# jiveapi Documentation

***Release 1.0.0***

**Jason Antman**

**Oct 13, 2019**

# Contents


Simple and limited Python client for Jive collaboration software ReST API v3, along with utilities for massaging HTML to display better on Jive. Also comes pre-installed in a Docker image and a Sphinx theme and builder for Jive-optimized HTML output.

**Note: Full documentation is hosted at:** jiveapi.readthedocs.io. **This README is just a short introduction.**

CHAPTER 1

# Scope and Status

**This project is effectively abandoned/unsupported and needs a new maintainer!** My employer to longer uses Jive, so I'm no longer using this project and also have no way of testing it. If you are interested in taking over as maintainer, please open an issue!

I'm writing this to be a working Python wrapper around a small portion of the Jive ReST API - specifically, uploading/publishing updating Documents, uploading embedded Images, and manipulating the input HTML to display better in Jive. I'm doing this in my personal time, but we'll be using the project at work for a very limited requirement: "syndicating" documentation that we publish on internal web servers (mostly Sphinx and Hugo static sites) to our corporate Jive instance. The main purpose for doing this is to reach a wider audience and for searchability, not to faithfully reproduce the layout and styling of the original HTML. I don't plan on adding support beyond what's required for that, but contributions are welcome.

This has been in use daily at my current employer for almost a year. It's stable for the particular way we use it, but some code paths may not have been fully exercised before.

Also be aware that Jive **heavily modifies** HTML, including stripping out and sometimes replacing `id` attributes, breaking any internal anchor links containing dashes, etc. The high-level methods in this package make a best effort to modify HTML to work in Jive, but nothing is guaranteed. Once again, this is focused on content not presentation.

## 1.1 Supported Actions

- Low-level API (direct interface to Jive API calls)
  - Get information on currently-authenticated user
  - Get API version information
  - Get Content, Create Content, and Update Content (i.e. Documents, Posts, etc.) in Jive from Python dictionary equivalents of the native Jive API types.
  - Get binary Image data and Create Images that can be embedded in Content (i.e. Documents and Posts).
  - Backdate Content items when creating or updating them.
  - List all Content in a Place.

- High-level wrapper API (provides assistance with generating parameters and massaging content):
  - Create and Update HTML Documents given HTML content and some parameters, including most of the common parameters such as the place to post in, visibility, published/draft status, and keywords.
  - *Not Yet Implemented:* Create and Update HTML Posts given HTML content and some parameters, including most of the common parameters such as the place to post in, visibility, published/draft status, and keywords.
  - Modify HTML formatting to use Jive UI conventions ("jive-ize" HTML).
  - Given a HTML string that contains image tags referring to local images and the filesystem path containing the images, upload each of them to Jive and modify the HTML to point to the images' Jive URLs. Return metadata about the content and images to the user for future updates. Use this metadata on future updates to prevent re-uploading the same image.
  - Option to modify HTML to insert Jive-style information/notice boxes as header and footer, such as information reminding users not to edit the document directly on Jive and giving links to the canonical source, commit, and build that last generated the content.
  - Option to add a Jive Table of Contents macro to the beginning of the content.
- jiveapi also includes a basic Sphinx theme (called `jiveapi`) and Builder (also called `jiveapi`) optimized for building single-page HTML for uploading to Jive.

CHAPTER 2

# Requirements

jiveapi is also available in a self-contained Docker image with all dependencies. See https://hub.docker.com/r/jantman/jiveapi/.

- Python 3.5+. Yes, this package is *only* developed and tested against Python3, specifically 3.5 or later. It *should* work under 2.7 as well, but that is neither tested nor supported.
- requests
- premailer (optional, only required for high-level JiveContent interface)
- lxml (optional, only required for high-level JiveContent interface)

CHAPTER 3

# Usage

**See the full documentation at:** http://jiveapi.readthedocs.io/

CHAPTER 4

# License

This software is licensed under the Affero General Public License, version 3 or later. If you're not redistributing or modifying this software, compliance with the license is simple: make sure anyone interacting with it (even remotely over a network) is informed of where the source code can be downloaded (the project URL in the Python package, or the `jiveapi.version.PROJECT_URL` string constant). If you intend on modifying it, the user must have a way of retrieving the exact running source code. If you're intending on distributing it outside your company, please read the full license and consult your legal counsel or Open Source Compliance policy.

CHAPTER 5

# Contents

## 5.1 Getting Started

### 5.1.1 Local Installation

```
pip install jiveapi
```

### 5.1.2 Use via Docker Image

`docker pull jantman/jiveapi:VERSION` where `VERSION` is the desired release version.

For Docker usage examples, see *Docker Examples*.

### 5.1.3 Authentication

Version 3 of the Jive ReST API is rather limited in terms of Authentication methods: OAuth is only supported for Jive Add-Ons. The alternative is HTTP Basic, which is not supported for federated/SSO accounts. This project uses HTTP Basic auth, which requires a Jive local (service) account.

### 5.1.4 Important Notes

#### 5.1.4.1 Content IDs

When a Content object (e.g. Document, Post, etc.) is created in Jive it is assigned a unique contentID. This contentID must be provided in order to update or delete the content. It is up to you, the user, to store the contentIDs generated by this package when you create content objects. For example use it's enough to record them from the CLI output. For actual production use, I recommend using the Python API and storing the returned IDs in a database or key/value store, or committing them back to the git repository. Also note that even though I've never seen a Jive contentID that isn't `^[0-9]+$`, the Jive API JSON presents and accepts them as strings and the API type documentation lists them as strings.

For most Jive objects, you can obtain the ID by viewing it in the web interface and appending `/api/v3` to the URL. i.e. if you have a Space at `https://sandbox.jiveon.com/community/developertest`, you can find its contentID in the JSON returned from `https://sandbox.jiveon.com/community/developertest/api/v3`. It is **important** to note that the "id" field of the JSON is *not* the same as the "contentID" field.

#### 5.1.4.2 HTML

Jive's HTML handling is somewhat finnicky. This package includes code that attempts to compensate for that. In addition to some very specific styling required to get input HTML to look correct in Jive, Jive also does some annoying things like:

- Removing or overwriting the `id` attributes of HTML elements.
- Assigning its own `id` attributes to anchors.
- Not allowing links to anchors with names including hyphens; they will be silently ignored and result in broken links.
- Requiring explicit `<br />\n` sequences in `<pre>` elements in order to preserve linebreaks.

The workarounds we have in place for this are described further in the *`jiveize_etree()`* method.

In addition, while Jive will happily accept a full HTML document as input, it appears to discard everything outside of the `<body>` tag, including CSS. As a workaround for this, the *`inline_css_etree()`* method calls out to the premailer library to convert all embedded CSS to inline CSS on the elements themselves.

## 5.2 Usage and Examples

### 5.2.1 JiveContent Return Dict Format

The *`JiveContent`* high-level wrapper methods that create or update content in Jive (specifically *`create_html_document()`* and *`update_html_document()`*) return a dictionary describing the content object that was created and various Jive attributes of it. This dict includes the `contentID`, which Jive uses to uniquely identify content objects and must be known in order to update content in Jive. It also contains an `images` key, described below under *JiveContent Images Dict Format*, that must be known in order to not re-upload all images when content is updated.

This dict **must be persisted** if you want to programmatically update the content object in the future. The format of the dict is as follows:

**`entityType`** [*(string)*] the entityType of the content object in Jive.

**`id`** [*(string)*] the ID of the content in Jive. This is only used internally to Jive and is distinct from the `contentID`.

**`html_ref`** [*(string)*] the URL to the content object in the Jive UI, i.e. the URL for users to access it at. Can possibly be an empty string.

**`contentID`** [*(string)*] the Jive content ID required to update this content object; must be passed in when updating existing content. While these IDs appear to only contain numeric characters, the Jive API documentation explicitly states that they are strings.

**`type`** [*(string)*] the content type of the object, i.e. `document` or `post`.

**`typeCode`** [*(int)*] the numeric content type code of the object.

**`images`** [*(dict)*] information about the images contained in the content that have already been uploaded. Details are in *JiveContent Images Dict Format*, below.

Aside from `images`, all of the values are taken from the Jive API representation of the content object; see Jive ReST API - Content Entity for further information.

### 5.2.2 JiveContent Images Dict Format

When images are uploaded in Jive, they are assigned unique identifiers. There is no simple way to determine if an image has already been uploaded to Jive (i.e. the system does not store, or at least does not expose via the API, any sort of checksum). As such, to prevent uploading the same image (as a new Jive object) on every update of a content object, we must store information about the images already uploaded for a given content object and determine on the client side if an image already exists in Jive.

The `images` dict, returned as the value of the `images` key of the *JiveContent Return Dict Format*, stores this information.

Keys of this dict are the string hexdigest of the sha256 checksum of the image's binary content, as returned by passing the image's content through `hashlib.sha256` (see *`_load_image_from_disk()`* for implementation). Values are dicts describing the image, namely:

**`location`** [*(string)*] The URI to the binary image itself as returned by Jive when uploading the image. This is the URI used when embedding the image in HTML.

**`jive_object`** [*(dict)*] The Image Object returned by Jive when uploading the image; see the Jive ReST API - Image Entity for details.

**`local_path`** [*(string)*] The original local path to the image in input HTML, i.e. the `src` attribute of the image tag in the original HTML passed to *`JiveContent`*.

If this dict is not persisted by the client and passed back in on subsequent method calls that update the content, images will be re-uploaded as distinct Jive objects every time.

### 5.2.3 Usage

jiveapi contains two main classes, *`JiveApi`* and *`JiveContent`*. The *`JiveApi`* class contains the low-level methods that map directly to Jive's API, such as creating and updating Content and Images. These methods generally require dicts (serialized to JSON objects in the API calls) that comply with the Jive API documentation for each object type. The *`JiveContent`* class wraps an instance of *`JiveApi`* and provides higher-level convenience methods for generating these API calls such as posting a string of HTML as a Document in a specific Place. *`JiveContent`* also contains static helper methods, such as for manipulating HTML to appear properly in Jive.

### 5.2.4 Examples

For examples of the use of the low-level methods in *`JiveApi`*, see the source code of the unit tests and of the high-level *`JiveContent`* class.

#### 5.2.4.1 Uploading HTML as a Document

In this example we assume that we have a HTML file, `index.html`, in our current directory that we want to upload to the Jive server at `http://jive.example.com` as a Document. If the HTML contains any images, they are either in our current directory or have paths relative to our current directory.

```
import json
from jiveapi import JiveApi, JiveContent
api = JiveApi('http://jive.example.com', 'username', 'password')
```

(continues on next page)

(continued from previous page)

```
jive = JiveContent(api)
with open('index.html', 'r') as fh:
    html = fh.read()
res = jive.create_html_document('My Title', html)
with open('jive_document.json', 'w') as fh:
    fh.write(json.dumps(res))
```

Note that we have JSON-serialized the return value of *create_html_document()*, which is a dict in the *Jive-Content Return Dict Format*. We will need this information when updating the Document in the future; this example just writes it to a file in the current directory, but any non-trivial use should probably store it in a database or key/value store.

#### 5.2.4.2 Updating an Existing Document

Following on the previous example, let's assume that we've made some edits to the HTML and replaced one of the images in it and want to make those changes in Jive. We'll use the *update_html_document()* method for this:

```
import json
from jiveapi import JiveApi, JiveContent
api = JiveApi('http://jive.example.com', 'username', 'password')
jive = JiveContent(api)
with open('index.html', 'r') as fh:
    html = fh.read()
with open('jive_document.json', 'r') as fh:
    doc = json.loads(fh.read())
res = jive.update_html_document(doc['contentID'], 'My Title', html, images=doc['images
↪'])
with open('jive_document.json', 'w') as fh:
    fh.write(json.dumps(res))
```

We should now have a properly-updated document in Jive. This process only uploads new images.

#### 5.2.4.3 Notable Options

The *create_html_document()* and *update_html_document()* methods share many common options. See their documentation for the full list, but here are some that may be of particular interest:

**tags** [*(list)*] a list of string tags to add to the content

**place_id** [*(string)*] the ID of a Place to create the content in. This can be obtained by browsing to a place in the Jive UI and appending `/api/v3` to the URL.

**set_datetime** [*(datetime.datetime)*] the Jive API allows you to explicitly specify the creation/update date on content, i.e. for use when migrating content in.

**toc** [*(boolean)*] prepend the Jive Table of Contents macro to the content.

**header_alert** [*(str or tuple)*] prepend a Jive Alert Box macro to the content, such as to remind users that it was created by an external system.

**footer_alert** [*(str or tuple)*] append a Jive Alert Box macro to the content, such as to link to the build that updated it.

### 5.2.5 Docker Examples

The jiveapi Docker image is an Alpine Linux / Python 3.6 image that comes with jiveapi, Sphinx, the Read The Docs Sphinx theme, rinohtype and boto3. They are all installed globally. The default entrypoint of the container is `/bin/bash`, dropping you into a root shell so that you can explore (i.e. run `python`). For normal use, you would most likely write a script in your current working directory to do whatever you need, mount your current working directory into the container, and then run that script.

For instance, one of the above examples could be saved as `./jive_upload.py` and then run in the Docker container with:

```
docker run -it --rm -v $(pwd):/app jantman/jiveapi:0.1.0 bash -c 'cd /app && python␣
↪jive_upload.py'
```

Please keep in mind that, since the container runs as root, any files it writes to your current directory will be owned by root.

### 5.2.6 Sphinx Theme and Builder

This package includes a Sphinx theme and builder that generate single-page HTML output optimized for uploading to Jive via jiveapi. The theme is based on sphinx' built-in "basic" theme and the builder is based on sphinx' built-in `sphinx.builders.singlehtml.SingleFileHTMLBuilder`.

To build your existing Sphinx documentation you need only install the jiveapi package and specify the "jiveapi" theme and "jiveapi" builder. For example, if your documentation source is in the `source/` directory, then you could build a single-page jive-optimized HTML file to `jivehtml/index.html` with:

```
python -msphinx source jivehtml -b jiveapi -D html_theme=jiveapi
```

### 5.2.7 Jive Sandbox for Testing

If you're interested in trying this against something other than your real Jive instance, Jive maintains https://sandbox.jiveon.com/ as a developer sandbox. There should be a How to Access Sandbox link in the header; as of the writing of this software, it's a completely automated process that should take less than five minutes (but result in a sales email that you can ignore if you wish). Be aware that the sandbox seems to be rather unstable and prone to outages and seemingly-random 500 errors.

## 5.3 Development and Testing

### 5.3.1 Installing for Development

1. Clone the git repo.
2. `virtualenv --python=python3.6 .`
3. `python setup.py develop`
4. `pip install tox`
5. Make changes as necessary. Run tests with `tox`.

### 5.3.2 Testing

Testing is done via tox and pytest. `pip install tox` then `tox` to run tests.

The package itself uses the wonderful requests package as a HTTP(S) client. Tests use the betamax package to record and replay HTTP(S) requests and responses. When adding a new test using betamax, set `JIVEAPI_TEST_MODE=--record` in your environment to capture and record new requests - otherwise, outgoing HTTP requests will be blocked. To re-record a test, delete the current capture from `tests/fixtures/cassettes`. Before committing test data, please inspect it and be sure that no sensitive information is included. To print all base64 bodies from a specific betamax "cassette", you can use `jiveapi/tests/fixtures/showcassette.py`.

## 5.4 jiveapi

### 5.4.1 jiveapi package

#### 5.4.1.1 Subpackages

##### jiveapi.sphinx_theme package

`jiveapi.sphinx_theme.`**`get_html_theme_path`**`()`
Return list of HTML theme paths.

`jiveapi.sphinx_theme.`**`setup`**`(`*app*`)`

#### Submodules

##### jiveapi.sphinx_theme.builder module

**class** `jiveapi.sphinx_theme.builder.`**`JiveHtmlTranslator`**`(`*builder*, **args*, ***kwds*`)`
Bases: `sphinx.writers.html.HTMLTranslator`

Subclass of sphinx's built-in `HTMLTranslator` to fix some output nuances. Mainly, Jive overwrites "id" elements on everything, so named anchors need to use the deprecated `name` attribute. We also need to identify internal hrefs that link to `index.html#something` and strip the leading filename.

**`add_permalink_ref`**`(`*node*, *title*`)`

**`depart_title`**`(`*node*`)`

**`visit_reference`**`(`*node*`)`

**class** `jiveapi.sphinx_theme.builder.`**`JiveapiBuilder`**`(`*app*`)`
Bases: `sphinx.builders.html.SingleFileHTMLBuilder`

Subclass of sphinx's built-in `SingleFileHTMLBuilder` to use `JiveHtmlTranslator` in place of sphinx's built-in `HTMLTranslator`.

**`default_translator_class`**

**`epilog = 'The Jive HTML page is in %(outdir)s.'`**

**`name = 'jiveapi'`**

**`script_files = []`**

`jiveapi.sphinx_theme.builder.`**`setup`**`(`*app*`)`

#### 5.4.1.2 Submodules

### jiveapi.api module

**class** jiveapi.api.**JiveApi**(*base_url*, *username*, *password*)
Bases: object

Low-level client for the Jive API, with methods mapping directly to the Jive API endpoints.

**Parameters**

- **base_url** (*str*) – Base URL to the Jive API. This should be the scheme, hostname, and optional port ending with a path of /api/ (i.e. https://sandbox.jiveon.com/api/).
- **username** (*str*) – Jive API username
- **password** (*str*) – Jive API password

**_get**(*path*, *autopaginate=True*)
Execute a GET request against the Jive API, handling pagination.

**Parameters**

- **path** (*str*) – path or full URL to GET
- **autopaginate** (*bool*) – If True, automatically paginate multi-page responses and return a list of the combined results. Otherwise, return the unaltered JSON response.

**Returns** deserialized response JSON. Usually dict or list.

**_get_content_id_by_html_url**(*path*)
Return contentID from given html/url contentID is unique identifier which is associated with majority type of contents in Jive Api for example you can look here https://developers.jivesoftware.com/api/v3/cloud/rest/DocumentEntity.html :param path: html or full URL to GET :type path: str :return: contentID from given url :rtype: str :variable aux: stored _get response

**_post_json**(*path*, *data*)
Execute a POST request against the Jive API, sending JSON.

**Parameters**

- **path** (*str*) – path or full URL to POST to
- **data** (dict or list) – Data to POST.

**Returns** deserialized response JSON. Usually dict or list.

**Raises** *RequestFailedException*

**_put_json**(*path*, *data*)
Execute a PUT request against the Jive API, sending JSON.

**Parameters**

- **path** (*str*) – path or full URL to PUT to
- **data** (dict or list) – Data to POST.

**Returns** deserialized response JSON. Usually dict or list.

**abs_url**(*path*)
Given a relative path under the base URL of the Jive instance, return the absolute URL formed by joining the base_url to the specified path.

**Parameters** **path** (*str*) – relative path on Jive instance

**Returns** absolute URL to `path` on the Jive instance

**Return type** str

**api_version**()
Get the Jive API version information

**Returns** raw API response dict for `/version` endpoint

**Return type** dict

**create_content**(*contents*, *publish_date=None*)
POST to create a new Content object in Jive. This is the low-level direct API call that corresponds to Create content. Please see the more specific wrapper methods if they suit your purposes.

**Parameters**

- **contents** (*dict*) – A JSON-serializable Jive content representation, suitable for POSTing to the `/contents` API endpoint.
- **publish_date** (*datetime.datetime*) – A backdated publish and update date to set on the content. This allows publishing content with backdated publish dates, for migration purposes.

**Returns** API response of Content object

**Return type** dict

**Raises** *RequestFailedException*, *ContentConflictException*

**get_content**(*content_id*)
Given the content ID of a content object in Jive, return the API (dict) representation of that content object. This is the low-level direct API call that corresponds to Get Content.

This GETs content with the "Silent Directive" that prevents Jive read counts from being incremented. See Silent Directive for Contents Service.

**Parameters** **content_id** (*str*) – the Jive contentID of the content

**Returns** content object representation

**Return type** dict

**get_content_in_place**(*place_id*)
Given the placeID of a Place in Jive, return a list of all Content in that Place. Note that this list can be extremely long. Each element of the list is a full representation of the Content object, including body, which should (theoretically) be identical to that returned by *get_content()*. This is the low-level direct API call that corresponds to PlaceService - Get Content.

**Note:**

1. The `place_id` for a Place in Jive can be found by viewing the place in the web UI and appending `/api/v3` to the URL. It will be the `placeID` field of the resulting JSON response.
2. For some reason, while the web UI shows blog posts in Places, they actually belong to a blog-specific child place and will not be returned in the response. To retrieve blog posts, view the JSON object for the place using the `/api/v3` URL and find the `ref` of the `blog` resource for it. You will then need to call this method a second time with that placeID.

**Parameters** **place_id** (*str*) – the Jive placeID of the Place to list Content in

**Returns** list of content object representation dicts for content in the place

**Return type** list of dict

**get_image**(*image_id*)
GET the image specified by image_id as binary content. This method currently can only retrieve the exact original image. This is the low-level direct API call that corresponds to Get Image.

**Parameters** **image_id** (*str*) – Jive Image ID to get. This can be found in a Content (i.e. Document or Post) object's contentImages list.

**Returns** binary content of Image

**Return type** bytes

**update_content**(*content_id*, *contents*, *update_date=None*)
PUT to update an existing Content object in Jive. This is the low-level direct API call that corresponds to Update content. Please see the more specific wrapper methods if they suit your purposes.

**Warning:** In current Jive versions, it appears that editing/updating a (blog) Post will change the date-based URL to the post, breaking all existing links to it!

**Parameters**

- **content_id** (*str*) – The Jive contentID of the content to update.
- **contents** (*dict*) – A JSON-serializable Jive content representation, suitable for POSTing to the /contents API endpoint.
- **update_date** (*datetime.datetime*) – A backdated update date to set on the content. This allows publishing content with backdated publish dates, for migration purposes.

**Returns** API response of Content object

**Return type** dict

**Raises** *RequestFailedException*, *ContentConflictException*

**upload_image**(*img_data*, *img_filename*, *content_type*)
Upload a new Image resource to be stored on the server as a temporary image, i.e. for embedding in an upcoming Document, Post, etc. Returns Image object and the user-facing URI for the image itself, i.e. https://sandbox.jiveon.com/api/core/v3/images/601174?a=1522503578891 . This is the low-level direct API call that corresponds to Upload New Image.

**Note:** Python's requests lacks streaming file support. As such, images sent using this method will be entirely read into memory and then sent. This may not work very well for extremely large images.

**Warning:** As far as I can tell, the user-visible URI to an image can *only* be retrieved when the image is uploaded. There does not seem to be a way to get it from the API for an existing image.

**Parameters**

- **img_data** (*bytes*) – The binary image data.
- **img_filename** (*str*) – The filename for the image. This is purely for display purposes.
- **content_type** (*str*) – The MIME Content Type for the image data.

**Returns** 2-tuple of (string user-facing URI to the image i.e. for use in HTML, dict Image object representation)

**Return type** tuple

**user**(*id_number='@me'*)
Return dict of information about the specified user.

**Parameters** **id_number** (*str*) – User ID number. Defaults to @me, the current user

**Returns** user information

**Return type** dict

jiveapi.api.**TIME_FORMAT = '%Y-%m-%dT%H:%M:%S.000%z'**
API url param timestamp format, like '2012-01-31T22:46:12.044+0000' note that sub-second time is ignored and set to zero.

### jiveapi.content module

**class** jiveapi.content.**JiveContent**(*api*, *image_dir=None*)
Bases: object

High-level Jive API interface that wraps *JiveApi* with convenience methods for common tasks relating to manipulating Content and Image objects.

Methods in this class that involve uploading images require storing state out-of-band. For information on that state, see *JiveContent Images Dict Format*.

**Parameters**

- **api** (jiveapi.api.JiveApi) – authenticated API instance
- **image_dir** (*str*) – The directory/path on disk to load images relative to. This should be an absolute path. If not specified, the result of os.getcwd() will be used.

**static _is_local_image**(*src*)
Given the string path to an image, return True if it appears to be a local image and False if it appears to be a remote image. We consider an image remote (return False) if urllib.parse.urlparse() returns an empty string for scheme, or local (return True) otherwise. Also returns False if src is None.

**Parameters** **src** (*str*) – the value of image tag's src attribute

**Returns** True if the image appears to be local (relative or absolute path) or False if it appears to be remote

**Return type** bool

**_load_image_from_disk**(*img_path*)
Given the path to an image taken from the src attribute of an img tag, load it from disk. If the path is relative, it will be loaded relative to self._image_dir. Return a 3-tuple of a string describing the Content-Type of the image, the raw bytes of the image data, and the sha256 sum of the image data. The content type is determined using the Python standard library's imghdr.what().

**Parameters** **img_path** (*str*) – path to the image on disk

**Returns** (str Content-Type, bytes binary image content read from disk, str hex sha256 sum of bytes)

**Return type** tuple

**_upload_images**(*root*, *images={}*)
Given the root Element of a (HTML) document, find all img tags. For any of them that have a src attribute pointing to a local image (as determined by *_is_local_image()*), read the corresponding image file from disk, upload it to the Jive server, and then replace the src attribute with the upload temporary URL and add an entry to the image dictionary (second element of the return value).

The format of the second element of the return value is the images dict format described in this class under *JiveContent Images Dict Format*.

**Important:** The images dict (second element of return value) must be externally persisted.

**Parameters** **root** (lxml.etree._Element) – root node of etree to inline CSS in

**Returns** 2-tuple of (root with attributes modified as appropriate, and a dict mapping the original image paths to the API response data for them)

**Return type** tuple

**create_html_document**(*subject*, *html*, *tags=[]*, *place_id=None*, *visibility=None*, *set_datetime=None*, *inline_css=True*, *jiveize=True*, *handle_images=True*, *editable=False*, *toc=False*, *header_alert=None*, *footer_alert=None*)

Create a HTML Document in Jive. This is a convenience wrapper around *create_content()* to assist with forming the content JSON, as well as to assist with HTML handling.

**Important:** In order to update the Document in the future, the entire return value of this method must be externally persisted and passed in to future method calls via the content_id and images parameters.

**Parameters**

- **subject** (*str*) – The subject / title of the Document.
- **html** (*str*) – The HTML for the Document's content. See the notes in the jiveapi package documentation about HTML handling.
- **tags** (*list*) – List of string tags to add to the Document
- **place_id** (*str*) – If specified, post this document in the Place with the specified placeID. According to the API documentation for the Document type (linked above), this requires visibility to be set to "place".
- **visibility** (*str*) – The visibility policy for the Document. Valid values per the API documentation are: all (anyone with appropriate permissions can see the content), hidden (only the author can see the content), or place (place permissions specify which users can see the content).
- **set_datetime** (*datetime.datetime*) – datetime.datetime to set as the publish time. This allows backdating Documents to match their source publish time.
- **inline_css** (*bool*) – if True, pass input HTML through *inline_css_etree()* to convert any embedded CSS to inline CSS so that Jive will preserve/respect it.
- **jiveize** (*bool*) – if True, pass input HTML through *jiveize_etree()* to make it look more like how Jive styles HTML internally.
- **handle_images** (*bool*) – if True, pass input HTML through *_upload_images()* to upload all local images to Jive.
- **editable** (*bool*) – set to True if the content HTML includes Jive RTE Macros. Otherwise, they will not be processed by Jive.
- **toc** (*bool*) – If True, insert the Jive RTE "Table of Contents" macro at the beginning of the html, after header_alert (if specified). Setting this to True forces editable to be True.
- **header_alert** (str or tuple) – If not None, insert a Jive RTE "Alert" macro at the beginning of the html (before the Table of Contents, if present). Setting this to forces editable to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.
- **footer_alert** (str or tuple) – If not None, insert a Jive RTE "Alert" macro at the end of the html. Setting this forces editable to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of

the string alert box type (one of "info", "success", "warning" or "danger") and the string content.

**Returns** dict describing the created content object in Jive. See *JiveContent Return Dict Format* for details.

**Return type** dict

**Raises** *RequestFailedException*, *ContentConflictException*

**dict_for_html_document**(*subject*, *html*, *tags=[]*, *place_id=None*, *visibility=None*, *inline_css=True*, *jiveize=True*, *handle_images=True*, *editable=False*, *toc=False*, *header_alert=None*, *footer_alert=None*, *images={}*)

Generate the API (dict/JSON) representation of a HTML Document in Jive, used by *create_html_document()*.

The format of the second element of the return value is the images dict format described in this class under *JiveContent Images Dict Format*.

**Important:** The images dict (second element of return value) must be externally persisted or else all images will be re-uploaded every time this is run.

**Parameters**

- **subject** (*str*) – The subject / title of the Document.
- **html** (*str*) – The HTML for the Document's content. See the notes in the jiveapi package documentation about HTML handling.
- **tags** (*list*) – List of string tags to add to the Document
- **place_id** (*str*) – If specified, post this document in the Place with the specified placeID. According to the API documentation for the Document type (linked above), this requires visibility to be set to "place".
- **visibility** (*str*) – The visibility policy for the Document. Valid values per the API documentation are: `all` (anyone with appropriate permissions can see the content), `hidden` (only the author can see the content), or `place` (place permissions specify which users can see the content).
- **set_datetime** (*datetime.datetime*) – datetime.datetime to set as the publish time. This allows backdating Documents to match their source publish time.
- **inline_css** (*bool*) – if True, pass input HTML through *inline_css_etree()* to convert any embedded CSS to inline CSS so that Jive will preserve/respect it.
- **jiveize** (*bool*) – if True, pass input HTML through *jiveize_etree()* to make it look more like how Jive styles HTML internally.
- **handle_images** (*bool*) – if True, pass input HTML through *_upload_images()* to upload all local images to Jive.
- **editable** (*bool*) – set to True if the content HTML includes Jive RTE Macros. Otherwise, they will not be processed by Jive.
- **toc** (*bool*) – If True, insert the Jive RTE "Table of Contents" macro at the beginning of the html, after header_alert (if specified). Setting this to True forces `editable` to be True.
- **header_alert** (`str` or `tuple`) – If not None, insert a Jive RTE "Alert" macro at the beginning of the html (before the Table of Contents, if present). Setting this to forces `editable` to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.

- **footer_alert** (str or tuple) – If not None, insert a Jive RTE "Alert" macro at the end of the html. Setting this forces editable to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.
- **images** (*dict*) – a dict of information about images that have been already uploaded for this Document. This parameter should be the value of the images key from the return value of this method (or of *create_html_document()* or *update_html_document()*).

**Returns** 2-tuple of (dict representation of the desired Document ready to pass to the Jive API, dict images data to persist for updates)

**Return type** tuple

**static etree_add_alert**(*root*, *alert_spec*, *header=True*)

Add an alert macro to the specified tree, either at the beginning or end of the body.

**Parameters**

- **root** (lxml.etree._Element) – root node of etree to add Alert macro to
- **alert_spec** (str or tuple) – The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.
- **header** (*bool*) – add to beginning of body element (header) if True, otherwise add to end of body element (footer)

**Returns** root node of etree containing modified HTML

**Return type** lxml.etree._Element or lxml.etree._ElementTree

**static etree_add_toc**(*root*)

Return the provided Element with a Jive RTE "Table of Contents" macro prepended to the body.

**Parameters** **root** (lxml.etree._Element) – root node of etree to add Table of Contents macro to

**Returns** root node of etree containing modified HTML

**Return type** lxml.etree._Element or lxml.etree._ElementTree

**static html_to_etree**(*html*)

Given a string of HTML, parse via etree.fromstring() and return either the roottree if a doctype is present or the root otherwise.

**Important Note:** If the document passed in has a doctype, it will be stripped out. That's fine, since Jive wouldn't recognize it anyway.

**Parameters** **html** (*str*) – HTML string

**Returns** root of the HTML tree for parsing and manipulation purposes

**Return type** lxml.etree._Element or lxml.etree._ElementTree

**static inline_css_etree**(*root*)

Given an etree root node, uses premailer's transform method to convert all CSS from embedded/internal/external to inline, as Jive only allows inline CSS.

**Parameters** **root** (lxml.etree._Element) – root node of etree to inline CSS in

**Returns** root node of etree with CSS inlined

**Return type** lxml.etree._Element or lxml.etree._ElementTree

**static inline_css_html**(*html*)
Wrapper around *inline_css_etree()* that takes a string of HTML and returns a string of HTML.

**Parameters** **html** (*str*) – input HTML to inline CSS in

**Returns** HTML with embedded/internal CSS inlined

**Return type** str

**static jiveize_etree**(*root*, *no_sourcecode_style=True*)
Given a lxml etree root, perform some formatting and style fixes to get each element in it to render correctly in the Jive UI:

- If no_sourcecode_style is True, remove the style attribute from any div elements with a class of sourceCode.
- In all <pre> elements, convert \n to <br />\n via *newline_to_br()*.
- For any HTML tags that are keys of *TAGSTYLES*, set their style attribute according to *TAGSTYLES*.
- Change the name attribute on all a elements to replace dashes with underscores, and do the same on the href attributes of any a elements that begin with #. Apparently Jive breaks anchor links that contain dashes.
- For any element with an id attribute, append a named anchor to it with a name the same as its' id. If it is an anchor, copy the id to the name. We do this because Jive removes or overwrites many id attributes.

Elements which have a "jivemacro" attribute present will not be modified.

**Parameters**

- **root** (lxml.etree._Element) – root node of etree to jive-ize
- **no_sourcecode_style** (*bool*) – If True, remove the style attribute from any div elements with a class of sourceCode.

**Returns** root node of etree containing jive-ized HTML

**Return type** lxml.etree._Element or lxml.etree._ElementTree

**static jiveize_html**(*html*, *no_sourcecode_style=True*)
Wrapper around *jiveize_etree()* that takes a string of HTML and returns a string of HTML.

**Parameters**

- **html** (*str*) – input HTML to Jive-ize
- **no_sourcecode_style** (*bool*) – If True, remove the style attribute from any div elements with a class of sourceCode.

**Returns** jive-ized HTML

**Return type** str

**update_html_document**(*content_id*, *subject*, *html*, *tags=[]*, *place_id=None*, *visibility=None*, *set_datetime=None*, *inline_css=True*, *jiveize=True*, *handle_images=True*, *editable=False*, *toc=False*, *header_alert=None*, *footer_alert=None*, *images={}*)
Update a HTML Document in Jive. This is a convenience wrapper around *update_content()* to assist with forming the content JSON, as well as to assist with HTML handling.

**Important:** In order to update the Document in the future, the entire return value of this method must be externally persisted and passed in to future method calls via the content_id and images parameters.

**Parameters**

- **content_id** (*str*) – the Jive contentID to update. This is the `contentID` element of the *JiveContent Return Dict Format* that is returned by this method or *create_html_document()*.
- **subject** (*str*) – The subject / title of the Document.
- **html** (*str*) – The HTML for the Document's content. See the notes in the jiveapi package documentation about HTML handling.
- **tags** (*list*) – List of string tags to add to the Document
- **place_id** (*str*) – If specified, post this document in the Place with the specified placeID. According to the API documentation for the Document type (linked above), this requires visibility to be set to "place".
- **visibility** (*str*) – The visibility policy for the Document. Valid values per the API documentation are: `all` (anyone with appropriate permissions can see the content), `hidden` (only the author can see the content), or `place` (place permissions specify which users can see the content).
- **set_datetime** (*datetime.datetime*) – datetime.datetime to set as the publish time. This allows backdating Documents to match their source publish time.
- **inline_css** (*bool*) – if True, pass input HTML through *inline_css_etree()* to convert any embedded CSS to inline CSS so that Jive will preserve/respect it.
- **jiveize** (*bool*) – if True, pass input HTML through *jiveize_etree()* to make it look more like how Jive styles HTML internally.
- **handle_images** (*bool*) – if True, pass input HTML through *_upload_images()* to upload all local images to Jive.
- **editable** (*bool*) – set to True if the content HTML includes Jive RTE Macros. Otherwise, they will not be processed by Jive.
- **toc** (*bool*) – If True, insert the Jive RTE "Table of Contents" macro at the beginning of the html, after header_alert (if specified). Setting this to True forces `editable` to be True.
- **header_alert** (`str` or `tuple`) – If not None, insert a Jive RTE "Alert" macro at the beginning of the html (before the Table of Contents, if present). Setting this to forces `editable` to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.
- **footer_alert** (`str` or `tuple`) – If not None, insert a Jive RTE "Alert" macro at the end of the html. Setting this forces `editable` to be True. The value of this parameter can either be a string which will be used as the content of a "info" alert box, or a 2-tuple of the string alert box type (one of "info", "success", "warning" or "danger") and the string content.
- **images** (*dict*) – a dict of information about images that have been already uploaded for this Document. This parameter should be the value of the `images` key from the *JiveContent Return Dict Format* that is returned by this method or *create_html_document()*.

**Returns** dict describing the created content object in Jive. See *JiveContent Return Dict Format* for details.

**Return type** dict

**Raises** *RequestFailedException*, *ContentConflictException*

jiveapi.content.**TAGSTYLES = {'blockquote': 'padding: 0 1em; color: #6a737d; border-left**
This is a mapping of certain HTML tags to the Jive styles to apply to them.

jiveapi.content.**newline_to_br**(*elem*)
Helper function for *jiveize_html()*. Given a html Element, convert it to a string, add explicit <br /> tags before all newlines, and return a new Element with that content.

**Parameters** **elem** (lxml.etree._Element) – element to modify

**Returns** modified element

**Return type** lxml.etree._Element

### jiveapi.exceptions module

**exception** jiveapi.exceptions.**ContentConflictException**(*response*)
Bases: *jiveapi.exceptions.RequestFailedException*

Exception raised when the Jive server response indicates that there is a conflict between the submitted content and content already in the system, such as two content objects of the same type with the same name.

**Parameters** **response** (requests.Response) – the response that generated this exception

**_message_for_response**(*resp*)

**exception** jiveapi.exceptions.**RequestFailedException**(*response*)
Bases: RuntimeError

Exception raised when a Jive server response contains a HTTP status code that indicates an error, or is not the expected status code for the request.

**Parameters** **response** (requests.Response) – the response that generated this exception

**_message_for_response**(*resp*)

### jiveapi.jiveresponse module

**class** jiveapi.jiveresponse.**JiveResponse**
Bases: requests.models.Response

Subclass of requests.Response to handle automatically trimming the JSON Security String from the beginning of Jive API responses.

**json**(***kwargs*)
Returns the json-encoded content of a response, if any, with the leading JSON Security String stripped off.

**Parameters** **kwargs** – Optional arguments that json.loads takes.

**Raises** **ValueError** – If the response body does not contain valid json.

jiveapi.jiveresponse.**requests_hook**(*response*, ***_*)
requests.Session response hook to return *JiveResponse* objects instead of plain requests.Response objects.

Add this to a requests.Session like session.hooks['response'].append(requests_hook)

### jiveapi.utils module

jiveapi.utils.**prettyjson**(*j*)
Return pretty-printed JSON.

**Parameters** **j** – object to JSON serialize

**Returns** pretty-printed JSON serialized version of j

**Return type** str

jiveapi.utils.**set_log_debug**(*logger*)
set logger level to DEBUG, and debug-level output format, via *set_log_level_format()*.

jiveapi.utils.**set_log_info**(*logger*)
set logger level to INFO via *set_log_level_format()*.

jiveapi.utils.**set_log_level_format**(*logger*, *level*, *format*)
Set logger level and format.

**Parameters**

- **logger** (*logging.Logger*) – the logger object to set on
- **level** (*int*) – logging level; see the logging constants.
- **format** (*str*) – logging formatter format string

### jiveapi.version module

jiveapi.version.**PROJECT_URL = 'https://github.com/jantman/jiveapi'**
Constant to hold the project URL, used both in setup.py and anywhere in the code that reports the version.

jiveapi.version.**VERSION = '1.0.0'**
Constant to hold this version of the package, used both in setup.py and anywhere in the code that reports the version.

## 5.5 Changelog

### 5.5.1 1.0.0 (2019-10-13)

- PR 13 - Add function to return contentID from URL (thanks to patricia1387).
- Announce abandonment of project / seeking new maintainer.

### 5.5.2 0.3.0 (2018-11-18)

- Switch order of optional toc and header_warning in rendered HTML for JiveContent.create_html_document() / JiveContent.update_html_document(), so that warning is not hidden by long TOC.

### 5.5.3 0.2.0 (2018-04-21)

- Add support to list Content in a specific Place.

### 5.5.4 0.1.0 (2018-04-07)

- Initial release

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## j

# Index

## Symbols



## A



## C



## D



## E



## G



## H



## I



## J

## N



## P



## R



## S



## T



## U



## V