
Imbo

Release 2.2.1

Apr 28, 2017

Contents

1	Installation guide	3
1.1	Requirements	3
1.2	Installation	3
1.3	Upgrading Imbo	10
1.4	Configuration	14
1.5	Customize your Imbo installation with event listeners	35
1.6	Command line tool	48
2	End user guide	51
2.1	Imbo's API	51
2.2	Transforming images on the fly	74
3	Extending/customizing Imbo	85
3.1	Working with events and event listeners	85
3.2	Implement your own database and/or storage adapter	90
3.3	Implement your own image transformations	91
3.4	Cache adapters	91
3.5	Contributing to Imbo	94

Imbo is an image “server” that can be used to add/get/delete images using a RESTful HTTP API. There is also support for adding meta data to the images stored in Imbo. The main idea behind Imbo is to have a place to store high quality original images and to use the API to fetch variations of the images. Imbo will resize, rotate and crop (amongst other transformations) images on the fly so you won’t have to store all the different variations.

Imbo is an open source ([MIT licensed](#)) project written in [PHP](#) and is [available on GitHub](#). If you find any issues or missing features please add an issue in the [issue tracker](#). If you want to know more feel free to join the #imbo channel on the [Freenode IRC network](#) (chat.freenode.net) as well.

Requirements

Imbo requires a web server (for instance [Apache](#), [Nginx](#) or [Lighttpd](#)) running `PHP >= 5.6` and the [Imagick](#) extension for PHP.

You will also need a backend for storing image information, like for instance [MongoDB](#) or [MySQL](#). If you want to use MongoDB as a database and/or [GridFS](#) for storage, you will need to install the [Mongo PECL](#) extension, and if you want to use a RDBMS (Relational Database Management System) like MySQL, you will need to install the [Doctrine Database Abstraction Layer](#).

Installation

To install Imbo on the server you can choose between two different methods, [Composer](#) (recommended) or [git clone](#).

Using composer

The recommended way of installing Imbo is by creating a `composer.json` file for your installation, and then install Imbo and optional 3rd party plug-ins via [Composer](#). You will need the following directory structure for this method to work:

```
/path/to/install/composer.json
/path/to/install/config/
```

where the `composer.json` file can contain:

```
{
  "name": "yourname/imbo",
  "require": {
    "imbo/imbo": "dev-master"
  }
}
```

```
}  
}
```

and the `config/` directory contains one or more configuration files that will be merged with the *default configuration*. Imbo will load **all** `.php` files in this directory, and the ones returning an array will be used as configuration.

If you want to install 3rd party plug-ins and/or for instance the Doctrine DBAL library simply add these to the `require` object in your `composer.json`:

```
{  
  "name": "yourname/imbo",  
  "require": {  
    "imbo/imbo": "dev-master",  
    "rexxars/imbo-hipsta": "dev-master",  
    "doctrine/dbal": "2.*"  
  }  
}
```

If some of the 3rd party plug-ins provide configuration files, you can link to these in the `config/` directory to have Imbo automatically load them:

```
cd /path/to/install/config  
ln -s ../vendor/rexxars/imbo-hipsta/config/config.php 01-imbo-hipsta.php
```

To be able to control the order that Imbo will use when loading the configuration files you should prefix them with a number, like `01` in the example above. Lower numbers will be loaded first, meaning that configuration files with higher numbers will override settings set in configuration files with a lower number.

When you have created the `composer.json` file you can install Imbo with Composer:

```
curl -s https://getcomposer.org/installer | php  
php composer.phar install -o --no-dev
```

After composer has finished installing Imbo and optional dependencies the Imbo installation will reside in `/path/to/install/vendor/imbo/imbo`. The correct web server document root in this case would be `/path/to/install/vendor/imbo/imbo/public`.

If you later want to update Imbo you can bump the version number you have specified in `composer.json` and run:

```
php composer.phar update -o --no-dev
```

Regarding the Imbo version you are about to install you can use `dev-master` for the latest released version, or you can use a specific version if you want to (recommended). Head over to [Packagist](#) to see the available versions. If you're more of a YOLO type of person you can use `dev-develop` for the latest development version. If you choose to use the `dev-develop` branch, expect things to break from time to time.

Imbo strives to keep full BC in minor and patch releases, but breaking changes can occur. The most secure way to install one or more Imbo servers is to specify a specific version (for instance `1.2.0`) in your `composer.json` file. Read the [ChangeLog](#) and the [Upgrading Imbo](#) chapter before doing an upgrade.

Using git clone

You can also install Imbo directly via git, and then use Composer to install the dependencies:

```
mkdir /path/to/install; cd /path/to/install  
git clone https://github.com/imbo/imbo.git  
cd imbo
```



```
curl -s https://getcomposer.org/installer | php
php composer.phar install -o --no-dev
```

In this case the correct web server document root would be `/path/to/install/imbo/public`. Remember to checkout the correct branch after cloning the repository to get the version you want, for instance `git checkout master`. If you use this method of installation you will have to modify Imbo's `composer.json` to install 3rd party libraries. You will also have to place your own `config.php` configuration file in the same directory as the default Imbo configuration file, which in the above example would be the `/path/to/install/imbo/config` directory.

If you want to contribute to Imbo, this is the obvious installation method. Read more about this in the *Contributing to Imbo* chapter.

Web server configuration

After installing Imbo by using one of the methods mentioned above you will have to configure the web server you want to use. Imbo ships with sample configuration files for [Apache](#) and [Nginx](#) that can be used with a few minor adjustments. Both configuration files assume the httpd runs on port 80. If you use [Varnish](#) or some other HTTP accelerator simply change the port number to the port that your httpd listens to.

Apache

You will need to enable `mod_rewrite` if you want to use Imbo with Apache. Below is an example on how to configure Apache for Imbo:

```
<VirtualHost *:80>
    # Servername of the virtual host
    ServerName imbo

    # Define aliases to use multiple hosts
    # ServerAlias imbo1 imbo2 imbo3

    # Document root where the index.php file is located
    DocumentRoot /path/to/install/vendor/imbo/imbo/public

    # Logging
    # CustomLog /var/log/apache2/imbo.access_log combined
    # ErrorLog /var/log/apache2/imbo.error_log

    # Rewrite rules that rewrite all requests to the index.php script
    <Directory /path/to/install/vendor/imbo/imbo/public>
        RewriteEngine on
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule .* index.php
    </Directory>
</VirtualHost>
```

You will need to update `ServerName` to match the host name you will use for Imbo. If you want to use several host names you can update the `ServerAlias` line as well. You must also update `DocumentRoot` and `Directory` to point to the `public` directory in the Imbo installation. If you want to enable logging update the `CustomLog` and `ErrorLog` lines. `RewriteCond` and `RewriteRule` should be left alone.

Nginx

Below is an example on how to configure Nginx for Imbo. This example uses PHP via [FastCGI](#):

```
server {
    # Listen on port 80
    listen 80;

    # Define the server name
    server_name imbo;

    # Use the line below instead of the server_name above if you want to use multiple_
↪host names
    # server_name imbo imbo1 imbo2 imbo3;

    # Path to the public directory where index.php is located
    root /path/to/install/vendor/imbo/imbo/public;
    index index.php;

    # Logs
    # error_log /var/log/nginx/imbo.error_log;
    # access_log /var/log/nginx/imbo.access_log main;

    location / {
        try_files $uri $uri/ /index.php?$args;
        location ~ \.php$ {
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME /path/to/install/vendor/imbo/imbo/public/
↪index.php;
            include fastcgi_params;
        }
    }
}
```

You will need to update `server_name` to match the host name you will use for Imbo. If you want to use several host names simply put several host names on that line. `root` must point to the `public` directory in the Imbo installation. If you want to enable logging update the `error_log` and `access_log` lines. You must also update the `fastcgi_param SCRIPT_FILENAME` line to point to the `public/index.php` file in the Imbo installation.

Lighttpd

Below is an example on how to configure Lighttpd for Imbo. Running PHP through FastCGI is recommended (not covered here).

```
# Use the line below instead of the next one if you want to use multiple host names
# $HTTP["host"] =~ "^(imbo|imbo1|imbo2|imbo3)$" {

$HTTP["host"] == "imbo" {
    # Listen on port 80
    server.port = 80

    # Path to the public directory where index.php is located
    server.document-root = "/path/to/install/vendor/imbo/imbo/public"

    # Logs
    # server.errorlog = "/var/log/lighttpd/imbo.error_log"
    # accesslog.filename = "/var/log/lighttpd/imbo.access_log"

    # Rewrite all to index.php
```

```
url.rewrite-if-not-file = ("^[^?]*(\\?.*)?$" => "index.php/$1")
}
```

You will need to set the correct host name(s) used with `$HTTP["host"]` and update the `server.document-root` to point to the correct path. If you want to enable logging remove the comments on the lines with `server.errorlog` and `accesslog.filename` and set the correct paths. If you want to specify a custom access log path you will need to enable the `mod_accesslog` module.

This example requires the `mod_rewrite` module to be loaded.

Varnish

Imbo strives to follow the [HTTP Protocol](#), and can because of this easily leverage [Varnish](#).

The only required configuration you need in your [VCL](#) is a default backend:

```
backend default {
    .host = "127.0.0.1";
    .port = "81";
}
```

where `.host` and `.port` is where Varnish can reach your web server.

If you use the same host name (or a sub-domain) for your Imbo installation as other services, that in turn uses [Cookies](#), you might want the VCL to ignore these Cookies for the requests made against your Imbo installation (unless you have implemented event listeners for Imbo that uses Cookies). To achieve this you can put the following snippet into your VCL file:

```
sub vcl_recv {
    if (req.http.host == "imbo.example.com") {
        unset req.http.Cookie;
    }
}
```

or, if you have Imbo installed in some path:

```
sub vcl_recv {
    if (req.http.host ~ "(www.)?example.com$" && req.url ~ "^/imbo/") {
        unset req.http.Cookie;
    }
}
```

if your Imbo installation is available on `[www.]example.com/imbo`.

Database setup

If you choose to use a RDBMS to store data in, you will need to manually create a database, a user and the tables Imbo stores information in. Below you will find schemas for different RDBMSs. You will find information regarding how to authenticate against the RDBMS of your choice in the [Configuration](#) topic.

MySQL

```

CREATE TABLE IF NOT EXISTS `imageinfo` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `user` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `imageIdentifier` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `size` int(10) unsigned NOT NULL,
  `extension` varchar(5) COLLATE utf8_danish_ci NOT NULL,
  `mime` varchar(20) COLLATE utf8_danish_ci NOT NULL,
  `added` int(10) unsigned NOT NULL,
  `updated` int(10) unsigned NOT NULL,
  `width` int(10) unsigned NOT NULL,
  `height` int(10) unsigned NOT NULL,
  `checksum` char(32) COLLATE utf8_danish_ci NOT NULL,
  `originalChecksum` char(32) COLLATE utf8_danish_ci NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `image` (`user`,`imageIdentifier`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `metadata` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `imageId` int(10) unsigned NOT NULL,
  `tagName` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `tagValue` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  PRIMARY KEY (`id`),
  KEY `imageId` (`imageId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `shorturl` (
  `shortUrlId` char(7) COLLATE utf8_danish_ci NOT NULL,
  `user` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `imageIdentifier` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `extension` char(3) COLLATE utf8_danish_ci DEFAULT NULL,
  `query` text COLLATE utf8_danish_ci NOT NULL,
  PRIMARY KEY (`shortUrlId`),
  KEY `params` (`user`,`imageIdentifier`,`extension`,`query` (255))
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

CREATE TABLE IF NOT EXISTS `storage_images` (
  `user` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `imageIdentifier` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `data` blob NOT NULL,
  `updated` int(10) unsigned NOT NULL,
  PRIMARY KEY (`user`,`imageIdentifier`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

CREATE TABLE IF NOT EXISTS `storage_image_variations` (
  `user` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `imageIdentifier` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `width` int(10) unsigned NOT NULL,
  `data` blob NOT NULL,
  PRIMARY KEY (`user`,`imageIdentifier`,`width`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

CREATE TABLE IF NOT EXISTS `imagevariations` (
  `user` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `imageIdentifier` varchar(255) COLLATE utf8_danish_ci NOT NULL,
  `width` int(10) unsigned NOT NULL,
  `height` int(10) unsigned NOT NULL,
  `added` int(10) unsigned NOT NULL,

```

```

PRIMARY KEY (`user`, `imageIdentifier`, `width`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

```

The `storage_images` table is only needed if you plan on storing the actual images in the database as well.

SQLite

```

CREATE TABLE IF NOT EXISTS imageinfo (
  id INTEGER PRIMARY KEY NOT NULL,
  user TEXT NOT NULL,
  imageIdentifier TEXT NOT NULL,
  size INTEGER NOT NULL,
  extension TEXT NOT NULL,
  mime TEXT NOT NULL,
  added INTEGER NOT NULL,
  updated INTEGER NOT NULL,
  width INTEGER NOT NULL,
  height INTEGER NOT NULL,
  checksum TEXT NOT NULL,
  originalChecksum TEXT NOT NULL,
  UNIQUE (user, imageIdentifier)
);

CREATE TABLE IF NOT EXISTS metadata (
  id INTEGER PRIMARY KEY NOT NULL,
  imageId KEY INTEGER NOT NULL,
  tagName TEXT NOT NULL,
  tagValue TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS shorturl (
  shortUrlId TEXT PRIMARY KEY NOT NULL,
  user TEXT NOT NULL,
  imageIdentifier TEXT NOT NULL,
  extension TEXT,
  query TEXT NOT NULL
);

CREATE INDEX shorturlparams ON shorturl (
  user,
  imageIdentifier,
  extension,
  query
);

CREATE TABLE IF NOT EXISTS storage_images (
  user TEXT NOT NULL,
  imageIdentifier TEXT NOT NULL,
  data BLOB NOT NULL,
  updated INTEGER NOT NULL,
  PRIMARY KEY (user, imageIdentifier)
);

CREATE TABLE IF NOT EXISTS storage_image_variations (
  user TEXT NOT NULL,
  imageIdentifier TEXT NOT NULL,
  width INTEGER NOT NULL,

```

```
data BLOB NOT NULL,  
PRIMARY KEY (user, imageIdentifier, width)  
);  
  
CREATE TABLE IF NOT EXISTS imagevariations (  
user TEXT NOT NULL,  
imageIdentifier TEXT NOT NULL,  
width INTEGER NOT NULL,  
height INTEGER NOT NULL,  
added INTEGER NOT NULL,  
PRIMARY KEY (user, imageIdentifier, width)  
);
```

The `storage_images` table is only needed if you plan on storing the actual images in the database as well.

Upgrading Imbo

Depending on the *installation method* you chose, upgrading Imbo can be done quite easily. If you went for the *recommended version* you will only have to bump the version number in your custom `composer.json` file and run `composer update`.

If you did a *git clone* you could simply do a `git pull` to update your working copy.

From time to time Imbo will introduce new features or fix bugs that might require you to update the contents of the database you choose to use. This chapter will contain all information you need to keep your installation up to date. Each of the following sections include the necessary steps you need to execute when upgrading to the different versions.

Imbo-2.0.0

Below are the changes you need to be aware of when upgrading to Imbo-2.0.0.

- *Public key is an authentication detail*
 - *Doctrine*
 - *MongoDB*
 - *GridFS*
- *Image identifiers are no longer MD5-sums*

Public key is an authentication detail

Versions prior to 2.0.0 had a 1:1 correlation between what a `user` and a `public key` was. In 2.0.0, a `user` is the entity which images belong to, while a `public key` is one part of the authentication scheme. `Public keys` each has their own set of permissions, which can grant them access to different resources within different users.

Imbo provides some level of backwards compatibility with the old `auth` part of the configuration file, but no longer supports multiple private keys per public key (such as read-only/read+write). If you are using this, Imbo will throw an exception on startup. We recommend moving away from the `auth` setup and onto the new *Access Control adapters* as soon as possible.

Prior to 2.0.0, both the database and HTTP endpoints used a field name of `publicKey` to identify the user. Going forward, apart from when working with authentication, `user` will be the new field name. This requires some database changes - see below.

Doctrine

If you use the *Doctrine database adapter* you'll need to rename the `publicKey` fields to `user`. The field has been updated in the *Database setup* section. The field should be renamed while there are no write operations pending, as a write could fail before upgrading Imbo itself.

Postgres/Other

```
ALTER TABLE imageinfo RENAME COLUMN publicKey TO user;
ALTER TABLE shorturl RENAME COLUMN publicKey TO user;
ALTER TABLE imagevariations RENAME COLUMN publicKey TO user;
```

MySQL

```
ALTER TABLE imageinfo CHANGE `publicKey` `user` VARCHAR(255) COLLATE utf8_danish_ci_
↳NOT NULL;
ALTER TABLE shorturl CHANGE `publicKey` `user` VARCHAR(255) COLLATE utf8_danish_ci_
↳NOT NULL;
ALTER TABLE imagevariations CHANGE `publicKey` `user` VARCHAR(255) COLLATE utf8_
↳danish_ci NOT NULL;
```

If you use the Doctrine storage adapter for images and/or image variations, you will have to rename fields in those databases too:

Postgres/Other

```
ALTER TABLE storage_images RENAME COLUMN publicKey TO user;
ALTER TABLE storage_image_variations RENAME COLUMN publicKey TO user;
```

MySQL

```
ALTER TABLE storage_images CHANGE `publicKey` `user` VARCHAR(255) COLLATE utf8_danish_
↳ci NOT NULL;
ALTER TABLE storage_image_variations CHANGE `publicKey` `user` VARCHAR(255) COLLATE_
↳utf8_danish_ci NOT NULL;
```

Note: The `imagevariations` and `storage_image_variations` table might not be present in your database unless you previously upgraded to 1.2.4. In this case, skip the queries affecting those tables and instead follow the instructions specified in the *Database setup* section.

MongoDB

If you use the MongoDB adapter, you will need to rename the `publicKey` field with the following queries:

```
db.image.update({}, { $rename: { 'publicKey': 'user' } }, { multi: true })
db.shortUrl.update({}, { $rename: { 'publicKey': 'user' } }, { multi: true })
db.imagevariation.update({}, { $rename: { 'publicKey': 'user' } }, { multi: true })
```

Note: The `imagevariation` collection might not be present in your database unless you previously upgraded to 1.2.4. In this case, skip the last query and instead follow the instructions specified in the *Database setup* section.

GridFS

If you use the GridFS adapter, you will need to rename the `publicKey` field with the following query:

```
db.fs.files.update({}, { $rename: { 'publicKey': 'user' } }, { multi: true })
```

Note: The default database names for the GridFS adapters are `imbo_storage` and `imbo_imagevariation_storage`. The query specified should be run on both databases. If the `imbo_imagevariation_storage` database does not exist, run the query on `imbo_storage` and follow the instructions specified in the *Database setup* section to create the appropriate indexes for the `imbo_imagevariation_storage` database.

Image identifiers are no longer MD5-sums

Previously, Imbo used the MD5 algorithm to generate the image identifier for an image. In Imbo 2.0.0 and onwards, image identifiers are simply randomly generated strings. This means that the same image can exist multiple times within the same user. If this is not what you want, you can check if the image already exists by querying the *images resource* and specifying the MD5-sum of the image as an `originalChecksum-filter`. Most Imbo-clients implement this already, as `imageExists()` or similar.

To accommodate the new image identifiers and the possibility of future changes in how they are represented, databases should be able to store an image identifier of up to 255 characters. If you are using the *Doctrine database adapter* with the suggested schema on a MySQL database, this will require some changes:

```
ALTER TABLE imageinfo MODIFY imageIdentifier varchar(255) COLLATE utf8_danish_ci NOT NULL;
ALTER TABLE shorturl MODIFY imageIdentifier varchar(255) COLLATE utf8_danish_ci NOT NULL;
ALTER TABLE imagevariations MODIFY imageIdentifier varchar(255) COLLATE utf8_danish_ci NOT NULL;
```

If you use the Doctrine storage adapter for images and/or image variations, you will have to rename fields in those databases too:

```
ALTER TABLE storage_images MODIFY imageIdentifier varchar(255) COLLATE utf8_danish_ci NOT NULL;
ALTER TABLE storage_image_variations MODIFY imageIdentifier varchar(255) COLLATE utf8_danish_ci NOT NULL;
```

Note: The `imagevariations` and `storage_image_variations` table might not be present in your database unless you previously upgraded to 1.2.4. In this case, skip the queries affecting those tables and instead follow the instructions specified in the *Database setup* section.

Imbo-1.2.4

A new *Image Variations* event listener was introduced. It is disabled by default, and to use it you will have to configure a database and storage adapter for it - depending on your choice of adapters, you might need to modify your database. See the *Database setup* section.

Imbo-1.2.0

Below are the changes you need to be aware of when upgrading to Imbo-1.2.0.

- *Response to metadata write operations*
- *Original checksum*
 - *Doctrine*
 - *MongoDB*
- *Short image URLs*

Response to metadata write operations

Versions prior to 1.2.0 contained the image identifier in the response to HTTP POST/PUT/DELETE against the *metadata resource*. Starting from Imbo-1.2.0 the response to these requests will contain the metadata attached to the image instead. Read more about the different responses in the *metadata resource* section.

Original checksum

Imbo-1.2.0 includes a new feature that lets you filter images based on the original checksum of the image when querying the *images resource*. For this to work you need to add a field to your database. You can also populate this field for all images if you want, but this is not required. If you have event listeners that update incoming images, the values already stored in the database under the `checksum` field (which is used to populate the `originalChecksum` field in the following examples) might not be the checksum of the original image. If you don't have such event listeners added to your configuration you should be able to update the data as explained below and end up with 100% correct results.

Doctrine

If you use the *Doctrine database adapter* you'll need to add the new `originalChecksum` field to the table. The field has also been added to the *Database setup* section. The field should be added while there are no write operations pending, as a write could fail before upgrading Imbo itself.

```
ALTER TABLE imageinfo ADD COLUMN `originalChecksum` char(32) COLLATE utf8_danish_ci_
↳NOT NULL;
```

When you have added the field to your database you can run the following query to update all rows in the database:

```
UPDATE `imageinfo` SET `originalChecksum` = `checksum`
```

This query will simply copy the value of the existing `checksum` field over to `originalChecksum`. If you have a lot of images this operation might take a while.

MongoDB

If you use the MongoDB adapter all you need to do is to update all entries in the image collection:

```
db.image.find().forEach(  
  function (elem) {  
    db.image.update(  
      { _id: elem._id },  
      { $set: { originalChecksum: elem.checksum } }  
    );  
  }  
)
```

Short image URLs

In versions prior to Imbo-1.2.0 short image URLs were created automatically whenever a user agent requested the image resource (with or without transformations), and sent in the response as the `X-Imbo-ShortUrl` header. This no longer done automatically. Refer to the [Short URLs resource - /users/<user>/images/<image>/shorturls](#) section for more information on how to generate short URLs from this version on.

Configuration

Imbo ships with a default configuration file that will be automatically loaded. You will have to create one or more configuration files of your own that will be automatically merged with the default configuration by Imbo. The location of these files depends on the [installation method](#) you choose. You should never have to edit the default configuration file provided by Imbo.

The configuration file(s) you need to create should simply return arrays with configuration data. All available configuration options are covered in this chapter.

- [Using callables in configuration](#)
- [Imbo access control - `accessControl`](#)
- [Database configuration - `database`](#)
- [Storage configuration - `storage`](#)
- [Image identifier generation - `imageIdentifierGenerator`](#)
- [HTTP cache headers - `httpCacheHeaders`](#)
- [Content negotiation for images - `contentNegotiateImages`](#)
- [Trusted proxies - `trustedProxies`](#)
- [Authentication protocol - `authentication`](#)

- *Event listeners* - `eventListeners`
- *Event listener initializers* - `eventListenerInitializers`
- *Image transformation presets* - `transformationPresets`
- *Custom resources and routes* - `resources and routes`

Using callables in configuration

Imbo supports providing callables for certain configuration options. In Imbo 2 these are the `accessControl`, `database`, `eventListeners[name]`, `resource` and `storage` options, while Imbo 3 adds callable support for `transformationPresets`. The callable receives two arguments (`$request` and `$response`) which map to the active request and response objects.

By using a callable you can extend Imbo with custom logic for most configuration options (or provide the `$request` or `$response` objects to your implementing class), such as switching storage modules based on which user performed the request.

Imbo access control - `accessControl`

Imbo catalogs stored images under a `user`. To add an image to a given user, you need a public and private key pair. This pair is used to sign requests to Imbo's API and ensures that the API can't be accessed without knowing the private key.

Multiple public keys can be given access to a user, and you can also configure a public key to have access to several users. It's important to note that a `user` doesn't have to be created in any way - as long as a public key is defined to have access to a given user, you're ready to start adding images.

Public keys can be configured to have varying degrees of access. For instance, you might want one public key for write operations (such as adding and deleting images) and a different public key for read operations (such as viewing images and applying transformations to them). Access is defined on a `resource` basis - which basically translates to an API endpoint and an HTTP method. To retrieve an image, for instance, you would give access to the `image.get` resource.

Specifying a long list of resources can get tedious, so Imbo also supports `resource groups` - basically just a list of different resources. When creating access rules for a public key, these can be used instead of specifying specific resources.

For the private keys you can for instance use a [SHA-256](#) hash of a random value. The private key is used by clients to sign requests, and if you accidentally give away your private key users can use it to delete all your images (given the public key it belongs to has write access). Make sure not to generate a private key that is easy to guess (like for instance the MD5 or SHA-256 hash of the public key). Imbo does not require the private key to be in a specific format, so you can also use regular passwords if you want. The key itself will never be a part of the payload sent to/from the server.

Imbo ships with a small command line tool that can be used to generate private keys for you using the `openssl_random_pseudo_bytes` function. The tool is located in the `bin` directory of the Imbo installation:

```
$ ./bin/imbo generate-private-key
3b98dde5f67989a878b8b268d82f81f0858d4f1954597cc713ae161cdfcc84a
```

The private key can be changed whenever you want as long as you remember to change it in both the server configuration and in the client you use. The user can not be changed easily as database and storage adapters use it when storing/fetching images and metadata.

Access control is managed by `adapters`. The simplest adapter is the `SimpleArrayAdapter`, which has a number of trade-offs in favor of being easy to set up. Mainly, it expects the public key to have the same name as the

user it should have access to, and that the public key should be given full read+write access to all resources belonging to that user.

Warning: It's not recommended that you use the same public key for both read and write operations. Read on to see how you can create different public keys for read and read/write access.

The adapter is set up using the `accessControl` key in your configuration file:

```
<?php
return [
    // ...

    'accessControl' => function() {
        return new Imbo\Auth\AccessControl\Adapter\SimpleArrayAdapter([
            'some-user' => 'my-super-secret-private-key',
            'other-user' => 'other-super-secret-private-key',
        ]);
    },
    // ...
];
```

It's usually a good idea to have separate public keys for read-only and read+write operations. You can achieve this by using a more flexible access control adapter, such as the `ArrayAdapter`:

```
<?php
use Imbo\Auth\AccessControl\Adapter\ArrayAdapter,
    Imbo\Resource;

return [
    // ...

    'accessControl' => function() {
        return new ArrayAdapter([
            [
                'publicKey' => 'some-read-only-pubkey',
                'privateKey' => 'some-private-key',
                'acl' => [[
                    'resources' => Resource::getReadOnlyResources(),
                    'users' => ['some-user']
                ]]
            ],
            [
                'publicKey' => 'some-read-write-pubkey',
                'privateKey' => 'some-other-private-key',
                'acl' => [[
                    'resources' => Resource::getReadWriteResources(),
                    'users' => ['some-user']
                ]]
            ]
        ]);
    },
    // ...
];
```

As you can see, the `ArrayAdapter` is much more flexible than the `SimpleArrayAdapter`. The above example

only shows part of this flexibility. You can also provide resource groups and multiple access control rules per public key. The following example shows this more clearly:

```
<?php
use Imbo\Auth\AccessControl\Adapter\ArrayAdapter,
    Imbo\Resource

return [
    // ...

    'accessControl' => function() {
        return new ArrayAdapter([
            [
                // A unique public key matching the following regular expression: [A-
                ↪Za-z0-9_-]{1,}
                'publicKey' => 'some-pubkey',

                // Some form of private key
                'privateKey' => 'some-private-key',

                // Array of rules for this public key
                'acl' => [
                    [
                        // An array of different resource names that the public key
                        ↪should have
                        // access to - see AdapterInterface::RESOURCE_* for available
                        ↪options.
                        'resources' => Resource::getReadOnlyResources(),

                        // Names of the users which the public key should have access
                        ↪to.
                        'users' => ['some', 'users'],
                    ],

                    // Multiple rules can be applied in order to make a single public
                    ↪key have
                    // different access rights on different users
                    [
                        'resources' => Resource::getReadWriteResources(),
                        'users' => ['different-user'],
                    ],

                    // You can also specify resource groups instead of explicitly
                    ↪setting them like
                    // in the above examples. Note that you cannot specify both
                    ↪resources and group
                    // in the same rule.
                    [
                        'group' => 'read-stats',
                        'users' => ['user1', 'user2']
                    ]
                ]
            ], [
                // Second argument to the ArrayAdapter being the available resource groups
                // Format: 'name' => ['resource1', 'resource2']
                'read-stats' => ['user.get', 'user.head', 'user.options'],
            ]
        ]);
    },
],
```

```
// ...  
];
```

Imbo also ships with a MongoDB access control adapter, which is mutable. This means you can manipulate the access control rules on the fly, using Imbo's API. The adapter uses PHP's [mongo extension](#). The following parameters are supported:

databaseName Name of the database to use. Defaults to `imbo`.

server The server string to use when connecting. Defaults to `mongodb://localhost:27017`.

options Options passed to the underlying adapter. Defaults to `['connect' => true, 'timeout' => 1000]`. See the [manual for the MongoClient constructor](#) for available options.

```
<?php  
return [  
    // ...  
  
    'accessControl' => function() {  
        return new Imbo\Auth\AccessControl\Adapter\MongoDB([  
            'databaseName' => 'imbo-acl'  
        ]);  
    },  
  
    // ...  
];
```

When using a mutable access control adapter, you will need to create an initial public key that can subsequently be used to create other public keys. The easiest way to create public keys when using a mutable adapter is to utilize the *add-public-key command* provided by the CLI tool that Imbo is shipped with.

Database configuration - database

The database adapter you decide to use is responsible for storing metadata and basic image information, like width and height for example, along with the generated short URLs. Imbo ships with some different database adapters that you can use. Remember that you will not be able to switch the adapter whenever you want and expect all data to be automatically transferred. Choosing a database adapter should be a long term commitment unless you have migration scripts available.

In the default configuration file the *MongoDB* database adapter is used. You can choose to override this in your configuration file by specifying a different adapter. You can either specify an instance of a database adapter directly, or specify a closure that will return an instance of a database adapter when executed. Which database adapter to use is specified in the `database` key in the configuration array:

```
<?php  
return [  
    // ...  
  
    'database' => function() {  
        return new Imbo\Database\MongoDB([  
            'databaseName' => 'imbo',  
        ]);  
    },  
  
    // or
```

```
'database' => new Imbo\Database\MongoDB([
    'databaseName' => 'imbo',
]),
// ...
];
```

Below you will find documentation on the different database adapters Imbo ships with.

- *Doctrine*
- *MongoDB*
- *Mongo*
- *Custom database adapter*

Doctrine

This adapter uses the [Doctrine Database Abstraction Layer](#). The options you pass to the constructor of this adapter is passed to the underlying classes, so have a look at the Doctrine DBAL documentation over at [doctrine-project.org](#). When using this adapter you need to create the required tables in the RDBMS first, as specified in the *Database setup* section.

Examples

Here are some examples on how to use the Doctrine adapter in the configuration file:

1. Use a PDO instance to connect to a SQLite database:

```
<?php
return [
    // ...

    'database' => function() {
        return new Imbo\Database\Doctrine([
            'pdo' => new PDO('sqlite:/path/to/database'),
        ]);
    },
    // ...
];
```

2. Connect to a MySQL database using PDO:

```
<?php
return [
    // ...

    'database' => function() {
        return new Imbo\Database\Doctrine([
            'dbname' => 'database',
            'user' => 'username',
            'password' => 'password',
        ]);
    },
    // ...
];
```

```
        'host'      => 'hostname',
        'driver'    => 'pdo_mysql',
    });
},
// ...
];
```

MongoDB

This adapter uses PHP's [mongo extension](#) to store data in [MongoDB](#). The following parameters are supported:

databaseName Name of the database to use. Defaults to `imbo`.

server The server string to use when connecting. Defaults to `mongodb://localhost:27017`.

options Options passed to the underlying adapter. Defaults to `['connect' => true, 'timeout' => 1000]`. See the [manual for the MongoClient constructor](#) for available options.

Examples

1. Connect to a local MongoDB instance using the default `databaseName`:

```
<?php
return [
    // ...

    'database' => function() {
        return new Imbo\Database\MongoDB();
    },

    // ...
];
```

2. Connect to a replica set:

```
<?php
return [
    // ...

    'database' => function() {
        return new Imbo\Database\MongoDB([
            'server' => 'mongodb://server1,server2,server3',
            'options' => [
                'replicaSet' => 'nameOfReplicaSet',
            ],
        ]);
    },

    // ...
];
```

Mongo

This adapter uses PHP's [mongodb extension](#). It can be configured in the same way as the [MongoDB](#) adapter.

Custom database adapter

If you need to create your own database adapter you need to create a class that implements the `Imbo\Database\DatabaseInterface` interface, and then specify that adapter in the configuration:

```
<?php
return [
    // ...

    'database' => function() {
        return new My\Custom\Adapter([
            'some' => 'option',
        ]);
    },
    // ...
];
```

You can read more about how to achieve this in the *Implement your own database and/or storage adapter* chapter.

Storage configuration - storage

Storage adapters are responsible for storing the original images you put into Imbo. As with the database adapter it is not possible to simply switch the adapter without having migration scripts available to move the stored images. Choose an adapter with care.

In the default configuration file the *GridFS* storage adapter is used. You can choose to override this in your configuration file by specifying a different adapter. You can either specify an instance of a storage adapter directly, or specify a closure that will return an instance of a storage adapter when executed. Which storage adapter to use is specified in the `storage` key in the configuration array:

```
<?php
return [
    // ...

    'storage' => function() {
        return new Imbo\Storage\Filesystem([
            'dataDir' => '/path/to/images',
        ]);
    },
    // or

    'storage' => new Imbo\Storage\Filesystem([
        'dataDir' => '/path/to/images',
    ]),
    // ...
];
```

Below you will find documentation on the different storage adapters Imbo ships with.

- *Amazon Simple Storage Service*
- *Doctrine*

- *Filesystem*
- *GridFS*
- *Custom storage adapter*

Amazon Simple Storage Service

This adapter stores your images in a bucket in the Amazon Simple Storage Service (S3). The parameters are:

key Your AWS access key

secret Your AWS secret key

bucket The name of the bucket you want to store your images in. Imbo will **not** create this for you.

This adapter creates subdirectories in the bucket in the same fashion as the *Filesystem storage adapter* stores the files on the local filesystem.

Examples

```
<?php
return [
    // ...

    'storage' => function() {
        new Imbo\Storage\S3([
            'key' => '<aws access key>'
            'secret' => '<aws secret key>',
            'bucket' => 'my-imbo-bucket',
        ]);
    },
    // ...
];
```

Doctrine

This adapter uses the [Doctrine Database Abstraction Layer](#). The options you pass to the constructor of this adapter is passed to the underlying classes, so have a look at the Doctrine DBAL documentation over at [doctrine-project.org](#). When using this adapter you need to create the required tables in the RDBMS first, as specified in the *Database setup* section.

Examples

Here are some examples on how to use the Doctrine adapter in the configuration file:

1. Use a PDO instance to connect to a SQLite database:

```
<?php
return [
    // ...

    'storage' => function() {
```

```

        return new Imbo\Storage\Doctrine([
            'pdo' => new PDO('sqlite:/path/to/database'),
        ]);
    },
    // ...
];

```

2. Connect to a MySQL database using PDO:

```

<?php
return [
    // ...

    'storage' => function() {
        return new Imbo\Storage\Doctrine([
            'dbname' => 'database',
            'user' => 'username',
            'password' => 'password',
            'host' => 'hostname',
            'driver' => 'pdo_mysql',
        ]);
    },
    // ...
];

```

Filesystem

This adapter simply stores all images on the file system. It has a single parameter, and that is the base directory of where you want your images stored:

dataDir The base path where the images are stored.

This adapter is configured to create subdirectories inside of `dataDir` based on the user and the checksum of the images added to Imbo. The algorithm that generates the path simply takes the three first characters of the user and creates directories for each of them, then the complete user, then a directory of each of the first characters in the image identifier, and lastly it stores the image in a file with a filename equal to the image identifier itself. For instance, an image stored under the user `foobar` with the image identifier `5c01e554-9fca-4231-bb95-a6eabf259b64` would be stored as `<dataDir>/f/o/o/foobar/5/c/0/5c01e554-9fca-4231-bb95-a6eabf259b64`.

Examples

1. Store images in `/path/to/images`:

```

<?php
return [
    // ...

    'storage' => function() {
        new Imbo\Storage\Filesystem([
            'dataDir' => '/path/to/images',
        ]);
    },
];

```

```
    // ...  
];
```

GridFS

The GridFS adapter is used to store the images in MongoDB using the [GridFS specification](#). This adapter has the following parameters:

databaseName The name of the database to store the images in. Defaults to `imbo_storage`.

server The server string to use when connecting to MongoDB. Defaults to `mongodb://localhost:27017`

options Options passed to the underlying adapter. Defaults to `['connect' => true, 'timeout' => 1000]`. See the [manual for the MongoClient constructor](#) for available options.

Examples

1. Connect to a local MongoDB instance using the default `databaseName`:

```
<?php  
return [  
    // ...  
  
    'storage' => function() {  
        return new Imbo\Storage\GridFS();  
    },  
  
    // ...  
];
```

2. Connect to a replica set:

```
<?php  
return [  
    // ...  
  
    'storage' => function() {  
        return new Imbo\Storage\GridFS([  
            'server' => 'mongodb://server1,server2,server3',  
            'options' => [  
                'replicaSet' => 'nameOfReplicaSet',  
            ],  
        ]);  
    },  
  
    // ...  
];
```

Custom storage adapter

If you need to create your own storage adapter you need to create a class that implements the `Imbo\Storage\StorageInterface` interface, and then specify that adapter in the configuration:

```
<?php
return [
    // ...

    'storage' => function() {
        return new My\Custom\Adapter([
            'some' => 'option',
        ]);
    },
    // ...
];
```

You can read more about how to achieve this in the *Implement your own database and/or storage adapter* chapter.

Image identifier generation - `imageIdentifierGenerator`

By default, Imbo will generate a random string of characters as the image identifier for added images. These are in the RegExp range `[A-Za-z0-9_-]` and by default, the identifier will be 12 characters long.

You can easily change the generation process to a different method. Imbo currently ships with two generators:

RandomString

The default, as stated above. This generator has the following parameters:

length The length of the randomly generated string. Defaults to 12.

Uuid

Generates 36-character v4 UUIDs, for instance `f47ac10b-58cc-4372-a567-0e02b2c3d479`. This generator does not have any parameters.

Usage:

```
<?php
return [
    // ...

    'imageIdentifierGenerator' => new Imbo\Image\Identifier\Generator\Uuid(),
    // ...
];
```

Custom generators

To create your own custom image identifier generators, simply create a class that implements `Imbo\Image\Identifier\Generator\GeneratorInterface` and ensure that the identifiers generated are in the character range `[A-Za-z0-9_-]` and are between one and 255 characters long.

HTTP cache headers - `httpCacheHeaders`

Imbo ships with reasonable defaults for which HTTP cache header settings it sends to clients. For some resources, however, it can be difficult to figure out a good middle ground between clients asking too often and too rarely. For instance, the `images` resource will change every time a new image has been added - but whether that happens once a second or once a year is hard to know.

To ensure that clients get fresh responses, Imbo sends `max-age=0`, `must-revalidate` on these kind of resources. You can however override these defaults in the configuration. For instance, if you wanted to set the `max-age` to 30 seconds, leave it up to the client if it should re-validate and tell intermediary proxies that this response is private, you could set the configuration to the following:

```
<?php
return [
    // ...

    'httpCacheHeaders' => [
        'maxAge' => 30,
        'mustRevalidate' => false,
        'public' => false,
    ],

    // ...
];
```

Content negotiation for images - `contentNegotiateImages`

By default, Imbo will do content negotiation for images. In other words, if a request is sent for an image with the `Accept`-header `image/jpeg`, it will try to deliver the image in JPEG-format.

If what you want is for images to be delivered in the format they were uploaded in, you can set `contentNegotiateImages` to `false` in the configuration. This will also ensure Imbo does not include `Accept` in the `Vary`-header for image requests, which will make caching behind reverse proxies more efficient.

You are still able to convert between formats by specifying an extension when requesting the image (`.jpg`, `.png`, `.gif` etc).

Trusted proxies - `trustedProxies`

If you find yourself behind some sort of reverse proxy (like a load balancer), certain header information may be sent to you using special `X-Forwarded-*` headers. For example, the `Host` HTTP-header is usually used to return the requested host. But when you're behind a proxy, the true host may be stored in an `X-Forwarded-Host` header.

Since HTTP headers can be spoofed, Imbo does not trust these proxy headers by default. If you are behind a proxy, you should manually whitelist your proxy. This can be done by defining the proxies IP addresses and/or using CIDR notations. Example:

```
<?php
return [
    // ...

    'trustedProxies' => ['192.0.0.1', '10.0.0.0/8'],

    // ...
];
```

Note: Not all proxies set the required `X-Forwarded-*` headers by default. A search for `X-Forwarded-Proto <your proxy here>` usually gives helpful answers to how you can add them to incoming requests.

Authentication protocol - authentication

Imbo generates access tokens and authentication signatures based on the incoming URL, and includes the protocol (by default). This can sometimes be problematic, for instance when Imbo is behind a load balancer which doesn't send `X-Forwarded-Proto` header, or if you want to use protocol-less image URLs on the client side (`//imbo.host/users/some-user/images/img`).

Setting the `protocol` option under `authentication` allows you to control how Imbo's authentication should behave. The option has the following possible values:

incoming Will try to detect the incoming protocol - this is based on `$_SERVER['HTTPS']` or the `X-Forwarded-Proto` header (given the `trustedProxies` option is configured). This is the default value.

both Will try to match based on both HTTP and HTTPS protocols and allow the request if any of them yields the correct signature/access token.

http Will always use `http` as the protocol, replacing `https` with `http` in the incoming URL, if that is the case.

https Will always use `https` as the protocol, replacing `http` with `https` in the incoming URL, if that is the case.

Example usage:

```
<?php
return [
    // ...

    'authentication' => [
        'protocol' => 'both',
    ],

    // ...
];
```

Event listeners - eventListeners

Imbo support event listeners that you can use to hook into Imbo at different phases without having to edit Imbo itself. An event listener is simply a piece of code that will be executed when a certain event is triggered from Imbo. Event listeners are added to the `eventListeners` part of the configuration array as associative arrays. If you want to disable some of the default event listeners simply specify the same key in your configuration file and set the value to `null` or `false`. Keep in mind that not all event listeners should be disabled.

Event listeners can be configured in the following ways:

1. A string representing a class name of a class implementing the `Imbo\EventListener\ListenerInterface` interface:

```
<?php
return [
    // ...

    'eventListeners' => [
        'accessToken' => 'Imbo\EventListener\AccessToken',
```

```
    ],  
    // ...  
];
```

2. Use an instance of a class implementing the `Imbo\EventListener\ListenerInterface` interface:

```
<?php  
return [  
    // ...  
    'eventListeners' => [  
        'accessToken' => new Imbo\EventListener\AccessToken(),  
    ],  
    // ...  
];
```

3. A closure returning an instance of a class implementing the `Imbo\EventListener\ListenerInterface` interface:

```
<?php  
return [  
    // ...  
    'eventListeners' => [  
        'accessToken' => function() {  
            return new Imbo\EventListener\AccessToken();  
        },  
    ],  
    // ...  
];
```

4. Use a class implementing the `Imbo\EventListener\ListenerInterface` interface together with an optional user filter:

```
<?php  
return [  
    // ...  
    'eventListeners' => [  
        'maxImageSize' => [  
            'listener' => new Imbo\EventListener\MaxImageSize(1024, 768),  
            'users' => [  
                'whitelist' => ['user'],  
                // 'blacklist' => ['someotheruser'],  
            ],  
            // 'params' => [ ... ]  
        ],  
    ],  
    // ...  
];
```

where `listener` is one of the following:

1. a string representing a class name of a class implementing the `Imbo\EventListener\ListenerInterface`

interface

2. an instance of the `Imbo\EventListener\ListenerInterface` interface
3. a closure returning an instance `Imbo\EventListener\ListenerInterface`

The `users` element is an array that you can use if you want your listener to only be triggered for some users. The value of this is an array with two elements, `whitelist` and `blacklist`, where `whitelist` is an array of users you **want** your listener to trigger for, and `blacklist` is an array of users you **don't want** your listener to trigger for. `users` is optional, and per default the listener will trigger for all users.

There also exists a `params` key that can be used to specify parameters for the event listener, if you choose to specify the listener as a string in the `listener` key:

```
<?php
return [
    // ...

    'eventListeners' => [
        'maxImageSize' => [
            'listener' => 'Imbo\EventListener\MaxImageSize',
            'users' => [
                'whitelist' => ['user'],
                // 'blacklist' => ['someotheruser'],
            ],
            'params' => [
                'width' => 1024,
                'height' => 768,
            ]
        ],
    ],
    // ...
];
```

The value of the `params` array will be sent to the constructor of the event listener class.

5. Use a closure directly:

```
<?php
return [
    // ...

    'eventListeners' => [
        'customListener' => [
            'callback' => function(Imbo\EventManager\EventInterface $event) {
                // Custom code
            },
            'events' => ['image.get'],
            'priority' => 1,
            'users' => [
                'whitelist' => ['user'],
                // 'blacklist' => ['someotheruser'],
            ],
        ],
    ],
    // ...
];
```

where `callback` is the code you want executed, and `events` is an array of the events you want it triggered for. `priority` is the priority of the listener and defaults to 0. The higher the number, the earlier in the chain your listener will be triggered. This number can also be negative. Imbo's internal event listeners uses numbers between 0 and 100. `users` uses the same format as described above. If you use this method, and want your callback to trigger for multiple events with different priorities, specify an associative array in the `events` element, where the keys are the event names, and the values are the priorities for the different events. This way of attaching event listeners should mostly be used for quick and temporary solutions.

All event listeners will receive an event object (which implements `Imbo\EventManager\EventInterface`), that is described in detail in the *The event object* section.

Listeners added by default

The default configuration file includes some event listeners by default:

- *Access token*
- *Authenticate*
- *Stats access*
- *Imagick*

as well as event listeners for image transformations:

- *autoRotate*
- *border*
- *canvas*
- *compress*
- *convert*
- *crop*
- *desaturate*
- *flipHorizontally*
- *flipVertically*
- *maxSize*
- *resize*
- *rotate*
- *sepia*
- *smartSize*
- *strip*
- *thumbnail*
- *transpose*
- *transverse*
- *vignette*
- *watermark*

Read more about these listeners (and more) in the *Customize your Imbo installation with event listeners* and *Transforming images on the fly* chapters. If you want to disable any of these you could do so in your configuration file in the following way:

```
<?php
return [
    // ...

    'eventListeners' => [
        'accessToken' => null,
        'auth' => null,
        'statsAccess' => null,
    ],
    // ...
];
```

Warning: Do not disable the event listeners used in the example above unless you are absolutely sure about the consequences. Your images can potentially be deleted by anyone.

Warning: Disabling image transformation event listeners is not recommended.

Event listener initializers - `eventListenerInitializers`

Some event listeners might require custom initialization, and if you don't want to do this in-line in the configuration, Imbo supports event listener initializer classes. This is handled via the `eventListenerInitializers` key. The value of this element is an associative array where the keys identify the initializers (only used in the configuration itself), and the values are strings representing class names, or implementations of the `Imbo\EventListener\Initializer\InitializerInterface` interface. If you specify strings the classes you refer to must also implement this interface.

The interface has a single method called `initialize` and receives instances of event listeners implementing the `Imbo\EventListener\ListenerInterface` interface. This method is called once for each event listener instantiated by Imbo's event manager. Example:

```
<?php
// Some event listener
class Listener implements Imbo\EventListener\ListenerInterface {
    public function setDependency($dependency) {
        // ...
    }

    // ...
}

class OtherListener implements Imbo\EventListener\ListenerInterface {
    public function setDependency($dependency) {
        // ...
    }

    // ...
}
```

```

// Event listener initializer
class Initializer implements Imbo\EventListener\Initializer\InitializerInterface {
    private $dependency;

    public function __construct() {
        $this->dependency = new SomeDependency();
    }

    public function initialize(Imbo\EventListener\ListenerInterface $listener) {
        if ($listener instanceof Listener || $listener instanceof OtherListener) {
            $listener->setDependency($this->dependency);
        }
    }
}

// Configuration
return [
    'eventListeners' => [
        'customListener' => 'Listener',
        'otherCustomListener' => 'OtherListener',
    ],

    'eventListenerInitializers' => [
        'initializerForCustomListener' => 'Initializer',
    ],
];

```

In the above example the `Initializer` class will be instantiated by Imbo, and in the `__construct` method it will create an instance of some dependency. When the event manager creates the instances of the two event listeners these will in turn be sent to the `initialize` method, and the same dependency will be injected into both listeners. An alternative way to accomplish this by using Closures in the configuration could look something like this:

```

<?php
$dependency = new SomeDependency();

return [
    'eventListeners' => [
        'customListener' => function() use ($dependency) {
            $listener = new Listener();
            $listener->setDependency($dependency);

            return $listener;
        },
        'otherCustomListener' => function() use ($dependency) {
            $listener = new OtherListener();
            $listener->setDependency($dependency);

            return $listener;
        },
    ],
];

```

Imbo itself includes an event listener initializer in the default configuration that is used to inject the same instance of Imagick to all image transformations.

Note: Only event listeners specified as strings (class names) in the configuration will be instantiated by Imbo, so

event listeners instantiated in the configuration array, either directly or via a Closures, will not be initialized by the configured event listener initializers.

Image transformation presets - `transformationPresets`

Through the configuration you can also combine image transformations to make presets (transformation chains). This is done via the `transformationPresets` key:

```
<?php
return [
    // ...

    'transformationPresets' => [
        'graythumb' => [
            'thumbnail',
            'desaturate',
        ],
        // ...
    ],
    // ...
];
```

where the keys are the names of the transformations as specified in the URL, and the values are arrays containing other transformation names (as used in the `eventListeners` part of the configuration). You can also specify hard coded parameters for the presets if some of the transformations in the chain supports parameters:

```
<?php
return [
    // ...

    'transformationPresets' => [
        'fixedGraythumb' => [
            'thumbnail' => [
                'width' => 50,
                'height' => 50,
            ],
            'desaturate',
        ],
        // ...
    ],
    // ...
];
```

By doing this the `thumbnail` part of the `fixedGraythumb` preset will ignore the `width` and `height` query parameters, if present. By only specifying for instance `'width' => 50` in the configuration the height of the thumbnail can be adjusted via the `query` parameter, but the `width` is fixed.

Note: The URLs will stay the same if you change the transformation chain in a preset. Keep this in mind if you use for instance Varnish or some other HTTP accelerator in front of your web server(s).

Custom resources and routes - resources and routes

Warning: Custom resources and routes is an experimental and advanced way of extending Imbo, and requires extensive knowledge of how Imbo works internally. This feature can potentially be removed in future releases, so only use this for testing purposes.

If you need to create a custom route you can attach a route and a custom resource class using the configuration. Two keys exists for this purpose: `resources` and `routes`:

```
<?php
return [
    // ...

    'resources' => [
        'users' => new ImboUsers();

        // or

        'users' => function() {
            return new ImboUsers();
        },

        // or

        'users' => 'ImboUsers',
    ],

    'routes' => [
        'users' => '#^/users(\.(?<extension>json|xml))?$#',
    ],

    // ...
];
```

In the above example we are creating a route for Imbo using a regular expression, called `users`. The route itself will match the following three requests:

- `/users`
- `/users.json`
- `/users.xml`

When a request is made against any of these endpoints Imbo will try to access a resource that is specified with the same key (`users`). The value specified for this entry in the `resources` array can be:

1. a string representing the name of the resource class
2. an instance of a resource class
3. an anonymous function that, when executed, returns an instance of a resource class

The resource class must implement the `Imbo\Resource\ResourceInterface` interface to be able to response to a request.

Below is an example implementation of the `ImboUsers` resource used in the above configuration:

```
<?php
use Imbo\Resource\ResourceInterface,
```

```

Imbo\EventManager\EventInterface,
Imbo\Model\ListModel;

class ImboUsers implements ResourceInterface {
    public function getAllowedMethods() {
        return ['GET'];
    }

    public static function getSubscribedEvents() {
        return [
            'users.get' => 'get',
        ];
    }

    public function get(EventInterface $event) {
        $model = new ListModel();
        $model->setList('users', 'user', array_keys($event->getConfig()['auth']));
        $event->getResponse()->setModel($model);
    }
}

```

This resource informs Imbo that it supports HTTP GET, and specifies a callback for the `users.get` event. The name of the event is the name specified for the resource in the configuration above, along with the HTTP method, separated with a dot.

In the `get()` method we are simply creating a list model for Imbo's response formatter, and we are supplying the keys from the `auth` part of your configuration file as data. When formatted as JSON the response looks like this:

```

{
  "users": [
    "someuser",
    "someotheruser"
  ]
}

```

and the XML representation looks like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<imbo>
  <users>
    <user>someuser</user>
    <user>someotheruser</user>
  </users>
</imbo>

```

Feel free to experiment with this feature. If you end up creating a resource that you think should be a part of Imbo, send a [pull request on GitHub](#).

Customize your Imbo installation with event listeners

Imbo ships with a collection of event listeners for you to use. Some of them are enabled in the default configuration file. Image transformations are also technically event listeners, but will not be covered in this chapter. Read the [Transforming images on the fly](#) chapter for more information regarding the image transformations.

- *Access token*
- *Authenticate*
- *Auto rotate image*
- *CORS (Cross-Origin Resource Sharing)*
- *EXIF metadata*
- *Image transformation cache*
- *Image variations*
- *Imagick*
- *Max image size*
- *Metadata cache*
- *Stats access*
- *Varnish HashTwo*

Access token

This event listener enforces the usage of access tokens on all read requests against user-specific resources. You can read more about how the actual access tokens work in the *Access tokens* part of the *Imbo's API* chapter.

To enforce the access token check this event listener subscribes to the following events:

- `user.get`
- `user.head`
- `images.get`
- `images.head`
- `image.get`
- `image.head`
- `metadata.get`
- `metadata.head`

This event listener has a single parameter that can be used to whitelist and/or blacklist certain image transformations, used when the current request is against an image resource. The parameter is an array with a single key: `transformations`. This is another array with two keys: `whitelist` and `blacklist`. These two values are arrays where you specify which transformation(s) to whitelist or blacklist. The names of the transformations are the same as the ones used in the request. See *Transforming images on the fly* for a complete list of the supported transformations.

Use `whitelist` if you want the listener to skip the access token check for certain transformations, and `blacklist` if you want it to only check certain transformations:

```
[
  'transformations' => [
    'whitelist' => [
      'border',
    ],
  ],
]
```



```
],
]
```

means that the access token will **not** be enforced for the *border* transformation.

```
[
  'transformations' => [
    'blacklist' => [
      'border',
    ],
  ],
]
```

means that the access token will be enforced **only** for the *border* transformation.

If both `whitelist` and `blacklist` are specified all transformations will require an access token unless it's included in `whitelist`.

This event listener is included in the default configuration file without specifying any transformation filters:

```
<?php
return [
    // ...

    'eventListeners' => [
        'accessToken' => 'Imbo\EventListener\AccessToken',
    ],

    // ...
];
```

Disable this event listener with care. Installations with no access token check is open for [DoS attacks](#).

Authenticate

This event listener enforces the usage of signatures on all write requests against user-specific resources. You can read more about how the actual signature check works in the *Signing write requests* section in the *Imbo's API* chapter.

To enforce the signature check for all write requests supported by Imbo this event listener subscribes to the following events:

- `images.post`
- `image.delete`
- `metadata.put`
- `metadata.post`
- `metadata.delete`

This event listener does not support any parameters and is enabled per default like this:

```
<?php
return [
    // ...

    'eventListeners' => [
        'auth' => 'Imbo\EventListener\Authenticate',
    ],
];
```

```
    ],  
    // ...  
];
```

Disable this event listener with care. User agents can delete all your images and metadata if this listener is disabled.

Auto rotate image

This event listener will auto rotate new images based on metadata embedded in the image itself (EXIF).

The listener does not support any parameters and can be enabled like this:

```
<?php  
return [  
    // ...  
    'eventListeners' => [  
        'autoRotateListener' => 'Imbo\EventListener\AutoRotateImage',  
    ],  
    // ...  
];
```

If you enable this listener all new images added to Imbo will be auto rotated based on the EXIF data. This might also cause the image identifier sent in the response to be different from the one used in the URI when storing the image. This can happen with all event listeners which can possibly modify the image before storing it.

CORS (Cross-Origin Resource Sharing)

This event listener can be used to allow clients such as web browsers to use Imbo when the client is located on a different origin/domain than the Imbo server is. This is implemented by sending a set of CORS-headers on specific requests, if the origin of the request matches a configured domain.

The event listener can be configured on a per-resource and per-method basis, and will therefore listen to any related events. If enabled without any specific configuration, the listener will allow and respond to the **GET**, **HEAD** and **OPTIONS** methods on all resources. Note however that no origins are allowed by default and that a client will still need to provide a valid access token, unless the *Access token listener* is disabled.

Here is an example on how to enable the CORS listener:

```
<?php  
return [  
    // ...  
    'eventListeners' => [  
        'cors' => [  
            'listener' => 'Imbo\EventListener\Cors',  
            'params' => [  
                'allowedOrigins' => ['http://some.origin'],  
                'allowedMethods' => [  
                    'image' => ['GET', 'HEAD'],  
                    'images' => ['GET', 'HEAD', 'POST'],  
                ],  
                'maxAge' => 3600,  
            ],  
        ],  
    ],  
];
```

```

    ],
  ],
  // ...
];

```

Below all supported parameters are listed:

allowedOrigins is an array of allowed origins. Specifying `*` as a value in the array will allow any origin.

allowedMethods is an associative array where the keys represent the resource (`shorturl`, `status`, `stats`, `user`, `images`, `image` and `metadata`) and the values are arrays of HTTP methods you wish to open up.

maxAge specifies how long the response of an `OPTIONS`-request can be cached for, in seconds. Defaults to 3600 (one hour).

EXIF metadata

This event listener can be used to fetch the EXIF-tags from uploaded images and adding them as metadata. Enabling this event listener will not populate metadata for images already added to Imbo.

The event listener subscribes to the following events:

- `images.post`
- `db.image.insert`

and the parameters given to the event listener supports a single element:

allowedTags The tags you want to be populated as metadata. Defaults to `exif:*`. When specified it will override the default value, so if you want to register all `exif` and `date` tags for example, you will need to specify them both.

and is enabled like this:

```

<?php
return [
  // ...

  'eventListeners' => [
    'exifMetadata' => [
      'listener' => 'Imbo\EventListener\ExifMetadata',
      'params' => [
        'allowedTags' => ['exif:*', 'date:*', 'png:gAMA'],
      ],
    ],
  ],
  // ...
];

```

which would allow all `exif` and `date` properties as well as the `png:gAMA` property. If you want to store **all** tags as metadata, use `['*']` as filter.

Image transformation cache

This event listener enables caching of image transformations. Read more about image transformations in the *Transforming images on the fly* section.

To achieve this the listener subscribes to the following events:

- `image.get`
- `response.send`
- `image.delete`

The parameters for the event listener supports a single element:

path Root path where the cached images will be stored.

and is enabled like this:

```
<?php
return [
    // ...

    'eventListeners' => [
        'imageTransformationCache' => [
            'listener' => 'Imbo\EventListener\ImageTransformationCache',
            'params' => [
                'path' => '/path/to/cache',
            ],
        ],
    ],
],
// ...
];
```

Note: This event listener uses a similar algorithm when generating file names as the *Filesystem* storage adapter.

Warning: It can be wise to purge old files from the cache from time to time. If you have a large amount of images and present many different variations of these the cache will use up quite a lot of storage.

An example on how to accomplish this:

```
$ find /path/to/cache -ctime +7 -type f -delete
```

The above command will delete all files in `/path/to/cache` older than 7 days and can be used with for instance `crontab`.

Image variations

This event listener can be used to generate multiple variations of incoming images so that a more suitable size can be used when performing scaling transformations. This will increase the amount of data stored by Imbo, but it will also improve performance. In cases where the original images are very large and the requested transformed images are significantly smaller, the difference will be quite drastic.

The event listener has two roles, one is to generate the variations when new images are added, and the other is to pick the most fitting image variation when clients request an image with a set of transformations applied that will alter the dimensions of the image, for instance *resize* or *thumbnail*.

Imbo ships with MongoDB and Doctrine adapters for storing metadata about these variations. If you want to use a different database, you can implement the `Imbo\EventListener\ImageVariations\Database\DatabaseInterface` interface and set the name of the class in the configuration of the event listener.

In the same way, Imbo ships three different adapters for storing the actual image variation data (the downscaled images): GridFS, Doctrine and Filesystem. See examples of their configuration below.

The event listener supports for following configuration parameters:

- (boolean) lossless** Have Imbo use a lossless format to store the image variations. This results in better quality images, but converting between formats can be slower and will use more disk space. Defaults to `false`.
- (boolean) autoScale** Have Imbo automatically figure out the widths of the image variations (based on other parameters). Defaults to `true`.
- (float) scaleFactor** The factor to use when scaling. Defaults to `0.5` which basically generates variants half the size of the previous one.
- (int) minDiff** When the difference of the width in pixels between two image variations fall below this limit, no more variants will be generated. Defaults to `100`.
- (int) minWidth** Do not generate image variations that ends up with a width in pixels below this level. Defaults to `100`.
- (int) maxWidth** Don't start to generate image variations before the width of the variation falls below this limit. Defaults to `1024`.
- (array) widths** An array of widths to use when generating variations. This can be used together with the auto generation, and will ignore the rules related to auto generation. If you often request images with the same dimensions, you can significantly speed up the process by specifying the width here. Defaults to `[]`.
- (array) database** The database adapter to use. This array has two elements:
 - (string) `adapter`: The class name of the adapter. The class must implement the `Imbo\EventListener\ImageVariations\Database\DatabaseInterface` interface.
 - (array) `params`: Parameters for the adapter (optional).
- (array) storage** The storage adapter to use. This array has two elements:
 - (string) `adapter`: The class name of the adapter. The class must implement the `Imbo\EventListener\ImageVariations\Storage\StorageInterface` interface.
 - (array) `params`: Parameters for the adapter (optional).

Examples:

1. Automatically generate image variations

Given the following configuration:

```
return [
    // ...

    'eventListeners' => [
        'imageVariations' => [
            'listener' => 'Imbo\EventListener\ImageVariations',
            'params' => [
                'database' => [
                    'adapter' =>
↪ 'Imbo\EventListener\ImageVariations\Database\MongoDB',
                ],
                'storage' => [
                    'adapter' =>
↪ 'Imbo\EventListener\ImageVariations\Storage\GridFS',
                ],
            ],
        ],
    ],
],
```

```

    ],
    // ...
];

```

when adding an image with dimensions 3082 x 2259, the following variations will be generated:

- 770 x 564
- 385 x 282
- 192 x 140

When later requesting this image with for instance `?t[]=resize:width=500` as transformation (read more about image transformations in the *Transforming images on the fly* chapter), Imbo will choose the image which is 770 x 564 pixels and downscale it to 500 pixels in width.

2. Specify image widths:

Given the following configuration:

```

return [
    // ...

    'eventListeners' => [
        'imageVariations' => [
            'listener' => 'Imbo\EventListener\ImageVariations',
            'params' => [
                'database' => [
                    'adapter' =>
↳ 'Imbo\EventListener\ImageVariations\Database\MongoDB',
                ],
                'storage' => [
                    'adapter' =>
↳ 'Imbo\EventListener\ImageVariations\Storage\GridFS',
                ],
                'autoScale' => false,
                'widths' => [1000, 500, 200, 100, 50],
            ],
        ],
    ],
    // ...
];

```

when adding an image with dimensions 3082 x 2259, the following variations will be generated:

- 1000 x 732
- 500 x 366
- 200 x 146
- 100 x 73
- 50 x 36

As you can see the `minDiff` and `minWidth` parameters are ignored when using the `width` parameter.

3. Configuring database and storage adapters:

As stated earlier, there are several different adapters to choose from when storing the variations metadata as well as the actual variation image files.

The default adapter for the metadata database is MongoDB, and the default storage adapter is GridFS. They have the same configuration parameters:

```
return [
    // ...

    'eventListeners' => [
        'imageVariations' => [
            'listener' => 'Imbo\EventListener\ImageVariations',
            'params' => [
                'database' => [
                    'adapter' =>
↳ 'Imbo\EventListener\ImageVariations\Database\MongoDB',
                    'params' => [
                        'databaseName' => 'imbo',
                        'server' => 'mongodb://localhost:27017',
                        'options' => ['connect' => true, 'connectTimeoutMS' =>
↳ 1000],
                    ],
                ],
            ],
        ],
        'storage' => [
            'adapter' =>
↳ 'Imbo\EventListener\ImageVariations\Storage\GridFS',
            'params' => [
                'databaseName' => 'imbo_storage',
                'server' => 'mongodb://localhost:27017',
                'options' => ['connect' => true, 'connectTimeoutMS' =>
↳ 1000],
            ],
        ],
    ],
],
// ...
];
```

The Doctrine adapter is an alternative for storing both metadata and variation data. This adapter uses the [Doctrine Database Abstraction Layer](#). When using this adapter you need to create the required tables in the RDBMS first, as specified in the *Database setup* section. Note that you can either pass a PDO instance (as the `pdo` parameter) or specify connection details. Example usage:

```
return [
    // ...

    'eventListeners' => [
        'imageVariations' => [
            'listener' => 'Imbo\EventListener\ImageVariations',
            'params' => [
                'database' => [
                    'adapter' =>
↳ 'Imbo\EventListener\ImageVariations\Database\Doctrine',
                    'params' => [
                        'dbname' => 'imbo',
                        'user' => 'imbo_rw',
                        'password' => 'imbo_password',
                        'host' => 'localhost',
                        'driver' => 'mysql',
                    ],
                ],
            ],
        ],
    ],
];
```

```

        'tableName' => 'imagevariations',

        // OR, pass a PDO instance
        'pdo'       => null,
    ],
    'storage' => [
        'adapter' =>
↪ 'Imbo\EventListener\ImageVariations\Storage\Doctrine',
        'params' => [] // Same as above
    ],
],
],
// ...
];

```

The third option for the storage adapter is the Filesystem adapter. It's fairly straightforward and uses a similar algorithm when generating file names as the *Filesystem* storage adapter. Example usage:

```

return [
    // ...

    'eventListeners' => [
        'imageVariations' => [
            'listener' => 'Imbo\EventListener\ImageVariations',
            'params' => [
                'storage' => [
                    'adapter' =>
↪ 'Imbo\EventListener\ImageVariations\Storage\Filesystem',
                    'params' => [
                        'dataDir' => '/path/to/image-variation-storage'
                    ]
                ],
            ],

            // Use any database adapter you want
            'database' => [
                'adapter' =>
↪ 'Imbo\EventListener\ImageVariations\Database\Doctrine',
            ],
        ],
    ],
    // ...
];

```

Note: When using the Filesystem adapter, the `dataDir` **must** be specified, and **must** be writable by the web server.

Imagick

This event listener is required by the image transformations that is included in Imbo, and there is no configuration options for it. Unless you plan on exchanging all the internal image transformations with your own (for instance implemented using Gmagick or GD) you are better off leaving this as-is.

Max image size

This event listener can be used to enforce a maximum size (height and width, not byte size) of **new** images. Enabling this event listener will not change images already added to Imbo.

The event listener subscribes to the following event:

- `images.post`

and the parameters includes the following elements:

width The max width in pixels of new images. If a new image exceeds this limit it will be downsized.

height The max height in pixels of new images. If a new image exceeds this limit it will be downsized.

and is enabled like this:

```
<?php
return [
    // ...

    'eventListeners' => [
        'maxImageSizeListener' => [
            'listener' => 'Imbo\EventListener\MaxImageSize',
            'params' => [
                'width' => 1024,
                'height' => 768,
            ],
        ],
    ],
];
```

which would effectively downsize all images exceeding a width of 1024 or a height of 768. The aspect ratio will be kept.

Metadata cache

This event listener enables caching of metadata fetched from the backend so other requests won't need to go all the way to the metadata backend to fetch it. To achieve this the listener subscribes to the following events:

- `db.metadata.load`
- `db.metadata.delete`
- `db.metadata.update`
- `db.image.delete`

and the parameters supports a single element:

cache An instance of a cache adapter. Imbo ships with *APC* and *Memcached* adapters, and both can be used for this event listener. If you want to use another form of caching you can simply implement the `Imbo\Cache\CacheInterface` interface and pass an instance of the custom adapter to the constructor of the event listener. See the *Implement a custom cache adapter* section for more information regarding this. Here is an example that uses the APC adapter for caching:

```
<?php
return [
    // ...

    'eventListeners' => [
        'metadataCache' => [
            'listener' => 'Imbo\EventListener\MetadataCache',
            'params' => [
                'cache' => new Imbo\Cache\APC('imbo'),
            ],
        ],
    ],
],
// ...
];
```

Stats access

This event listener controls the access to the *stats resource* by using white listing of IPv4 and/or IPv6 addresses. CIDR-notations are also supported.

This listener is enabled per default, and only allows 127.0.0.1 and ::1 to access the statistics:

```
<?php
return [
    // ...

    'eventListeners' => [
        'statsAccess' => [
            'listener' => 'Imbo\EventListener\StatsAccess',
            'params' => [
                'allow' => ['127.0.0.1', '::1'],
            ],
        ],
    ],
],
// ...
];
```

The event listener also supports a notation for “allowing all”, simply by placing '*' somewhere in the list:

```
<?php
return [
    // ...

    'eventListeners' => [
        'statsAccess' => [
            'listener' => 'Imbo\EventListener\StatsAccess',
            'params' => [
                [
                    'allow' => ['*'],
                ],
            ],
        ],
    ],
],
// ...
];
```

```

        ],
    ],
    ],
    // ...
];

```

The above example will allow all clients access to the statistics.

Varnish HashTwo

This event listener can be enabled if you want Imbo to include [HashTwo headers](#) in responses to image requests. These headers can be used by [Varnish](#) for more effective cache invalidation strategies. The listener, when enabled, subscribes to the following events:

- `image.get`
- `image.head`

The parameters supports a single element:

headerName Set the header name to use. Defaults to X-HashTwo.

```

<?php
return [
    // ...

    'eventListeners' => [
        'hashTwo' => 'Imbo\EventListener\VarnishHashTwo',
    ],

    // ...
];

```

or, if you want to use a non-default header name:

```

<?php
return [
    // ...

    'eventListeners' => [
        'hashTwo' => [
            'listener' => 'Imbo\EventListener\VarnishHashTwo',
            'params' => [
                'headerName' => 'X-Custom-HashTwo-Header-Name',
            ],
        ],
    ],

    // ...
];

```

The header appears multiple times in the response, with slightly different values:

```

X-HashTwo: imbo;image;<publicKey>;<imageIdentifier>
X-HashTwo: imbo;user;<publicKey>

```

Command line tool

Imbo ships with a command line tool which can aid you when setting up Imbo and keeping it up to date. This chapter lists all available commands and what they are capable of.

The binary can be found in one of two places, depending on the *Installation* method you chose. If you installed Imbo via Composer, the binary can be found in `vendor/bin/imbo`, and if you used git clone as the installation method it can be found in `bin/imbo`.

- *Add a public key - add-public-key*
- *Generate a private key - generate-private-key*
- *Help - help*
- *List commands - list*

Add a public key - add-public-key

When using a mutable access control adapter (usually meaning it's backed by a database or similar), this command helps you with adding public/private key pairs and associated rules. It's an alternative to using Imbo's public REST API for this purpose, and is the only way to add an initial public key with access to create and modify other public keys.

Example:

```
./bin/imbo add-public-key somePublicKey
```

The above command will start an interactive session that will guide you through creating a public key with the name `somePublicKey`, given it does not already exist.

Generate a private key - generate-private-key

The script that was earlier called `scripts/generatePrivateKey.php` is now included in the CLI tool. This command does not support any arguments.

Example:

```
./bin/imbo generate-private-key
```

The above command will simply output a secret key that can be used as a private key for an Imbo user.

Help - help

Use this command to get a detailed description of another command along with available arguments and their effect on the command.

Example:

```
./bin/imbo help generate-private-key
```

The above command will provide a description of the *generate-private-key* command.

List commands - `list`

This command can be used to simply list all commands along with their short description. This is the default command that is executed when running `./bin/imbo` with no arguments.

Imbo's API

In this chapter you will learn more about how Imbo's API works, and how you as a user are able to read from and write to Imbo. Most examples listed in this chapter will use `cURL`, so while playing around with the API it's encouraged to have `cURL` easily available. For the sake of simplicity the access tokens and authentication information is not used in the examples. See the *Access tokens* and *Signing write requests* sections for more information regarding this.

Resources/endpoints

In this section you will find information on the different resources Imbo's RESTful API expose, along with their capabilities:

Available resources

- *Index resource* - `/`
- *Stats resource* - `/stats`
- *Status resource* - `/status`
- *Global short URL resource* - `/s/<id>`
- *User resource* - `/users/<user>`
- *Images resource* - `/users/<user>/images`
- *Image resource* - `/users/<user>/images/<image>`
- *Short URLs resource* - `/users/<user>/images/<image>/shorturls`
- *Short URL resource* - `/users/<user>/images/<image>/shorturls/<id>`
- *Metadata resource* - `/users/<user>/images/<image>/metadata`

- *Global images resource* - `/images`
- *Public key resource* - `/keys/<publicKey>`
- *Groups resource* - `/groups`
- *Group resource* - `/groups/<groupName>`
- *Access rules resource* - `/keys/<publicKey>/access`
- *Access rule resource* - `/keys/<publicKey>/access/<ruleId>`

Index resource - /

The index resource shows the version of the Imbo installation along with some external URLs for Imbo-related information, and some internal URLs for the available endpoints.

```
curl -H"Accept: application/json" http://imbo
```

results in:

```
{
  "version": "dev",
  "urls": {
    "site": "http://www.imbo-project.org",
    "source": "https://github.com/imbo/imbo",
    "issues": "https://github.com/imbo/imbo/issues",
    "docs": "http://docs.imbo-project.org"
  },
  "endpoints": {
    "status": "http://imbo/status",
    "stats": "http://imbo/stats",
    "user": "http://imbo/users/{user}",
    "images": "http://imbo/users/{user}/images",
    "image": "http://imbo/users/{user}/images/{imageIdentifier}",
    "globalShortImageUrl": "http://imbo/s/{id}",
    "metadata": "http://imbo/users/{user}/images/{imageIdentifier}/metadata",
    "shortImageUrls": "http://imbo/users/{user}/images/{imageIdentifier}/shorturls",
    "shortImageUrl": "http://imbo/users/{user}/images/{imageIdentifier}/shorturls/{id}"
  }
}
```

This resource does not support any extensions in the URI, so you will need to use the `Accept` header to fetch different representations of the data.

The index resource does not require any authentication per default.

Typical response codes:

- 200 Hell Yeah

Note: The index resource is not cache-able.

Stats resource - /stats

Imbo provides an endpoint for fetching simple statistics about the data stored in Imbo.

```
curl http://imbo/stats.json
```

results in:

```
{
  "numImages": 12,
  "numUsers": 2,
  "numBytes": 3898294
  "custom": {}
}
```

if the client making the request is allowed access.

Access control

The access control for the stats endpoint is controlled by an event listener, which is enabled per default, and only allows connections from 127.0.0.1 (IPv4) and ::1 (IPv6). Read more about how to configure this event listener in the *Stats access event listener* section.

Custom statistics

The stats resource enables users to attach custom statistics via event listeners by using the data model as a regular associative array. The following example attaches a simple event listener in the configuration file that populates some custom data in the statistics model:

```
<?php
return [
    // ...

    'eventListeners' => [
        'customStats' => [
            'events' => ['stats.get'],
            'callback' => function($event) {
                // Fetch the model from the response
                $model = $event->getResponse()->getModel();

                // Set some values
                $model['someValue'] = 123;
                $model['someOtherValue'] = [
                    'foo' => 'bar',
                ];
                $model['someList'] = [1, 2, 3];
            }
        ],
    ],
    // ...
];
```

When requesting the stats endpoint, the output will look like this:

```
{
  "users": {
    "someuser": {
      "numImages": 11,
      "numBytes": 3817197
    },
    "someotheruser": {
      "numImages": 1,
      "numBytes": 81097
    }
  },
  "total": {
    "numImages": 12,
    "numUsers": 2,
    "numBytes": 3898294
  },
  "custom": {
    "someValue": 123,
    "someOtherValue": {
      "foo": "bar"
    },
    "someList": [1, 2, 3]
  }
}
```

Use cases for this might be to simply store data in some backend in various events (for instance `image.get` or `metadata.get`) and then fetch these and display them when requesting the stats endpoint (`stats.get`).

Note: The stats resource is not cache-able.

Status resource - `/status`

Imbo includes a simple status resource that can be used with for instance monitoring software.

```
curl http://imbo/status.json
```

results in:

```
{
  "timestamp": "Tue, 24 Apr 2012 14:12:58 GMT",
  "database": true,
  "storage": true
}
```

where `timestamp` is the current timestamp on the server, and `database` and `storage` are boolean values informing of the status of the current database and storage adapters respectively. If both are `true` the HTTP status code is 200 OK, and if one or both are `false` the status code is 503. When the status code is 503 the reason phrase will inform you whether it's the database or the storage adapter (or both) that is having issues. As soon as the status code does not equal 200 Imbo will no longer work as expected.

The reason for adapter failures depends on what kind of adapter you are using. The *file system storage adapter* will for instance return a failure if it can no longer write to the storage directory. The *MongoDB* and *Doctrine* database adapters will fail if they can no longer connect to the server they are configured to communicate with.

Typical response codes:

- 200 OK
- 503 Database error
- 503 Storage error
- 503 Storage and database error

Note: The status resource is not cache-able.

Global short URL resource - /s/<id>

Images in Imbo can have short URLs associated with them, which are generated on demand when interacting with the *short URLs resource*. These URLs can be used in place of the rather long original URLs which includes both access tokens and transformations.

The format of the random ID part of the short URL can be matched with the following [regular expression](#):

```
/^[a-zA-Z0-9]{7}$/
```

There are some caveats regarding the short URLs:

1. If the data used to generate the short URL contained an image extension, content negotiation will not be applied to the short URL. You will always get the mime type associated with the extension used to generate the short URL.
2. If the data used to generate the short URL did not contain an image extension you can use the `Accept` header to decide the mime type of the generated image when requesting the short URL.
3. Short URLs do not support extensions, so you can not append `.jpg` to force `image/jpeg`. If you need to make sure the image is always a JPEG, simply add `jpg` as an extension when generating the short URL.

You can read more about how to generate these URLs in the *short URLs section*.

Note: In Imbo only images have short URLs

User resource - /users/<user>

The user resource represents a single user on the current Imbo installation. The output contains basic user information:

```
curl http://imbo/users/<user>.json
```

results in:

```
{
  "user": "<user>",
  "numImages": 42,
  "lastModified": "Wed, 18 Apr 2012 15:12:52 GMT"
}
```

where `user` is the user (the same used in the URI of the request), `numImages` is the number of images the user has stored in Imbo and `lastModified` is when the user last uploaded or deleted an image, or when the user last updated metadata of an image. If the user has not added any images yet, the `lastModified` value will be set to the current time on the server.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 User not found

Images resource - `/users/<user>/images`

The images resource is the collection of images owned by a specific user. This resource can be used to search added images, and is also used to add new images to a collection.

Add an image

To be able to display images stored in Imbo you will first need to add one or more images. This is done by requesting this endpoint with an image attached to the request body, and changing the HTTP METHOD to `POST`. The body of the response for such a request contains a JSON object containing the image identifier of the added image:

```
curl -XPOST http://imbo/users/<user>/images --data-binary @<file to add>
```

results in:

```
{
  "imageIdentifier": "<imageIdentifier>",
  "width": <width>,
  "height": <height>,
  "extension": "<extension>"
}
```

The `<imageIdentifier>` in the response is the identifier of the added image. This is used with the *image resource*<>. The response body also contains the width, height and extension of the image that was just added.

Typical response codes:

- 200 OK
- 201 Created
- 400 Bad request

Get image collections

The images resource can also be used to gather information on which images a user owns. This is done by requesting this resource using HTTP `GET`. Supported query parameters are:

page The page number. Defaults to 1.

limit Number of images per page. Defaults to 20.

metadata Whether or not to include metadata in the output. Defaults to 0, set to 1 to enable.

from Fetch images starting from this [Unix timestamp](#).

to Fetch images up until this timestamp.

fields[] An array with fields to display. When not specified all fields will be displayed.

sort[] An array with fields to sort by. The direction of the sort is specified by appending `asc` or `desc` to the field, delimited by `:`. If no direction is specified `asc` will be used. Example: `?sort[]=size&sort[]=width:desc` is the same as `?sort[]=size:asc&sort[]=width:desc`. If no `sort` is specified Imbo will sort by the date the images was added, in a descending fashion.

ids[] An array of image identifiers to filter the results by.

checksums[] An array of image checksums to filter the results by.

originalChecksums[] An array of the original image checksums to filter the results by.

```
curl "http://imbo/users/<user>/images.json?limit=1&metadata=1"
```

results in:

```
{
  "search": {
    "hits": 3,
    "page": 1,
    "limit": 1,
    "count": 1
  },
  "images": [
    {
      "added": "Mon, 10 Dec 2012 11:57:51 GMT",
      "updated": "Mon, 10 Dec 2012 11:57:51 GMT",
      "checksum": "<checksum>",
      "originalChecksum": "<originalChecksum>",
      "extension": "png",
      "size": 6791,
      "width": 1306,
      "height": 77,
      "mime": "image/png",
      "imageIdentifier": "<image>",
      "user": "<user>",
      "metadata": {
        "key": "value",
        "foo": "bar"
      }
    }
  ]
}
```

The search object is data related to pagination, where `hits` is the number of images found by the query, `page` is the current page, `limit` is the current limit, and `count` is the number of images in the visible collection.

The images list contains image objects. Each object has the following fields:

- `added`: A formatted date of when the image was added to Imbo.
- `updated`: The formatted date of when the image was last updated (read: when metadata attached to the image was last updated, as the image itself never changes).
- `checksum`: The MD5 checksum of the image blob stored in Imbo.
- `originalChecksum`: The MD5 checksum of the original image. Might differ from `<checksum>` if event listeners that might change incoming images have been enabled. This field was added to Imbo in version 1.2.0. If this field is `null` when you query the images resource, you will need to manually update the database. If you have event listeners changing incoming images you might not want to simply set the original checksum to `<checksum>` as that might not be true.

- `extension`: The original image extension.
- `size`: The size of the image in bytes.
- `width`: The width of the image in pixels.
- `height`: The height of the image in pixels.
- `mime`: The mime type of the image.
- `imageIdentifier`: The image identifier stored in Imbo.
- `user`: The user who owns the image.
- `metadata`: A JSON object containing metadata attached to the image. This field is only available if the `metadata` query parameter described above is used.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 User not found

Image resource - `/users/<user>/images/<image>`

The image resource represents specific images owned by a user. This resource is used to retrieve and remove images. It's also responsible for transforming the images based on the transformation parameters in the query.

Fetch images

Fetching images added to Imbo is done by requesting the image identifiers of the images.

```
curl http://imbo/users/<user>/images/<image>
```

results in:

```
<binary data of the original image>
```

When fetching images Imbo also sends a set of custom HTTP response headers related to the image:

```
X-Imbo-Originalextension: png
X-Imbo-Originalmimetype: image/png
X-Imbo-Originalfilesize: 45826
X-Imbo-Originalheight: 390
X-Imbo-Originalwidth: 380
```

These are all related to the image that was just requested.

How to use this resource to generate image transformations is described in the *Transforming images on the fly* chapter.

Typical response codes:

- 200 OK
- 304 Not modified
- 400 Bad request
- 404 Image not found

Delete images

Deleting images from Imbo is accomplished by requesting the image URIs using HTTP DELETE. All metadata attached to the image will be removed as well.

```
curl -XDELETE http://imbo/users/<user>/images/<image>
```

results in:

```
{
  "imageIdentifier": "<image>"
}
```

where <image> is the image identifier of the image that was just deleted (the same as the one used in the URI).

Typical response codes:

- 200 OK
- 400 Bad request
- 404 Image not found

Short URLs resource - /users/<user>/images/<image>/shorturls

This resource is used to create short URLs for images on demand, as well as removing all short URLs associated with a single image.

Create a short URL

Creating a short URL is done by requesting this resource using HTTP POST while including some parameters for the short URL in the request body. The parameters must be specified as a JSON object, and the object supports the following fields:

- `imageIdentifier`: The same image identifier as the one in the requested URI.
- `user`: The same user as the one in the requested URI.
- `extension`: An optional extension to the image, for instance `jpg` or `png`.
- `query`: The query string with transformations that will be applied. The format is the same as when requesting the image resource with one or more transformations. See the *Transforming images on the fly* chapter for more information regarding the transformation of images.

The generated ID of the short URL can be found in the response:

```
curl -XPOST http://imbo/users/<user>/images/<image>/shorturls.json -d '{
  "imageIdentifier": "<image>",
  "user": "<user>",
  "extension": "jpg",
  "query": "t[]=thumbnail:width=75,height=75&t[]=desaturate"
}'
```

results in:

```
{
  "id": "<id>"
}
```

where `<id>` can be used with the *global short URL resource* for requesting the image with the configured extension / transformations applied.

Delete all short URLs associated with an image

If you want to remove all short URLs associated with an image, you can request this resource using HTTP DELETE:

```
curl -XDELETE http://imbo/users/<user>/images/<image>/shorturls.json
```

results in:

```
{
  "imageIdentifier": "<image>"
}
```

Short URL resource - `/users/<user>/images/<image>/shorturls/<id>`

This resource can be used to remove a single short URL for a specific image variation.

This is achieved by simply requesting the resource with HTTP DELETE, specifying the ID of the short URL in the URI:

```
curl -XDELETE http://imbo/users/<user>/images/<image>/shorturls/<id>
```

results in:

```
{
  "id": "<id>"
}
```

Metadata resource - `/users/<user>/images/<image>/metadata`

Imbo can also be used to attach metadata to the stored images. The metadata is based on a simple `key => value` model, for instance:

- `category: Music`
- `band: Koldbrann`
- `genre: Black metal`
- `country: Norway`

Values can be nested `key => value` pairs.

Adding/replacing metadata

To add (or replace all existing metadata) on an image a client should make a request against this resource using HTTP PUT with the metadata attached in the request body as a JSON object. The response body will contain the added metadata.


```
curl -XPUT http://imbo/users/<user>/images/<image>/metadata.json -d '{
  "beer": "Dark Horizon First Edition",
  "brewery": "Nøgne Ø",
  "style": "Imperial Stout"
}'
```

results in:

```
{
  "beer": "Dark Horizon First Edition",
  "brewery": "Nøgne Ø",
  "style": "Imperial Stout"
}
```

Note: When using the *Doctrine database adapter*, metadata keys can not contain : : .

Typical response codes:

- 200 OK
- 400 Bad request
- 400 Invalid metadata (when using the *Doctrine* adapter, and keys contain : :)
- 404 Image not found

Partially updating metadata

Partial updates to metadata attached to an image is done by making a request with HTTP POST and attaching metadata to the request body as a JSON object. If the object contains keys that already exists in the metadata on the server the old values will be replaced by the ones found in the request body. New keys will be added to the metadata. The response will contain all metadata attached to the image after the update.

```
curl -XPOST http://imbo/users/<user>/images/<image>/metadata.json -d '{
  "ABV": "16%",
  "score": "100/100"
}'
```

results in:

```
{
  "beer": "Dark Horizon First Edition",
  "brewery": "Nøgne Ø",
  "style": "Imperial Stout",
  "ABV": "16%",
  "score": "100/100"
}
```

if the image already included the first three keys as metadata.

Note: When using the *Doctrine database adapter*, metadata keys can not contain : : .

Typical response codes:

- 200 OK

Imbo, Release 2.2.1

- 400 Bad request
- 400 Invalid metadata (when using the *Doctrine* adapter, and keys contain : :)
- 404 Image not found

Fetch metadata

Requests using HTTP GET on this resource returns all metadata attached to an image.

```
curl http://imbo/users/<user>/images/<image>/metadata.json
```

results in:

```
{}
```

when there is no metadata stored, or for example

```
{
  "category": "Music",
  "band": "Koldbrann",
  "genre": "Black metal",
  "country": "Norway"
}
```

if the image has metadata attached to it.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 Image not found

Remove metadata

To remove metadata attached to an image a request using HTTP DELETE can be made.

```
curl -XDELETE http://imbo/users/<user>/images/<image>/metadata.json
```

results in:

```
{}
```

Typical response codes:

- 200 OK
- 400 Bad request
- 404 Image not found

Global images resource - /images

The global images resource is used to search for images across users, given that the public key has access to the images of these users.

This resource is read only, and behaves in the same way as described in the *Get image collections* section of *Images resource* - `/users/<user>/images`. In addition to the parameters specified for *Get image collections*, the following query parameter must be specified:

users [] An array of users to get images for.

```
curl "http://imbo/images?users[]=foo&users[]=bar"
```

results in a response with the exact same format as shown under *Get image collections*.

Public key resource - `/keys/<publicKey>`

The public key resource provides a way for clients to dynamically add, remove and update public keys to be used as part of the access control routines. Not all access control adapters implement this functionality - in this case the configuration is done through configuration files.

A private key does not have any specific requirements, while a public key must match the following [regular expression](#):

```
/^[a-zA-Z0-9_-]{1,}$/
```

Add a public key

Every public key must also have a private key, which is used to sign and generate access tokens for requests. This is the only required parameter in the request body.

```
curl -XPUT http://imbo/keys/<publicKey>.json -d '{"privateKey":"<privateKey>"}
```

Check if a public key exist

A HEAD request can be used if you want to check if a public key exist. The public key used to sign the request must have access to the `keys.head` resource.

```
curl -XHEAD http://imbo/keys/<publicKey>
```

Typical response codes:

- 200 OK
- 404 Public key not found

Change private key for a public key

Use the same method as when adding a public key to change the private key.

Remove a public key

Public keys can be removed using a DELETE request. The public key used to sign the request must have access to the `keys.delete` resource.

```
curl -XDELETE http://imbo/keys/<publicKey>.json
```

Typical response codes:

- 200 OK
- 201 Created
- 400 Bad request
- 404 Public key not found
- 405 Access control adapter is immutable

Note: The keys resource is not cache-able.

Groups resource - /groups

The groups resource can list available resource groups, used in the access control routines.

List resource groups

Requests using HTTP GET on this resource returns all available resource groups. Supported query parameters are:

page The page number. Defaults to 1.

limit Number of groups per page. Defaults to 20.

```
curl http://imbo/groups.json
```

results in:

```
{"search":{"hits":0,"page":1,"limit":20,"count":0},"groups":[]}
```

when there are no resource groups defined, or for example

```
{
  "search": {
    "hits": 1,
    "page": 1,
    "limit": 20,
    "count": 1
  },
  "groups": [
    {
      "name": "read-stats",
      "resources": [
        "user.get",
        "user.head",
        "user.options"
      ]
    }
  ]
}
```

if there are resource groups defined.

Typical response codes:

- 200 OK

Group resource - /groups/<groupName>

The group resource enables adding, modifying and deleting resource groups used in the access control routine. Not all access control adapters allow modification of groups - in this case the configuration is done through configuration files, and PUT/DELETE operations will result in an HTTP 405 response.

List resources of a group

Requests using HTTP GET on this resource returns all the resources the group consists of.

```
curl http://imbo/groups/<group>.json
```

results in:

```
{
  "resources": [
    "user.get",
    "user.head",
    "user.options"
  ]
}
```

Add/modify resources for a group

Requests using HTTP PUT on this resource either adds a new group with the given name, or if it already exists, updates it. The request body should contain an array of resources the group should consist of.

```
curl -XPUT http://imbo/groups/<group>.json -d '[
  "user.get",
  "stats.get"
]'
```

Delete a resource group

Requests using HTTP DELETE on this resource will remove the entire resource group. Note that any access control rules that are using this resource group will also be deleted, since they are now invalid.

```
curl -XDELETE http://imbo/groups/<group>.json
```

Typical response codes:

- 200 OK
- 201 Created
- 404 Group not found
- 405 Access control adapter is immutable

Access rules resource - /keys/<publicKey>/access

The access rules endpoint allows you to add rules that give a public key access to a specified set of resources. These rules can also be defined on a per-user basis. Instead of defining a list of resources, you also have the option to specify a *resource group*.

Listing access control rules

Requests using HTTP GET on this resource returns all the access control rules defined for the given public key.

```
curl http://imbo/keys/<publicKey>/access.json
```

results in:

```
[
  {
    "id": 1,
    "resources": ['images.get', 'image.get', 'images.post', 'image.delete'],
    "users": [
      "user1",
      "user2"
    ]
  },
  {
    "id": 2,
    "group": "read-stats",
    "users": [
      "user1",
      "user2"
    ]
  }
]
```

Adding access control rules

Requests using HTTP POST on this resource adds new rules to the given public key. The request body should contain an array of rules. The parameters for a rule must be specified as JSON objects, where the object supports the following fields:

- **users:** Defines on which users the public key should have access to the defined resources. Either an array of users or the string * (all users).
- **resources:** An array of resources you want the public key to have access to.
- **group:** A resource group the public key should have access to.

Note: A rule must contain *either* resources or group, not both. users is required.

```
curl -XPOST http://imbo/keys/<publicKey>/access -d '{
  "resources": ["user.get", "image.get", "images.get", "metadata.get"],
  "users": "*"
}'
```

Typical response codes:

- 200 OK
- 400 No access rule data provided
- 404 Public key not found
- 405 Access control adapter is immutable

Access rule resource - /keys/<publicKey>/access/<ruleId>

The access rule endpoint allows you to see which resources and users a given access control rule contains. It also allows you to remove a specific access control rule.

Get access rule details

```
curl http://imbo/keys/<publicKey>/access/<ruleId>.json
```

results in:

```
{
  "id": 1,
  "resources": ['images.get', 'image.get', 'images.post', 'image.delete'],
  "users": [
    "user1",
    "user2"
  ]
}
```

Removing an access rule

Requests using HTTP DELETE on this resource removes the access control rule, given the access control adapter supports mutations.

```
curl -XDELETE http://imbo/keys/<publicKey>/access/<ruleId>
```

Typical response codes:

- 200 OK
- 404 Public key not found
- 404 Access rule not found
- 405 Access control adapter is immutable

Access tokens

Access tokens are enforced by an event listener that is enabled in the default configuration file. The access tokens are used to prevent DoS attacks so think twice before you disable the event listener.

An access token, when enforced by the event listener, must be supplied in the URI using the `accessToken` query parameter and without it, most GET and HEAD requests will result in a 400 Bad request response. The value of the `accessToken` parameter is a [Hash-based Message Authentication Code \(HMAC\)](#). The code is a [SHA-256](#) hash of the URI itself using the private key of the user as the secret key. It is very important that the URI is **not** URL-encoded when generating the hash. Below is an example on how to generate a valid access token for a specific image using PHP:

```
1 <?php
2 $user      = "<user>";           // The user id
3 $publicKey = "<public key>";      // The public key of the user
4 $privateKey = "<secret value>";    // The private key of the user
5 $image     = "<image>";          // The image identifier
```

```

6
7 // Image transformations
8 $query = [
9     "t[]=thumbnail:width=40,height=40,fit=outbound",
10    "t[]=border:width=3,height=3,color=000",
11    "t[]=canvas:width=100,height=100,mode=center"
12 ];
13
14 // Add a query parameter for public key if it differs from the user
15 if ($user != $publicKey) {
16     $query[] = 'publicKey=' . $publicKey;
17 }
18
19 // The URI
20 $uri = sprintf("http://imbo/users/%s/images/%s?%s", $user, $image, implode('&',
    ↪$query));
21
22 // Generate the token
23 $accessToken = hash_hmac("sha256", $uri, $privateKey);
24
25 // Output the URI with the access token
26 echo sprintf("%s&accessToken=%s", $uri, $accessToken) . PHP_EOL;

```

and Python:

```

1 from hashlib import sha256
2 import hmac
3
4 user      = "<user>"          # The user id
5 publicKey = "<public key>"    # The public key of the user
6 privateKey = "<secret value>" # The private key of the user
7 image     = "<image>"        # The image identifier
8
9 # Image transformations
10 query = [
11     "t[]=thumbnail:width=40,height=40,fit=outbound",
12     "t[]=border:width=3,height=3,color=000",
13     "t[]=canvas:width=100,height=100,mode=center"
14 ]
15
16 # Specify public key if it differs from user
17 if user != publicKey:
18     query.append('publicKey=%s' % (publicKey))
19
20 # The URI
21 uri = "http://imbo/users/%s/images/%s?%s" % (user, image, "&".join(query))
22
23 # Generate the token
24 accessToken = hmac.new(privateKey, uri, sha256)
25
26 # Output the URI with the access token
27 print "%s&accessToken=%s" % (uri, accessToken.hexdigest())

```

and Ruby:

```

1 require "digest"
2
3 user      = "<user>"          # User id

```



```

4  publicKey = "<public key>" # The public key of the user
5  privateKey = "<secret value>" # The private key of the user
6  image = "<image>" # The image identifier
7
8  # Image transformations
9  queryParts = [
10     "t[]=thumbnail:width=40,height=40,fit=outbound",
11     "t[]=border:width=3,height=3,color=000",
12     "t[]=canvas:width=100,height=100,mode=center"
13 ]
14
15 # Specify public key if it differs from user
16 if user != publicKey
17     queryParts.push("publicKey=#{publicKey}")
18 end
19
20 # Join the query parts
21 query = queryParts.join("&")
22
23 # The URI
24 uri = "http://imbo/users/#{user}/images/#{image}?#{query}"
25
26 # Generate the token
27 accessToken = Digest::HMAC.hexdigest(uri, privateKey, Digest::SHA256)
28
29 # Output the URI with the access token
30 puts "#{uri}&accessToken=#{accessToken}"

```

If the event listener enforcing the access token check is removed, Imbo will ignore the `accessToken` query parameter completely. If you wish to implement your own form of access token you can do this by implementing an event listener of your own (see *Writing an event listener* for more information).

Prior to Imbo-1.0.0 there was no rule that the URL had to be completely URL-decoded prior to generating the access token in the clients. Because of this Imbo will also try to re-generate the access token server side by using the URL as-is. This feature has been added to ease the transition to Imbo >= 1.0.0, and will be removed some time in the future.

Signing write requests

To be able to write to Imbo the user agent will have to specify two request headers: `X-Imbo-Authenticate-Signature` and `X-Imbo-Authenticate-Timestamp`.

`X-Imbo-Authenticate-Signature` is, like the access token, an HMAC (also using SHA-256 and the private key of the user).

The data for the hash is generated using the following elements:

- HTTP method (PUT, POST or DELETE)
- The URI
- Public key
- GMT timestamp (YYYY-MM-DDTHH:MM:SSZ, for instance: 2011-02-01T14:33:03Z)

These elements are concatenated in the above order with `|` as a delimiter character, and a hash is generated using the private key of the user. The following snippet shows how this can be accomplished in PHP when deleting an image:

```

1 <?php
2 $user = "<user>"; // User id

```

```

3 $publicKey = "<public key>";           // The public key of the user
4 $privateKey = "<secret value>";       // The private key of the user
5 $timestamp = gmdate("Y-m-d\TH:i:s\Z"); // Current timestamp (UTC)
6 $image     = "<image>";              // The image identifier
7 $method    = "DELETE";               // HTTP method to use
8
9 // The URI
10 $uri = sprintf("http://imbo/users/%s/images/%s", $user, $image);
11
12 // Data for the hash
13 $data = implode("|", [$method, $uri, $publicKey, $timestamp]);
14
15 // Generate the token
16 $signature = hash_hmac("sha256", $data, $privateKey);
17
18 // Request the URI
19 $response = file_get_contents($uri, false, stream_context_create([
20     "http" => [
21         "method" => $method,
22         "header" => [
23             "X-Imbo-PublicKey" : $publicKey,
24             "X-Imbo-Authenticate-Signature" : $signature,
25             "X-Imbo-Authenticate-Timestamp" : $timestamp,
26         ],
27     ],
28 ]));

```

and Python (using the Requests library):

```

1 import hmac, requests
2 from hashlib import sha256
3 from datetime import datetime
4
5 user = "<user>" # The user ID
6 publicKey = "<public key>" # The public key of the
7 privateKey = "<secret value>" # The private key of
8 timestamp = datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ") # Current timestamp
9 image = "<image>" # The image identifier
10 method = "DELETE" # HTTP method to use
11
12 # The URI
13 uri = "http://imbo/users/%s/images/%s" % (user, image)
14
15 # Data for the hash
16 data = "|".join([method, uri, publicKey, timestamp])
17
18 # Generate the token
19 signature = hmac.new(privateKey, data, sha256).hexdigest()
20
21 # Request the URI
22 response = requests.delete(uri, headers = {
23     "X-Imbo-PublicKey": publicKey,
24     "X-Imbo-Authenticate-Signature": signature,
25     "X-Imbo-Authenticate-Timestamp": timestamp
26 })

```

and Ruby (using the `httpclient` gem):

```

1 require "digest"
2 require "httpclient"
3
4 user      = "<user>"           # The user ID
5 publicKey = "<public key>"     # The public key of the user
6 privateKey = "<secret value>"  # The private key of the user
7 timestamp = Time.now.utc.strftime("%Y-%m-%dT%H:%M:%SZ") # Current timestamp (UTC)
8 image     = "<image>"         # The image identifier
9 method    = "DELETE"          # HTTP method to use
10
11 # The URI
12 uri = "http://imbo/users/#{user}/images/#{image}"
13
14 # Data for the hash
15 data = [method, uri, publicKey, timestamp].join("|")
16
17 # Generate the token
18 signature = Digest::HMAC.hexdigest(data, privateKey, Digest::SHA256)
19
20 # Request the URI
21 client = HTTPClient.new
22 response = client.delete(uri, {}, {
23   "X-Imbo-PublicKey" => publicKey,
24   "X-Imbo-Authenticate-Signature" => signature,
25   "X-Imbo-Authenticate-Timestamp" => timestamp
26 })

```

Imbo requires that `X-Imbo-Authenticate-Timestamp` is within ± 120 seconds of the current time on the server.

As with the access token the signature check is enforced by an event listener that can also be disabled. If you disable this event listener you effectively open up for writing from anybody, which you probably don't want to do.

If you want to implement your own authentication paradigm you can do this by creating a custom event listener.

Supported content types

Imbo currently responds with images (jpg, gif and png), **JSON** and **XML**, but only accepts images (jpg, gif and png) and JSON as input.

Imbo will do content negotiation using the `Accept` header found in the request, unless you specify a file extension, in which case Imbo will deliver the type requested without looking at the `Accept` header.

The default content type for non-image responses is JSON. Examples in this chapter uses the `.json` extension. Change it to `.xml` to get the XML representation instead. You can also skip the extension and force a specific content type using the `Accept` header:

```
curl http://imbo/status.json
```

and

```
curl -H "Accept: application/json" http://imbo/status
```

will end up with the same content type. Use `application/xml` for XML.

If you use JSON you can wrap the content in a function (**JSONP**) by using one of the following query parameters:

- `callback`
- `jsonp`
- `json`

```
curl http://imbo/status.json?callback=func
```

will result in:

```
func(  
  {  
    "date": "Mon, 05 Nov 2012 19:18:40 GMT",  
    "database": true,  
    "storage": true  
  }  
)
```

For images the default mime-type is the original mime-type of the image. If you add an `image/gif` image and fetch that image with `Accept: */*` or `Accept: image/*` the mime-type of the image returned will be `image/gif`. To choose a different mime type either change the `Accept` header, or use `.jpg` or `.png` (for `image/jpeg` and `image/png` respectively).

An exception to this is if the configuration option `contentNegotiateImages` is set to `false`, in which case Imbo will not convert the image to a different format than the original, unless explicitly told to do so by specifying an extension (`.jpg`, `.png`, `.gif` etc).

Cache headers

Most responses from Imbo includes a set of cache-related headers that enables shared caches and user agents to cache content.

Cache-Control

Some responses from Imbo are not cache-able. These will typically include `Cache-Control: max-age=0, no-store, private`. The following resources are not cache-able:

- *Index resource - /*
- *Stats resource - /stats*
- *Status resource - /status*

All other resources will include `Cache-Control: public`. The *image* and *short url* resources will also set a `max-age`, resulting in the following header: `Cache-Control: max-age=31536000, public`.

ETag

Imbo provides [entity tags](#) for cache validation mechanisms. User agents can use the `ETag` response header to do conditional requests further down the road (by specifying the original `ETag` value in the `If-None-Match` request header). This results in saved bandwidth as web caches and Imbo servers no longer need to send the response body, as the one cached by the user agent can be re-used. This is achieved by sending `304 Not Modified` back to the user agent, instead of `200 OK`.

The following resources in Imbo will include an `ETag`:

- *User resource - /users/<user>*

- *Images resource* - `/users/<user>/images`
- *Image resource* - `/users/<user>/images/<image>`
- *Metadata resource* - `/users/<user>/images/<image>/metadata`
- *Global short URL resource* - `/s/<id>`

The value of the `ETag` header is simply the MD5 sum of the content in the response body, enclosed in quotes. For instance `ETag: "fd2fd87a2f5288be31c289e70e916123"`.

Last-Modified

Imbo also includes a `Last-Modified` response header for resources that has a know last modification date, and these resources are:

- *User resource* - `/users/<user>`: The date of when the user last added or deleted an image, or manipulated the metadata of an image. If the user don't have any images yet, the value of this date will be the current timestamp.
- *Images resource* - `/users/<user>/images`: The date of when the user last modified an image in the collection (either the image itself, or metadata attached to the image).
- *Image resource* - `/users/<user>/images/<image>`: The date of when the image was added (or replaced), or when the metadata of the image was last modified.
- *Metadata resource* - `/users/<user>/images/<image>/metadata`: The date of when the metadata of the image was last modified.
- *Global short URL resource* - `/s/<id>`: Same as the date of the original image.

User agents can use the value of the `Last-Modified` header in the `If-Modified-Since` request header to make a conditional request. The value of the `Last-Modified` header is an `HTTP-date`, for instance `Last-Modified: Wed, 12 Feb 2014 09:46:02 GMT`.

Errors

When an error occurs Imbo will respond with a fitting HTTP response code along with a JSON object explaining what went wrong.

```
curl -g "http://imbo/users/<user>/foobar"
```

results in:

```
{
  "error": {
    "imboErrorCode": 0,
    "date": "Wed, 12 Dec 2012 21:15:01 GMT",
    "message": "Not Found",
    "code": 404
  }
}
```

The `code` is the HTTP response code, `message` is a human readable error message, `date` is when the error occurred on the server, and `imboErrorCode` is an internal error code that can be used by the user agent to distinguish between similar errors (such as 400 Bad request).

The JSON object will also include `imageIdentifier` if the request was made against the image or the metadata resource.

Transforming images on the fly

What you as an end-user of an Imbo installation will be doing most of the time, is working with images. This is what Imbo was originally made for, and this chapter includes details about all the different image transformations Imbo supports.

All image transformations can be triggered by specifying the `t` query parameter. This parameter must be used as an array so that you can provide several image transformations. The transformations will be applied to the image in the same order as they appear in the URL. Each element in this array represents a single transformation with optional parameters, specified as a string. If the `t` query parameter is not an array or if any of its elements are not strings, Imbo will respond with `HTTP 400`.

Below you will find all image transformations supported “out of the box”, along with their parameters. Some transformations are rarely used with `HTTP GET`, but are instead used by event listeners that transform images when they are added to Imbo (`HTTP POST`). If this is the case it will be mentioned in the description of the transformation.

Auto rotate image based on EXIF data - `t[]=autoRotate`

This transformation will auto rotate the image based on EXIF data stored in the image. This transformation is rarely used per request, but is typically used by the *Auto rotate image* event listener when adding images to Imbo.

Examples:

- `t[]=autoRotate`

Blur the image - `t[]=blur`

This transformation can be used to blur the image.

Parameters:

mode The blur type (optional). Defaults to `gaussian`. Possible values are:

gaussian When adding gaussian blur, the `radius` and `sigma` parameters are required.

adaptive When adding adaptive blur, the `radius` and `sigma` parameters are required. Adaptive blur decrease the blur in the part of the picture near to the edge of the image canvas.

motion When adding motion blur, the `radius`, `sigma` and `angle` parameters are required.

radial When adding radial blur, the `angle` parameter is required.

radius The radius of the Gaussian, in pixels, not counting the center pixel.

sigma The standard deviation of the Gaussian, in pixels.

angle The number of degrees to rotate the image.

Examples:

- `t[]=blur:radius=1,sigma=2`
- `t[]=blur:type=adaptive,radius=2,sigma=4`

Add an image border - `t[]=border`

This transformation will apply a border around the image.

Parameters:

color Color of the border in hexadecimal. Defaults to 000000 (You can also specify short values like f00 (ff0000)).

width Width of the border in pixels on the left and right sides of the image. Defaults to 1.

height Height of the border in pixels on the top and bottom sides of the image. Defaults to 1.

mode Mode of the border. Can be `inline` or `outbound`. Defaults to `outbound`. `Outbound` places the border outside of the image, increasing the dimensions of the image. `inline` paints the border inside of the image, retaining the original width and height of the image.

Examples:

- `t[]=border`
- `t[]=border:mode=inline`
- `t[]=border:color=000`
- `t[]=border:color=f00,width=2,height=2`

Expand the image canvas - `t[]=canvas`

This transformation can be used to change the canvas of the original image.

Parameters:

width Width of the surrounding canvas in pixels. If omitted the width of `<image>` will be used.

height Height of the surrounding canvas in pixels. If omitted the height of `<image>` will be used.

mode The placement mode of the original image. `free`, `center`, `center-x` and `center-y` are available values. Defaults to `free`.

x X coordinate of the placement of the upper left corner of the existing image. Only used for modes: `free` and `center-y`.

y Y coordinate of the placement of the upper left corner of the existing image. Only used for modes: `free` and `center-x`.

bg Background color of the canvas. Defaults to `ffffff` (also supports short values like `f00` (`ff0000`)).

Examples:

- `t[]=canvas:width=200,mode=center`
- `t[]=canvas:width=200,height=200,x=10,y=10,bg=000`
- `t[]=canvas:width=200,height=200,x=10,mode=center-y`
- `t[]=canvas:width=200,height=200,y=10,mode=center-x`

Compress the image - `t[]=compress`

This transformation compresses images on the fly resulting in a smaller payload. It is advisable to only use this transformation in combination with an image type in the URL (for instance `.jpg` or `.png`). This transformation is not applied to images of type `image/gif`.

Parameters:

level The level of the compression applied to the image. The effect this parameter has on the image depends on the type of the image. If the image in the response is an `image/jpeg` a `high level` means high quality, usually resulting in larger files. If the image in the response is an `image/png` a `high level` means high compression,

usually resulting in smaller files. If you do not specify an image type in the URL the result of this transformation is not deterministic as clients have different preferences with regards to the type of images they want to receive (via the `Accept` request header).

Examples:

- `t[]=compress:level=40`

Change image contrast - `t[]=contrast`

This transformation can be used to change the contrast of the colors in the image.

Parameters:

alpha Used to adjust the intensity differences between the lighter and darker elements of the image. Can also be negative. Note: this parameter was named `sharpen` in Imbo 1.x.

beta Where the midpoint of the gradient will be. This value should be in the range 0 to 1. Default: 0.5.

Examples:

- `t[]=contrast:alpha=3`

Note: If you are getting different results than expected when using negative alpha values, your `imagemagick` extension is probably built against an old version of ImageMagick.

Convert the image type - `.jpg/.gif/.png`

This transformation can be used to change the image type. It is not applied like the other transformations, but is triggered when specifying a custom extension to the `<image>`. Currently Imbo can convert to:

- `image/jpeg`
- `image/png`
- `image/gif`

Examples:

- `curl http://imbo/users/<user>/images/<image>.gif`
- `curl http://imbo/users/<user>/images/<image>.jpg`
- `curl http://imbo/users/<user>/images/<image>.png`

Crop the image - `t[]=crop`

This transformation is used to crop the image.

Parameters:

x The X coordinate of the cropped region's top left corner.

y The Y coordinate of the cropped region's top left corner.

width The width of the crop in pixels.

height The height of the crop in pixels.

mode The crop mode (optional). Possible values are:

center When using the center mode the `x` and `y` parameters are ignored, and the center of the cropped area is placed in the center of the original image.

center-x Center the crop on the x-axis. Use the `y` parameter to control the upper edge of the crop.

center-y Center the crop on the y-axis. Use the `x` parameter to control the left edge of the crop.

Examples:

- `t[]=crop:x=10,y=25,width=250,height=150`
- `t[]=crop:width=100,height=100,mode=center`
- `t[]=crop:width=50,height=50,mode=center-x,y=15`
- `t[]=crop:width=50,height=50,mode=center-y,x=15`

Make a gray scaled image - `t[]=desaturate`

This transformation desaturates the image (in practice, gray scales it).

Examples:

- `t[]=desaturate`

Draw points of interest - `t[]=drawPois`

This transformation will draw an outline around all the POIs (points of interest) stored in the metadata for the image. The format of the metadata is documented under the *smartSize* transformation.

Parameters:

color Color of the border in hexadecimal format. Defaults to `ff0000` (You can also specify short values like `f0f` (`ff00ff`)).

borderSize Width of the border in pixels. Defaults to 2.

pointSize The diameter (in pixels) of the circle drawn around points of interest that do not have a height and width specified. Defaults to 30.

Examples:

- `t[]=drawPois`
- `t[]=drawPois:borderSize=10`
- `t[]=drawPois:color=0f0`
- `t[]=drawPois:color=00f,borderSize=10,pointSize=100`

Note: This transformation has a bug/limitation: all coordinates are based on the original image. In other words, applying this at the end of a transformation chain which resizes/crops/rotates the image can lead to unexpected results. This will hopefully change in the future.

Make a mirror image - `t[]=flipHorizontally`

This transformation flips the image horizontally.

Examples:

- `t[]=flipHorizontally`

Flip the image upside down - `t[]=flipVertically`

This transformation flips the image vertically.

Examples:

- `t[]=flipVertically`

Generate a histogram of the image - `t[]=histogram`

This transformation will convert the image into a histogram of the image itself, with a size of 256x158 pixels. The size of the generated image can be overridden by using one or more of the supported parameters.

Parameters:

scale The amount to scale the histogram. Defaults to 1.

ratio The ratio to use when calculating the height of the image. Defaults to 1.618.

red The color to use when drawing the graph for the red channel. Defaults to #D93333.

green The color to use when drawing the graph for the green channel. Defaults to #58C458.

blue The color to use when drawing the graph for the blue channel. Defaults to #3767BF.

Examples:

- `t[]=histogram`
- `t[]=histogram:scale=2`
- `t[]=histogram:red=f00,green=0f0,blue=00f`

Adjust levels of the image - `t[]=level`

This transformation will adjust the levels of an image. You are able to specify individual channels to adjust - by default it will apply to all channels.

Parameters:

channel The channel to adjust. `r` (red), `g` (green), `b` (blue), `c` (cyan), `m` (magenta), `y` (yellow), `k` (black) and `all` (all channels) are available values. These channels can also be combined, if multiple channels should be adjusted. Defaults to `all`.

amount The amount to adjust by. Range is from -100 to 100. Defaults to 1.

Examples:

- `t[]=level`
- `t[]=level:channel=r,amount=30`
- `t[]=level:channel=rg,amount=-45`

Enforce a max size of an image - `t[]=maxSize`

This transformation will resize the image using the original aspect ratio. Two parameters are supported and at least one of them must be supplied to apply the transformation.

Note the difference from the *resize* transformation: given both `width` and `height`, the resulting image will not be the same width and height as specified unless the aspect ratio is the same.

Parameters:

width The max width of the resulting image in pixels. If not specified the width will be calculated using the same aspect ratio as the original image.

height The max height of the resulting image in pixels. If not specified the height will be calculated using the same aspect ratio as the original image.

Examples:

- `t[]=maxSize:width=100`
- `t[]=maxSize:height=100`
- `t[]=maxSize:width=100,height=50`

Modulate the image - `t[]=modulate`

This transformation can be used to control the brightness, saturation and hue of the image.

Parameters:

b Brightness of the image in percent. Defaults to 100.

s Saturation of the image in percent. Defaults to 100.

h Hue percentage. Defaults to 100.

Examples:

- `t[]=modulate:b=150`
- `t[]=modulate:b=120,s=130,h=90`

Make a progressive image - `t[]=progressive`

This transformation makes the image progressive.

Examples:

- `t[]=progressive`

Resize the image - `t[]=resize`

This transformation will resize the image. Two parameters are supported and at least one of them must be supplied to apply the transformation.

Parameters:

width The width of the resulting image in pixels. If not specified the width will be calculated using the same aspect ratio as the original image.

height The height of the resulting image in pixels. If not specified the height will be calculated using the same aspect ratio as the original image.

Examples:

- `t[]=resize:width=100`
- `t[]=resize:height=100`
- `t[]=resize:width=100,height=50`

Rotate the image - `t[]=rotate`

This transformation will rotate the image clock-wise.

Parameters:

angle The number of degrees to rotate the image (clock-wise).

bg Background color in hexadecimal. Defaults to 000000 (also supports short values like f00 (ff0000)).

Examples:

- `t[]=rotate:angle=90`
- `t[]=rotate:angle=45,bg=fff`

Apply a sepia color tone - `t[]=sepia`

This transformation will apply a sepia color tone transformation to the image.

Parameters:

threshold Threshold ranges from 0 to QuantumRange and is a measure of the extent of the sepia toning. Defaults to 80

Examples:

- `t[]=sepia`
- `t[]=sepia:threshold=70`

Sharpen the image - `t[]=sharpen`

This transformation can be used to change the sharpness in the image.

Parameters:

radius The radius of the Gaussian operator in pixels. Defaults to 2.

sigma The standard deviation of the Gaussian, in pixels. Defaults to 1.

gain The percentage of the difference between the original and the blur image that is added back into the original. Defaults to 1.

threshold The threshold in pixels needed to apply the difference gain. Defaults to 0.05.

preset Different presets that can be used. The presets are:

- `light` (radius = 2, sigma = 1, gain = 1, threshold = 0.05)
- `moderate` (radius = 2, sigma = 1, gain = 2, threshold = 0.05)

- `strong` (radius = 2, sigma = 1, gain = 3, threshold = 0.025)
- `extreme` (radius = 2, sigma = 1, gain = 4, threshold = 0)

When using any of the presets the different parameters can be overridden by specifying `radius`, `sigma`, `gain` and/or `threshold`. Not specifying any parameters at all is the same as using the `light` preset.

Examples:

- `t[]=sharpen`
- `t[]=sharpen:preset=light` (same as above)
- `t[]=sharpen:preset=extreme,gain=10` (use the `extreme` preset, but use a gain value of 10 instead of 4)
- `t[]=sharpen:radius=2,sigma=1,gain=1,threshold= 0.05` (same as using `t[]=sharpen:preset=light`, or simply `t[]=sharpen`)

Smart size the image - `t[]=smartSize`

This transformation is used to crop the image based on a point of interest (POI) provided either as a transformation parameter or from the image metadata.

Metadata format

The smart size transformation supports reading the POI from the metadata of the image. The POI information is expected to be stored on the `poi` property in metadata. Below is an example of a valid metadata object containing a 600, 240 POI:

```
{
  "poi": [
    {
      x: 600,
      y: 240
    }
  ]
}
```

Note: The smart size transformation currently takes only the first object into account when cropping the image, but the POIs is stored as an array of objects in order to be easy to expand with more information for a more sophisticated smart size algorithm in the future.

Parameters:

width The width of the crop in pixels.

height The height of the crop in pixels.

poi The POI coordinate `x, y` to crop around. The parameter is optional if the POI exists in metadata.

crop The closeness of the crop (optional). Possible values are:

`close medium wide`

Examples:

- `t[]=smartSize:width=250,height=250,poi=300,200`
- `t[]=smartSize:width=250,height=250,poi=300,200,crop=close`

Strip image properties and comments - `t[]=strip`

This transformation removes all properties and comments from the image. If you want to strip EXIF tags from the image for instance, this transformation will do that for you.

Examples:

- `t[]=strip`

Create a thumbnail of the image - `t[]=thumbnail`

This transformation creates a thumbnail of `<image>`.

Parameters:

width Width of the thumbnail in pixels. Defaults to 50.

height Height of the thumbnail in pixels. Defaults to 50.

fit Fit style. Possible values are: `inset` or `outbound`. Default to `outbound`.

Examples:

- `t[]=thumbnail`
- `t[]=thumbnail:width=20,height=20,fit=inset`

Create a vertical mirror image - `t[]=transpose`

This transformation transposes the image.

Examples:

- `t[]=transpose`

Create a horizontal mirror image - `t[]=transverse`

This transformation transverses the image.

Examples:

- `t[]=transverse`

Add a vignette to the image - `t[]=vignette`

This transformation can be used to add a vignette to the image.

Parameters:

inner Color at the center of the image, in hexadecimal. Defaults to `none`, which means transparent. (You can also specify short values like `f00` (`ff0000`)).

outer Color at the edge of the image, in hexadecimal. Defaults to `000`.

scale Scale factor of the vignette. `2` will create a vignette twice the size of the original image. Defaults to `1.5`.

Examples:

- `t[]=vignette`

- `t[]=vignette:outer=ccc`
- `t[]=vignette:scale=1,outer=333`

Add a watermark to the image - `t[]=watermark`

This transformation can be used to apply a watermark on top of the original image.

Parameters:

img Image identifier of the image to apply as watermark. Can be set to a default value in configuration by using `<setDefaultImage>`.

width Width of the watermark image in pixels. If omitted the width of `` will be used.

height Height of the watermark image in pixels. If omitted the height of `` will be used.

position The placement of the watermark image. `top-left`, `top-right`, `bottom-left`, `bottom-right` and `center` are available values. Defaults to `top-left`.

x Number of pixels in the X-axis the watermark image should be offset from the original position (defined by the `position` parameter). Supports negative numbers. Defaults to 0

y Number of pixels in the Y-axis the watermark image should be offset from the original position (defined by the `position` parameter). Supports negative numbers. Defaults to 0

opacity Can be an integer between 0 and 100 where 0 is fully transparent, and 100 is fully opaque. Defaults to 100

Examples:

- `t[]=watermark:img=f5f7851c40e2b76a01af9482f67bbf3f`
- `t[]=watermark:img=f5f7851c40e2b76a01af9482f67bbf3f,width=200,x=5`
- `t[]=watermark:img=f5f7851c40e2b76a01af9482f67bbf3f,height=50,x=-5,y=-5,position=bottom-right,opacity=50`

If you want to set the default watermark image you will have to do so in the configuration:

```
<?php
return [
    // ...

    'eventListeners' => [
        'watermark' => function() {
            $transformation = new Imbo\Image\Transformation\Watermark();
            $transformation->setDefaultImage('some image identifier');

            return $transformation;
        },
    ],
    // ...
];
```

When you have specified a default watermark image you are not required to use the `img` option for the transformation, but if you do so it will override the default one.

Working with events and event listeners

Imbo uses an event dispatcher to trigger certain events from inside the application that you can subscribe to by using event listeners. In this chapter you can find information regarding the events that are triggered, and how to be able to write your own event listeners for Imbo.

Events

When implementing an event listener you need to know about the events that Imbo triggers. The most important events are combinations of the accessed resource along with the HTTP method used. Imbo currently provides these resources:

- *index*
- *stats*
- *status*
- *user*
- *images*
- *image*
- *globalshorturl*
- *metadata*

Examples of events that are triggered:

- `image.get`
- `images.post`
- `image.delete`
- `metadata.get`

- `status.head`
- `stats.get`

As you can see from the above examples the events are built up by the resource name and the HTTP method, lower-cased and separated by `..`.

Some other notable events:

- `storage.image.insert`
- `storage.image.load`
- `storage.image.delete`
- `db.image.insert`
- `db.image.load`
- `db.image.delete`
- `db.metadata.update`
- `db.metadata.load`
- `db.metadata.delete`
- `response.send`

Image transformations also use the event dispatcher when triggering events. The events triggered for this is prefixed with `image.transformation.` and ends with the transformation as specified in the URL, lowercased. If you specify `t[]=thumbnail&t[]=flipHorizontally` as a query parameter when requesting an image the following events will be triggered:

- `image.transformation.thumbnail`
- `image.transformation.fliphorizontally`

All image transformation events adds the image and parameters for the transformation as arguments to the event, which can be fetched by the transformation via the `$event` object passed to the methods which subscribe to the transformation events.

Writing an event listener

When writing an event listener for Imbo you can choose one of the following approaches:

1. Implement the `Imbo\EventListener\ListenerInterface` interface that comes with Imbo
2. Implement a callable piece of code, for instance a class with an `__invoke` method
3. Use a [Closure](#)

Below you will find examples on the approaches mentioned above.

Note: Information regarding how to **attach** the event listeners to Imbo is available in the *event listener configuration* section.

Implement the `Imbo\EventListener\ListenerInterface` interface

Below is the complete interface with comments:

```

1 <?php
2 /**
3  * This file is part of the Imbo package
4  *
5  * (c) Christer Edvartsen <cogo@starzinger.net>
6  *
7  * For the full copyright and license information, please view the LICENSE file that
8  * was
9  * distributed with this source code.
10 */
11 namespace Imbo\EventListener;
12
13 /**
14  * Event listener interface
15  *
16  * @author Christer Edvartsen <cogo@starzinger.net>
17  * @package Event\Listeners
18  */
19 interface ListenerInterface {
20     /**
21      * Return an array with events to subscribe to
22      *
23      * Single callbacks can use the simplest method, defaulting to a priority of 0
24      *
25      * return [
26      *     'event' => 'someMethod',
27      *     'event2' => 'someOtherMethod',
28      * ];
29      *
30      * If you want to specify multiple callbacks and/or a priority for the
31      * callback(s):
32      *
33      * return [
34      *     'event' => [
35      *         'someMethod', // Defaults to priority 0, same as 'someMethod' => 0
36      *         'someOtherMethod' => 10, // Will trigger before "someMethod"
37      *         'someThirdMethod' => -10, // Will trigger after "someMethod"
38      *     ],
39      *     'event2' => 'someOtherMethod',
40      * ];
41      *
42      * @return array
43      */
44     static function getSubscribedEvents();
45 }

```

The only method you need to implement is called `getSubscribedEvents` and that method should return an array where the keys are event names, and the values are callbacks. You can have several callbacks to the same event, and they can all have specific priorities.

Below is an example of how the *Authenticate* event listener implements the `getSubscribedEvents` method:

```

<?php
// ...

public static function getSubscribedEvents() {

```

```
$callbacks = [];  
$events = [  
    'images.post',  
    'image.delete',  
    'metadata.put',  
    'metadata.post',  
    'metadata.delete'  
];  
  
foreach ($events as $event) {  
    $callbacks[$event] = ['authenticate' => 100];  
}  
  
return $callbacks;  
}  
  
public function authenticate(Imbo\EventManager\EventInterface $event) {  
    // Code that handles all events this listener subscribes to  
}  
  
// ...
```

In the snippet above the same method (`authenticate`) is attached to several events. The priority used is 100, which means it's triggered early in the application flow.

The `authenticate` method, when executed, receives an instance of *the event object* that it can work with. The fact that the above code only uses a single callback for all events is an implementation detail. You can use different callbacks for all events if you want to.

Use a class with an `__invoke` method

You can also keep the listener definition code out of the event listener entirely, and specify that piece of information in the Imbo configuration instead. An invocable class could for instance look like this:

```
<?php  
class SomeEventListener {  
    public function __invoke(Imbo\EventManager\EventInterface $event) {  
        // some custom code  
    }  
}
```

where the `$event` object is the same as the one passed to the `authenticate` method in the previous example.

Use a Closure

For testing and/or debugging purposes you can also write the event listener directly in the configuration, by using a Closure:

```
<?php  
return [  
    // ...  
  
    'eventListeners' => [  
        'customListener' => [  
            'callback' => function(Imbo\EventManager\EventInterface $event) {  
                // Custom code  
            }  
        ]  
    ]  
];
```

```

        },
        'events' => ['image.get'],
    ],
],
// ...
];

```

The `$event` object passed to the function is the same as in the previous two examples. This approach should mostly be used for testing purposes and quick hacks. More information regarding this approach is available in the [event listener configuration](#) section.

Subscribing to events using a wild card

When subscribing to one or more events Imbo let's you use wild cards. This means that you can have your event listener subscribe to all the different image events by specifying `'image.*'` as the event you are subscribing to. If you want a listener to subscribe to **all** events you can simply use `'*'`. Use `$event->getName()` in your handler to figure out which event is actually being triggered.

One other thing to keep in mind while using wild card events is priorities. Global wild card event listeners (that listens to `'*'`) is triggered before all other listeners, and other wild card listeners are triggered before the ones who has absolute event names they are subscribing to.

Example:

```

<?php
return [
    // ...

    'eventListeners' => [
        'listener1' => [
            'callback' => function ($e) { /* ... */ },
            'events' => ['*'],
            'priority' => 100,
        ],
        'listener2' => [
            'callback' => function ($e) { /* ... */ },
            'events' => ['*'],
            'priority' => 200,
        ],
        'listener3' => [
            'callback' => function ($e) { /* ... */ },
            'events' => ['image.*'],
            'priority' => 300,
        ],
        'listener4' => [
            'callback' => function ($e) { /* ... */ },
            'events' => ['image.*'],
            'priority' => 400,
        ],
        'listener5' => [
            'callback' => function ($e) { /* ... */ },
            'events' => ['image.get'],
            'priority' => PHP_INT_MAX,
        ],
    ],
],
];

```

```
    // ...  
};
```

Given the configuration above the execution order would be:

- listener2
- listener1
- listener4
- listener3
- listener5

Even though `listener5` has the highest possible priority the wild card listeners are executed first, because they are in their own priority queues.

The event object

The object passed to the event listeners is an instance of the `Imbo\EventManager\EventInterface` interface. This interface has some methods that event listeners can use:

getName() Get the name of the current event. For instance `image.delete`.

getHandler() Get the name of the current event handler, as specified in the configuration. Can come in handy when you have to dynamically register more callbacks based on constructor parameters for the event listener. Have a look at the implementation of *the CORS event listener* for an example on how to achieve this.

getRequest() Get the current request object (an instance of `Imbo\Http\Request\Request`)

getResponse() Get the current response object (an instance of `Imbo\Http\Response\Response`)

getDatabase() Get the current database adapter (an instance of `Imbo\Database\DatabaseInterface`)

getStorage() Get the current storage adapter (an instance of `Imbo\Storage\StorageInterface`)

getManager() Get the current event manager (an instance of `Imbo\EventManager\EventManager`)

getConfig() Get the complete Imbo configuration. This should be used with caution as it includes all authentication information regarding the Imbo users.

stopPropagation() If you want your event listener to force Imbo to skip all following listeners for the same event, call this method in your listener.

isPropagationStopped() This method is used by Imbo to check if a listener wants the propagation to stop. Your listener will most likely never need to use this method.

getArgument() This method can be used to fetch arguments given to the event. This method is used by all image transformation event listeners as the image itself and the parameters for the transformation is stored as arguments to the event.

With these methods you have access to most parts of Imbo. Be careful when using the database and storage adapters as these grant you access to all data stored in Imbo, with both read and write permissions.

Implement your own database and/or storage adapter

If the adapters shipped with Imbo does not fit your needs you can implement your own set of database and/or storage adapters and have Imbo use them pretty easily. A set of interfaces exists for you to implement, and then all that's left

to do is to enable the adapters in your configuration file. See the *Database configuration* and *Storage configuration* sections for more information on how to enable different adapters in the configuration.

Custom database adapters must implement the `Imbo\Database\DatabaseInterface` interface, and custom storage adapters must implement the `Imbo\Storage\StorageInterface` interface.

If you implement an adapter that you think should be a part of Imbo feel free to send a pull request on [GitHub](#).

Implement your own image transformations

Imbo also supports custom image transformations. All you need to do is to create an event listener, and configure your transformation:

```
<?php
class My\Custom\Transformation implements Imbo\EventListener\ListenerInterface {
    public static function getSubscribedEvents() {
        return ['image.transformation.cooltransformation' => 'transform'];
    }

    public function transform($event) {
        $image = $event->getArgument('image');
        $params = $event->getArgument('params'); // If the transformation allows,
        ↪params in the URL

        // ...
    }
}

return [
    // ..

    'eventListeners' => [
        'coolTransformation' => 'My\Custom\Transformation',
    ],

    // ...
];
```

Whenever someone requests an image using `?t[]=coolTransformation:width=100,height=200` Imbo will trigger the `image.transformation.cooltransformation` event, and assign the following value to the `params` argument of the event:

```
[
    'width' => '100',
    'height' => '200',
]
```

Take a look at the existing transformations included with Imbo for more information.

Cache adapters

If you want to leverage caching in a custom event listener, Imbo ships with some different solutions:

APC

This adapter uses the [APCu](#) extension for caching. If your Imbo installation consists of a single httpd this is a good choice. The adapter has the following parameters:

\$namespace (optional) A namespace for your cached items. For instance: “imbo”

Example:

```
<?php
$adapter = new Imbo\Cache\APC('imbo');
$adapter->set('key', 'value');

echo $adapter->get('key'); // outputs "value"
echo apc_fetch('imbo:key'); // outputs "value"
```

Memcached

This adapter uses [Memcached](#) for caching. If you have multiple httpd instances running Imbo this adapter lets you share the cache between all instances automatically by letting the adapter connect to the same Memcached daemon. The adapter has the following parameters:

\$memcached An instance of the pecl/memcached class.

\$namespace (optional) A namespace for your cached items. For instance: “imbo”.

Example:

```
<?php
$memcached = new Memcached();
$memcached->addServer('hostname', 11211);

$adapter = new Imbo\Cache\Memcached($memcached, 'imbo');
$adapter->set('key', 'value');

echo $adapter->get('key'); // outputs "value"
echo $memcached->get('imbo:key'); // outputs "value"
```

Implement a custom cache adapter

If you want to use some other cache mechanism an interface exists ([Imbo\Cache\CacheInterface](#)) for you to implement:

```
1 <?php
2 /**
3  * This file is part of the Imbo package
4  *
5  * (c) Christer Edvartsen <cogo@starzinger.net>
6  *
7  * For the full copyright and license information, please view the LICENSE file that
8  * ↪ was
9  * distributed with this source code.
10 */
11 namespace Imbo\Cache;
12
```



```

13 /**
14  * Cache adapter interface
15  *
16  * An interface for cache adapters.
17  *
18  * @author Christer Edvartsen <cogo@starzinger.net>
19  * @package Cache
20  */
21 interface CacheInterface {
22     /**
23      * Get a cached value by a key
24      *
25      * @param string $key The key to get
26      * @return mixed Returns the cached value or null if key does not exist
27      */
28     function get($key);
29
30     /**
31      * Store a value in the cache
32      *
33      * @param string $key The key to associate with the item
34      * @param mixed $value The value to store
35      * @param int $expire Number of seconds to keep the item in the cache
36      * @return boolean True on success, false otherwise
37      */
38     function set($key, $value, $expire = 0);
39
40     /**
41      * Delete an item from the cache
42      *
43      * @param string $key The key to remove
44      * @return boolean True on success, false otherwise
45      */
46     function delete($key);
47
48     /**
49      * Increment a value
50      *
51      * @param string $key The key to use
52      * @param int $amount The amount to increment with
53      * @return int|boolean Returns new value on success or false on failure
54      */
55     function increment($key, $amount = 1);
56
57     /**
58      * Decrement a value
59      *
60      * @param string $key The key to use
61      * @param int $amount The amount to decrement with
62      * @return int|boolean Returns new value on success or false on failure
63      */
64     function decrement($key, $amount = 1);
65 }

```

If you choose to implement this interface you can also use your custom cache adapter for all the event listeners Imbo ships with that leverages a cache.

If you implement an adapter that you think should be a part of Imbo feel free to send a pull request on [GitHub](#).

Contributing to Imbo

Imbo is an open source project licensed with the [MIT license](#). All contributions should ideally be sent in form of a pull request on GitHub. Please use feature branches with descriptive names, and remember to send the pull request against the `develop` branch.

If you have found a bug in Imbo, please leave an issue in the [issue tracker](#).

Build script

Imbo uses [Rake](#) for building, and if you have Rake installed you can simply run the `rake` command after cloning Imbo to run the complete build. You might need to install some additional tools for the whole build to complete successfully. If you need help getting the build script to work with no errors drop by the `#imbo` channel on IRC (Freenode) or simply add an issue in the issue tracker on GitHub.

Running the complete suite is not necessary for all contributions. If you skip the build script and simply want to get Imbo up and running for contributing you can run the following commands in the directory where you cloned Imbo:

```
curl -s https://getcomposer.org/installer | php
php composer.phar install
```

Remember to **not** include the `--no-dev` argument to composer. If you include that argument the development requirements will not be installed.

Requirements

When contributing to Imbo (or any of the other related packages) there are some guidelines you should follow.

Coding standard

Imbo has a coding standard that is partially defined as a [PHP Code Sniffer](#) standard. The standard is [available on GitHub](#) and is installable via [PEAR](#). There are some details that might not be covered by the standard, so if you send a PR you might notice some nitpicking from my part regarding stuff not covered by the standard. Browse existing code to understand the general look and feel.

Tests

When introducing new features you are required to add tests. Unit/integration tests ([PHPUnit](#)) and/or [Behat](#) scenarios is sufficient. To run the PHPUnit test suite you can execute the following command in the project root directory after installing Imbo:

```
./vendor/bin/phpunit -c tests/phpunit
```

If you want to generate code coverage as well you can run the test suite by using a Rake task:

```
rake phpunit
```

For the Behat test suite you can run similar commands:

```
./vendor/bin/behat --strict --profile no-cc --config tests/behat/behat.yml
```

to skip code coverage, or

```
rake behat
```

for code coverage of the Behat tests. If you want to run both suites and collect code coverage you can execute:

```
rake test
```

Code coverage is located in `build/coverage` and `build/behat-coverage` respectively.

If you find a bug that you want to fix please add a test first that confirms the bug, and then fix the bug, making the newly added test pass.

Documentation

API documentation is written using [phpDocumentor](#), and can be generated via a Rake task:

```
rake apidocs
```

End user documentation (the ones you are reading now) is written using [Sphinx](#) and is located in the `docs/` directory in the project root. To generate the HTML version of the docs you can execute the following command:

```
rake readthedocs
```

This task also includes a spell checking stage.

Pull requests on GitHub

If you want to send a pull request, please do so from a publicly available fork of Imbo, using a feature branch with a self descriptive name. The pull request should be sent to the `develop` branch. If your pull request is fixing an open issue from [the issue tracker](#) your branch can be named after the issue number, for instance `issue-312`.