imboclient-js Documentation

Release dev

Espen Hovlandsdal

Sep 27, 2017

Contents

This is the official Javascript-based client for Imbo servers.

CHAPTER 1

Requirements

The client requires Node.js >= 0.10 or a modern browser.

CHAPTER 2

Installation

imboclient can be installed using npm or bower:

```
# NPM:
npm install imboclient

# Bower:
bower install imboclient
```

CHAPTER 3

Usage

Below you will find documentation covering most features of the client.

- *Get user info*

Loading the client

The client is exposed using UMD (Universal Module Definition), meaning it should work in most environments.

1. Node.js / CommonJS environments:

```
var Imbo = require('imboclient');
```

2. AMD (Asynchronous Module Definition):

```
define(['imboclient'], function(Imbo) { });
```

3. Browser global:

```
console.log(window.Imbo);
```

Note: Global is only available if CommonJS/AMD environments are not detected.

Instantiating the client

To create an instance of the client, pass an object of options to the constructor:

```
var Imbo = require('imboclient');
var client = new Imbo.Client({
    hosts: 'http//imbo.example.com',
    user: 'someuser',
    publicKey: '<publicKey>',
    privateKey: '<privateKey>'
});
```

You may also pass multiple hostnames:

```
var Imbo = require('imboclient');
var client = new Imbo.Client({
    hosts: [
        'http//imbo1.example.com',
        'http//imbo2.example.com',
        'http//imbo3.example.com'
    ],
    user: 'someuser',
    publicKey: '<publicKey>',
    privateKey: '<privateKey>'
});
```

If you use multiple hostnames when instantiating the client, it will choose different image URLs based on the image identifier and the number of available hostnames. The client will generate the same URL for the same image identifier, as long as the number of hostnames specified does not change.

Following the recommendation of the HTTP 1.1 specification, browsers typically default to two simultaneous requests per hostname. Specifying multiple hostnames might speed up the loading time for your users.

Error handling

The client performs its operations asynchronously and returns its results using callbacks. The client follows the node.js convention where the first parameter of any callback is an optional error object/message.

Add an image

The first thing you might want to do is to start adding images. This can be done in several ways:

1. Add an image from a local path (node.js):

```
client.addImage('/path/to/image.jpg', function(err, imageIdentifier, body) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('Image added! Image identifier: ' + imageIdentifier);
    console.log('Size of image: ' + body.width + 'x' + body.height);
});
```

2. Add an image from a URL:

```
client.addImageFromUrl('http://example.com/some/image.png', function(err,␣
↪imageIdentifier, body) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('Image added! Image identifier: ' + imageIdentifier);
    console.log('Size of image: ' + body.width + 'x' + body.height);
});
```

3. From a `File` instance in the browser:

```
fileInput.addEventListener('change', function(evt) {
    client.addImage(evt.files[0], function(err, imageIdentifier, body) {
        if (err) {
            return console.error('An error occured: ' + err);
        }

        console.log('Image added! Image identifier: ' + imageIdentifier);
        console.log('Size of image: ' + body.width + 'x' + body.height);
    });
}, false);
```

The image identifier returned from these methods is the identifier you will use when generating URLs to the image later on. The `body` also has some other information that you might find useful:

(string) imageIdentifier As mentioned above, the ID of the added image.

(int) width The width of the added image.

(int) height The height of the added image.

(string) extension The extension of the added image.

The `width` and `height` can differ from the original image if the server has added event listeners that might change incoming images. Some changes that might occur is auto rotating based on EXIF-data embedded into the image, and if a max image size is being enforced by the server.

Get image properties

You can fetch properties of the image by using the `getImageProperties` method, specifying the image identifier of an image:

```
client.getImageProperties('image identifier', function(err, properties) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('Image width: '  + properties.width);
    console.log('Image height: ' + properties.height);
    console.log('File size: '    + properties.filesize);
    console.log('Extension: '    + properties.extension);
    console.log('Mime type: '    + properties.mimetype);
});
```

The properties returned is an object containing the following elements:

(int) width The width of the image in pixels.

(int) height The height of the image in pixels.

(int) filesize The file size of the image in bytes.

(string) extension The extension of the image.

(string) mimetype The mime type of the image.

Delete an image

If you want to delete an image from the server, you can use the `deleteImage` method:

```
client.deleteImage('image identifier', function(err) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('Image deleted!');
});
```

Check for the existence of images on the server

If you want to see if a local image exists on the server, use the `imageExists(path)` method:

```
var path = '/path/to/image.jpg';
client.imageExists(path, function(err, exists) {
    if (err) {
        return console.error('An error occured: ' + err);
```

```
    }

    console.log('"' + path + '" ' + (exists ? 'exists' : 'does not exist') + ' on the␣
↪server');
});
```

You can also check for the existence of an image identifier on the server by using the `imageIdentifierExists(imageIdentifier)` method.

Get the number of added images

If you want to fetch the number of images owned by the current user you can use the `getNumImages` methods:

```
client.getNumImages(function(err, numImages) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('The user has ' + numImages + ' images.');
});
```

Get the binary image data

If you want to fetch the binary data of an image you can use `getImageData(imageIdentifier)`. If you have an instance of an image URL you can use the `getImageDataFromUrl(imageUrl)` method:

```
client.getImageData(imageIdentifier, function(err, data) {
    console.log(err ? 'An error occured' : ('image data: ' + data));
});

// or

var url = client.getImageUrl(imageIdentifier).thumbnail().border();
client.getImagedataFromUrl(url, function(err, data) {
    console.log(err ? 'An error occured' : ('image data: ' + data));
});
```

You can read more about the image URLs in the *Imbo URLs* section.

Search for images

The client also let's you search for images on the server. This is done via the `getImages` method:

```
client.getImages(function(err, images, search) {
    console.log('Images on the server: ' + search.hits);
    images.forEach(function(image) {
        console.log(image.imageIdentifier)
    });
});
```

The callback passed to `getImages` will receive four arguments : `err`, `images`, `search` and `response`. `search` is an object with information related to pagination of your query:

`(int) hits` The number of hits from your query.

`(int) page` The current page.

`(int) limit` The maximum number of images per page.

`(int) count` The number of images in the returned set.

`images` is an array where each entry represents an image. Each image is an object which includes the following keys:

- `added`
- `updated`
- `checksum`
- `extension`
- `size`
- `width`
- `height`
- `mime`
- `imageIdentifier`
- `user`
- `metadata` (only if the query explicitly enabled metadata in the response, which is off by default).

Some of these elements might not be available if the query excludes some fields (more on that below).

The `getImages` method can also take a parameter which specifies a query to execute. The parameter is an instance of the `Imbo.Query` class. This class has a set of methods that can be used to customize your query. All methods can be chained when used with a parameter (when setting a value). If you skip the parameter, the methods will return the current value instead:

`page(page = null)` Set or get the `page` value. Functions like an offset (`limit` × `page`). Defaults to `1`.

`limit(limit = null)` Set or get the `limit` value. Defines the maximum number of images to return per page. Defaults to `20`.

`metadata(metadata = null)` Set to true to return metadata attached to the images. Defaults to `false`. Setting this to `true` will make the client include the `metadata` element mentioned above in the images in the collection.

`from(from = null)` Specify a `Date` instance which represents the oldest image you want returned in the collection. Defaults to `null`.

`to(to = null)` Specify a `Date` instance which represents the newest image you want returned in the collection. Defaults to `null`.

`fields(fields = null)` Specify (as an array) which fields should be available per image in the `images` element of the response. Defaults to `null` (all fields). The fields to include are mentioned above.

Note: If you want to include metadata in the response, remember to include `metadata` in the set of fields, **if** you specify custom fields.

`sort(sort = null)` Specify which field(s) to sort by. Defaults to `date:desc`. All fields mentioned above can be sorted by, and they all support `asc` and `desc`. If you don't specify a sort order `asc` will be used.

ids(ids = null) Only include these image identifiers in the collection. Defaults to `null`.

checksums(checksums = null) Only include these MD5 checksums in the collection. Defaults to `null`.

originalChecksums(originalChecksums = null) Same as `checksums()` except the checksums are compared before any event listeners have modified the image. Defaults to `null`.

Here are some examples of how to use the query object:

1. Fetch (at most) 10 images added within the last 24 hours, sorted by the image byte size (ascending) and then the width of the image (descending):

```
var yesterday = new Date();
yesterday.setDate(yesterday.getDate() - 1);

var query = new Imbo.Query();
query
    .limit(10)
    .from(yesterday)
    .sort(['size', 'width:desc']);

client.getImages(query, function(err, images, search) {

});
```

2. Include metadata in the response:

```
var query = new Imbo.Query();
query.metadata(true);

client.getImages(query, function(err, images, search) {

});
```

3. Only fetch the `width` and `height` fields on a set of images:

```
var query = new Imbo.Query();
query.ids(['id1', 'id2', 'id3']).fields(['width', 'height']);

client.getImages(query, function(err, images, search) {

});
```

If you want to return metadata, and happen to specify custom fields you will need to explicitly add the `metadata` field. If you don't use the `fields` method this is not necessary:

```
query.metadata(true).fields(['size']); // Does include the metadata field
query.metadata(true).fields(['size', 'metadata']); // Includes the size and metadata␣
↪fields
query.metadata(true); // Includes all fields, including metadata
query.metadata(false); // Exclude the metadata field (default behaviour)
```

Get metadata

Images in Imbo can have metadata attached to them. If you want to fetch this data you can use the `getMetadata` method:

```
client.getMetadata('image identifier', function(err, data) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    for (var key in data) {
        console.log(key + ': ' + data[key]);
    }
});
```

Update metadata

If you have added an image and want to edit its metadata you can use the `editMetadata` method:

```
client.editMetadata('image identifier', {
    'key': 'value',
    'other key': 'other value',
}, function(err, metadata) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('Updated metadata: ', metadata);
});
```

This method will partially update existing metadata.

Replace metadata

If you want to replace all existing metadata with something else you can use the `replaceMetadata` method:

```
client.replaceMetadata('image identifier', {
    'key': 'value',
    'other key': 'other value',
}, function(err, metadata) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('New metadata: ', metadata);
});
```

This will first remove existing (if any) metadata, and add the metadata specified as the second parameter.

Delete metadata

If you want to remove all metadata attached to an image you can use the `deleteMetadata` method:

```
client.deleteMetadata('image identifier', function(err) {
    if (err) {
        return console.error('An error occured: ' + err);
```

```
    }
});
```

Imbo URLs

Imbo uses access tokens in the URLs to prevent DoS attacks, and the client includes functionality that does this automatically:

`getStatusUrl()` Fetch URL to the status endpoint.

`getStatsUrl()` Fetch URL to the stats endpoint.

`getUserUrl()` Fetch URL to the user information of the current user (specified by setting the correct public key when instantiating the client)‘‘.

`getImagesUrl()` Fetch URL to the images endpoint.

`getImageUrl(imageIdentifier)` Fetch URL to a specific image.

`getMetadataUrl(imageIdentifier)` Fetch URL to the metadata of a specific image.

`getShortUrl(imageUrl, callback)` Fetch the short URL to an image (with optional image transformations added).

All these methods return instances of different classes, and all can be used in string context to get the URL with the access token added. The instance returned from the `getImageUrl` is somewhat special since it will let you chain a set of transformations before generating the URL as a string:

```
var imageUrl = client.getImageUrl('image identifier');
imageUrl.thumbnail().border().jpg();

document.write('<img src="' + imageUrl + '">');
```

The available transformation methods are:

- `autoRotate()`
- `border({ color: '000000', width: 1, height: 1, mode: 'outbound' })`
- `canvas({ width: null, height: null, mode: null, x: null, y: null, bg: null })`
- `compress({ level: 75 })`
- `contrast({ sharpen: 2 })`
- `crop({ x: null, y: null, width: null, height: null })`
- `desaturate()`
- `flipHorizontally()`
- `flipVertically()`
- `maxSize({ width: null, height: null })`
- `modulate({ brightness: 100, saturation: 100, hue: 100 })`
- `progressive()`
- `resize({ width: null, height: null })`
- `rotate({ angle: null, bg: '000000' })`

- `sepia({ threshold: 80 })`
- `sharpen({ radius: 3, sigma: 2, gain: 1.5, threshold: 0.07 })`
- `smartSize({ width: null, height: null, crop: 'mode', poi: 'x,y' })`
- `strip()`
- `thumbnail({ width: 50, height: 50, fit: 'outbound' })`
- `transpose()`
- `transverse()`
- `watermark({ img: null, width: null, height: null, position: 'top-left', x: 0, y: 0 })`

Please refer to the server documentation for details about the image transformations.

There are also some other methods available:

`append(transformation)` Can be used to add a custom transformation (that needs to be available on the server):

```
url.append('foobar'); // results in t[]=foobar being added to the URL
```

`convert(type)` Convert the image to one of the supported types:

- `jpg`
- `gif`
- `png`

`gif()` Proxies to `convert('gif')`.

`jpg()` Proxies to `convert('jpg')`.

`png()` Proxies to `convert('png')`.

`reset()` Removes all transformations added to the ImageUrl instance.

`clone()` Creates a clone of the ImageUrl instance.

The methods related to the image type (`convert` and the proxy methods) can be added anywhere in the chain. Otherwise all transformations will be applied to the image in the same order as they appear in the chain.

Parsing image URLs

`ImageUrl`-instances can also be constructed from strings:

```
var url = 'http://imbo01.host.com/users/user/images/83b2931724639325abe.jpg?
↪t[]=flipHorizontally&accessToken=01379d1861fb5b26';

var imageUrl = client.parseImageUrl(url);
imageUrl.sepia().thumbnail({ width: 320 }).png();

console.log('Sepia thumbnail URL: ', imageUrl.toString());
```

If the image is stored under a different user than the client has been instantiated with, you will need to pass the private key belonging to the user who owns the image:

```
var url = 'http://imbo01.host.com/users/some-other-user/images/8329f110695abe.jpg?
↪t[]=crop%3Ax%3D0%2Cy%3D0%2Cwidth%3D927%2Cheight%3D621';

var imageUrl = client.parseImageUrl(url, 'private key for "some-other-user"');
imageUrl.desaturate().thumbnail({ width: 320 }).png();

console.log('Desaturated thumbnail URL: ', imageUrl.toString());
```

ShortUrls

With the host, user, image identifier, transformations and access tokens all being part of an image URL, the URLs can become quite long. Imbo supports making shorter URLs, which follows this pattern: `http://imbo.host/s/ShortId`.

Instances of `ShortUrl` contains both the short URL (retrieved by calling `shortUrl.toString()`) and the ID of the short URL (`shortUrl.getId()`). This ID can be used with `deleteShortUrlForImage` if you should wish to remove the short URL at a later time.

The available methods related to short URLs are:

`getShortUrl(imageUrl, callback)` Generates a `ShortUrl`. `imageUrl` is an instance of `Imbo.ImageUrl`

`deleteAllShortUrlsForImage(imageIdentifier, callback)` Deletes every short URL that has been generated for the given image identifier.

`deleteShortUrlForImage(imageIdentifier, shortUrl, callback)` Deletes a specific short URL. `shortUrl` can be either a `ShortUrl` instance or the ID of a short URL.

```
var url = client.getImageUrl(imageIdentifier).thumbnail();

client.getShortUrl(url, function(err, shortUrl) {
    if (err) {
        return console.error('An error occured: ' + err);
    }

    console.log('ShortUrl generated: ' + shortUrl.toString());

    // To delete the short URL:
    client.deleteShortUrlForImage(imageIdentifier, shortUrl, function(err) {
        console.log(err ? ('An error occured: ' + err) : 'ShortUrl deleted');
    });
});
```

Get server status

If you want to get the server status, you can use the `getServerStatus` method:

```
client.getServerStatus(function(err, status) {
    console.log(err ? 'An error occured: ' : 'Status: ', err || status);
});
```

The `status` value above is an object and includes the following elements:

(boolean) database Whether or not the configured database works as expected on the server.

(boolean) storage Whether or not the configured storage works as expected on the server.

(Date) date The server date/time.

(int) status The HTTP status code.

Get server statistics

If you have access to the server statistics and want to fetch these, you can use the `getServerStats` method:

```
client.getServerStats(function(err, statistics) {
    console.log(err ? 'An error occured: ' : 'Stats: ', err || statistics);
});
```

`statistics` is an object and includes the following elements:

(object) users An object of users where the keys are user names (public keys) and values are objects with the following elements:

- `(int) numImages`: Number of images owned by this user
- `(int) numBytes`: Number of bytes stored by this user

(object) total An object with aggregated values. The object includes the following elements:

- `(int) numImages`: The number of images on the server
- `(int) numUsers`: The number of users on the server
- `(int) numBytes`: The number of bytes stored on the server

(object) custom If the server has configured any custom statistics, these are available in this element.

Get user info

Get some information about the user configured with the client:

```
client.getUserInfo(function(err, info) {
    console.log(err ? 'An error occured: ' : 'Info: ', err || info);
});
```

`info` is an object and includes the following elements:

(string) user The name of the user (the same as the one used when instantiating the client).

(int) numImages The number of images owned by the user.

(Date) lastModified A `Date` instance representing when the user last modified any data on the server.