imboclient-php Documentation

Release 2.0.3

Christer Edvartsen

March 17, 2016

Contents

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

CHAPTER 1

Requirements

The client requires PHP >= 5.3.3.

CHAPTER 2

Installation

ImboClient can be installed using Composer by requiring `imbo/imboclient` in your `composer.json` file, or by running the following commands:

```
curl -s https://getcomposer.org/installer | php
php composer.phar create-project imbo/imboclient [<dir>] [<version>]
```

Available versions can be located at packagist.

CHAPTER 3

Usage

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

3.1 Instantiating the client

This can be achieved in two different ways:

1. Use the factory:

```
$client = ImboClient\ImboClient::factory([
    'serverUrls' => ['http://imbo.example.com'],
    'publicKey' => 'public key',
    'privateKey' => 'private key',
    'user' => 'username',
]);
```

2. Use the constructor:

```
$client = new ImboClient\ImboClient('http://imbo.example.com', [
    'publicKey' => 'public key',
    'privateKey' => 'private key',
    'user' => 'username',
]);
```

The main difference is that the first argument to the factory method requires you to specify the host name(s) of the Imbo server(s) as an array, while the constructor requires you to pass a string. If you want to use the example number 2 above, and still want to use multiple host names you can use the `setServerUrls` method:

```
$client->setServerUrls([
    'http//imbo1.example.com',
    'http//imbo2.example.com',
    'http//imbo3.example.com',
]);
```

If you use multiple URLs when instantiating the client it will choose different image URLs based on the image identifier and the number of available host names. If you have a site which includes a lot of `<img>` tags against an Imbo server, using multiple hosts might speed up the loading time for your users. If you don't change the amount of server URLs the client will always pick the same host name given the same image identifier.

3.2 Error handling

Most methods will throw a `Guzzle\Common\Exception\GuzzleException` exception if the server responds with an error (as in HTTP 4** or 5**). Some methods might also throw an `InvalidArgumentException` exception of the provided parameter to a method is invalid (for instance if you try to add an image and provide a local path to a file that does not exist). Remember to use `try/catch` if you want to handle these errors gracefully.

3.3 Get server status

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

```
$status = $client->getServerStatus();
```

The `$status` value above can be used as an associative array, 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.

(int) status The HTTP status code.

(string) message The HTTP response reason phrase.

3.4 Get server statistics

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

```
$stats = $client->getServerStats();
```

The return value from this method can be used as an associative array, and includes the following elements:

(array) users An array of users where the keys are user names and values are arrays with the following elements:

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

`(array) total` An array with aggregated values. The array 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

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

3.5 Get user info

Get some information about the user configured with the client:

```
$info = $client->getUserInfo();
```

The value returned from the `getUserInfo` method includes the following elements:

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

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

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

3.6 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:

```
$response = $client->addImage('/path/to/image.jpg');
```

2. Add an image from a URL:

```
$response = $client->addImageFromUrl('http://example.com/some/image.jpg');
```

3. Add an in-memory image:

```
$response = $client->addImageFromString(file_get_contents('/path/to/image.jpg'));
```

The `$response` returned from these methods holds the resulting image identifier of the image, and can be fetched by using the response as an associative array:

```
echo 'Image added, identifier: ' . $response['imageIdentifier'];
```

This is the identifier you will use when generating URLs to the image later on. The response 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.

(int) status The HTTP status of the response from the server. Should be `200` or `201`.

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.

3.7 Get image properties

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

```
$properties = $client->getImageProperties('image identifier');
```

The return value can be used as an associative array, and contains 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.

3.8 Delete an image

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

```
$response = $client->deleteImage('identifier');
```

where `'identifier'` is the value of the `imageIdentifier` key of the response returned when adding images.

3.9 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:

```
$path = '/path/to/image.jpg';
$exists = $client->imageExists($path);

echo '"' . $path . '" ' . ($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.

3.10 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:

```
echo 'The user "' . $client->getUser() . '" has ' . $client->getNumImages() . ' images.';
```

3.11 Get the binary image data

If you want to fetch the binary data of an image as a string you can use `getImageData($imageIdentifier)`. If you have an instance of an image URL you can use the `getImageDataFromUrl(ImboClient\Http\ImageUrl $imageUrl)` method:

```
$imageData = $client->getImageData($imageIdentifier);

// or

$imageData = $client->getImagedataFromUrl($client->getImageUrl($imageIdentifier)->thumbnail()->borde
```

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

3.12 Search for images

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

```
$collection = $client->getImages();

echo '<h1>Images on the server:</h1>';
echo '<ul>';

foreach ($collection['images'] as $image) {
    echo '<li>' . $image['imageIdentifier'] . '</li>';
}

echo '</ul>';
```

The `$collection` variable returned from the `getImages` methods has two elements: `search` and `images`. `search` is an array related to pagination and holds information about the images returned by your query:

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

(int) page The current page.

(int) limit Limit the number of images per page.

(int) count The number of images currently on the page.

and the `images` element is a traversable where each element represents an image. Each image is an associative array which includes the following elements:

- `added`
- `updated`
- `checksum`
- `originalChecksum`
- `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 `ImboClient\ImagesQuery` 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. Defaults to `1`.

`limit($limit = null)` Set or get the `limit` value. 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 Unix timestamp which represents the oldest image you want returned in the collection. Defaults to `null`.

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

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

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

`originalChecksums(array $checksums = null)` Only include these original MD5 checksums in the collection. 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):

```
$current = time();
$query = new ImboClient\ImagesQuery();
$query->limit(10)->from($current - 3600 * 24)->sort(['size', 'width:desc']);

$collection = $client->getImages($query);
```

2. Include metadata in the response:

```
$query = new ImboClient\ImagesQuery();
$query->metadata(true);

$collection = $client->getImages($query);
```

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

```
$query = new ImboClient\ImagesQuery();
$query->ids(['id1', 'id2', 'id3'])->fields(['width', 'height']);

$collection = $client->getImages($query);
```

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)
```

3.13 Get metadata

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

```
$metadata = $client->getMetadata('image identifier');

echo '<dl>';

foreach ($metadata as $key => $value) {
    echo '<dt>' . $key . '</dt>';
    echo '<dd>' . $value . '</dd>';
}

echo '</dl>';
```

3.14 Update metadata

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

```
$metadata = $client->editMetadata('image identifier', [
    'key' => 'value',
    'other key' => 'other value',
]);
```

This method will partially update existing metadata, and the response contains all metadata attached to the image.

3.15 Replace metadata

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

```
$metadata = $client->replaceMetadata('image identifier', [
    'key' => 'value',
    'other key' => 'other value',
]);
```

This will first remove existing (if any) metadata, and add the metadata specified as the second parameter. The response contains the metadata of the image, in this case the same as the data being sent to the server.

3.16 Delete metadata

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

```
$metadata = $client->deleteMetadata('image identifier');
```

The response is the existing metadata, which in this case is an empty object.

3.17 Generate short image URL

To be able to generate short image URLs you can use the `generateShortUrl` method, and simply specify an instance of the image URL you want to shorten:

```
// Create an image URL with some optional transformations
$imageUrl = $client->getImageUrl('image identifier')->thumbnail()->desaturate()->jpg();

// Pass the image URL instance to the generateShortUrl method
$response = $client->generateShortUrl($imageUrl);

echo 'Short URL ID: ' . $response['id'];
```

The generated ID can be used with the global short URL resource in Imbo.

3.18 Get resource groups

To retrieve resource groups available on the Imbo server, you can use the `getResourceGroups` method:

```
$collection = $client->getResourceGroups();

echo '<h1>Available resource groups:</h1>';
echo '<ul>';

foreach ($collection['groups'] as $group) {
    echo '<li>' . $group['name'] . '</li>';
}

echo '</ul>';
```

The `$collection` variable returned from the `getResourceGroups` methods has two elements: `search` and `groups`. `search` is an array related to pagination and holds information about the groups returned by your query:

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

(int) page The current page.

(int) limit Limit the number of groups per page.

(int) count The number of groups currently on the page.

and the `groups` element is a traversable where each element represents a group. Each group is an associative array which includes the following elements:

- `name` - name of the group
- `resources` - array of strings defining the resources the group consists of

The `getResourceGroups` method can also take a parameter which specifies a query to execute. The parameter is an instance of the `ImboClient\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. Defaults to `1`.

`limit($limit = null)` Set or get the `limit` value. Defaults to `20`.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.19 Get specific resource group

To retrieve a single resource group, you can use the `getResourceGroup` method:

```
$group = $client->getResourceGroup('group-name');

echo '<h1>"group-name" consists of the following resources:</h1>';
echo '<ul>';

foreach ($group['resources'] as $resource) {
    echo '<li>' . $resource . '</li>';
}

echo '</ul>';
```

The `$group` variable returned from the `getResourceGroup` method currently only has a single element: `resources`, which represents the resources the group consists of.

This method will throw an exception if the group name is invalid, already exists or an error occurs.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.20 Add a resource group

Resource groups can be created using the `addResourceGroup` method:

```
$client->addResourceGroup('group-name', [
    'image.get',
    'image.head',
    'images.post',
    'images.get',
    'images.head'
]);
```

This method will throw an exception if the group name is invalid, already exists or an error occurs.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.21 Edit a resource group

Resource groups can be edited using the `editResourceGroup` method:

```
$client->editResourceGroup('group-name', [
    'image.get',
    'image.head',
    'images.post',
    'images.get',
    'images.head'
]);
```

It's important to note that if the resource group with the given name does not already exist, it will be created. If it exists, the resources provided in the second argument will **overwrite** the existing resources for that group. If you need to add more resources to an existing group, first retrieve it's resources using the `getResourceGroup`-method and merge the resources returned with the ones you want to add.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.22 Delete a resource group

Resource groups can be deleted using the `deleteResourceGroup` method:

```
$client->deleteResourceGroup('group-name');
```

Note: Any access control rules that are using this resource group will also be deleted, since they are now invalid.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.23 Check for the existence of a resource group

Calling the `resourceGroupExists` method will return whether a resource group exists:

```
if ($client->resourceGroupExists('group-name')) {
    echo 'Resource group exists';
} else {
    echo 'Resource group does NOT exist';
}
```

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.24 Creating a new public/private key pair

Adding new public keys (and an associated private key) can be achieved by using the `addPublicKey` method:

```
$client->addPublicKey('new-pub-key', 'associated-priv-key');
```

This method will throw an exception if the public key name is invalid, already exists or an error occurs.

Note: This function sends the private and public key as plain text to the Imbo server, and should only be used over HTTPS.

Note: Private keys should be hard to guess. We advise you to use a secure password generator to generate one.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.25 Editing a public/private key pair

Editing existing public/private key pairs can be achieved by using the `editPublicKey` method:

```
$client->editPublicKey('public-key', 'new-private-key');
```

This method will throw an exception if the public key name is invalid or an error occurs.

Note: All the same considerations should be taken as when using the `addPublicKey` method - data is sent in plain text, do not use unless you are communicating over HTTPS!

3.26 Deleting a public key

Deleting a public key (and the associated private key) can be achieved by using the `deletePublicKey` method:

```
$client->deletePublicKey('public-key');
```

This method will throw an exception if the public key name is invalid or an error occurs.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.27 Check for the existence of a public key

Calling the `publicKeyExists` method will return whether a public key exists:

```
if ($client->publicKeyExists('public-key')) {
    echo 'Public key exists';
} else {
    echo 'Public key does NOT exist';
}
```

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.28 Getting list of ACL-rules for a public key

To retrieve a list of the defined access control rules for a given public key, you can use the `getAccessControlRules` method:

```
$aclRules = $client->getAccessControlRules('public-key');
```

The return value of this method is a traversable where each element represents a single ACL-rule. See the documentation of `getAccessControlRule` below for the details on the contents of these rules.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.29 Getting a specific ACL-rule for a public key

To retrieve a specific access control rule, you can use the `getAccessControlRule` method:

```
$aclRule = $client->getAccessControlRule('public-key', 'acl-rule-id');
```

The return value of this method is a collection (accessible as an array), containing the following keys:

(string) id The ID of the ACL-rule (same as the one specified when retrieving the rule).

(string) group Name of the resource group which defines which resources this rule should apply for. Only present if `resources` is not.

(array) resources An array of the resources this ACL-rule grants access to. Only present if `group` is not.

(array|string) users Either an array of users which this ACL-rule grants access to, or the string `*`, meaning it gives access to the given resources for **all** users.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.30 Adding ACL-rules for a public key

To add new access control rules, you can use the `addAccessControlRules`. It accepts an array of ACL-rules:

```
$client->addAccessControlRules('public-key', [
    [
        'group' => 'some-group',
        'users' => ['user1', 'user2']
    ],
    [
        'resources' => ['image.get', 'image.head', 'image.options'],
        'users' => '*'
    ]
]);
```

The ACL-rules you want to create should have the same pattern as documented in `getAccessControlRule`, expect no `id` should be defined.

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.31 Deleting an ACL-rule for a public key

Deleting an access control rule can be achieve by using the `deleteAccessControlRule` method:

```
$client->deleteAccessControlRule('public-key', 'acl-rule-id');
```

Note: Not all public keys have (and usually shouldn't have) access to this functionality.

3.32 Imbo URLs

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

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

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

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

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

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

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

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:

```
$imageUrl = $client->getImageUrl('image identifier');
$imageUrl->thumbnail()->border()->jpg();

echo '<img src="' . $imageUrl . '">';
```

The available transformation methods are:

- `autoRotate()`
- `blur($params)`
- `border($color = '000000', $width = 1, $height = 1, $mode = 'outbound')`
- `canvas($width, $height, $mode = null, $x = null, $y = null, $bg = null)`
- `compress($level = 75)`
- `contrast($alpha = null, $beta = null)`
- `crop($x, $y, $width, $height, $mode)`

- desaturate()
- drawPois()
- flipHorizontally()
- flipVertically()
- histogram($scale = null, $ratio = null, $red = null, $green = null, $blue = null)
- level($amount = 1, $channel = null)
- maxSize($maxWidth = null, $maxHeight = null)
- modulate($brightness = null, $saturation = null, $hue = null)
- progressive()
- resize($width = null, $height = null)
- rotate($angle, $bg = '000000')
- sepia($threshold = 80)
- sharpen($params = null)
- smartSize($width, $height, $crop = null, $poi = null)
- strip()
- thumbnail($width = 50, $height = 50, $fit = 'outbound')
- transpose()
- transverse()
- vignette($scale = null, $outerColor = null, $innerColor = null)
- watermark($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:

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

```
$url->addTransformation('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 URL 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.

3.33 Migrating from ImboClient < 1.3.0

From ImboClient 1.3.0, the client fully supports Imbo 2.0. While the client itself is fully backwards-compatible, we encourage all users to add the `user` property when instantiating the client. For users who are still using Imbo 1.x, the user will have the same value as `publicKey`, while in Imbo 2, these two values can be different.

3.34 Migrating from ImboClient < 1.0.0

ImboClient's API changed somewhat with the release of version 1.0.0. This section should help you migrate from an older version of the client.

3.34.1 Instantiating the client

From version 1.0.0 ImboClient comes with a factory that should be used to instantiate the client:

```
$client = ImboClient\ImboClient::factory([
    'serverUrls' => ['http://imbo.example.com'],
    'publicKey' => 'public key',
    'privateKey' => 'private key',
    'user' => 'username',
]);
```

More examples on how to instantiate the client are available in the *Instantiating the client* section.

3.34.2 Response objects

All methods now return response objects that can be used as arrays, whereas the old client returned objects with accessor methods for the image identifier and more. Below is an example that shows the difference:

```
// New client
$response = $client->addImage('/path/to/image.jpg');
echo "Image identifier: " . $response['imageIdentifier'];

// Old client
$response = $client->addImage('/path/to/image.jpg');
echo "Image identifier: " . $response->getImageIdentifier();
```

3.34.3 Translating old URLs

If you for some reason have stored complete Imbo URLs (including access tokens), **which you should really try to avoid**, you might want to re-generate these if you get some "incorrect access token" errors from the server. This can be done in the following fashion:

```
// Create an instance of an image URL, using the old URL with the faulty access token and the
// current private key of the user as input
$url = ImboClient\Http\ImageUrl::factory(
    'http://imbo/users/user/images/image?t[]=resize:width=100&accessToken=<incorrect token>',
    'your private key'
);

// Remove the incorrect access token from the query parameters
$url->getQuery()->remove('accessToken');
```

```
// Convert the URL to a string to get the new URL, including the correct access token
echo "New URL: " . $url;
```

3.34.4 Exceptions

All exceptions thrown by the client related to response errors from the server implement the `Guzzle\Common\Exception\GuzzleException` interface. Earlier versions of the threw `ImboClient\Exception\ServerException` exceptions. This exception no longer exists.

The client can also throw `InvalidArgumentException` on some occasions if you provide invalid arguments to some methods, whereas the old client threw either `ImboClient\Exception\InvalidArgumentException` or `ImboClient\Exception\RuntimeException`. None of these two exceptions exist anymore.