
gibbon

Gibbon is an API wrapper for MailChimp's API.

build passing

Important Notes

Please read MailChimp's Getting Started Guide.

Gibbon 3.0.0+ returns a `Gibbon::Response` instead of the response body directly. `Gibbon::Response` exposes the parsed response `body` and `headers`.

Installation

```
1 $ gem install gibbon
```

Requirements

A MailChimp account and API key. You can see your API keys [here](#).

Usage

First, create a *one-time use instance* of `Gibbon::Request`:

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key")
```

Note Only reuse instances of Gibbon after terminating a call with a verb, which makes a request. Requests are light weight objects that update an internal path based on your call chain. When you terminate a call chain with a verb, a request instance makes a request and resets the path.

You can set an individual request's `timeout` and `open_timeout` like this:

```
1 gibbon.timeout = 30
2 gibbon.open_timeout = 30
```

You can read about `timeout` and `open_timeout` in the `Net::HTTP` doc.

Now you can make requests using the resources defined in MailChimp's docs. Resource IDs are specified inline and a **CRUD** (`create`, `retrieve` (or `get`), `update`, `upsert`, or `delete`) verb initiates

the request. `upsert` lets you update a record, if it exists, or insert it otherwise where supported by MailChimp's API.

Note `upsert` requires Gibbon version 2.1.0 or newer!

You can specify `headers`, `params`, and `body` when calling a CRUD method. For example:

```
1 gibbon.lists.retrieve(headers: {"SomeHeader": "SomeHeaderValue"},
  params: {"query_param": "query_param_value"})
```

Note `get` can be substituted for `retrieve` as of Gibbon version 3.4.1 or newer!

Of course, `body` is only supported on `create`, `update`, and `upsert` calls. Those map to HTTP POST, PATCH, and PUT verbs respectively.

You can set `api_key`, `timeout`, `open_timeout`, `faraday_adapter`, `proxy`, `symbolize_keys`, `logger`, and `debug` globally:

```
1 Gibbon::Request.api_key = "your_api_key"
2 Gibbon::Request.timeout = 15
3 Gibbon::Request.open_timeout = 15
4 Gibbon::Request.symbolize_keys = true
5 Gibbon::Request.debug = false
```

For example, you could set the values above in an `initializer` file in your `Rails` app (e.g. `your_app/config/initializers/gibbon.rb`).

Assuming you've set an `api_key` on Gibbon, you can conveniently make API calls on the class itself:

```
1 Gibbon::Request.lists.retrieve
```

You can also set the environment variable `MAILCHIMP_API_KEY` and Gibbon will use it when you create an instance:

```
1 gibbon = Gibbon::Request.new
```

Note Substitute an underscore if a resource name contains a hyphen.

Pass `symbolize_keys: true` to use symbols (instead of strings) as hash keys in API responses.

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key", symbolize_keys:
  true)
```

MailChimp's resource documentation is a list of available resources.

Debug Logging

Pass `debug: true` to enable debug logging to STDOUT.

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key", debug: true)
```

Custom logger

Ruby `Logger.new` is used by default, but it can be overridden using:

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key", debug: true,
  logger: MyLogger.new)
```

Logger can be also set by globally:

```
1 Gibbon::Request.logger = MyLogger.new
```

Examples

Lists

Fetch first page of lists:

```
1 gibbon.lists.retrieve
```

Retrieving a specific list looks like:

```
1 gibbon.lists(list_id).retrieve
```

Retrieving a specific list's members looks like:

```
1 gibbon.lists(list_id).members.retrieve
```

Subscribers

Get first page of subscribers for a list:

```
1 gibbon.lists(list_id).members.retrieve
```

By default the Mailchimp API returns 10 results. To set the count to 50:

```
1 gibbon.lists(list_id).members.retrieve(params: {"count": "50"})
```

And to retrieve the next 50 members:

```
1 gibbon.lists(list_id).members.retrieve(params: {"count": "50", "offset": "50"})
```

And to retrieve only subscribed members

```
1 gibbon.lists(list_id).members.retrieve(params: {"count": "50", "offset": "50", "status": "subscribed"})
```

Subscribe a member to a list:

```
1 gibbon.lists(list_id).members.create(body: {email_address: "foo@bar.com", status: "subscribed", merge_fields: {FNAME: "First Name", LNAME: "Last Name"}})
```

If you want to `upsert` instead, you would do the following:

```
1 gibbon.lists(list_id).members(lower_case_md5_hashed_email_address).upsert(body: {email_address: "foo@bar.com", status: "subscribed", merge_fields: {FNAME: "First Name", LNAME: "Last Name"}})
```

You can also unsubscribe a member from a list:

```
1 gibbon.lists(list_id).members(lower_case_md5_hashed_email_address).update(body: { status: "unsubscribed" })
```

Get a specific member's information (open/click rates etc.) from MailChimp:

```
1 gibbon.lists(list_id).members(lower_case_md5_hashed_email_address).retrieve
```

Permanently delete a specific member from a list:

```
1 gibbon.lists(list_id).members(lower_case_md5_hashed_email_address).actions.delete_permanent.create
```

Tags

Tags are a flexible way to organize (slice and dice) your list: for example, you can send a campaign directly to one or more tags.

Add tags to a subscriber:

```
1 gibbon.lists(list_id).members(Digest::MD5.hexdigest(lower_case_email_address)).tags.create(  
2   body: {  
3     tags: [{name:"referred-from-xyz", status:"active"},{name:"pro-plan", status:"active"}]  
4   }  
)
```

Batch Operations

Any API call that can be made directly can also be organized into batch operations. Performing batch operations requires you to generate a hash for each individual API call and pass them as an [Array](#) to the Batch endpoint.

```
1 # Create a new batch job that will create new list members
2 gibbon.batches.create(body: {
3   operations: [
4     {
5       method: "POST",
6       path: "lists/#{ list_id }/members",
7       body: "{...}" # The JSON payload for PUT, POST, or PATCH
8     },
9     ...
10  ]
11 })
```

This will create a new batch job and return a Batch response. The response will include an [id](#) attribute which can be used to check the status of a particular batch job.

Checking on a Batch Job

```
1 gibbon.batches(batch_id).retrieve
```

Response Body (i.e. `response.body`)

```
1 {
2   "id"=>"0ca62e43cc",
3   "status"=>"started",
4   "total_operations"=>1,
5   "finished_operations"=>1,
6   "errored_operations"=>0,
7   "submitted_at"=>"2016-04-19T01:16:58+00:00",
8   "completed_at"=>"",
9   "response_body_url"=>""
10 }
```

Note This response truncated for brevity. Reference the MailChimp API documentation for Batch Operations for more details.

Fields

Only retrieve ids and names for fetched lists:

```
1 gibbon.lists.retrieve(params: {"fields": "lists.id,lists.name"})
```

Only retrieve emails for fetched lists:

```
1 gibbon.lists(list_id).members.retrieve(params: {"fields": "members.email_address"})
```

Campaigns

Get first page of campaigns:

```
1 campaigns = gibbon.campaigns.retrieve
```

Fetch the number of opens for a campaign

```
1 email_stats = gibbon.reports(campaign_id).retrieve["opens"]
```

Create a new campaign:

```
1 recipients = {
2   list_id: list_id,
3   segment_opts: {
4     saved_segment_id: segment_id
5   }
6 }
7 settings = {
8   subject_line: "Subject Line",
9   title: "Name of Campaign",
10  from_name: "From Name",
11  reply_to: "my@email.com"
12 }
13
14 body = {
15   type: "regular",
16   recipients: recipients,
17   settings: settings
18 }
19
20 begin
21   gibbon.campaigns.create(body: body)
22 rescue Gibbon::MailChimpError => e
23   puts "Houston, we have a problem: #{e.message} - #{e.raw_body}"
24 end
```

Add content to a campaign:

(Please note that Mailchimp does not currently support dynamic replacement of mc:edit areas in their drag-and-drop templates using their API. Custom templates can be used instead.)

```
1 body = {
2   template: {
3     id: template_id,
4     sections: {
5       "name-of-mc-edit-area": "Content here"
6     }
7   }
8 }
9
10 gibbon.campaigns(campaign_id).content.upsert(body: body)
```

Send a campaign:

```
1 gibbon.campaigns(campaign_id).actions.send.create
```

Schedule a campaign:

```
1 body = {
2   schedule_time: "2016-06-27 20:00:00"
3 }
```

```
1 gibbon.campaigns(campaign_id).actions.schedule.create(body: body)
```

Interests

Interests are a little more complicated than other parts of the API, so here's an example of how you would set interests during at subscription time or update them later. The ID of the interests you want to opt in or out of must be known ahead of time so an example of how to find interest IDs is also included.

Subscribing a member to a list with specific interests up front:

```
1 gibbon.lists(list_id).members.create(body: {email_address:
   user_email_address, status: "subscribed", interests: {
   some_interest_id: true, another_interest_id: true}})
```

Updating a list member's interests:

```
1 gibbon.lists(list_id).members(member_id).update(body: {interests: {
   some_interest_id: true, another_interest_id: false}})
```

So how do we get the interest IDs? When you query the API for a specific list member's information:

```
1 gibbon.lists(list_id).members(member_id).retrieve
```

The response body (i.e. `response.body`) looks something like this (unrelated things removed):

```
1 {"id"=>"...", "email_address"=>"...", ..., "interests"=>{"3def637141"=>true, "f7cc4ee841"=>false, "fcdc951b9f"=>false, "3daf3cf27d"=>true, "293a3703ed"=>false, "72370e0d1f"=>false, "d434d21a1c"=>false, "bdb1ff199f"=>false, "a54e78f203"=>false, "c4527fd018"=>false} ...}
```

The API returns a map of interest ID to boolean value. Now we to get interest details so we know what these interest IDs map to. Looking at this doc page, we need to do this:

```
1 gibbon.lists(list_id).interest_categories.retrieve
```

To get a list of interest categories. That gives us something like (again, this is the `response.body`):

```
1 {"list_id"=>"...", "categories"=>[{"list_id"=>"...", "id"=>"0ace7aa498", "title"=>"Food Preferences", ...}] ...}
```

In this case, we're interested in the ID of the "Food Preferences" interest, which is `0ace7aa498`. Now we can fetch the details for this interest group:

```
1 gibbon.lists(list_id).interest_categories("0ace7aa498").interests.retrieve
```

That response gives the interest data, including the ID for the interests themselves, which we can use to update a list member's interests or set them when we call the API to subscribe her or him to a list.

Error handling

Gibbon raises an error when the API returns an error.

`Gibbon::MailChimpError` has the following attributes: `title`, `detail`, `body`, `raw_body`, `status_code`. Some or all of these may not be available depending on the nature of the error. For example:

```
1 begin
2   gibbon.lists(list_id).members.create(body: body)
3 rescue Gibbon::MailChimpError => e
4   puts "Houston, we have a problem: #{e.message} - #{e.raw_body}"
5 end
```

Other

Overriding Gibbon's API endpoint (i.e. if using an access token from OAuth and have the `api_endpoint` from the metadata):

```
1 Gibbon::Request.api_endpoint = "https://us1.api.mailchimp.com"
2 Gibbon::Request.api_key = your_access_token_or_api_key
```

You can set an optional proxy url like this (or with an environment variable MAILCHIMP_PROXY):

```
1 gibbon.proxy = 'http://your_proxy.com:80'
```

You can set a different Faraday adapter during initialization:

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key", faraday_adapter:
  :net_http)
```

Migrating from Gibbon 1.x

Gibbon 2.x+ exposes a different API from version 1.x. This is because Gibbon maps to MailChimp's API and because version 3 of the API is quite different from version 2. First, the name of the primary class has changed from `API` to `Request`. And the way you pass an API key during initialization is different. A few examples below.

Initialization Gibbon 1.x:

```
1 gibbon = Gibbon::API.new("your_api_key")
```

Gibbon 2.x+:

```
1 gibbon = Gibbon::Request.new(api_key: "your_api_key")
```

MailChimp API 3 is a RESTful API, so Gibbon's syntax now requires a trailing call to a verb, as described above.

Fetching Lists Gibbon 1.x:

```
1 gibbon.lists.list
```

Gibbon 2.x+:

```
1 gibbon.lists.retrieve
```

Fetching List Members Gibbon 1.x:

```
1 gibbon.lists.members({:id => list_id})
```

Gibbon 2.x+:

```
1 gibbon.lists(list_id).members.retrieve
```

Subscribing a Member to a List Gibbon 1.x:

```
1 gibbon.lists.subscribe({:id => list_id, :email => {:email => "foo@bar.com"}, :merge_vars => {:FNAME => "Bob", :LNAME => "Smith"}})
```

Gibbon 2.x+:

```
1 gibbon.lists(list_id).members.create(body: {email_address: "foo@bar.com", status: "subscribed", merge_fields: {FNAME: "Bob", LNAME: "Smith"}})
```

Thanks

Thanks to everyone who has contributed to Gibbon's development.

Copyright

- Copyright (c) 2010-2022 Amro Mousa. See LICENSE.txt for details.
- MailChimp (c) 2001-2022 The Rocket Science Group.