary will contain error information if `create_user()` was unsuccessful.

`user` is a python dictionary containing the user information to send to TimeSync. The syntax is `"key": "value"`. `user` requires the following fields:

- `"username"`
- `"password"`

Additionally, the following parameters may be optionally included:

- `"display_name"`
- `"email"`
- `"site_admin"` - sitewide permission, must be a boolean
- `"site_spectator"` - sitewide permission , must be a boolean
- `"site_manager"` - sitewide permission, must be a boolean
- `"active"` - user status, usually set internally, must be a boolean

Example usage:

```
>>> user = {
...     "username": "example",
...     "password": "password",
...     "display_name": "X. Ample User",
...     "email": "example@example.com"
...}
>>> ts.create_user(user=user)
{u'username': u'example', u'deleted_at': None, u'display_name': u'X. Ample User', u'site_admin':
>>>
```

TimeSync.**update_user(user, username)**

Update an existing user by `username` on the TimeSync instance specified by the baseurl provided when instantiating the TimeSync object. This method will return a python dictionary containing the updated user if successful. The dictionary will contain error information if `update_user()` was unsuccessful.

`user` is a python dictionary containing the user information to send to TimeSync. The syntax is `"key": "value"`.

`username` is a string containing the username of the user to be updated.

You only need to pass the fields you want to update in `user`.

`user` accepts the following fields to update a user object:

- `"username"`
- `"password"`
- `"display_name"`
- `"email"`
- `"site_admin"`
- `"site_manager"`
- `"site_spectator"`

Example usage:

```
>>> user = {
...     "username": "red-leader",
...     "email": "red-leader@yavin.com"
...}
>>> ts.update_user(user=user, username="example")
{u'username': u'red-leader', u'display_name': u'Mr. Example', u'site_admin': False, u'site_spect
>>>
```

TimeSync.**delete_user(username)**

Allows the currently authenticated admin user to delete a user record by username.

`username` is a string containing the username of the user to be deleted.

delete_user() returns a `{"status": 200}` if successful or an error message if unsuccessful.

Example usage:

```
>>> ts.delete_user(username="username")
{u'status": 200}
>>>
```

CHAPTER 2

Testing Code That Uses Pymesync

Contents

Testing code that calls external modules can be difficult if those modules make expensive API calls, like pymesync. Often, the code that uses pymesync relies on the data that pymesync/TimeSync returns, so mocking pymesync is unrealistic.

Because of this, pymesync has a built-in test mode that allows users of the module to test their code. When in test mode, pymesync returns *representations* of what would be returned upon a successful TimeSync API call. Pymesync still runs all internal error checking while in test mode.

To start test mode, it must be set in the constructor with `test=True`:

```
>>> import pymesync
>>>
>>> ts = pymesync.TimeSync(baseurl="http://timesync.example.com/v1", test=True)
>>> ts.authenticate(username="test-user", password="test-password", auth_type="password")
[{'token': 'TESTTOKEN'}]
>>>
```

Individual methods, like `create_time()`, take all the parameters specified in *pymesync - Communicate with a TimeSync API*. In test mode, those methods return valid representations of TimeSync objects (according to the TimeSync API) using the data that was passed to pymesync.

An (almost) exhaustive example of test mode:

```
>>> import pymesync
>>>
>>> ts = pymesync.TimeSync(baseurl="http://timesync.example.com/v1", test=True)
>>>
>>> time = {
...     "duration": 12,
...     "user": "example-2",
...     "project": "ganeti_web_manager",
...     "activities": ["docs"],
...     "notes": "Worked on documentation toward settings configuration.",
...     "issue_uri": "https://github.com/osuosl/ganeti_webmgr/issues",
...     "date_worked": "2014-04-17"
...}
>>> ts.create_time(time=time)
[{'pymesync error': 'Not authenticated with TimeSync, call self.authenticate() first'}]
```

```
>>>
>>> ts.authenticate(username="test-user", password="test-pass", auth_type="password")
[{'token': 'TESTTOKEN'}]
>>>
>>> ts.token_expiration_time()
datetime.datetime(2016, 1, 13, 11, 45, 34)
>>>
>>> ts.create_time(time=time)
[{'activities': ['docs'], 'deleted_at': None, 'date_worked': '2014-04-17', 'uuid': '838853e3-3635-401
>>>
>>> time = {
...     "duration": 19,
...     "user": "red-leader",
...     "activities": ["hello", "world"],
...}
>>> ts.update_time(time=time, uuid="some-uuid")
[{'activities': ['hello', 'world'], 'date_worked': '2015-08-07', 'updated_at': '2015-10-18', 'user':
>>>
>>> time = {
...     "duration": "0h30m",
...     "user": "red-leader",
...     "project": "ganeti_web_manager",
...     "activities": ["docs"],
...     "notes": "Worked on documentation toward settings configuration.",
...     "issue_uri": "https://github.com/osuosl/ganeti_webmgr/issues",
...     "date_worked": "2014-04-17"
...}

>>> ts.create_time(time=time)
[{'activities': ['docs'], 'deleted_at': None, 'date_worked': '2014-04-17', 'uuid': '838853e3-3635-401
>>>
>>> time = {
...     "duration": "1h50m",
...     "user": "red-leader",
...     "activities": ["hello", "world"],
...}
>>> ts.update_time(time=time, uuid="some-uuid")
[{'activities': ['hello', 'world'], 'date_worked': '2015-08-07', 'updated_at': '2015-10-18', 'user':
>>>
>>> project = {
...     "uri": "https://code.osuosl.org/projects/timesync",
...     "name": "TimeSync API",
...     "slugs": ["timesync", "time"],
...     "users": {"tschuy": {"member": True, "spectator": False, "manager": True},
...               "mrsj": {"member": True, "spectator": False, "manager": False},
...               "patcht": {"member": True, "spectator": False, "manager": True},
...               "oz": {"member": False, "spectator": True, "manager": False}
...              }
...}
>>>
>>> ts.create_project(project=project)
[{'users': {'tschuy': {'member': true, 'spectator': false, 'manager': true}, 'mrsj': {'member': true,
>>>
>>> project = {
...     "uri": "https://code.osuosl.org/projects/timesync",
...     "name": "pymesync",
...}
>>> ts.update_project(project=project, slug="ps")
```

```
[{'users': {'tschuy': {'member': true, 'spectator': true, 'manager': false}, 'mrsj': {'member': true,
>>>
>>> activity = {
...     "name": "Quality Assurance/Testing",
...     "slug": "qa"
...}
>>> ts.create_activity(activity=activity)
[{'uuid': 'cfa07a4f-d446-4078-8d73-2f77560c35c0', 'created_at': '2013-07-27', 'updated_at': None, 'de
>>>
>>> activity = {"name": "Code in the wild"}
>>> ts.update_activity(activity=activity, slug="ciw")
[{'uuid': '3cf78d25-411c-4d1f-80c8-a09e5e12cae3', 'created_at': '2014-04-16', 'updated_at': '2014-04-
>>>
>>> user = {
...     "username": "example",
...     "password": "password",
...     "display_name": "X. Ample User",
...     "email": "example@example.com"
...}
>>> ts.create_user(user=user)
[{'username': 'example', 'deleted_at': None, 'display_name': 'X. Ample User', 'site_manager': False,
>>>
>>> user = {
...     "username": "red-leader",
...     "email": "red-leader@yavin.com"
...}
>>> ts.update_user(user=user, username="example")
[{'username': 'red-leader', 'display_name': 'Mr. Example', 'site_manager': False, 'site_spectator': F
>>>
>>> ts.get_times()
[{'activities': ['docs', 'planning'], 'date_worked': '2014-04-17', 'updated_at': None, 'user': 'usero
>>>
>>> ts.get_projects()
[{'users': {'managers': ['tschuy'], 'spectators': ['tschuy'], 'members': ['patcht', 'tschuy']}, 'uuid
>>>
>>> ts.get_activities()
[{'uuid': 'adf036f5-3d49-4a84-bef9-062b46380bbf', 'created_at': '2014-04-17', 'updated_at': None, 'na
>>>
>>> ts.get_users()
[{'username': 'userone', 'display_name': 'One Is The Loneliest Number', 'site_manager': False, 'site_
>>>
>>> ts.get_users("admin")
[{'username': 'admin', 'display_name': 'X. Ample User', 'site_manager': False, 'site_admin': True, 's
>>>
>>> ts.get_users("manager")
[{'username': 'manager', 'display_name': 'X. Ample User', 'site_manager': True, 'site_admin': False,
>>>
>>> ts.get_users("spectator")
[{'username': 'spectator', 'display_name': 'X. Ample User', 'site_manager': False, 'site_admin': Fals
>>>
>>> ts.get_times({"uuid": "some-uuid"})
[{'activities': ['docs', 'planning'], 'date_worked': '2014-04-17', 'updated_at': None, 'user': 'usero
>>>
>>> ts.delete_time(uuid="some-uuid")
[{"status": 200}]
>>>
>>> ts.delete_user(username="username")
[{"status": 200}]
```

```
>>>
>>> ts.project_users(project="ff")
{'malcolm': ['member', 'manager'], 'jayne': ['member'], 'kaylee': ['member'], 'zoe': ['member'], 'hob
>>>
```

CHAPTER 3

Developer Documentation for Pymesync

Introduction

When developing for pymesync, there are several things that need to be considered, including communication with TimeSync, return formats, error messages, testing, internal vs. external methods, test mode, and documenting changes.

Contents

Communicating with TimeSync

Pymesync communicates with a user-defined TimeSync implementation using the Python requests library. All POST requests to TimeSync must be in proper JSON by passing the data to the json variable in the POST request.

TimeSync returns either a single JSON object or a list of several JSON objects. These must be converted to a python dictionary or list of dictionary as described in the next section.

Return Format

Pymesync usually returns a dictionary or a list of zero or more python dictionaries (in the case of get methods). The return format is decided by the information that will be returned by TimeSync. If TimeSync could return multiple objects, Pymesync returns the dicts in a list, even if zero or one object is returned.

Following this format, the user can use the same logic and syntax to process a `get_<endpoint>()` method that returns one object as they do for a `get_<endpoint>()` method that returns many objects. This is important because filtering parameters can be passed to those methods that will get an unknown number of objects from TimeSync.

The exception to this rule is for simple data returns like `token_expiration_time()`, which returns a python datetime.

Error Messages

Local pymesync error messages and TimeSync error messages returned from the API should be returned in the same format as their parent method. Simple data returns such as `token_expiration_time()` should return a python dictionary.

The key for the error message is set as a class variable in the `pymesync.TimeSync` class constructor. This class variable is called `error` and sets the key name throughout the module, including in the tests. The value stored at the key location must be descriptive enough to help the user debug their issue.

The TimeSync API also returns its own errors in a different format, like so:

```
[{"status": 401, "error": "Authentication failure", "text": "Invalid username or password"}]
```

Testing

Pymesync makes some very expensive API calls to the TimeSync API. These calls can tie up TimeSync resources or even change the state of the TimeSync database.

To test any method that makes an API call or uses an external resource, you should mock it. Mocking in python involves a somewhat steep learning curve. Read the official documentation and review the current pymesync tests that rely on mocking to familiarize yourself.

External and Internal Methods

There are several methods in pymesync that are available to the user, such as `get_times()`, `create_times()`, `update_activity()`. Some are only usable by an authenticated TimeSync administrator, but all are public. Write these methods as you would write any other.

Several public methods accomplish very similar tasks and use an internal method to keep the code DRY. The trick is that in Python, there aren't really any *truly* private methods. We prefix a method name with ___ (double underscore) to indicate that it is private. Python then "name mangles" the method name to prevent name collisions with another class (again, see the documentation)

The result of the name mangling for a developer writing internal functions is in the testing phase. When accessed externally, the `___internal()` method of the `TimeSync` class is renamed like the following:

```
ts_object._TimeSync__internal()
```

By using the mangled name, you can unit test the internal method.

Test Mode

Pymesync provides a *Testing Code That Uses Pymesync* mode so users can test their code without having to mock pymesync. It just returns what the TimeSync API says it should return on proper inputs.

If you write a new public method for pymesync, make sure you add it to the `mock_pymesync.py` file with a proper return. In the method you write, include this logic so the test mode method is called instead when test mode is on:

```
if self.test:
    return # your test mode method
```

Make sure you are returning your test mode method *after* all error checking is complete.

Documenting Changes

When you add a public method, please document it in the usage docs and the test mode docs. Follow the format for already-existing methods.

Uploading to PyPi

When new features are added or bugs are fixed it is necessary to push Pymesync to PyPi so users can `pip install` those changes. This can only be done by owners and maintainers of Pymesync, listed below. Email support@osuosl.org or visit us at #osuosl on freenode if you believe a new version is warranted.

Developer	IRC nick	Role
Matthew Johnson	mrsj	Owner
Ken Lett	kennric	Owner
Alex Taylor	subnomo	Maintainer

There are several steps that a developer must take before submitting Pymesync to PyPi:

1. Follow the article How to submit a package to PyPi by Peter Downs to create accounts and set up your local configuration file.
2. Open a PR and merge the `develop` branch into `master`. This should be reviewed by an owner or maintainer to ensure the update is necessary.
3. Directly on the `master` branch, an owner or maintainer should bump the `version` in `setup.py` following the Semantic Versioning Specification (SemVer). Tag that commit with the version number.
4. Upload Pymesync to PyPi Test first to make sure that everything is working.

```
(venv) $ python setup.py register -r pypitest
[... register to pypitest success ...]
(venv) $ python setup.py sdist upload -r pypitest
[... upload to pypitest success ...]
```

> **Note:** `pypitest` is a configuration set in `.pypirc` from step 1.

   Now visit https://testpypi.python.org/pypi, search for `pymesync`, and make sure the version is up to date.

5. Upload Pymesync to PyPi.

```
(venv) $ python setup.py register -r pypi
[... register to pypi success ...]
(venv) $ python setup.py sdist upload -r pypi
[... upload to pypi success ...]
```

> **Note:** `pypi` is a configuration set in `.pypirc` from step 1.

   Now visit https://pypi.python.org/pypi to make sure the version is up to date.

6. Inform Pymesync users there has been an update in whatever way is standard for your community.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search