amity

Version 2.8.0

API-Handbuch

Einführung, Dokumentation und Fallbeispiele

Stand: October 22, 2014

dialogue1

amity offers a REST-like API to access contacts, lists and other data, as well as performing actual sendings.

This document is aimed at developers, who want to build an application or integration into another system, using this API. It describes the *basics* of the interface like the structure of requests and responses, the *authentication* as well as the available *resources* and how to interact with them.

CHAPTER 1

Basics

The amity API is based on HTTP and uses JSON as the data format of choice. In general, clients work on **resources**, which can be accessed and manipulated by using the HTTP verbs `GET`, `PUT`, `POST` and `DELETE`. So for example, an email contact is a resource, much like a list or a folder are resources.

The API is generally available at `/api` below the customer domain, e.g. `http://news.mybusiness.com/api/...`.

Note: Throughout the documentation, the example JSON documents in requests and responses are usually shortened, to only show the relevant parts. Always refer to the resource structure definition to know what fields exist and what form they take!

1.1 Versioning

The API is versioned, so that clients continue to work, even when never versions break with backwards compatibility. The version number is embedded in the URL and the first element after the API root (`/api`), for example:

> http://news.mybusiness.com/api/v2/...

Throughout the documentation, the `/api` and the version number are omitted for brevity. So when we talk about `/contacts`, we are really talking about `/api/v2/contacts`.

1.2 Request

A simple request could look like this:

```
GET /api/v2/contacts/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
```

The request above would retrieve (`GET`) the contact `lm9`.

Besides the resource's URL (`/api/v2/contacts/lm9`), a request also contains a number of HTTP headers, especially those containing authentication information.

1.2.1 Headers

The following headers are of general interest and to be included in most of the requests.

`Accept` **(mandatory)** the desired representation (output) format; has to be `application/json` in all cases at the moment; future versions of the API might support different formats

`User-Agent` *(recommended)* a unique string to identify the client software and version, for example `initech/myclient 1.0`

There are additional headers to control the *authentication*, which differ by what method a client choses. The documentation chapter has more information on this.

1.3 Response

The API uses the standard HTTP status codes to indicate the type of result of a request. For example, the code 200 represents a successful execution of the request, whereas 500 indicates a server-side error.

If not stated otherwise, the API always responds with a JSON document, which contains either a resource's representation or an error. Both cases are described in greater detail below.

A possible answer to the request given above could look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Jan 2014 13:58:42 GMT
Expires: Mon, 20 Jan 2013 13:58:42 GMT
Server: Apache/2.4.6
Cache-Control: private

{
  "data": {
    "id": "lm9"
  }
}
```

Note: The JSON document in the actual API's responses is not prettyprinted like in these examples. Clients should not depend on any formatting in the responses!

1.3.1 Success

A **successful** (HTTP code 2xx or 3xx) response consists of the JSON representation of the ressource as a JSON object, where the actual data is below the `data` key. It does not contain any special status fields like `status`. Have a look at the documentation of the specific resource to learn more about its structure.

For a contact list, the JSON could look like this:

```
{
  "data": {
    "id": "lm9",
    "label": "Standardliste",
    "test": false,
    "disabled": false,
    "links": [
      {
        "rel": "self",
        "uri": "/v2/lists/lm9"
      },
      {
        "rel": "contacts",
        "uri": "/v2/lists/lm9/contacts"
      },
      {
        "rel": "folder",
        "uri": "/v2/folder/lm9"
      }
    ]
  }
}
```

1.3.2 Errors

In case of an error, the response will contain a JSON document with a status code and a human readable error description. The status code consists of the HTTP code (e.g. 400) and a detailed, four-digit code. The value for `status` is computed by doing `($httpCode * 1000) + $detailCode`, so `404127` stands for HTTP 404 (Not Found) and the detail code 127 (which could stand for something like ``parent element not found'').

The detail codes depend on the type of request and are documented along with the *resources*. To get the detail code out of the `status` field, perform `$status - ((int) ($status / 1000)) * 1000` in your client.

With all this said, this is how an error would look like:

```
{
   "status": 400023,
   "message": "The required parameter 'name' is missing."
}
```

Implementations should use the HTTP status code to differentiate between successes and failures, and then can use the `status` field and the contained detail code to look deeper into the source of the problem.

The transmitted `message` is meant to be read by the developer and **not suited** to be displayed to the end user directly. Instead, clients should use the `status` field to switch between error messages.

1.4 Embedding

The amity API allows you to embed related resources when fetching data. This concept is sometimes also known as including subresources or ``side-loading'' them.

This mechanism allows you to fetch a contact and also -- in the same request -- get all the lists this contact is on, as well as the folders in which those lists are placed. Each resource defines its own possible embeds.

To embed something, use the query parameter `embed`, which needs to be a string. If you want to embed, say, embed the lists for a contact, use `embed=lists`.

```
GET /api/v2/contacts/lm9?embed=lists HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
```

This will lead to the list being embedded in the response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Jan 2014 13:58:42 GMT
Expires: Mon, 20 Jan 2013 13:58:42 GMT
Server: Apache/2.4.6
```

```
Cache-Control: private

{
  "data": {
    "id": "lm9",
    "name1": "Amy",
    "name2": "Pond",
    "lists": {
      "data": [
        {
          "id": "hag",
          "label": "Test List"
        },
        {
          "id": "h352",
          "label": "Another List"
        }
      ]
    }
  }
}
```

As you can see, the contact now contains a `list` element, which contains the same structure as if you requested a list of lists, i.e. it has a `data` element.

If a resource allows multiple embeds, separate those with a comma, e.g. `embed=lists,children` for fetching folders:

```
GET /api/v2/folders/lm9?embed=lists,children HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
```

The request above will fetch a single folder (`lm9`) and embed both the lists contained it in, as well as the child folders it has.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Jan 2014 13:58:42 GMT
Expires: Mon, 20 Jan 2013 13:58:42 GMT
Server: Apache/2.4.6
Cache-Control: private

{
  "data": {
    "id": "lm9",
    "label": "Foo",
    "parent": null,
    "children": {
      "data": [
        {
          "id": "hs62g",
          "label": "A child folder of 'Foo'"
        },
        {
          "id": "ha62g",
          "label": "Another child"
        }
      ]
    },
    "lists": {
      "data": [
        {
          "id": "hag",
          "label": "Test List in 'Foo'"
        },
        {
          "id": "h352",
          "label": "Another List in 'Foo'"
        }
      ]
    }
  }
}
```

Embedding works recursively. So for example, you cannot just fetch a folder's lists, but also for each of its lists the contacts therein and for each of those contacts the lists that they are in. Or you can fetch a folder, its children (the folders directly inside it), their children, their children's children etc.

Use a dot to separate multiple recursive levels, e.g. `embed=children.children` to fetch the direct children and their children as well. Or use `embed=children.contacts` to embed the child folders and the contacts for each of those child folders.

```
GET /api/v2/folders/lm9?embed=children.children HTTP/1.1
Host: news.mybusiness.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Jan 2014 13:58:42 GMT
Expires: Mon, 20 Jan 2013 13:58:42 GMT
Server: Apache/2.4.6
Cache-Control: private

{
  "data": {
    "id": "lm9",
    "label": "Foo",
    "parent": null,
    "children": {
      "data": [
        {
          "id": "hs62g",
          "label": "A child folder of 'Foo'",
          "children": {
            "data": [
              {
                "id": "hs62g",
                "label": "A grandchild of 'Foo'"
              }
            ]
          }
        },
        {
          "id": "ha62g",
          "label": "Another child",
          "children": {
            "data": []
          }
        }
      ]
    }
  }
}
```

You can nest embeds up to three levels deep.

CHAPTER 2

Authentication

To access the API, the requests need to be authenticated. At the moment, there is one authentication scheme available.

2.1 Signed Requests

Signing requests works by creating an HMAC over the entire request, using an API key as the secret. This way, only an authorized user can create valid signatures, which can even be transmitted over insecure (speak: non-TLS) connections. The API key is available within the amity webapp.

Warning: Never send your API key in requests. It is only used to create signatures, which in turn are sent to the server in order to authenticate requests.

The following things are all part of the **request signature**:

- the HTTP method (e.g. `GET`)
- the HTTP URI, starting from the API root (e.g. `/api/v2/contacts/lm9`)
- all query string parameters (e.g. `?test=1&foo=bar`)
- the request payload (the ``body'', i.e. the JSON)

The signature is calculated using a **SHA256-HMAC** and sent along as a special HTTP header, together with your client ID (so amity knows which secret it needs to use to verify your signatures).

2.1.1 Calculating the Signature

It's very important to follow the building scheme for signatures exactly, as even a slight variation will lead to unusable signatures. You can have a look at the example source code, provided at the end of this page.

1. Start with an empty list (or vector or array or however a simple list of strings is called in your programming environment).
2. Put the HTTP verb, capitalized, as the first element in your list (e.g. `GET`).
3. Put the request URI as the second element in the list. Note that the URI always starts with a `/`.
4. Put the encoded query string in the list. The query string must be sorted by the parameter names (so it's `foo=bar&xyz=example`, because `foo` comes before `xyz`). Also, make sure to encode the space character (`0x20`) as `%20` (some implementations will generate +).
5. If your request contains a payload, put it in as the last element in your list.

6. Join your list items with a newline character (`\n`).

You should now have a string that looks like this:

```
GET
/api/v2/contacts
param=value&something=else&the=end
```

Or, for a POST request:

```
POST
/api/v2/contacts

{
  "contact": {
    "name": "Who",
    "title": "Dr"
  }
}
```

Notice the empty line. If you have no query string, like in this request, leave the 3rd line blank, but do not leave it out!

Now you can create the HMAC of this string in combination with your API key as the secret.

2.1.2 Sending the Signature

Each request must contain a signature. You have to send it in form of a special HTTP header, called `X-Signature`. Do not quote it, just send the verbatim signature, like in this example:

```
GET /api/v2/contacts HTTP/1.1
Host: news.mybusiness.com
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

2.1.3 Identify Yourself

In order for amity to validate your signature, it needs to know what secret to use. In order to do so, you have to provide your client ID as well. Similar to the signature, it needs to be sent as part of every request and is called `X-Client`.

```
GET /api/v2/contacts HTTP/1.1
Host: news.mybusiness.com
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
X-Client: lm9
```

2.2 Failures

In general, if the provided authentication is invalid for any reason, the API will respond with a generic error message. In this case, the HTTP code is **403** (Forbidden). The `status` field (see the *Errors* section) may contain a special detail code, which would be documented along with the operation.

An authentication failure could look like this:

```
{
   "status": 403951,
   "message": "The given signature is invalid."
}
```

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**950**: No `X-Client` header given.
- 400**951**: No `X-Signature` header given.

HTTP 403 (Unauthenticated)

- 403**950**: The requested client ID is invalid.
- 403**951**: The given signature is invalid.

2.3 Example Code

The following PHP function creates a signature for a request.

```php
<?php

function createSignature($secret, $method, $uri, array $params = array(), $body = null) {
  // make sure the parameters in the query string have a well-defined order and that
  // spaces are escaped as %20 instead of +.
  ksort($params);

  $params = http_build_query($params, '', '&');
  $params = str_replace(array(' ', '+'), '%20', $params);

  // the payload is what needs to be signed and should include all relevant request data
  $payload = array(
    $method,
    $uri,
    $params
  );

  // append the request body, if any, to the payload
  if ($body !== null) {
    $payload[] = $body;
  }

  $payload = implode("\n", $payload);

  // $payload now looks like this:

  /*
  POST
  /api/v2/contacts/lm9
  abc=xyz&param=foo
  {"contact":{"name1":"Amy","name2":"Pond","email":"amy@tardis.org"}}
  */

  // calculate the HMAC for this payload using the known, good $secret
  return hash_hmac('sha256', $payload, $secret);
}

$body = json_encode(array(
  'contact' => array(
    'name1' => 'Amy',
    'name2' => 'Pond',
    'email' => 'amy@tardis.org'
  )
));

$params = array(
  'param' => 'foo',
  'abc'   => 'xyz'
);

$signature = createSignature('<your-api-key-here>', 'POST', '/api/v2/contacts/lm9', $params, $body);
print $signature;
```

CHAPTER 3

Resources

This chapter describes the available resources in amity.

3.1 Contacts

Contacts can be grouped in lists, and a single contact can be in multiple lists at the same time.

The lists a contact belongs to can be managed via the *contact list* subresource.

You can *embed* a contact's list(s) by using `embed=lists`.

3.1.1 Structure

Each contact, represented as a JSON object, consists of the following fields:

`id` **(string)** automatically generated, unique ID

`gender` **(string)** the contact's gender, can be `m` for male, `f` for female or `x` for unknown

`active` **(boolean)** whether or not the contact is active

`language` **(string)** the contact's language ID

`anonymized` **(boolean)** if true, the link tracking for the contact will be anonymized, as some countries forbid tracking personal information

`title` **(string)** the contact's academic title, like `Dr.`, can be empty

`name1` **(string)** the contact's firstname

`name2` **(string)** the contact's lastname

`email` **(string)** the contact's email address

`zip` **(string)** the contact's zipcode, can be empty

`city` **(string)** the contact's city, can be empty

`province` **(string)** the contact's province, can be empty

`nation` **(string)** the contact's province, can be empty

`company` **(string)** the contact's company, can be empty

`mobile` **(string)** the contact's mobile phone number, can be empty

`phone` **(string)** the contact's home phone number, can be empty `fax` **(string)** the contact's fax number, can be empty

`registered` **(date, ISO 8601[1])** time and date when the contact was registered

`address1` **(string)** first address line, can be empty

`address2` **(string)** second address line, can be empty

`internalid` **(string)** an internal identifier for the contact, usage varies from customer to customer; mostly used to customer numbers and similar values; can be empty

`next` **(date, ISO 8601[2])** the next deadline, can be empty

`birthday` **(date, ISO 8601[3])** the contact's birthday, can be empty

`fromlabel` **(string)** the name used in `From:` headers for mailings to this contact, can be empty

`replytolabel` **(string)** the name used in `Reply-To:` headers for mailings to this contact, can be empty

`replyto` **(string)** the email address used in `Reply-To:` headers for mailings to this contact, can be empty

Example:

```
{
  "id": "lm9",
  "gender": "m",
  "active": true,
  "language": "lm9",
  "anonymized": true,
  "title": "Dr.",
  "name1": "",
  "name2": "Who",
  "email": "thedoctor@tardis.org",
  "zip": null,
  "city": null,
  "province": null,
  "nation": null,
  "company": null,
  "mobile": null,
  "phone": null,
  "fax": null,
  "registered": "2014-04-08T15:18:57Z",
  "address1": null,
  "address2": null,
  "internalid": null,
  "next": null,
  "birthday": null,
  "fromlabel": null,
  "replytolabel": null,
  "replyto": null
}
```

3.1.2 Fetch all contacts

To get a list of your contacts, you have to send a `GET` request to `/contacts`.

By default, you will receive **up to 50** contacts, starting from the first one, but you can (and need to, if you want to get all) include query parameters to control both the page size (`size`) and the page number (`page`). Note that the page number starts at zero and that you cannot request more than 500 contacts in one request.

You can also filter the list to only show contacts whose email address (partially) matches a given value. Use the `email` query parameter for the value you want to filter and set the optional `substring` parameter to `1` if you want partial matches as well. Note that you have to give at least 3 characters for the email address.

Filtering by gender (using the `gender` query parameter with a value of `f`, `m` or `x`) is possible as well, just like only fetching `active` contacts. If you give multiple filters, they will be applied as a conjunction (i.e. they must all match).

A few examples should make it easier to understand:

`GET /contacts` fetches the first 50 contacts

`GET /contacts?page=1` fetches the next 50 contacts (51-100)

`GET /contacts?size=200&page=2` fetches contacts 401 to 600

`GET /contacts?email=someone@somewhere.de` performs an exact-match search for the given address; responds with either zero or one contact

`GET /contacts?email=@somewhere.de&substring=1` finds the first 50 contacts having an email address containing `@somewhere.de`

`GET /contacts?gender=f&active=1` finds the first 50 active, female contatcs

`GET /contacts?gender=f&email=@web.de&substring=1&size=100` finds the first 100 female contacts with an email address containing `@web.de`

Request Example

This request will fetch the first 50 contacts.

```
GET /api/v2/contacts HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The `data` element in the response contains a flat list of contacts.

```
{
  "data": [
    {
      "id": "lm9",
      "title": "Dr",
      "name2": "Who"
    },
    {
      "id": "kn9n",
      "name1": "Amy",
      "name2": "Pond"
    }
  ]
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The given `email` query parameter is too short.
- 400**002**: The given `gender` query parameter is invalid.

3.1.3 Fetch a single contact

To fetch a single contact, simply perform a `GET` request to `/contacts/<contact-id>`.

Request Example

This request will fetch the contact with the ID `lm9`.

```
GET /api/v2/contacts/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

You receive the data of this contact.

```
{
  "data": {
    "id": "lm9",
    "title": "Dr",
    "name2": "Who"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given `contact` ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.

3.1.4 Create a new contact

To create a new contact, perform a `POST` request on the contact collection, that is `/contacts`. In your request payload, you have to send the JSON structure with the contact data like name and email address. The contact ID will be automatically generated.

Note: You need appropriate permission to create new contacts.

The request payload must be a JSON object, having a `contact` element which contains the contact information. It should look like this:

```
{
  "contact": {
    "name1": "Amy",
    "name2": "Pond"
  }
}
```

Because this is such a common operation, you can also put the contact into any number of lists in the same request. To do so, include another `lists` element in your payload, which must be an array of list IDs.

```
{
  "contact": {
    "name1": "Amy",
    "name2": "Pond"
  },
  "lists": [
    "kn9n",
    "abc3",
    "g57a"
  ]
}
```

Similarly, you can also trigger any number of events. This is useful for sending welcome mails to the new contact. To do so, include a list of event IDs:

```
{
  "contact": {
    "name1": "Amy",
    "name2": "Pond"
  },
  "lists": [
    "kn9n",
    "abc3",
    "g57a"
  ],
  "events": [
    "bc34"
  ]
}
```

Both `lists` and `events` are completely optional. Note that in both lists, invalid IDs are silently ignored and will not lead to cancelled requests.

Request Example

The following request will create a new contact, *Amy Pond*.

```
POST /api/v2/contacts HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "contact": {
    "name1": "Amy",
    "name2": "Pond",
    "email": "amy@tardis.org"
  }
}
```

Response Example

In response, you will get the contact back (the same as if you did `GET /contacts/<id>`).

```
{
  "data": {
    "id": "bca9",
    "name1": "Amy",
    "name2": "Pond",
    "email": "amy@tardis.org"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `contact` element.
- 400**100**: The given `email` address is invalid.
- 400**101**: Any of the dates given is invalid (can be `next` or `birthday`).
- 400**102**: The given `gender` is invalid (needs to be either `f`, `m` or `x`).
- 400**103**: The given language ID could not be found.
- 400**501**: The given `contact` ID is invalid.
- 400**502**: The given `language` ID is invalid.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

3.1.5 Update a contact

To update an existing contact, perform a `PUT` operation on him, e.g. `PUT /contacts/bca9`. You have to send the same data as when creating a new contact, except that you cannot control the list association of the contact in your request (so `lists` is not possible). It is however possible to trigger events.

To manage a contact's lists, use the `lists` subresource (`/contacts/bca9/lists`).

Note: You need appropriate permission to update contacts.

You don't have to send **all** fields of the contact. To avoid conflicts and accidental overwrites, only send the fields that you want to change.

Request Example

The following request will update Amy's email address and then trigger an event to do some magic.

```
PUT /api/v2/contacts/bca9 HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 123

{
  "contact": {
    "email": "amy@tardis.co.uk"
  },
  "events": [
    "lm44"
  ]
}
```

Response Example

The response contains the updated contact.

```
{
  "data": {
    "id": "bca9",
    "name1": "Amy",
    "name2": "Pond",
    "email": "amy@tardis.co.uk"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `contact` element.
- 400**100**: The given `email` address is invalid.
- 400**101**: Any of the dates given is invalid (can be `next` or `birthday`).
- 400**102**: The given `gender` is invalid (needs to be either `f`, `m` or `x`).
- 400**103**: The given language ID could not be found.
- 400**501**: The given `contact` ID is invalid.
- 400**502**: The given `language` ID is invalid.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.

3.1.6 Delete a contact

There are two ways to handle ``deleting'' a user. Either perform a real deletion or just move the contact into the trash bin.

To just do a ``soft delete'', **update** the contact (see above) and set the `recyclebin` flag to `true` (to un-delete the contact, do the same but set the flag to `false`).

```
{
  "contact": {
    "recyclebin": true
  }
}
```

To permanently delete a contact instead, perform a `DELETE` request on him, e.g. `DELETE /contacts/bca9`. There is no additional payload in this request (so you cannot trigger any events).

Note: You need appropriate permission to delete contacts.

Warning: **There is no un-do for this.** Once this operation has been executed, the contact is gone for good.

Request Example

This request will delete the contact with the ID `lm9`.

```
DELETE /api/v2/contacts/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The response will just contain a single `success` element.

```
{
  "success": true
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.

3.2 Lists

Lists are collections of *contacts*. Contacts can be in multiple lists at the same time and can be freely added to or removed from lists.

A list belongs to exactly one *folder*. Folders can be nested, but lists cannot (i.e. there are no sub-lists).

Each list has a label, which has to be unique within a folder.

This page describes how to manage what lists exists, how they are named and structured. To move contacts to and from lists, use their *list subresource* instead.

You can *embed* a the contacts on a list (`embed=cotacts`) as well as the folder in which a list is placed (`embed=folder`).

3.2.1 Structure

A JSON representation of a list consists of the following fields:

`id` **(string)** automatically generated, unique ID

`label` **(string)** the list's label

`test` **(boolean)** whether or not this is a test list, used for sending tests (i.e. testing how much a mailing looks like spam)

`disabled` **(boolean)** disabled lists are hidden when creating a new mailing

`folder` **(string)** the ID of the folder where the list is located

Example:

```
{
  "id": "lm9",
  "label": "New Subscribers",
  "test": true,
  "disabled": false,
  "folder": "bc23"
}
```

3.2.2 Fetch all lists

To get a list of your lists, you have to send a `GET` request to `/lists`. You will always receive all lists.

Request Example

```
GET /api/v2/lists HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The `data` element in the response contains a flat list of lists.

```
{
  "data": [
    {
      "id": "lm9",
      "label": "Tests",
      "test": true
    },
    {
      "id": "kn9n",
      "label": "New Subscribers",
      "test": false
    }
  ]
}
```

Error Conditions

There are no expected special error conditions.

3.2.3 Fetch a single list

To fetch a single list, simply perform a GET request to `/lists/<list-id>`.

Request Example

```
GET /api/v2/lists/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

```
{
  "data": {
    "id": "lm9",
    "label": "New Subscribers",
    "test": true
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given list ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested list was not found.

3.2.4 Create a new list

To create a new list, perform a POST request on the list collection, that is `/lists`. In your request payload, you have to send the JSON structure with the list data like the label and the folder to put the list in. The list ID will be automatically generated.

Note: You need appropriate permission to create new lists.

The `folder` ID is optional. If not given, the list will be placed in the root folder.

The request payload must be a JSON object, having a `list` element which contains the list information. It should look like this:

```
{
  "list": {
    "label": "New Subscribers",
    "folder": "bc23",
    "test": false
  }
}
```

For the sake of convenience you are able to create a new list *and* add contacts with one single request. To do so, include another `contacts` element in your payload, which must be an array of contacts IDs.

```
{
  "list": {
    "label": "New Subscribers",
    "folder": "bc23",
    "test": false
  },
  "contacts": [
    "kn9n",
    "abc3",
    "g57a"
  ]
}
```

Note that invalid IDs are silently ignored and will not lead to cancelled requests.

Request Example

The following request will create a new list, *New Subscribers*.

```
POST /api/v2/lists HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "list": {
    "label": "New Subscribers",
    "folder": "bc23",
    "test": false
  }
}
```

Response Example

In response, you will get the list back (the same as if you did `GET /lists/<id>`).

```
{
  "data": {
    "id": "n4g3"
    "label": "New Subscribers",
    "folder": "bc23",
    "test": false
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `list` element.
- 400**002**: The `folder` ID is invalid.
- 400**003**: The folder could not be found.
- 400**004**: The given `label` is empty.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

HTTP 409 (Conflict)

- 409**001**: There is already a list with that label in the given folder.

3.2.5 Update a list

To update an existing list, perform a `PUT` operation on it, e.g. `PUT /lists/bca9`. You have to send the same data as when creating a new list, except that you cannot control the contacts in the list in your request (so `contacts` is not possible).

Note: You need appropriate permission to update lists.

You don't have to send **all** fields of the list. To avoid conflicts and accidental overwrites, only send the fields that you want to change.

Request Example

The following request will update a list's label.

```
PUT /api/v2/lists/bca9 HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 123

{
  "list": {
    "label": "A New Hope for Lists"
  }
}
```

Response Example

The response contains the updated list.

```
{
  "data": {
    "id": "bca9",
    "label": "A New Hope for Lists",
    "folder": "bc23"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `list` element.
- 400**002**: The `folder` ID is invalid.
- 400**003**: The folder could not be found.
- 400**004**: The given `label` is empty.
- 400**501**: The given list ID is invalid.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

HTTP 409 (Conflict)

- 409**001**: There is already a list with that label in the given folder.

HTTP 404 (Not Found)

- 404**001**: The requested list could not be found.

3.2.6 Delete a list

To delete a list, perform a `DELETE` request on it, e.g. `DELETE /lists/bca9`. Note that deleting a list does **not** delete the contacts in it.

Note: You need appropriate permission to delete lists.

Request Example

```
DELETE /api/v2/lists/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The response will just contain a single `success` element.

```
{
  "success": true
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 404 (Not Found)

- 404**001**: The requested folder was not found.

3.3 Contact Lists

This subresource on *contacts* can be used to manage the lists a contact is assigned to. It's available as `/lists` for each contact.

If you want to change what lists exist and where they are located (in what folder), use the *lists resource* instead.

There are no sub-resources available for *embedding*.

3.3.1 Structure

Each contact-list association is represented like any normal list representation, except custom links.

```
{
  "id": "lm9",
  "label": "My Test List",
  "test": true,
  "disabled": false,
  "folder": "bag452",
  "links": [
    {
      "rel": "self",
      "uri": "/v2/lists/lm9"
    },
    {
      "rel": "remove",
      "uri": "/v2/contacts/ha64g/lists/lm9"
    },
    {
      "rel": "folder",
      "uri": "/v2/folders/bag452"
    }
  ]
}
```

3.3.2 Get current lists

To get a list of all lists a contact is currently on, perform a `GET` request on the subresource of a given contact, e.g. `/contacts/lm9/lists`. You will get a flat list of lists back.

Request Example

```
GET /api/v2/contacts/lm9/lists HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

```
{
  "data": [
    {
      "id": "kn9n",
      "label": "Another Test List"
    },
    {
      "id": "lm9",
      "label": "Testlist"
    }
  ]
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given `contact` ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.

3.3.3 Add contact to a list

To put an existing contact on an existing list, issue a `PUT` request on that particular contact list's subresource, e.g. `/contact/lm9/lists/hg23`.

Note: You need appropriate permission to add a contact to a list.

Request Example

Note how this request does not have a body (``payload'').

```
PUT /api/v2/contacts/bca9/lists/gta546 HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 0
```

Response Example

The response contains the newly created contact-list association.

```
{
  "data": {
    "id": "gta546",
    "label": "Some Random List"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given `contact` ID is invalid.
- 400**502**: The given `list` ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.
- 404**002**: The requested list could not be found.

HTTP 409 (Conflict)

- 404**001**: The requested contact is already on the requested list.

3.3.4 Remove contact from a list

To remove a contact from a list, issue a `DELETE` request on the association. Note that this does not delete either the contact or the list, but only the connection between the two.

Note: You need appropriate permission to remove contacts from lists.

Request Example

```
DELETE /api/v2/contacts/bca9/lists/gta546 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The response will just contain a single `success` element.

```
{
  "success": true
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given `contact` ID is invalid.
- 400**502**: The given `list` ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested contact could not be found.
- 404**002**: The requested list could not be found.
- 404**003**: The requested contact-list association could not be found.

3.4 Folders

Folders are used to structure *contact lists*. A folder can contain an unlimited number of lists and sub folders. Each folder has therefore exactly one parent folder.

There is a predefined root folder. The root folder cannot be edited or moved. All folders which are not epxlicitely put into a folder are automatically children of this root folder.

You can *embed* a folder's lists (`embed=lists`) as well as its child folders (`embed=children`).

3.4.1 Structure

A JSON representation of a folder consists of the following fields:

`id` **(string)** automatically generated, unique ID

`label` **(string)** the folder label, a freely choosable text

`parent` **(string)** the parent folder's ID, is empty for the root folder (cannot be empty for user-created folders)

Example:

```
{
  "id": "lm9",
  "label": "My folder",
  "parent": "ha62g"
}
```

3.4.2 Fetch root folder

To get the top-most (``root'') folder, perform a GET operation on /folders. You will receive **a list** of folders, like when you fetch child folders, but this list will always only contain one element.

Request Example

```
GET /api/v2/folders HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

```
{
  "data": [
    {
      "id": "lm9",
      "label": "My Folder"
    }
  ]
}
```

Error Conditions

There are no expected special error conditions.

3.4.3 Fetch a single folder

To fetch a single folder, simply perform a GET request to /folders/<folder-id>.

Request Example

```
GET /api/v2/folders/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

```
{
  "data": {
    "id": "lm9",
    "label": "My Folder"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given folder ID is invalid.

HTTP 404 (Not Found)

- 404**001**: The requested folder was not found.

3.4.4 Fetch child folders

To get all child folders of a given folder, you can access the /children subresource by issuing a GET request. Note that this only includes direct children (i.e. it's not recursive) and that you cannot manipulate the children list using this subresource.

Request Example

```
GET /api/v2/folders/lm9/children HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The data element in the response contains a flat list of folders.

```
{
  "data": [
    {
      "id": "kn62",
      "label": "First Subfolder"
    },
    {
      "id": "kn9n",
      "label": "Second Subfolder"
    }
  ]
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**501**: The given folder ID is malformed.

HTTP 404 (Not Found)

- 404**001**: The requested folder was not found.

3.4.5 Create a new folder

To create a new folder, perform a `POST` request on the folder collection, that is `/folders`. In your request payload, you have to send the JSON structure with the folder data like the label and the parent folder. The folder ID will be automatically generated.

Note: You need appropriate permission to create new folders.

The `parent` ID is optional. If not given, the list will be placed in the root folder.

The request payload must be a JSON object, having a `folder` element which contains the folder information. It should look like this:

```
{
  "folder": {
    "label": "New Subscribers",
    "parent": "bc23"
  }
}
```

Request Example

The following request will create a new folder, *My Awesome Folder*.

```
POST /api/v2/folders HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "list": {
    "label": "My Awesome Folder",
    "parent": "bc23"
  }
}
```

Response Example

In response, you will get the folder back (the same as if you did `GET /folders/<id>`).

```
{
  "data": {
    "id": "n4g3",
    "label": "My Awesome Folder",
    "parent": "bc23"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `folder` element.
- 400**002**: The `parent` folder ID is invalid.
- 400**003**: The parent folder could not be found.
- 400**004**: The given `label` is empty.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

3.4.6 Update a folder

To update an existing folder, perform a `PUT` operation on it, e.g. `PUT /folders/bca9`. You have to send the same data as when creating a new folder.

Note: You need appropriate permission to update folders.

You don't have to send **all** fields of the folder. To avoid conflicts and accidental overwrites, only send the fields that you want to change.

Request Example

The following request will update a folder's label.

```
PUT /api/v2/lists/bca9 HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 123

{
  "folder": {
    "label": "An Even Better Title"
  }
}
```

Response Example

The response contains the updated folder.

```
{
  "data": {
    "id": "bca9",
    "label": "An Even Better Title",
    "parent": "bc23"
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `folder` element.
- 400**004**: The given `label` is empty.
- 400**501**: The given folder ID is invalid.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

HTTP 404 (Not Found)

- 404**001**: The requested folder was not found.

3.4.7 Delete a folder

To delete a folder, perform a `DELETE` request on it, e.g. `DELETE /folders/bca9`.

Warning: Deleting a folder will also delete all child folders as well as all contact lists contained in it. It does **not** delete the contacts, though.

Note: You need appropriate permission to delete folders.

Request Example

```
DELETE /api/v2/folders/lm9 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

The response will just contain a single `success` element.

```
{
  "success": true
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 404 (Not Found)

- 404**001**: The requested folder was not found.

3.5 Sendings

This resource can be used to manage sendings (mailings); at the moment, you can just start a new mailing. In a mailing, you are sending an *asset* to a number of contacts (controlled via the *lists*).

You can *embed* the lists (`embed=lists`) used by a mailing.

3.5.1 Structure

A JSON representation of a mailing consists of the following fields:

`id` **(string)** automatically generated, unique ID

`created` **(date, ISO 8601[4])** the date and time when the asset has been created

`creator` **(string)** the ID of the user who created this asset

`label` **(string)** the mailings's label

`asset` **(string)** ID of the asset to send in the mailing

`category` **(string)** ID of the mailing's category (categories can be used to control default behaviour for various settings of a mailing)

`approved` **(boolean)** a mailing is sent once it has been approved

`schedule` **(date, ISO 8601[5])** the date and time for when the sending is scheduled to start

Example:

```
{
  "id": "hd72h",
  "label": "My Test Mailing",
  "created": "2014-05-12T12:23:34Z",
  "creator": "lm9",
  "asset": "8hb23",
  "category": "gdt34",
  "approved": false,
  "schedule": "2014-05-13T10:00:00Z"
}
```

3.5.2 Create a new mailing

Creating a new mailing works by sending a `POST` request to the `/mailings` resource. If you want to, the mailing can start immediately or at a later point in time. In both cases, the actual sending is perform asynchronously in the background, so you don't need and can't wait for the mailing to be finished.

Note: You need appropriate permission to create new mailings.

The request payload must be a JSON object, having a `mailing` element which contains the mailing information. It should look like this:

```
{
  "mailing": {
    "asset": "8hb23",
    "category": "gdt34",
    "label": "My Test Mailing"
  }
}
```

To set the contact lists that are included for this mailing, add another element, `lists` to your request:

```
{
  "mailing": {
    "asset": "8hb23",
    "category": "gdt34",
    "label": "My Test Mailing"
  },
  "lists": [
    "123bcg",
    "7ah3m",
    "ha72h"
  ]
}
```

Creating a mailing without any lists is a pointless endeavour.

Scheduling

Instead of sending a fixed `schedule` date, you have to give a symbolic value.

`now` If you set `schedule` to `now`, the sending will start right away, if `approved` has been set to `true`.

`today` With this scheduling type, you also have to send a `time` value, being a ISO 6801 formatted time without seconds (i.e. `HH:MM`, seconds would simply be ignored). The date will be simply the current date.

`later` Much like `today`, but you also have to send a `date` value with an ISO 6801 formatted date (i.e. `YYYY-MM-DD`).

The reason for splitting the scheduling up into these three modes is that the server may adjust the automatically determined values (like the date for `today`) to accomodate server workload.

Note: All dates and times are interpreted as UTC values.

Of course, you cannot give a schedule date that is in the past, the same way that you cannot plan more than 10 years ahead.

This example will send a mail on Christmas at 12:00 (UTC):

```
{
  "mailing": {
    "asset": "8hb23",
    "category": "gdt34",
    "label": "My Test Mailing",
    "schedule": "later",
    "date": "2015-12-24",
    "time": "12:00"
  },
  "lists": [
    "123bcg"
  ]
}
```

Request Example

```
POST /api/v2/mailings HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "mailing": {
    "asset": "8hb23",
    "category": "gdt34",
    "label": "My Test Mailing",
    "schedule": "later",
    "date": "2015-12-24",
    "time": "12:00"
  },
  "lists": [
    "123bcg",
    "7ah3m",
    "ha72h"
  ]
}
```

Response Example

You will receive the newly created mailing back.

```
{
  "data": [
    {
      "id": "hd72h",
      "label": "My Test Mailing",
      "created": "2014-05-12T12:23:34Z",
      "creator": "lm9",
      "asset": "8hb23",
      "category": "gdt34",
      "approved": false,
      "schedule": "2014-05-13T10:00:00Z"
    }
  ]
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `mailing` element.
- 400**002**: No `asset` ID given.
- 400**003**: No `category` ID given.
- 400**004**: No or empty `label` given.

- 400**005**: The given `schedule` is invalid.
- 400**006**: The `asset` ID is invalid.
- 400**007**: The `category` ID is invalid.
- 400**008**: The requested `asset` could not be found.
- 400**009**: The requested `category` could not be found.
- 400**010**: The given sending time is malformed.
- 400**011**: The given sending time is invalid.
- 400**012**: The given sending date is malformed.
- 400**013**: The given sending date is invalid.
- 400**014**: The given sending date and time are in the past.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

3.6 Assets

Assets are templates for *mailings*, written in HTML with placeholders. You can also give a plaintext, to create a multi-part asset.

This resource allows you to create new assets.

There are no sub-resources available for *embedding*.

3.6.1 Structure

The JSON representation of an asset is as follows:

`id` **(string)** automatically generated, unique ID

`title` **(string)** the asset's title, a freely choosable string

`name` **(string)** the asset's name, a freely choosable string

`subject` **(string)** when sent in a mailing, this will be the mailing's subject

`format` **(string)** the asset's format, can be either `html` for an asset containing only HTML, or `multipart` for an asset with both HTML and plaintext content

`created` **(date, ISO 8601[6])** the date and time when the asset has been created

`creator` **(string)** the ID of the user who created this asset

Example:

```
{
  "id": "hd72h",
  "created": "2014-05-12T12:23:34Z",
  "creator": "lm9",
  "name": "An Asset to send",
  "title": "My Awesome Asset",
  "subject": "Our Newsletter for Jan 2014",
  "format": "multipart"
}
```

3.6.2 Create a new asset

Creating works by sending a `POST` request to the `/assets` resource.

Note: You need appropriate permission to create new assets.

The request payload must be a JSON object, having a `asset` element which contains the asset information. It should look like this:

```
{
  "asset": {
    "title": "My Awesome Asset",
    "name": "An Asset to send",
    "subject": "Our Newsletter for Jan 2014",
    "html": "<!DOCTYPE><html><p>Hello [Nachname], this is our newsletter!</p></html>",
    "linktracking": true,
    "multipart": false
  }
}
```

Depending on your needs, you can include either `plain` (for the plaintext) or `html` in the payload. You can also send both, if you set `multipart` to `true`.

If you fear that your HTML markup might get mangled during transmission, you can also Base64 encode it and send it as `html_base64` instead. This also works for the plaintext.

```
{
  "asset": {
    "title": "My Awesome Asset",
    "name": "An Asset to send",
    "subject": "Our Newsletter for Jan 2014",
    "html_base64": "PCFETONUWVBFIGhObWw+Li4u",
    "linktracking": true,
    "multipart": false
  }
}
```

The non-encoded versions have preceendence, however. So if you send both `html` and `html_base64`, `html` wins.

The following JSON fields are available for this endpoint:

`title` **(string)** **mandatory**; the asset's title

`name` **(string)** **mandatory**; the asset's name

`subject` **(string)** **mandatory**; the asset's subject

`html` **/** `html_base64` **(string)** **mandatory**; the asset's HTML content (you must send one of the two, but never both)

`plain` **/** `plain_base64` **(string)** *optional*; the asset's plaintext content (never send both), is only useful if you set `multipart` to true.

`linktracking` **(boolean)** *optional*; whether or not to enable tracking the links that are embedded in the asset's HTML), defaults to `false` if not given

`multipart` **(boolean)** *optional*; whether or not this asset is a multipart asset, defaults to `false` if not given

Request Example

This example creates a new multipart asset.

```
POST /api/v2/assets HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "asset": {
    "title": "My Awesome Asset",
    "name": "An Asset to send",
    "subject": "Our Newsletter for Jan 2014",
    "html": "<!DOCTYPE><html><p>Hello [Nachname], this is our newsletter!</p></html>",
    "plain": "Hello [Nachname], this is our newsletter!",
    "linktracking": false,
    "multipart": true
  }
}
```

Response Example

You will receive the newly created asset back.

```
{
  "data": {
    "id": "hd72h",
    "created": "2014-05-12T12:23:34Z",
    "creator": "lm9",
    "name": "An Asset to send",
    "title": "My Awesome Asset",
    "subject": "Our Newsletter for Jan 2014",
    "format": "multipart"
  }
}
```

The `format` field can be either `html` or `multipart`.

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The request does not contain an `asset` element.
- 400**002**: No `title` given.
- 400**003**: No `name` given.
- 400**004**: No `subject` given.
- 400**005**: No `html` and no `html_base64` given.
- 400**006**: The given title is empty.
- 400**007**: The given name is empty.
- 400**008**: The given subject is empty.

- 400**009**: The given HTML code is empty.
- 400**900**: The request payload is not valid JSON.
- 400**901**: The request payload does not decode into a JSON object.

3.7 Files

In many cases, *assets* (written in HTML) will contain links to embedded resources, like images. amity allows you to host these images on its servers and this resource can be used to upload files.

Note: There are only a few restrictions on the files you can upload. You are advised to be sensible and not store gigabytes worth of files on amity servers. Abusing this hosting feature may lead to your account or this resource being suspended.

Note: Similarly, abusing amity as your generic file hoster is not allowed. Uploaded files may only be used for mailings.

There are no sub-resources available for *embedding*.

3.7.1 Structure

The JSON representation of a file consists of the following fields:

`url` **(string)** the absolute URL of the file hosted on amity servers

`size` **(int)** the filesize in bytes

Example:

```
{
  "url": "http://news.mybusiness.com/public/img/companylogo.png",
  "size": 54281
}
```

3.7.2 Upload a new file

In order to upload a file, it has to be hosted somewhere already. amity can download the file from a remote location and store it on its own server. Within the API, you have to perform a `PUT` request on `/files/<type>/<your-desired-filename>`.

Note: You need appropriate permission to upload files.

The `type` in the URI must be either `img` or `data`, depending on whether you upload an image or something else (like a PDF).

The request payload must be a JSON object, having a `file` element which contains the source information. It should look like this:

```
{
  "file": {
    "source": "http://www.example.com/files/myfile.png"
  }
}
```

Note that the following restrictions apply here:

- Existing files on the amity server cannot be overwritten (because that could interfere with older mailings).
- The target filename must not be empty and follow the form `something.ext` (i.e. you cannot create ``hidden'' files, e.g. `.png`).
- The following file extensions are allowed: `js`, `css`, `html`, `htm`, `pdf`, `gif`, `jpg`, `jpeg`, `png`, `tiff`, `xml` and `txt`
- The source URL must be an absolute URL, using HTTP or HTTPS. Note that the validity of an SSL certificate is **not** checked, so do not use this to upload sensitive files.
- The connection attempt times out after 3 seconds.
- The file download may at most last 20 seconds before it's interrupted and cancelled.
- There may be at most 3 HTTP redirects involved.

Request Example

This example uploads the company logo as an image.

```
PUT /api/v2/files/img/companylogo.png HTTP/1.1
Host: news.mybusiness.com
Content-Type: application/json; charset=UTF-8
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
Content-Length: 93

{
  "file": {
    "source": "http://www.initech.com/assets/images/logo.png"
  }
}
```

Response Example

You will receive some information about the new file back.

```
{
  "data": {
    "url": "http://news.mybusiness.com/public/img/companylogo.png",
    "size": 54281
  }
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The requested filename does not have a name (e.g. is `.txt`) or no extension.
- 400**002**: The requested filename has an invalid extension.
- 400**003**: The requested filetype is invalid (must be either `img` or `data`).
- 400**004**: The request payload does not contain a `file` element.

- 400**005**: The request payload does not contain a `source` element.
- 400**006**: The given `source` URL is invalid.
- 400**007**: The given `source` URL is neither HTTP nor HTTPS.
- 400**008**: The given `source` URL points to localhost.

HTTP 409 (Bad Request)

- 409**001**: The request file exists already.

HTTP 502 (Bad Gateway)

- 502**001**: The request to the source server has timed out.
- 502**002**: The source server responded with an error.

3.8 Bounces

A bounce means that an ail was sent to a specific address, but the mail server that received the email for that person has sent it back, saying it could not be delivered. Those bounces are recorded and made available via this resource. API clients can access the log and see which *contact* could not receive a sending.

There are no sub-resources available for *embedding*.

3.8.1 Structure

Each bounce record is presented in JSON as follows:

`id` **(unsigned int)** automatically generated, unique ID

`mailing` **(string)** the ID of the affected mailing

`email` **(string)** the recipient's email address

`date` **(date, ISO 8601[7])** the date and time when the bounce occured

`type` **(string)** the bounce type, one of `hardbounce`, `softbounce` or `autoresponse`

`reaction` **(string)** amity's reaction to the bounce, one of `none`, `excluded`, `recycled`, `recycledexcluded` or `deleted`

`error` **(string)** a human readable description of the bounce reason, e.g. `Mailbox full`, as defined in RFC 1893[8]

Example:

```
{
  "id": 1624,
  "mailing": "hga52",
  "email": "kenny@toronto.ca",
  "date": "2014-04-03T12:45:07Z",
  "type": "softbounce",
  "reaction": "excluded",
  "error": "Mailing list expansion problem"
}
```

[8] http://www.ietf.org/rfc/rfc1893.txt

3.8.2 Fetching bounces

To fetch bounce records, you have to send a GET request to /bounces. In your query string, you can filter the result by email address, mailing, month or the bounce reason. You can also combine multiple filters (which is what you usually will do to get meaningful data).

Filtering by email address works just like filtering contacts. You can give the email query string parameter, which must be at least 3 characters long. If you don't also give substring, an exact match search will be performed, otherwise the value of email must just be contained somewhere in the email address.

To further restrict the result, you can give mailing, which needs to be the ID of one mailing (e.g. ?mailing=hgd523). An invalid mailing ID will lead to an error message.

You can restrict the time frame the result by using the month parameter. It needs to be a string in the form of YYYY-MM (e.g. 2014-05 for May 2014). Invalid months will lead to an error.

Lastly, you can filter the result by the bounce reaction, using the reaction parameter. It needs to be one of the following values:

- recycled
- excluded
- recycledexcluded (recycled and excluded)
- anything (everything except *none*)

Request Example

The following request will fetch all bounces of April 2014 for contacts having an email address containing .ca (which is more or less saying ``all Canadian contacts'').

```
POST /api/v2/bounces?email=.ca&substring=1&month=2014-04 HTTP/1.1
Host: news.mybusiness.com
Accept: application/json
X-Client: lm9
X-Signature: beccf85d099bb5168c635d748198c94c065ac85f1e4ba90fcc989d1db6a471fb
```

Response Example

You will receive a flat list of bounces.

```
{
  "data": [
    {
      "id": 1624,
      "mailing": "hga52",
      "email": "kenny@toronto.ca",
      "date": "2014-04-03T12:45:07Z",
      "type": "softbounce",
      "reaction": "excluded",
      "error": "Mailing list expansion problem"
    },
    {
      "id": 1629,
      "mailing": "bc61",
      "email": "some@one.ca",
      "date": "2014-04-06T07:23:18Z",
      "type": "hardbounce",
      "reaction": "excluded",
      "error": "System incorrectly configured"
    }
  ]
}
```

Error Conditions

Read more about the error handling in the *API basics*.

HTTP 400 (Bad Request)

- 400**001**: The given `email` address is too short (needs to be at least 3 characters long).
- 400**002**: The given `mailing` ID is invalid.
- 400**003**: The given `month` is invalid (i.e. not in the `YYYY-MM` format).
- 400**004**: The given `reaction` is invalid.