# BioBlend Documentation

***Release 0.2.3-dev***

**Enis Afgan**

March 14, 2013

# CONTENTS

CHAPTER

ONE

# ABOUT

BioBlend is a Python (2.6 or higher) library for interacting with CloudMan and Galaxy's API.

Conceptually, it makes it possible to script and automate the process of cloud infrastructure provisioning and scaling via CloudMan, and running of analyses via Galaxy. In reality, it makes it possible to do things like this:

- Create a CloudMan compute cluster, via an API and directly from your local machine:

```
from bioblend.cloudman import CloudManConfig
from bioblend.cloudman import CloudManInstance
cfg = CloudManConfig('<your cloud access key>', '<your cloud secret key>', 'My CloudMan',  'ami-
cmi = CloudManInstance.launch_instance(cfg)
cmi.get_status()
```

- Reconnect to an existing CloudMan instance and manipulate it:

```
from bioblend.cloudman import CloudManInstance
cmi = CloudManInstance("<instance IP>", "<password>")
cmi.add_nodes(3)
cluster_status = cmi.get_status()
cmi.remove_nodes(2)
```

- Interact with Galaxy via a straightforward API:

```
from bioblend.galaxy import GalaxyInstance
gi = GalaxyInstance('<Galaxy IP>', key='your API key')
libs = gi.libraries.get_libraries()
gi.workflows.show_workflow('workflow ID')
gi.workflows.run_workflow('workflow ID', input_dataset_map)
```

**Note:** Although this library allows you to blend these two services into a cohesive unit, the library itself can be used with any single service irrespective of the other. For example, you can use it to just manipulate CloudMan clusters or to script the interactions with an instance of Galaxy running on your laptop.

CHAPTER

TWO

# INSTALLATION

Stable releases of BioBlend are best installed via `pip` or `easy_install` from PyPI using something like:

```
$ pip install bioblend
```

Alternatively, you may install the most current source code from our Git repository, or fork the project on Github. To install from source, do the following:

```
# Clone the repository to a local directory
$ git clone https://github.com/afgane/bioblend.git
# Install the library
$ cd bioblend
$ python setup.py install
```

After installing the library, you will be able to simply import it into your Python environment with `import bioblend`. For details on the available functionality, see the API documentation.

BioBlend requires a number of Python libraries. These libraries are installed automatically when BioBlend itself is installed, regardless whether it is installed via PyPi or by running `python setup.py install` command. The current list of required libraries is always available from setup.py in the source code repository and it is also replicated here: `requests>=1.1.0, poster, simplejson, boto, nose, mock, pyyaml`.

CHAPTER

THREE

# USAGE

To get started using BioBlend, install the library as described above. Once the library becomes available on the given system, it can be developed against. The developed scripts do not need to reside in any particular location on the system.

It is probably best to take a look at the example scripts in `docs/examples` source directory and browse the API documentation. Beyond that, it's up to your creativity :).

CHAPTER

FOUR

# DEVELOPMENT

Anyone interested in contributing or tweaking the library is more then welcome to do so. To start, simply fork the Git repository on Github and start playing with it. Then, issue pull requests.

CHAPTER

FIVE

# API DOCUMENTATION

BioBlend's API focuses around and matches the services it wraps. Thus, there are two top-level sets of APIs, each corresponding to a separate service and a corresponding step in the automation process. *Note* that each of the service APIs can be used completely independently of one another.

Effort has been made to keep the structure and naming of those API's consistent across the library but because they do bridge different services, some discrepancies may exist. Feel free to point those out and/or provide fixes.

## 5.1 CloudMan API

API used to manipulate the instantiated infrastructure. For example, scale the size of the compute cluster, get infrastructure status, get service status.

### 5.1.1 API documentation for interacting with CloudMan

#### CloudManLauncher

**class** `bioblend.cloudman.launch.`**`CloudManLauncher`**(*access_key*, *secret_key*, *cloud=None*)

Define the environment in which this instance of CloudMan will be launched.

Besides providing the credentials, optionally provide the `cloud` object. This object must define the properties required to establish a boto connection to that cloud. See this method's implementation for an example of the required fields. Note that as long the as provided object defines the required fields, it can really by implemented as anything (e.g., a Bunch, a database object, a custom class). If no value for the `cloud` argument is provided, the default is to use the Amazon cloud.

**`connect_ec2`**(*a_key*, *s_key*, *cloud=None*)

Create and return an EC2-compatible connection object for the given cloud.

See `_get_cloud_info` method for more details on the requirements for the `cloud` parameter. If no value is provided, the class field is used.

**`create_cm_security_group`**(*sg_name='CloudMan'*)

Create a security group with all authorizations required to run CloudMan. If the group already exists, check its rules and add the missing ones. Return the name of the created security group.

**`create_key_pair`**(*key_name='cloudman_key_pair'*)

Create a key pair with the provided `key_name`. Return the name of the key or `None` if there was an error creating the key.

**get_status**(*instance_id*)
Check on the status of an instance. If `instance_id` is not provided, the ID obtained when launching *the most recent* instance is used. Note that this assumes the instance being checked on was launched using this class. Also note that the same class may be used to launch multiple instances but only the most recent `instance_id` is kept while any others will need to be explicitly specified.

This method also allows the required `ec2_conn` connection object to be provided at invocation time. If the object is not provided, credentials defined for the class are used (ability to specify a custom `ec2_conn` helps in case of stateless method invocations).

Return a `state` dict with the current `instance_state`, `public_ip`, `placement`, and `error` keys, which capture the current state (the values for those keys default to empty string if no data is available from the cloud).

**launch**(*cluster_name, image_id, instance_type, password, kernel_id=None, ramdisk_id=None, key_name='cloudman_key_pair', security_groups=['CloudMan'], placement='', **kwargs*)
Check all the prerequisites (key pair and security groups) for launching a CloudMan instance, compose the user data based on the parameters specified in the arguments and the cloud properties as defined in the object's `cloud` field.

For the current list of user data fields that can be provided via `kwargs`, see http://wiki.g2.bx.psu.edu/CloudMan/UserData

Return a dict containing the properties and info with which an instance was launched, namely: `sg_names` containing the names of the security groups, `kp_name` containing the name of the key pair, `kp_material` containing the private portion of the key pair (*note* that this portion of the key is available and can be retrieved *only* at the time the key is created, which will happen only if no key with the name provided in the `key_name` argument exists), `rs` containing the boto `ResultSet` object, `instance_id` containing the ID of a started instance, and `error` containing an error message if there was one.

**rule_exists**(*rules*, *from_port*, *to_port*, *ip_protocol='tcp'*, *cidr_ip='0.0.0.0/0'*)
A convenience method to check if an authorization rule in a security group already exists.

### CloudManInstance

API for interacting with a CloudMan instance.

**class** `bioblend.cloudman.`**CloudManConfig**(*access_key=None, secret_key=None, cluster_name=None, image_id=None, instance_type='m1.medium', password=None, cloud_metadata=None, cluster_type=None, initial_storage_size=10, key_name='cloudman_key_pair', security_groups=['CloudMan'], placement='', kernel_id=None, ramdisk_id=None, block_till_ready=False, **kwargs*)
Initializes a CloudMan launch configuration object.

**Parameters**

- **access_key** (*string*) – Access credentials.
- **secret_key** (*string*) – Access credentials.
- **cluster_name** (*string*) – Name used to identify this CloudMan cluster.
- **image_id** (*string*) – Machine image ID to use when launching this CloudMan instance.
- **instance_type** (*string*) – The type of the machine instance, as understood by the chosen cloud provider. (e.g., `m1.medium`)
- **password** (*string*) – The administrative password for this CloudMan instance.

- **cloud_metadata** (*Bunch*) – This object must define the properties required to establish a boto connection to that cloud. See this method's implementation for an example of the required fields. Note that as long the as provided object defines the required fields, it can really by implemented as anything (e.g., a Bunch, a database object, a custom class). If no value for the `cloud` argument is provided, the default is to use the Amazon cloud.
- **kernel_id** (*string*) – The ID of the kernel with which to launch the instances
- **ramdisk_id** (*string*) – The ID of the RAM disk with which to launch the instances
- **key_name** (*string*) – The name of the key pair with which to launch instances
- **security_groups** (*list of strings*) – The ID of the security groups with which to associate instances
- **placement** (*string*) – The availability zone in which to launch the instances
- **cluster_type** (*string*) – The `type`, either 'Galaxy', 'Data', or 'SGE', defines the type of cluster platform to initialize.
- **initial_storage_size** (*int*) – The initial storage to allocate for the instance. This only applies if `cluster_type` is set to either `Galaxy` or `Data`.
- **block_till_ready** (*boolean*) – Specifies whether the launch method will block till the instance is ready and only return once all initialization is complete. The default is True. If False, the launch method will return immediately without blocking. However, any subsequent calls made will automatically block if the instance is not ready and initialized. The blocking timeout and polling interval can be configured by providing extra parameters to the `CloudManInstance.launch_instance` method.

**static `CustomTypeDecoder`**(*dct*)

**class `CustomTypeEncoder`**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, *bigint_as_string=False*, *item_sort_key=None*)

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, long, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming unicode characters escaped. If ensure_ascii is false, the output will be unicode object.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a string, then JSON array elements and object members will be pretty-printed with a newline followed by that string repeated for each level of nesting. `None` (the default) selects the most compact representation without any newlines. For backwards compatibility with versions of simplejson earlier than 2.1.0, an integer is also accepted and is converted to a string with that many spaces.

If specified, separators should be a (item_separator, key_separator) tuple. The default is (', ', ': '). To get the most compact JSON representation you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

If encoding is not None, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

If use_decimal is true (not the default), `decimal.Decimal` will be supported directly by the encoder. For the inverse, decode JSON with `parse_float=decimal.Decimal`.

If namedtuple_as_object is true (the default), objects with `_asdict()` methods will be encoded as JSON objects.

If tuple_as_array is true (the default), tuple (and subclasses) will be encoded as JSON arrays.

If bigint_as_string is true (not the default), ints 2**53 and higher or lower than -2**53 will be encoded as strings. This is to avoid the rounding that happens in Javascript otherwise.

If specified, item_sort_key is a callable used to sort the items in each dictionary. This is useful if you want to sort items other than in alphabetical order by key.

**default**(*obj*)

**static** CloudManConfig.**load_config**(*fp*)

CloudManConfig.**save_config**(*fp*)

CloudManConfig.**set_connection_parameters**(*access_key*, *secret_key*, *cloud_metadata=None*)

CloudManConfig.**set_extra_parameters**(***kwargs*)

CloudManConfig.**set_post_launch_parameters**(*cluster_type=None*, *initial_storage_size=10*)

CloudManConfig.**set_pre_launch_parameters**(*cluster_name, image_id, instance_type, password, kernel_id=None, ramdisk_id=None, key_name='cloudman_key_pair', security_groups=['CloudMan'], placement='', block_till_ready=False*)

CloudManConfig.**validate**()

**class** bioblend.cloudman.**CloudManInstance**(*url*, *password*, ***kwargs*)

Create an instance of the CloudMan API class, which is to be used when manipulating that given CloudMan instance.

The `url` is a string defining the address of CloudMan, for example "http://115.146.92.174". The `password` is CloudMan's password, as defined in the user data sent to CloudMan on instance creation.

**add_nodes**(**args*, ***kwargs*)

Add a number of worker nodes to the cluster, optionally specifying the type for new instances. If `instance_type` is not specified, instance(s) of the same type as the master instance will be started. Note that the `instance_type` must match the type of instance available on the given cloud.

`spot_price` applies only to AWS and, if set, defines the maximum price for Spot instances, thus turning this request for more instances into a Spot request.

**adjust_autoscaling**(**args*, ***kwargs*)

Adjust the autoscaling configuration parameters.

The number of worker nodes in the cluster is bounded by the optional `minimum_nodes` and `maximum_nodes` parameters. If a parameter is not provided then its configuration value does not change.

**autoscaling_enabled**(**args*, ***kwargs*)
Returns a boolean indicating whether autoscaling is enabled.

**cloudman_url**
Returns the URL for accessing this instance of CloudMan.

**disable_autoscaling**(**args*, ***kwargs*)
Disable autoscaling, meaning that worker nodes will need to be manually added and removed.

**enable_autoscaling**(**args*, ***kwargs*)
Enable cluster autoscaling, allowing the cluster to automatically add, or remove, worker nodes, as needed.

The number of worker nodes in the cluster is bounded by the `minimum_nodes` (default is 0) and `maximum_nodes` (default is 19) parameters.

**galaxy_url**
Returns the base URL for this instance, which by default happens to be the URL for Galaxy application.

**get_cloudman_version**(**args*, ***kwargs*)
Returns the cloudman version from the server. Versions prior to Cloudman 2 does not support this call, and therefore, the default is to return 1

**get_cluster_size**(**args*, ***kwargs*)
Get the size of the cluster in terms of the number of nodes; this count includes the master node.

**get_cluster_type**(**args*, ***kwargs*)
Get the `cluster type` for this CloudMan instance. See the CloudMan docs about the available types. Returns a dictionary, for example: `{u'cluster_type': u'SGE'}`.

**get_galaxy_state**(**args*, ***kwargs*)
Get the current status of Galaxy running on the cluster.

**get_master_id**(**args*, ***kwargs*)
Returns the instance ID of the master node in this CloudMan cluster

**get_master_ip**(**args*, ***kwargs*)
Returns the public IP of the master node in this CloudMan cluster

**get_nodes**(**args*, ***kwargs*)
Get a list of nodes currently running in this CloudMan cluster.

**get_static_state**(**args*, ***kwargs*)
Get static information on this CloudMan instance. i.e. state that doesn't change over the lifetime of the cluster

**get_status**(**args*, ***kwargs*)
Get status information on this CloudMan instance.

**initialize**(**args*, ***kwargs*)
Initialize CloudMan platform. This needs to be done before the cluster can be used.

The `cluster_type`, either 'Galaxy', 'Data', or 'SGE', defines the type of cluster platform to initialize.

**static launch_instance**(*cfg*, ***kwargs*)
Launches a new instance of CloudMan on the specified cloud infrastructure.

**Parameters** **cfg** (*CloudManConfig*) – A CloudManConfig object containing the initial parameters for this launch.

**reboot_node**(**args*, ***kwargs*)
Reboot a specific worker node.

The `instance_id` parameter defines the ID, as a string, of a worker node to reboot.

**remove_node**(**args*, ***kwargs*)
Remove a specific worker node from the cluster.

The `instance_id` parameter defines the ID, as a string, of a worker node to remove from the cluster. The `force` parameter (defaulting to False), is a boolean indicating whether the node should be forcibly removed rather than gracefully removed.

**remove_nodes**(**args*, ***kwargs*)
Remove worker nodes from the cluster.

The `num_nodes` parameter defines the number of worker nodes to remove. The `force` parameter (defaulting to False), is a boolean indicating whether the nodes should be forcibly removed rather than gracefully removed.

**terminate**(**args*, ***kwargs*)
Terminate this CloudMan cluster. There is an option to also terminate the master instance (all worker instances will be terminated in the process of cluster termination), and delete the whole cluster.

> **Warning:** Deleting a cluster is irreversible - all of the data will be permanently deleted.

**update**()
Update the local object's fields to be in sync with the actual state of the CloudMan instance the object points to. This method should be called periodically to ensure you are looking at the current data. New in version 0.2.2.

**class** `bioblend.cloudman.`**GenericVMInstance**(*launcher*, *launch_result*)
Create an instance of the CloudMan API class, which is to be used when manipulating that given CloudMan instance.

The `url` is a string defining the address of CloudMan, for example "http://115.146.92.174". The `password` is CloudMan's password, as defined in the user data sent to CloudMan on instance creation.

**get_machine_status**()
Check on the underlying VM status of an instance. This can be used to determine whether the VM has finished booting up and if CloudMan is up and running.

Return a `state` dict with the current `instance_state`, `public_ip`, `placement`, and `error` keys, which capture the current state (the values for those keys default to empty string if no data is available from the cloud).

**instance_id**
Returns the ID of this instance (e.g., `i-87ey32dd`) if launch was successful or `None` otherwise.

**key_pair_material**
Returns the private portion of the generated key pair. It does so only if the instance was properly launched and key pair generated; `None` otherwise.

**key_pair_name**
Returns the name of the key pair used by this instance. If instance was not launched properly, returns `None`.

**wait_till_instance_ready**(*vm_ready_timeout=300*, *vm_ready_check_interval=10*)
Wait until the VM state changes to ready/error or timeout elapses. Updates the host name once ready.

**exception** `bioblend.cloudman.`**VMLaunchException**(*value*)

`bioblend.cloudman.`**block_till_vm_ready**(*func*)
This decorator exists to make sure that a launched VM is ready and has received a public IP before allowing the wrapped function call to continue. If the VM is not ready, the function will block till the VM is ready. If the VM

does not become ready till the vm_ready_timeout elapses or the VM status returns an error, a VMLaunchException will be thrown.

This decorator relies on the wait_till_instance_ready method defined in class GenericVMInstance. All methods to which this decorator is applied must be members of a class which inherit from GenericVMInstance.

The following two optional keyword arguments are recognized by this decorator:

**Parameters**

- **vm_ready_timeout** (*int*) – Maximum length of time to block before timing out. Once the timeout is reached, a VMLaunchException will be thrown.
- **vm_ready_check_interval** (*int*) – The number of seconds to pause between consecutive calls when polling the VM's ready status.

### 5.1.2 Usage documentation

This page describes some sample use cases for CloudMan API and provides examples for these API calls. In addition to this page, there are functional examples of complete scripts in `docs/examples` directory of the BioBlend source code repository.

#### Setting up custom cloud properties

CloudMan supports Amazon, OpenStack, OpenNebula, and Eucalyptus based clouds and BioBlend can be used to programatically manipulate CloudMan on any of those clouds. Once launched, the API calls to CloudMan are the same irrespective of the cloud. In order to launch an instance on a given cloud, cloud properties need to be provided to `CloudManLauncher`. If cloud properties are not specified, `CloudManLauncher` will default to Amazon cloud properties.

If we want to use a different cloud provider, we need to specify additional cloud properties when creating an instance of the `CloudManLauncher` class. For example, if we wanted to create a connection to NeCTAR, Australia's national research cloud, we would use the following properties:

```
from bioblend.util import Bunch
nectar = Bunch(
    name='NeCTAR',
    cloud_type='openstack',
    bucket_default='cloudman-os',
    region_name='NeCTAR',
    region_endpoint='nova.rc.nectar.org.au',
    ec2_port=8773,
    ec2_conn_path='/services/Cloud',
    cidr_range='115.146.92.0/22',
    is_secure=True,
    s3_host='swift.rc.nectar.org.au',
    s3_port=8888,
    s3_conn_path='/')
```

---

**Note:** These properties are cloud-specific and need to be obtained from a given cloud provider.

---

#### Launching a new cluster instance

In order to launch a CloudMan cluster on a chosen cloud, we do the following (continuing from the previous example):

```
from bioblend.cloudman import CloudManConfig
from bioblend.cloudman import CloudManInstance
cmc = CloudManConfig('<your AWS access key', 'your AWS secret key', 'Cluster name',
     'ami-<ID>', 'm1.medium', 'choose_a_password_here', nectar)
cmi = CloudManInstance.launch_instance(cmc)
```

---

**Note:** If you already have an existing instance of CloudMan, just create an instance of the `CloudManInstance` object directly by calling its constructor and connecting to it (the password you provide must match the password you provided as part of user data when launching this instance). For example:

```
cmi = CloudManInstance('http://115.146.92.174', 'your_UD_password')
```

---

We now have a `CloudManInstance` object that allows us to manage created CloudMan instance via the API. Once launched, it will take a few minutes for the instance to boot and CloudMan start. To check on the status of the machine, (repeatedly) run the following command:

```
>>> cmi.get_machine_status()
{'error': '',
 'instance_state': u'pending',
 'placement': '',
 'public_ip': ''}
>>> cmi.get_machine_status()
{'error': '',
 'instance_state': u'running',
 'placement': u'melbourne-qh2',
 'public_ip': u'115.146.86.29'}
```

Once the instance is ready, although it may still take a few moments for CloudMan to start, it is possible to start interacting with the application.

---

**Note:** The `CloudManInstance` object (e.g., `cmi`) is a local representation of the actual CloudMan instance. As a result, the local object can get out of sync with the remote instance. To update the state of the local object, call the `update` method on the `cmi` object:

```
>>> cmi.update()
```

---

### Manipulating an existing cluster

Having a reference to a `CloudManInstance` object, we can manage it via the available *CloudManInstance* API:

```
>>> cmi.initialized
False
>>> cmi.initialize('SGE')
>>> cmi.get_status()
{u'all_fs': [],
 u'app_status': u'yellow',
 u'autoscaling': {u'as_max': u'N/A',
 u'as_min': u'N/A',
 u'use_autoscaling': False},
 u'cluster_status': u'STARTING',
 u'data_status': u'green',
 u'disk_usage': {u'pct': u'0%', u'total': u'0', u'used': u'0'},
 u'dns': u'#',
 u'instance_status': {u'available': u'0', u'idle': u'0', u'requested': u'0'},
```

```
 u'snapshot': {u'progress': u'None', u'status': u'None'}}
>>> cmi.get_cluster_size()
1
>>> cmi.get_nodes()
[{u'id': u'i-00006016',
  u'instance_type': u'm1.medium',
  u'ld': u'0.0 0.025 0.065',
  u'public_ip': u'115.146.86.29',
  u'time_in_state': u'2268'}]
>>> cmi.add_nodes(2)
{u'all_fs': [],
 u'app_status': u'green',
 u'autoscaling': {u'as_max': u'N/A',
  u'as_min': u'N/A',
  u'use_autoscaling': False},
 u'cluster_status': u'READY',
 u'data_status': u'green',
 u'disk_usage': {u'pct': u'0%', u'total': u'0', u'used': u'0'},
 u'dns': u'#',
 u'instance_status': {u'available': u'0', u'idle': u'0', u'requested': u'2'},
 u'snapshot': {u'progress': u'None', u'status': u'None'}}
>>> cmi.get_cluster_size()
3
```

## 5.2 Galaxy API

API used to manipulate genomic analyses within Galaxy, including data management and workflow execution.

### 5.2.1 API documentation for interacting with Galaxy

#### GalaxyInstance

**class** bioblend.galaxy.**GalaxyInstance**(*url*, *key*)

A base representation of an instance of Galaxy, identified by a URL and a user's API key.

After you have created an `GalaxyInstance` object, access various modules via the class fields (see the source for the most up-to-date list): `libraries`, `histories`, `workflows`, `datasets`, and `users` are the minimum set supported. For example, to work with histories, and get a list of all the user's histories, the following should be done:

```
from bioblend import galaxy

gi = galaxy.GalaxyInstance(url='http://127.0.0.1:8000', key='your_api_key')

hl = gi.histories.get_histories()
print "List of histories:", hl
```

**Parameters**

- **url** (*string*) – A FQDN or IP for a given instance of Galaxy. For example: http://127.0.0.1:8080
- **key** (*string*) – User's API key for the given instance of Galaxy, obtained from the user preferences.

**__init__**(*url*, *key*)
A base representation of an instance of Galaxy, identified by a URL and a user's API key.

After you have created an `GalaxyInstance` object, access various modules via the class fields (see the source for the most up-to-date list): `libraries`, `histories`, `workflows`, `datasets`, and `users` are the minimum set supported. For example, to work with histories, and get a list of all the user's histories, the following should be done:

```
from bioblend import galaxy

gi = galaxy.GalaxyInstance(url='http://127.0.0.1:8000', key='your_api_key')

hl = gi.histories.get_histories()
print "List of histories:", hl
```

**Parameters**

- **url** (*string*) – A FQDN or IP for a given instance of Galaxy. For example: http://127.0.0.1:8080
- **key** (*string*) – User's API key for the given instance of Galaxy, obtained from the user preferences.

**make_delete_request**(*url*, *payload=None*, *params=None*)
Make a DELETE request using the provided `url` and the optional arguments. The `payload` must be a dict that can be converted into a JSON object (via `simplejson.dumps`)

If the `params` are not provided, use `default_params` class field. If params are provided and the provided dict does not have `key` key, the default `self.key` value will be included in what's passed to the server via the request.

**make_get_request**(*url*, *params=None*)
Make a GET request using the provided url.

If the `params` are not provided, use `default_params` class field. If params are provided and the provided dict does not have `key` key, the default `self.key` value will be included in what's passed to the server via the request.

**make_post_request**(*url*, *payload*, *params=None*, *files_attached=False*)
Make a POST request using the provided `url` and `payload`. The `payload` must be a dict that contains the request values. The payload dict may contain file handles (in which case the files_attached flag must be set to true).

If the `params` are not provided, use `default_params` class field. If params are provided and the provided dict does not have `key` key, the default `self.key` value will be included in what's passed to the server via the request.

The return value will contain the response body as a JSON object.

## Libraries

Contains possible interactions with the Galaxy Data Libraries

**class** bioblend.galaxy.libraries.**LibraryClient**(*galaxy_instance*)

**create_folder**(*library_id*, *folder_name*, *description=None*, *base_folder_id=None*)
Create a folder in the given library and the base folder. If `base_folder_id` is not provided, the new folder will be created in the root folder.

**create_library**(*name*, *description=None*, *synopsis=None*)
Create a data library with the properties defined in the arguments. Return a list of JSON dicts, looking like so:

```
[{"id": "f740ab636b360a70",
  "name": "Library from bioblend",
  "url": "/api/libraries/f740ab636b360a70"}]
```

**get_libraries**(*library_id=None*, *name=None*, *deleted=False*)
Get all the libraries or filter for specific one(s) via the provided name or ID. Provide only one argument: `name` or `library_id`.

If `name` is set and multiple names match the given name, all the libraries matching the argument will be returned.

Return a list of JSON formatted dicts each containing basic information about a library.

**show_library**(*library_id*, *contents=False*)
Get information about a library.

If want to get contents of the library (rather than just the library details), set `contents` to `True`.

Return a list of JSON formatted dicts containing library details.

**upload_file_contents**(*library_id*, *pasted_content*, *folder_id=None*, *file_type='auto'*, *dbkey='?'*)
Upload pasted_contents to a data library as a new file. If `folder_id` is not specified, the file will be placed in the root folder.

**upload_file_from_local_path**(*library_id*, *file_local_path*, *folder_id=None*, *file_type='auto'*, *dbkey='?'*)
Read local file contents from file_local_path and upload data to a library. If `folder_id` is not specified, the file will be placed in the root folder.

**upload_file_from_server**(*library_id*, *server_dir*, *folder_id=None*, *file_type='auto'*, *dbkey='?'*, *link_data_only=None*, *roles=''*)
Upload a file to a library from a path on the server where Galaxy is running. If `folder_id` is not provided, the file will be placed in the root folder.

Note that for this method to work, the Galaxy instance you're connecting to must have the configuration option `library_import_dir` set in `universe_wsgi.ini`. The value of that configuration option should be a base directory from where more specific directories can be specified as part of the `server_dir` argument. All and only the files (ie, no folders) specified by the `server_dir` argument will be uploaded to the data library.

**upload_file_from_url**(*library_id*, *file_url*, *folder_id=None*, *file_type='auto'*, *dbkey='?'*)
Upload a file to a library from a URL. If `folder_id` is not specified, the file will be uploaded to the root folder.

**upload_from_galaxy_filesystem**(*library_id*, *filesystem_paths*, *folder_id=None*, *file_type='auto'*, *dbkey='?'*, *link_data_only=None*, *roles=''*)
Upload a file from filesystem paths already present on the Galaxy server.

Provides API access for the 'Upload files from filesystem paths' approach.

**link_data_only – whether to copy data into Galaxy. Setting to 'link_to_files'** symlinks data instead of copying

### Histories

Contains possible interactions with the Galaxy Histories

**class** bioblend.galaxy.histories.**HistoryClient**(*galaxy_instance*)

**create_history**(*name=None*)
Create a new history, optionally setting the `name`.

**delete_history**(*history_id*, *purge=False*)
Delete a history.

If `purge` is set to `True`, also purge the history. Note that for the purge option to work, `allow_user_dataset_purge` option must be set in the Galaxy's configuration file `universe_wsgi.ini`

**download_dataset**(*history_id*, *dataset_id*, *file_path=None*, *use_default_filename=True*, *to_ext=None*)

**get_current_history**()
Returns the current user's most recently used history object (not deleted)

**get_histories**(*history_id=None*, *name=None*, *deleted=False*)
Get all histories or filter the specific one(s) via the provided `name` or `history_id`. Provide only one argument, `name` or `history_id`, but not both.

If `deleted` is set to `True`, return histories that have been deleted.

Return a list of history element dicts. If more than one history matches the given `name`, return the list of all the histories with the given name.

**get_status**(*history_id*)
Returns the state of this history as a dictionary, with the following keys. 'state' = This is the current state of the history, such as ok, error, new etc. 'state_details' = Contains individual statistics for various dataset states. 'percent_complete' = The overall number of datasets processed to completion.

**show_dataset**(*history_id*, *dataset_id*)
Get details about a given history dataset. The required `history_id` can be obtained from the datasets's history content details.

**show_history**(*history_id*, *contents=False*)
Get details of a given history. By default, just get the history meta information. If `contents` is set to `True`, get the complete list of datasets in the given history.

**undelete_history**(*history_id*)
Undelete a history

**upload_dataset_from_library**(*history_id*, *lib_dataset_id*)
Upload a dataset into the history from a library. Requires the library dataset ID, which can be obtained from the library contents.

---

### Workflows

Contains possible interactions with the Galaxy Workflows

**class** bioblend.galaxy.workflows.**WorkflowClient**(*galaxy_instance*)

**export_workflow_json** (*workflow_id*)
Exports a workflow in json format

**Parameters** **workflow_id** (*string*) – Encoded workflow ID

**export_workflow_to_local_path** (*workflow_id*, *file_local_path*, *use_default_filename=True*)
Exports a workflow in json format to a given local path.

**Parameters**

- **workflow_id** (*string*) – Encoded workflow ID
- **file_local_path** (*string*) – Local path to which the exported file will be saved. (Should not contain filename if use_default_name=True)
- **use_default_name** (*boolean*) – If the use_default_name parameter is True, the exported file will be saved as file_local_path/Galaxy-Workflow-%s.ga, where %s is the workflow name. If use_default_name is False, file_local_path is assumed to contain the full file path including filename.

**get_workflows** ()
Get a list of all workflows

**Return type** list

**Returns**

A list of workflow dicts. For example:

```
[{u'id': u'92c56938c2f9b315',
  u'name': u'Simple',
  u'url': u'/api/workflows/92c56938c2f9b315'}]
```

**import_workflow_from_local_path** (*file_local_path*)
Imports a new workflow given the path to a file containing a previously exported workflow.

**import_workflow_json** (*workflow_json*)
Imports a new workflow given a json representation of a previously exported workflow.

**run_workflow** (*workflow_id*, *dataset_map*, *history_id=None*, *history_name=None*, *import_inputs_to_history=False*)
Run the workflow identified by `workflow_id`

**Parameters**

- **workflow_id** (*string*) – Encoded workflow ID
- **dataset_map** (*string or dict*) – A mapping of workflow inputs to datasets. The datasets source can be a LibraryDatasetDatasetAssociation (`ldda`), LibraryDataset (`ld`), or HistoryDatasetAssociation (`hda`). The map must be in the following format: `{'<input>': {'id': <encoded dataset ID>, 'src': '[ldda, ld, hda]'}}` (eg, `{'23': {'id': '29beef4fadeed09f', 'src': 'ld'}}`)
- **history_id** (*string*) – The encoded history ID where to store the workflow output. `history_id` OR `history_name` should be provided but not both!
- **history_name** (*string*) – Create a new history with the given name to store the workflow output. `history_id` OR `history_name` should be provided but not both!
- **import_inputs_to_history** (*bool*) – If `True`, used workflow inputs will be imported into the history. If `False`, only workflow outputs will be visible in the given history.

**Return type** dict

**Returns**

A dict containing the history ID where the outputs are placed as well as output dataset IDs. For example:

```
{u'history': u'64177123325c9cfd',
 u'outputs': [u'aa4d3084af404259']}
```

**show_workflow**(*workflow_id*)

Display information needed to run a workflow

**Parameters** **workflow_id** (*string*) – Encoded workflow ID

**Return type** list

**Returns**

A description of the workflow and its inputs as a JSON object. For example:

```
{u'id': u'92c56938c2f9b315',
 u'inputs': {u'23': {u'label': u'Input Dataset', u'value': u''}},
 u'name': u'Simple',
 u'url': u'/api/workflows/92c56938c2f9b315'}
```

---

## Datasets

Contains possible interactions with the Galaxy Datasets

**class** bioblend.galaxy.datasets.**DatasetClient**(*galaxy_instance*)

**download_dataset**(*dataset_id*, *file_path=None*, *use_default_filename=True*, *wait_for_completion=False*, *maxwait=12000*)

Downloads the dataset identified by 'id'.

**Parameters**

- **dataset_id** (*string*) – Encoded Dataset ID
- **file_path** (*string*) – If the file_path argument is provided, the dataset will be streamed to disk at that path (Should not contain filename if use_default_name=True). If the file_path argument is not provided, the dataset content is loaded into memory and returned by the method (Memory consumption may be heavy as the entire file will be in memory).
- **use_default_name** (*boolean*) – If the use_default_name parameter is True, the exported file will be saved as file_local_path/%s, where %s is the dataset name. If use_default_name is False, file_local_path is assumed to contain the full file path including filename.
- **wait_for_completion** (*boolean*) – If wait_for_completion is True, this call will block till the dataset is ready. If the dataset state becomes invalid, a DatasetStateException will be thrown.
- **maxwait** (*float*) – Time (in seconds) to wait for dataset to complete. If the dataset state is not complete within this time, a DatasetTimeoutException will be thrown.

**Return type** dict

**Returns** If a file_path argument is not provided, returns a dict containing the file_content. Otherwise returns nothing.

**show_dataset**(*dataset_id*, *deleted=False*)
Display information about and/or content of a dataset. This can be a history or a library dataset.

**exception** bioblend.galaxy.datasets.**DatasetStateException**(*value*)

**exception** bioblend.galaxy.datasets.**DatasetTimeoutException**(*value*)

---

### Users

Contains possible interaction dealing with Galaxy users.

These methods must be executed by a registered Galaxy admin user.

**class** bioblend.galaxy.users.**UserClient**(*galaxy_instance*)

**create_user**(*user_email*)
Create a new Galaxy user.

---

**Note:** For this method to work, the Galaxy instance must have `allow_user_creation` and `use_remote_user` options set to `True` in the `universe_wsgi.ini` configuration file. Also note that setting `use_remote_user` will require an upstream authentication proxy server; however, if you do not have one, access to Galaxy via a browser will not be possible.

---

**get_current_user**()
Returns the user id associated with this Galaxy connection

**get_users**(*deleted=False*)
Get a list of all registered users. If `deleted` is set to `True`, get a list of deleted users.

**Return type** list

**Returns**

A list of dicts with user details. For example:

```
[{u'email': u'a_user@example.com',
  u'id': u'dda47097d9189f15',
  u'url': u'/api/users/dda47097d9189f15'}]
```

**show_user**(*user_id*, *deleted=False*)
Display information about a user. If `deleted` is set to `True`, display information about a deleted user.

## 5.2.2 Usage documentation

This page describes some sample use cases for the Galaxy API and provides examples for these API calls. In addition to this page, there are functional examples of complete scripts in the `docs/examples` directory of the BioBlend source code repository.

### Connect to a Galaxy server

To connect to a running Galaxy server, you will need an account on that Galaxy instance and an API key for the account. Instructions on getting an API key can be found at http://wiki.galaxyproject.org/Learn/API .

To open a connection call:

```
from bioblend.galaxy import GalaxyInstance

gi = GalaxyInstance(url='http://example.galaxy.url', key='your-API-key')
```

We now have a `GalaxyInstance` object which allows us to interact with the Galaxy server under our account, and access our data. If the account is a Galaxy admin account we also will be able to use this connection to carry out admin actions.

### View Histories and Datasets

Methods for accessing histories and datasets are grouped under `GalaxyInstance.histories.*` and `GalaxyInstance.datasets.*` respectively.

To get information on the Histories currently in your account, call:

```
>>> gi.histories.get_histories()
[{u'id': u'f3c2b0f3ecac9f02',
  u'name': u'RNAseq_DGE_BASIC_Prep',
  u'url': u'/api/histories/f3c2b0f3ecac9f02'},
 {u'id': u'8a91dcf1866a80c2',
  u'name': u'June demo',
  u'url': u'/api/histories/8a91dcf1866a80c2'}]
```

This returns a list of dictionaries containing basic metadata, including the id and name of each History. In this case, we have two existing Histories in our account, 'RNAseq_DGE_BASIC_Prep' and 'June demo'. To get more detailed information about a History we can pass its id to the `show_history` method:

```
>>> gi.histories.show_history('f3c2b0f3ecac9f02', contents=False)
{u'annotation': u'',
 u'contents_url': u'/api/histories/f3c2b0f3ecac9f02/contents',
 u'id': u'f3c2b0f3ecac9f02',
 u'name': u'RNAseq_DGE_BASIC_Prep',
 u'nice_size': u'93.5 MB',
 u'state': u'ok',
 u'state_details': {u'discarded': 0,
      u'empty': 0,
      u'error': 0,
      u'failed_metadata': 0,
      u'new': 0,
      u'ok': 7,
      u'paused': 0,
      u'queued': 0,
      u'running': 0,
      u'setting_metadata': 0,
      u'upload': 0 },
 u'state_ids': {u'discarded': [],
      u'empty': [],
      u'error': [],
      u'failed_metadata': [],
      u'new': [],
      u'ok': [u'd6842fb08a76e351',
              u'10a4b652da44e82a',
              u'81c601a2549966a0',
              u'a154f05e3bcee26b',
              u'1352fe19ddce0400',
              u'06d549c52d753e53',
              u'9ec54455d6279cc7'],
      u'paused': [],
```

```
      u'queued': [],
      u'running': [],
      u'setting_metadata': [],
      u'upload': []
      }
  }
```

This gives us a dictionary containing the History's metadata. With `contents=False` (the default), we only get a list of ids of the datasets contained within the History; with `contents=True` we would get metadata on each dataset. We can also directly access more detailed information on a particular dataset by passing its id to the `show_dataset` method:

```
>>> gi.datasets.show_dataset('10a4b652da44e82a')
{u'data_type': u'fastqsanger',
 u'deleted': False,
 u'file_size': 16527060,
 u'genome_build': u'dm3',
 u'id': 17499,
 u'metadata_data_lines': None,
 u'metadata_dbkey': u'dm3',
 u'metadata_sequences': None,
 u'misc_blurb': u'15.8 MB',
 u'misc_info': u'Noneuploaded fastqsanger file',
 u'model_class': u'HistoryDatasetAssociation',
 u'name': u'C1_R2_1.chr4.fq',
 u'purged': False,
 u'state': u'ok',
 u'visible': True}
```

### View Data Libraries

Methods for accessing Data Libraries are grouped under `GalaxyInstance.libraries.*`. Most Data Library methods are available to all users, but as only administrators can create new Data Libraries within Galaxy, the `create_folder` and `create_library` methods can only be called using an API key belonging to an admin account.

We can view the Data Libraries available to our account using:

```
>>> gi.libraries.get_libraries()
[{u'id': u'8e6f930d00d123ea',
  u'name': u'RNA-seq workshop data',
  u'url': u'/api/libraries/8e6f930d00d123ea'},
 {u'id': u'f740ab636b360a70',
  u'name': u'1000 genomes',
  u'url': u'/api/libraries/f740ab636b360a70'}]
```

This gives a list of metadata dictionaries with basic information on each library. We can get more information on a particular Data Library by passing its id to the `show_library` method:

```
>>> gi.libraries.show_library('8e6f930d00d123ea')
{u'contents_url': u'/api/libraries/8e6f930d00d123ea/contents',
 u'description': u'RNA-Seq workshop data',
 u'name': u'RNA-Seq',
 u'synopsis': u'Data for the RNA-Seq tutorial'}
```

### Upload files to a Data Library

We can get files into Data Libraries in several ways: by uploading from our local machine, by retrieving from a URL, by passing the new file content directly into the method, or by importing a file from the filesystem on the Galaxy server.

For instance, to upload a file from our machine we might call:

```
>>> gi.libraries.upload_file_from_local_path('8e6f930d00d123ea', '/local/path/to/mydata.fastq', file_
```

Note that we have provided the id of the destination Data Library, and in this case we have specified the type that Galaxy should assign to the new dataset. The default value for `file_type` is 'auto', in which case Galaxy will attempt to guess the dataset type.

### View Workflows

Methods for accessing workflows are grouped under `GalaxyInstance.workflows.*`.

To get information on the Workflows currently in your account, use:

```
>>> gi.workflows.get_workflows()
[{u'id': u'e8b85ad72aefca86',
  u'name': u"TopHat + cufflinks part 1",
  u'url': u'/api/workflows/e8b85ad72aefca86'},
 {u'id': u'b0631c44aa74526d',
  u'name': u'CuffDiff',
  u'url': u'/api/workflows/b0631c44aa74526d'}]
```

This returns a list of metadata dictionaries. We can get the details of a particular Workflow, including its steps, by passing its id to the `show_workflow` method:

```
>>> gi.workflows.show_workflow('e8b85ad72aefca86')
{u'id': u'e8b85ad72aefca86',
 u'inputs':
    {u'252':
       {u'label': u'Input RNA-seq fastq',
        u'value': u''
        }
     },
 u'name': u"TopHat + cufflinks part 1",
 u'steps':
    {u'250':
       {u'id': 250,
        u'input_steps':
           {u'input1':
              {u'source_step': 252,
               u'step_output': u'output'
               }
           },
        u'tool_id': u'tophat',
        u'type': u'tool'
        },
     u'251':
        {u'id': 251,
         u'input_steps':
            {u'input':
               {u'source_step': 250,
                u'step_output': u'accepted_hits'
```

```
                }
            },
         u'tool_id': u'cufflinks',
         u'type': u'tool'
         },
     u'252':
        {u'id': 252,
         u'input_steps': {},
         u'tool_id': None,
         u'type': u'data_input'
         }
     },
 u'url': u'/api/workflows/e8b85ad72aefca86'
 }
```

### Run a Workflow

To run a Workflow, we need to tell Galaxy which datasets to use for which workflow inputs. We can use datasets from Histories or Data Libraries.

Examine the Workflow above. We can see that it takes only one input file. That is:

```
>>> wf = gi.workflows.show_workflow('e8b85ad72aefca86')
>>> wf['inputs']
{u'252':
    {u'label':
        u'Input RNA-seq fastq',
        u'value': u''
    }
}
```

There is one input, labelled 'Input RNA-seq fastq'. This input is passed to the Tophat tool and should be a fastq file. We will use the dataset we examined above, under *View Histories and Datasets*, which had name 'C1_R2_1.chr4.fq' and id '10a4b652da44e82a'.

To specify the inputs, we build a data map and pass this to the `run_workflow` method. This data map is a nested dictionary object which maps inputs to datasets. We call:

```
>>> datamap = dict()
>>> datamap['252'] = { 'src':'hda', 'id':'10a4b652da44e82a' }
>>> gi.workflows.run_workflow('e8b85ad72aefca86', datamap, history_name='New output history')
{u'history': u'0a7b7992a7cabaec',
 u'outputs': [u'33be8ad9917d9207',
              u'fbee1c2dc793c114',
              u'85866441984f9e28',
              u'1c51aa78d3742386',
              u'a68e8770e52d03b4',
              u'c54baf809e3036ac',
              u'ba0db8ce6cd1fe8f',
              u'c019e4cf08b2ac94'
              ]}
```

In this case the only input id is '252' and the corresponding dataset id is '10a4b652da44e82a'. We have specified the dataset source to be 'hda' (HistoryDatasetAssociation) since the dataset is stored in a History. See the *Workflows* API reference for allowed dataset specifications. We have also requested that a new History be created and used to store the results of the run, by setting `history_name='New output history'`.

The `run_workflow` call submits all the jobs which need to be run to the Galaxy workflow engine, with the appropriate dependencies so that they will run in order. The call returns immediately, so we can continue to submit new jobs while waiting for this workflow to execute. `run_workflow` returns the id of the output History and of the datasets that will be created as a result of this run. Note that these dataset ids are valid immediately, so we can specify these datasets as inputs to new jobs even before the files have been created, and the new jobs will be added to the queue with the appropriate dependencies.

If we view the output History after calling `run_workflow`, we will see something like:

```
>>> gi.histories.show_history('0a7b7992a7cabaec')
{u'annotation': u'',
 u'contents_url': u'/api/histories/0a7b7992a7cabaec/contents',
 u'id': u'0a7b7992a7cabaec',
 u'name': u'New output history',
 u'nice_size': u'0 bytes',
 u'state': u'queued',
 u'state_details': {u'discarded': 0,
     u'empty': 0,
     u'error': 0,
     u'failed_metadata': 0,
     u'new': 0,
     u'ok': 0,
     u'paused': 0,
     u'queued': 8,
     u'running': 0,
     u'setting_metadata': 0,
     u'upload': 0},
 u'state_ids': {u'discarded': [],
     u'empty': [],
     u'error': [],
     u'failed_metadata': [],
     u'new': [],
     u'ok': [],
     u'paused': [],
     u'queued': [u'33be8ad9917d9207',
                 u'fbee1c2dc793c114',
                 u'85866441984f9e28',
                 u'1c51aa78d3742386',
                 u'a68e8770e52d03b4',
                 u'c54baf809e3036ac',
                 u'ba0db8ce6cd1fe8f',
                 u'c019e4cf08b2ac94'],
     u'running': [],
     u'setting_metadata': [],
     u'upload': []
    }
}
```

In this case, because the submitted jobs have not had time to run, the output History contains 8 datasets in the 'queued' state and has a total size of 0 bytes. If we make this call again later we should instead see completed output files.

### View Users

User management is only available to Galaxy administrators, that is, the API key used to connect to Galaxy must be that of an admin account.

To get a list of users, call:

```
>>> gi.users.get_users()
[{u'email': u'userA@unimelb.edu.au',
  u'id': u'975a9ce09b49502a',
  u'quota_percent': None,
  u'url': u'/api/users/975a9ce09b49502a'},
 {u'email': u'userB@student.unimelb.edu.au',
  u'id': u'0193a95acf427d2c',
  u'quota_percent': None,
  u'url': u'/api/users/0193a95acf427d2c'}]
```

CHAPTER

SIX

# CONFIGURATION

BioBlend allows library-wide configuration to be set in external files. These configuration files can be used to specify access keys, for example.

## 6.1 Configuration documents for BioBlend

### 6.1.1 BioBlend

**class** bioblend.**NullHandler**(*level=0*)
Initializes the instance - basically setting the formatter to None and the filter list to empty.

**emit**(*record*)

bioblend.**get_version**()
Returns a string with the current version of the library (e.g., "0.2.0")

bioblend.**init_logging**()
Initialize BioBlend's logging from a configuration file.

bioblend.**set_file_logger**(*name*, *filepath*, *level=20*, *format_string=None*)

bioblend.**set_stream_logger**(*name*, *level=10*, *format_string=None*)

### 6.1.2 Config

**class** bioblend.config.**Config**(*path=None*, *fp=None*, *do_load=True*)
BioBlend allows library-wide configuration to be set in external files. These configuration files can be used to specify access keys, for example. By default we use two locations for the BioBlend configurations:

- System wide: `/etc/bioblend.cfg`
- Individual user: `~/.bioblend` (which works on both Windows and Unix)

**get**(*section*, *name*, *default=None*)

**get_value**(*section*, *name*, *default=None*)

**getbool**(*section*, *name*, *default=False*)

**getfloat**(*section*, *name*, *default=0.0*)

**getint**(*section*, *name*, *default=0*)

CHAPTER

SEVEN

# TESTING

The unit tests, in the `tests` folder, can be run using nose. From the project root:

```
$ nosetests
```

CHAPTER

EIGHT

# GETTING HELP

If you've run into issues, found a bug, or can't seem to find an answer to your question regarding the use and functionality of BioBlend, please use Github Issues page to ask your question.

CHAPTER

NINE

# RELATED DOCUMENTATION

Links to other documentation and libraries relevant to this library:

- Galaxy API documentation
- Blend4j: Galaxy API wrapper for Java
- clj-blend: Galaxy API wrapper for Clojure

CHAPTER

TEN

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

# PYTHON MODULE INDEX

## b