REST Principles and Conventions

Olivier Liechti & Simon Oulevay
COMEM Web Services 2016

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

URL Structure (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In most applications, you have **several types of resources** and there are relationships between them:
  - In a blog management platform, **Blog** authors create **BlogPosts** that can have associated **Comments**.
  - In a school management system, **Courses** are taught by **Professors** in **Rooms**.
- When you design your REST API, you have to define identifiers and URL patterns to give access to the resources. There are often different ways to define these.

URL Structure (2)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
GET /blogs/amtBlog/posts/892/comments HTTP/1.1
```

VS

```
GET /comments?blogId=amtBlog&postId=892 HTTP/1.1
```

```
GET /professors/liechti/courses HTTP/1.1
```

VS

```
GET /courses?professorName=liechti HTTP/1.1
```

URL Structure (3)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- How do you choose between these two approaches (flat vs deep structure)? There is no “right or wrong” answer and you find both styles in popular REST APIs.
- One rule that you can use to make your decision is whether you have an **aggregation** or **composition** relationship between resources. In other words, does the existence of one resource depend on the existence of another one. If that is the case, then it probably makes sense to use a hierarchical URL pattern.
- For instance, the existence of a **comment** depends on the existence of a **blog entry**. For this reason, I would probably go for:
  - /blogs/amtBlog/posts/892/comments/
- On the other hand, the existence of a **course** is not dependent on the existence of a **professor** (if one of you murders me, someone else will takeover the AMT course). Therefore, I would likely go for:
  - /courses?professorName=liechti

Linked resources

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In most domain models, you have relationships between domain entities:
  - Example: one-to-many relationship between "Company" and "Employee"
- Imagine that you have the following REST endpoints:
  - GET /companies/{id} to retrieve one company by id
- Question: what payload do you expect when invoking this URL?

Linked resources

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
{
  "name": "Apple",
  "address" : {},
  "employees" : [
    {
      "firstName" : "Tim",
      "lastName" : "Cook",
      "title" : "CEO"
    },
    {
      "firstName" : "Jony",
      "lastName" : "Ive",
      "title" : "CDO"
    }
  ]
}
```

Embedding
(reduces "chattiness", often good if there are "few" linked resources; company-employee is not a good example)

```
{
  "name": "Apple",
  "address" : {},
  "employeeIds" : [134, 892, 918, 9928]
}
```

References via IDs
(the client must know the URL structure to retrieve an an employee)

```
{
  "name": "Apple",
  "address" : {},
  "employeeURLs" : [
    "/companies/89/employees/134",
    "/companies/89/employees/892",
    "/companies/89/employees/918",
    "/contractors/255/employees/9928",
 ]
}
```

References via URLs
(better: decouples client and server implementation)

Resources & Actions (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In some situations, it is fairly easy to **identify resource** and to map related **actions** to HTTP request patterns.
- For instance, in an **academic management system**, one would probably come up with a Student resource and the associated HTTP request patterns:
  - `GET /students` to retrieve a list of students
  - `GET /students/{id}` to retrieve a student by id
  - `POST /students` to create a student
  - `PUT /students/{id}` to update a student
  - `DELETE /students/{id}` to delete a student

Resources & Actions (2)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Some situations are not as clear and subject to debate. For instance, let us imagine that with your system, you can **exclude students** if they have cheated at an exam. How do you implement that with a REST API?
- Some people would propose something like this:
  - `POST /students/{id}/exclude`
    - Notice that “exclude” is a verb. In that case, there is no request body and we do not introduce a new resource (we only have student).
- Other people (like me) would prefer something like this:
  - `POST /students/{id}/exclusions/`
    - In that case, we have introduced a new resource: an exclusion request (think about a form that the Dean has to fill out and file). In that case, we would have a request body (with the reasons for the exclusion, etc.).

Pagination (1)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In most cases, you need to deal with **collections of resources that can grow** and where it is not possible to get the list of resources in a single HTTP request (for performance and efficiency reasons).
  - `GET /phonebook/entries?zip=1700`
- Instead, you want to be able to **successively retrieve chunks of the collection**. The typical use case is that you have a UI that presents a "page" of *n* resources, with controls to move to the previous, the next or an arbitrary page.
- In terms of API, it means that you want to be able to request a page, by providing an offset and a page size. In the response, you expect to find the number of results and a way to display navigation links.

Pagination (2)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **At a minimum, what you need to do:**
  - When you **process an HTTP request**, you need a **page number** and a **page size**. You can use these to query a page from the database (do not transfer the whole table from the DB to the business tier!). You need to decide how the client is sending these values (query params, headers, defaults values).
  - When you generate the **HTTP response**, you need to **send the total number of results** (so that the client can compute the number of pages and generate the pagination UI), **and/or** send **ready-to-use links** that point to the first, last, prev and next pages. You use HTTP headers to send these informations.

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

Pagination (3)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Examples:**
  - http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#pagination
  - https://developer.github.com/v3/#pagination
  - https://dev.evrythng.com/documentation/api

http://tools.ietf.org/html/rfc5988#page-6

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

Pagination (4)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Example:**

Pagination

When retrieving a collection the API will return a paginated response. The pagination information is made available in the `X-Pagination` header containing four values separated by semicolons. These four values respectively correspond to the number of items per page, the current page number (starting at 1), the number of pages and the total number of elements in the collection.

For instance, the header `X-Pagination: 30;1;3;84` has the following meaning:

- `30` : There are 30 items per page
- `1` : The current page is the first one
- `3` : There are 3 pages in total
- `84` : There is a total of 84 items in the collection

To iterate through the list, you need to use the `page` and `pageSize` query parameters when doing a `GET` request on a collection. If you do not specify those parameters, the default values of 1 (for `page`) and 30 (for `pageSize`) will be assumed.

Example: The request `GET /myResources?page=2&pageSize=5 HTTP/1.1` would produce a response comparable to the following:

```
HTTP/1.1 200 OK
X-Pagination: 5;2;7;35
...

{
  [
    { "id": 6 },
    { "id": 7 },
    { "id": 8 },
    { "id": 9 },
    { "id": 10 }
  ]
}
```

Sorting and Filtering

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- Most REST APIs provide a mechanism to sort and filter collections.
- Think about GETting the list of all students who have a last name starting with a ‘B’, or all students who have an average grade above a certain threshold.
- Think about GETting the list of all students, sorted by rank or by age.
- The standard way to specify the sorting and filtering criteria is to use query string parameters.
- IMPORTANT: be consistent across your resources. The developer of client applications should be able to use the same mechanism (same parameter names and conventions) for all resources in your API!

Authentication

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In most cases, REST APIs are invoked over a secure channel (**HTTPS**).
- For that reason, the **basic authentication scheme** is often considered acceptable.
- Every request contains an “Authorization” header that contains either **user credentials** (user id + password) or some kind of **access token** previously obtained by the user.
- When the server receives an HTTP request, it extracts the credentials from the HTTP header, validates them against what is stored in the database and either grants/rejects the access.

OAuth

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- In many REST APIs, OAuth 2.0 is used for **authorization** and **access delegation**:
  - when you use a **Facebook Application** (e.g. a game), you are asked whether you agree to **authorize** this third-party Application to access some of **your Facebook data** (and actions, such as posting to your wall).
  - If you agree, the Facebook Application receives a **bearer token**. When it sends HTTP requests to the **Facebook API**, it sends this token in a HTTP header (typically in the Authorization header). Because the Application has a valid token, Facebook grants access to your data.
  - In other words, using OAuth is similar to handing your car keys to a concierge.

API versioning

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

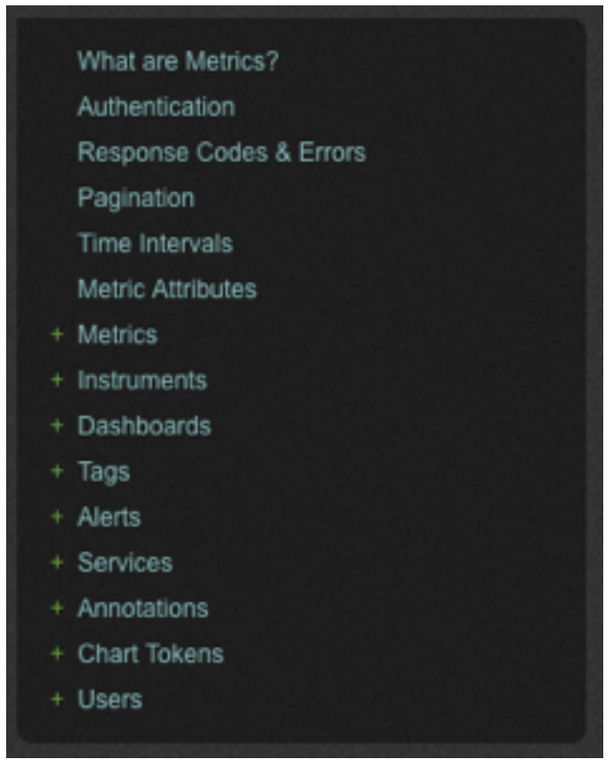
- If you think about the **medium and long term evolution of your service** (think about Twitter), your API is very likely to evolve over time:
  - You may add new types of resources
  - You may add/remove query string parameters
  - You may change the structure of the payloads
  - You may introduce new mechanisms (authentication, pagination, etc.)

- When you introduce a change in your API (and in the corresponding documentation), you will have a **compatibility issue**. Namely, you will have to support **some clients that still use the old version** of the API and **others that use the new version of the API**.

- For this reason, when you receive an HTTP request, you need to know which version is used by the client talking to you. As usual, there are different ways to pass this information (path element, query string parameter, header).

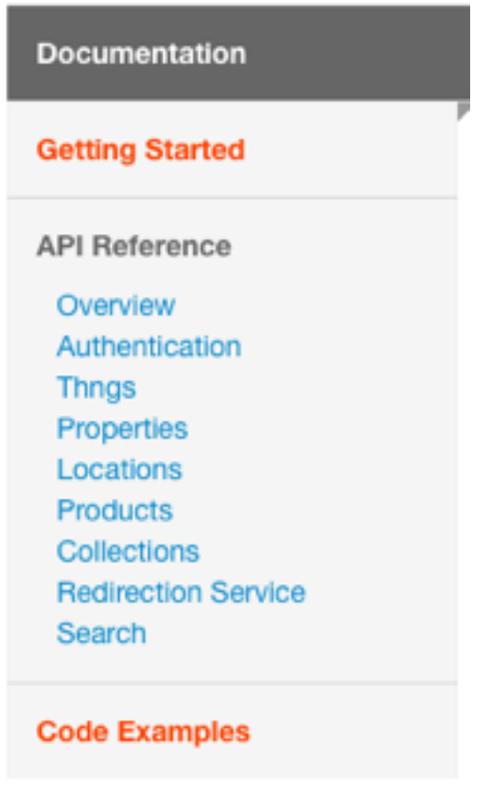
- A lot of REST APIs include the version number in the path, e.g. `http://www.myservice.com/api/v2/students/7883`

Look at Some Examples



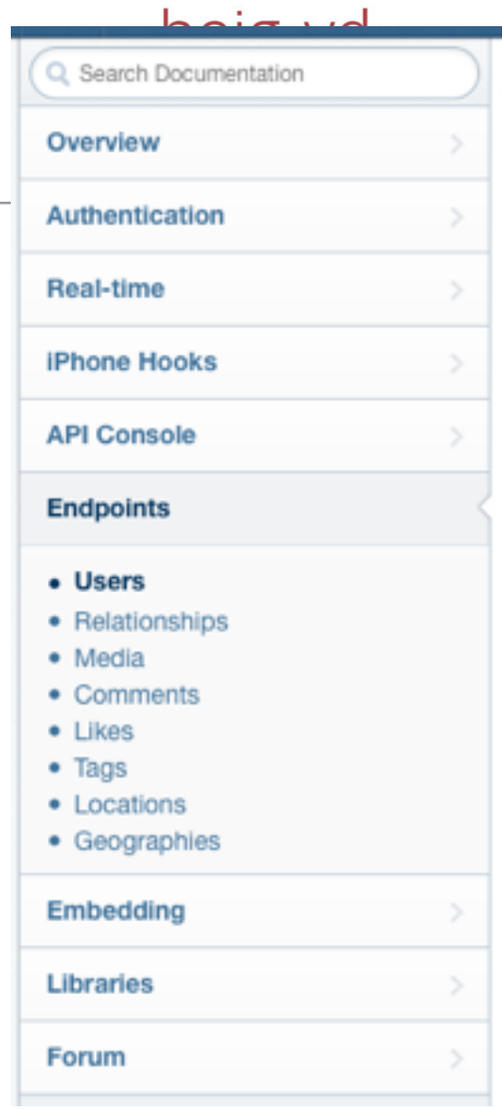
http://dev.librato.com/v1

librato



https://dev.evrythng.com/documentation/api

EVRYTHNG™



http://instagram.com/developer/endpoints/

Instagram

Short description of the resource (domain model)

heig-vd

Examples & payload structure

CRUD method description

navigator

What Are Metrics?

Metrics are custom measurements stored in Librato's Metrics service. These measurements are created and may be accessed programatically through a set of RESTful API calls. There are currently two types of metrics that may be stored in Librato Metrics, **gauges** and **counters**.

Gauges

Gauges capture a series of measurements where each measurement represents the value under observation at one point in time. The value of a gauge typically varies between some known minimum and maximum. Examples of gauge measurements include the requests/second serviced by an application, the amount of available disk space, the current value of $AAPL, etc.

Counters

Counters track an increasing number of occurrences of some event. A counter is unbounded and always monotonically increasing in any given *run*. A new *run* is started anytime that counter is reset to zero. Examples of counter measurements include the number of connections made to an app, the number of visitors to a website, the number of a times a write operation failed, etc.

Metric Properties

Some common properties are supported across all types of metrics:

`name`

Each metric has a name that is unique to its class of metrics e.g. a gauge name must be un
gauges. The name identifies a metric in subsequent API calls to store/query individual meas
can be up to 63 characters in length. Valid characters for metric names are 'A-Za-z0-9.:-_'.

`period`

The **period** of a metric is an integer value that describes (in seconds) the standard reportin
metric. Setting the period enables Metrics to detect abnormal interruptions in reporting and

Examples

Return the metric named *cpu_temp* with up to four measurements at resolution 60.

```
curl \
  -u <user>:<token> \
  -X GET \
  https://metrics-api.librato.com/v1/metrics/cpu_temp?resolution=60&count=4
```

Response Code

```
200 OK
```

Response Headers

```
** NOT APPLICABLE **
```

Response Body

```
{
  "type": "gauge",
  "display_name": "cpu_temp",
  "resolution": 60,

: {
e.com": [

  1,
_time": 1234567890,
  84.5

  1,
_time": 1234567950,
  86.7

  1,
_time": 1234568010,
  84.6

  1,
_time": 1234568070,
  89.7

(
": 0,
nsform": null,
ts_short": "&#176;F",
ua": "librato-metrics/0.7.4 (ruby; 1.9.3p194; x86_64-linux) direct-faraday/0.8.
": null,
ts_long": "Fahrenheit",
cked": true

  "Current CPU temperature in Fahrenheit",
emp"
```

GET /v1/metrics/:name

API VERSION 1.0

Description

Returns information for a specific metric. If time interval search parameters are specified will also include a set of metric measurements for the given time span.

URL

```
https://metrics-api.librato.com/v1/metrics/:name
```

Method

```
GET
```

Measurement Search Parameters

If optional time interval search parameters are specified, the response includes the set of metric measurements covered by the time interval. Measurements are listed by their originating *source* name if one was specified when the measurement was created. All measurements that were created without an explicit source name are listed with the source name **unassigned**.

`source`

Deprecated: Use **sources** with a single source name, e.g **[mysource]**.

`sources`

If **sources** is specified, the response is limited to measurements from those sources. The **sources** parameter should be specified as an array of source names. The response is limited to the set of sources specified in the array.

- What are Metrics?
- Authentication
- Response Codes & Errors
- Pagination
- Time Intervals
- Metric Attributes
- Metrics
  - GET /metrics
  - POST /metrics
  - DELETE /metrics
  - GET /metrics/:name
  - PUT /metrics/:name
  - DELETE /metrics/:name
- + Instruments
- + Dashboards
- + Tags
- + Alerts
- + Services
- + Annotations
- + Chart Tokens
- + Users

heig-vd

Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Short description of the whole domain model

Overview

The central data structure in our engine are `Thngs`, which are data containers to store all the data generated by and about any physical object. Various `Properties` can be attached to any Thng, and the content of each property can be updated any time, while preserving the history of those changes. Thngs can be added to various `Collections` which makes it easier to share a set of Thngs with other `Users` within the engine.

`Thng`

An abstract notion of an object which has location & property data associated to it. Also called Active Digital Identities (ADIs), these resources can model real-world elements such as persons, places, cars, guitars, mobile phones, etc.

`Property`

A Thng has various properties: arbitrary key/value pairs to store any data. The values can be updated individually at any time, and can be retrieved historically (e.g. "Give me the values of property X between 10 am and 5 pm on the 16th August 2012").

`Location`

Each Thng also has a special type of Properties used to store snapshots of its geographic position over time (for now only GPS coordinates - latitude and longitude).

`User`

Each interaction with the EVRYTHNG back-end is authenticated and a user is associated with each action. This dictates security access.

`Collection`

A collection is a grouping of Thngs. Col
one collection.

CRUD method description

Creating a new Product

To create a new `Product`, simply POST a JSON document that describes a product to the `/products` endpoint.

```
POST /products
Content-Type: application/json
Authorization: $EVRYTHNG_API_KEY
```

```
{
  *"fn": <String>,
  "description": <String>,
  "brand": <String>,
  "categories": [<String>, ...],
  "photos": [<String>, ...],
  "url": <String>,
  "identifiers": {
    <String>: <String>,
    ... },
  "properties": {
    <String>: <String>,
    ... },
  "tags": [<String>, ...]
}
```

Mandatory Parameters

`fn`

<String> The functional name of the product.

Optional Parameters

`description`

<String> An string that gives more details about the product, a short description.

More details about the Product resource (domain model) & payload structure

Products

Products are very similar to thngs, but instead of modeling an individual object instance, products are used to model a class of objects. Usually, they are used for general classes of thngs, usually a particular model with specific characteristics. Let's take for example a specific TV model (e.g. this one), which has various properties such as a model number, a description, a brand, a category, etc. Products are useful to captor the properties that are common to a set of thngs (so you don't replicate a property "model name" or "weight" for thousands of thngs that are individual instances of a same product category).

The Product document model used in our engine has been designed to be compatible with the hProduct microformat, therefore it can easily be integrated with the hProduct data model and applications supporting microformats.

The Product document model is as follows:

```
<Product>={
  "id": <String>,
  "createdAt": <timestamp>,
  "updatedAt": <timestamp>,
  "fn": <String>,
  "description": <String>,
  "brand": <String>,
  "categories": [<String>, ...],
  "photos": [<String>, ...],
  "url": <String>,
  "identifiers": {
    <String>: <String>,
    ... },
  "properties": {
    <String>: <String>,
        ... },
  "tags": [<String>, ...]
}
```

Cross-cutting concerns

Pagination

Requests that return multiple items will be paginated to 30 items by default. You can specify further pages with the `?page` parameter. You can also set a custom page size up to 100 with the `?per_page` parameter.

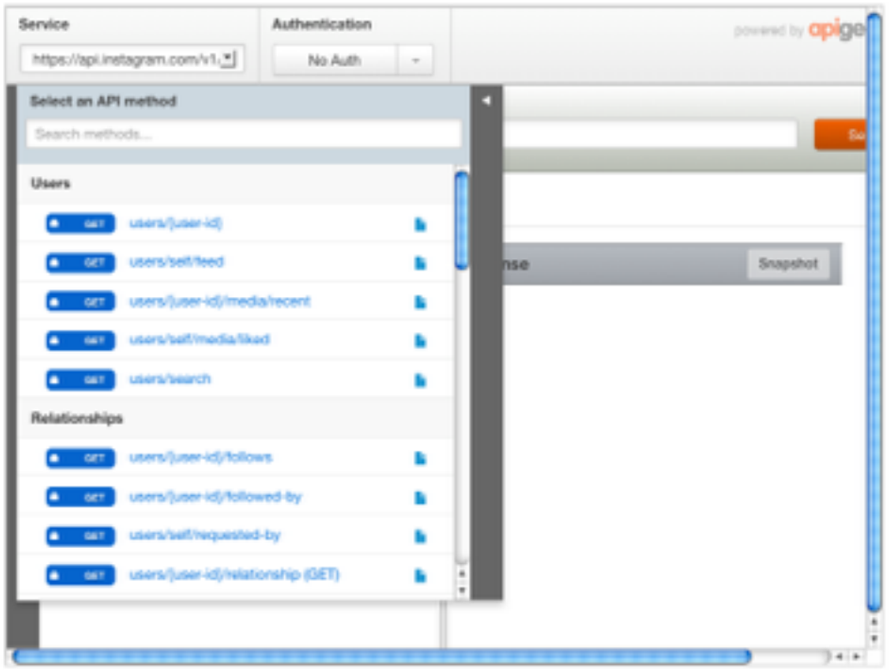
Authentication

Access to our API is done via HTTPS requests to the `https://api.evrythng.com` domain. Unencrypted HTTP requests are accepted (`http://api.evrythng.com` for low-power device without SSL support), but we **strongly** suggest to use only HTTPS if you store any valuable data in our engine. Every request to our API must include an API key using `Authorization` HTTP header to identify the user or application issuing the request and execute it if authorized.

Interactive test console

API Console

Our API console is provided by Apigee. Tap the Lock icon, select OAuth, and you can experiment with making requests to our API. See it in full screen →



List of supported CRUD methods for each resource (R; RW)

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

User Endpoints

GET	/users/ **user-id**	Get basic information about a user.
GET	/users/self/feed	See the authenticated user's feed.
GET	/users/ **user-id** /media/recent	Get the most recent media published by a user.
GET	/users/self/media/liked	See the authenticated user's list of liked media.
GET	/users/search	Search for a user by name.

Comment Endpoints

GET	/media/ **media-id** /comments	Get a full list of comments on a media.
POST	/media/ **media-id** /comments	Create a comment on a media. Please email apide...
DEL	/media/ **media-id** /comments/ **comment-id**	Remove a comment.

GET /media/ **media-id** /comments

https://api.instagram.com/v1/media/555/comments?access_token=ACCESS-TOKEN RESPONSE


```
{
    "meta": {
        "code": 200
    },
    "data": [
        {
            "created_time": "1280780324",
            "text": "Really amazing photo!",
            "from": {
                "username": "snoopdogg",
                "profile_picture": "http://images.instagram.com/profiles/profile_16_75sq_1305612434.jpg",
                "id": "1574083",
                "full_name": "Snoop Dogg"
            },
            "id": "420"
        },
        ...
    ]
}
```


Get a full list of comments on a media.

Required scope: comments

CRUD method description

Cross-cutting concerns

Limits

Be nice. If you're sending too many requests too quickly, we'll send back a 503 error code (server unavailable).

You are limited to 5000 requests per hour per access_token or client_id overall. Practically, this means you should (when possible) authenticate users so that limits are well outside the reach of a given user.

PAGINATION

Sometimes you just can't get enough. For this reason, we've provided a convenient way to access more data in any request for sequential data. Simply call the url in the next_url parameter and we'll respond with the next set of data.

The Envelope

Every response is contained by an envelope. That is, each response has a predictable set of keys with which you can expect to interact:


```
{
    "meta": {
        "code": 200
    },
    "data": {
        ...
    },
    "pagination": {
        "next_url": "...",
        "next_max_id": "13872296"
    }
}
```