Imbo

Release 1.0.2

February 14, 2014

Contents

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.

CHAPTER 1

Installation guide

1.1 Requirements

Imbo requires a web server (for instance Apache, Nginx or Lighttpd) running PHP >= 5.4 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.

1.2 Installation

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

1.2.1 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`:

```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.

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. 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, so you should be able to use Composer's Next Significant Release feature when specifying which Imbo version you want to install. Reading the ChangeLog and other related sources of information before upgrading an installation is always recommended.

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
```

1.2.2 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.

1.2.3 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`.

1.2.4 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 you choice in the *Configuration* topic.

MySQL

```
CREATE TABLE IF NOT EXISTS `imageinfo` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `publicKey` varchar(255) COLLATE utf8_danish_ci NOT NULL,
    `imageIdentifier` char(32) 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,
    PRIMARY KEY (`id`),
    UNIQUE KEY `image` (`publicKey`,`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,
    `publicKey` varchar(255) COLLATE utf8_danish_ci NOT NULL,
    `imageIdentifier` char(32) 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` (`publicKey`,`imageIdentifier`,`extension`,`query`(255))
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_danish_ci;

CREATE TABLE IF NOT EXISTS `storage_images` (
    `publicKey` varchar(255) COLLATE utf8_danish_ci NOT NULL,
    `imageIdentifier` char(32) COLLATE utf8_danish_ci NOT NULL,
    `data` blob NOT NULL,
    `updated` int(10) unsigned NOT NULL,
    PRIMARY KEY (`publicKey`,`imageIdentifier`)
) 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,
    publicKey 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,
    UNIQUE (publicKey,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,
    publicKey TEXT NOT NULL,
    imageIdentifier TEXT NOT NULL,
    extension TEXT,
    query TEXT NOT NULL
);
```

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

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

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

1.3 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.

1.3.1 Imbo users - `auth`

Every user that wants to store images in Imbo needs a public and private key pair. These keys are stored in the `auth` part of your configuration file:

```
<?php
return array(
    // ...

    'auth' => array(
        'username'  => '95f02d701b8dc19ee7d3710c477fd5f4633cec32087f562264e4975659029af7',
        'otheruser' => 'b312ff29d5da23dcd230b61ff4db1e2515c862b9fb0bb59e7dd54ce1e4e94a53',
    ),

    // ...
);
```

The public keys can consist of the following characters:

- a-z (only lowercase is allowed)
- 0-9
- _ and -

and must be at least 3 characters long.

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. 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 script is located in the `scripts` directory of the Imbo installation and does not require any arguments:

```
$ php scripts/generatePrivateKey.php
3b98dde5f67989a878b8b268d82f81f0858d4f1954597cc713ae161cdffcc84a
```

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 public key can not be changed easily as database and storage adapters use it when storing/fetching images and metadata.

1.3.2 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 array(
    // ...

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

    // or

    'database' => new Imbo\Database\MongoDB(array(
        'databaseName' => 'imbo',
    )),

    // ...
);
```

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

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 array(
    // ...

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

    // ...
);
```

2. Connect to a MySQL database using PDO:

```
<?php
return array(
    // ...

    'database' => function() {
        return new Imbo\Database\Doctrine(array(
            '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 `array('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
<?php
return array(
    // ...

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

    // ...
);
```

2. Connect to a replica set:

```php
<?php
return array(
    // ...

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

    // ...
);
```

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
<?php
return array(
    // ...

    'database' => function() {
        return new My\Custom\Adapter(array(
            'some' => 'option',
        ));
    },

    // ...
);
```

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

1.3.3 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
<?php
return array(
    // ...

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

    // or

    'storage' => new Imbo\Storage\Filesystem(array(
        'dataDir' => '/path/to/images',
    )),

    // ...
);
```

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

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 array(
    // ...

    'storage' => function() {
        new Imbo\Storage\S3(array(
            '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 array(
    // ...

    'storage' => function() {
        return new Imbo\Storage\Doctrine(array(
            'pdo' => new PDO('sqlite:/path/to/database'),
        ));
    },

    // ...
);
```

2. Connect to a MySQL database using PDO:

```
<?php
return array(
    // ...

    'storage' => function() {
        return new Imbo\Storage\Doctrine(array(
            '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 public key of 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 public key and creates directories for each of them, then the full public key, 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.

Examples

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

```
<?php
return array(
    // ...

    'storage' => function() {
        new Imbo\Storage\Filesystem(array(
            '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 `array('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 array(
    // ...

    'storage' => function() {
        return new Imbo\Storage\GridFS();
    },

    // ...
);
```

2. Connect to a replica set:

```
<?php
return array(
    // ...

    'storage' => function() {
        return new Imbo\Storage\GridFS(array(
            'server' => 'mongodb://server1,server2,server3',
            'options' => array(
                '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 array(
    // ...

    'storage' => function() {
        return new My\Custom\Adapter(array(
            'some' => 'option',
        ));
    },

    // ...
);
```

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

1.3.4 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\ListenerInteface` interface:

```php
<?php
return array(
    // ...

    'eventListeners' => array(
        'accessToken' => 'Imbo\EventListener\AccessToken',
    ),

    // ...
);
```

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

```php
<?php
return array(
    // ...

    'eventListeners' => array(
        'accessToken' => new Imbo\EventListener\AccessToken(),
    ),

    // ...
);
```

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

```php
<?php
return array(
    // ...

    'eventListeners' => array(
        'accessToken' => function() {
            return new Imbo\EventListener\AccessToken();
        },
    ),

    // ...
);
```

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

```php
<?php
return array(
    // ...

    'eventListeners' => array(
        'maxImageSize' => array(
            'listener' => new Imbo\EventListener\MaxImageSize(1024, 768),
            'publicKeys' => array(
                'whitelist' => array('user'),
                // 'blacklist' => array('someotheruser'),
            ),
            // 'params' => array( ... )
        ),
    ),

    // ...
);
```

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 `publicKeys` element is an array that you can use if you want your listener to only be triggered for some users (public keys). The value of this is an array with two elements, `whitelist` and `blacklist`, where `whitelist` is an array of public keys you **want** your listener to trigger for, and `blacklist` is an array of public keys you **don't want** your listener to trigger for. `publicKeys` 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
<?php
return array(
    // ...

    'eventListeners' => array(
        'maxImageSize' => array(
            'listener' => 'Imbo\EventListener\MaxImageSize',
            'publicKeys' => array(
                'whitelist' => array('user'),
                // 'blacklist' => array('someotheruser'),
            ),
            'params' => array(
                '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
<?php
return array(
    // ...

    'eventListeners' => array(
        'customListener' => array(
            'callback' => function(Imbo\EventManager\EventInterface $event) {
                // Custom code
            },
            'events' => array('image.get'),
            'priority' => 1,
            'publicKeys' => array(
                'whitelist' => array('user'),
                // 'blacklist' => array('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. `publicKeys` 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*
- *strip*
- *thumbnail*
- *transpose*
- *transverse*
- *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 array(
    // ...

    'eventListeners' => array(
        '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.

1.3.5 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 array(
    'eventListeners' => array(
        'customListener' => 'Listener',
        'otherCustomListener' => 'OtherListener',
    ),

    'eventListenerInitializers' => array(
        '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 array(
    'eventListeners' => array(
        '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.

1.3.6 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 array(
    // ...

    'transformationPresets' => array(
        'graythumb' => array(
            '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 array(
    // ...

    'transformationPresets' => array(
        'fixedGraythumb' => array(
            'thumbnail' => array(
                '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 URL's 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).

1.3.7 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
<?php
return array(
    // ...

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

        // or

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

        // or

        'users' => 'ImboUsers',
    ),

    'routes' => array(
        '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
<?php
use Imbo\Resource\ResourceInterface,
    Imbo\EventManager\EventInterface,
    Imbo\Model\ListModel;

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

    public static function getSubscribedEvents() {
        return array(
            '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.

1.4 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.

1.4.1 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:

```
array(
    'transformations' => array(
        'whitelist' => array(
            'border',
        ),
    ),
)
```

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

```
array(
    'transformations' => array(
        'blacklist' => array(
            '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 array(
    // ...

    'eventListeners' => array(
        'accessToken' => 'Imbo\EventListener\AccessToken',
    ),

    // ...
);
```

Disable this event listener with care. Installations with no access token check is open for DoS attacks.

1.4.2 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 array(
    // ...

    'eventListeners' => array(
        'authenticate' => 'Imbo\EventListener\Authenticate',
    ),

    // ...
);
```

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

1.4.3 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
<?php
return array(
    // ...

    'eventListeners' => array(
        '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.

1.4.4 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
<?php
return array(
    // ...

    'eventListeners' => array(
        'cors' => array(
            'listener' => 'Imbo\EventListener\Cors',
            'params' => array(
                'allowedOrigins' => array('http://some.origin'),
                'allowedMethods' => array(
                    'image'  => array('GET', 'HEAD'),
                    'images' => array('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).

1.4.5 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
<?php
return array(
    // ...

    'eventListeners' => array(
        'exifMetadata' => array(
            'listener' => 'Imbo\EventListener\ExifMetadata',
            'params' => array(
                'allowedTags' => array('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 `array('*')` as filter.

1.4.6 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 array(
    // ...

    'eventListeners' => array(
        'imageTransformationCache' => array(
            'listener' => 'Imbo\EventListener\ImageTransformationCache',
            'params' => array(
                '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.

1.4.7 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.

1.4.8 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 array(
    // ...

    'eventListeners' => array(
        'maxImageSizeListener' => array(
            'listener' => 'Imbo\EventListener\MaxImageSize',
            'params' => array(
                '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.

1.4.9 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 array(
    // ...

    'eventListeners' => array(
        'metadataCache' => array(
            'listener' => 'Imbo\EventListener\MetadataCache',
            'params' => array(
                'cache' => new Imbo\Cache\APC('imbo'),
            ),
        ),
    ),

    // ...
);
```

1.4.10 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 array(
    // ...

    'eventListeners' => array(
        'statsAccess' => array(
            'listener' => 'Imbo\EventListener\StatsAccess',
            'params' => array(
                'allow' => array('127.0.0.1', '::1'),
            ),
        ),
    ),

    // ...
);
```

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

```
<?php
return array(
    // ...

    'eventListeners' => array(
        'statsAccess' => array(
            'listener' => 'Imbo\EventListener\StatsAccess',
            'params' => array(
                array(
                    'allow' => array('*'),
                )
            ),
        ),
    ),

    // ...
);
```

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

1.4.11 Varnish HashTwo

This event listener can be enabled if you want Imbo to send a HashTwo header optionally used by Varnish. The listener when enabled subscribes to the following event:

- `image.get`

The parameters supports a single element:

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

```
<?php
return array(
    // ...

    'eventListeners' => array(
        'hashTwo' => 'Imbo\EventListener\VarnishHashTwo',
    ),

    // ...
);
```

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

```php
<?php
return array(
    // ...

    'eventListeners' => array(
        'hashTwo' => array(
            'listener' => 'Imbo\EventListener\VarnishHashTwo',
            'params' => array(
                'headerName' => 'X-Custom-HashTwo-Header-Name',
            ),
        ),
    ),

    // ...
);
```

The value of the header is a combination of the public key and the current image identifier, separated by `|`.

CHAPTER 2

End user guide

2.1 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.

2.1.1 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`
- User resource - `/users/<user>`
- Images resource - `/users/<user>/images`
- Image resource - `/users/<user>/images/<image>`
- ShortURL resource - `/s/<id>`
- Metadata resource - `/users/<user>/images/<image>/metadata`

Index resource - /

The index resource shows the version of the Imbo installation along with some external URL's for Imbo-related information, and some internal URL's 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/{publicKey}",
    "images": "http://imbo/users/{publicKey}/images",
    "image": "http://imbo/users/{publicKey}/images/{imageIdentifier}",
    "shortImageUrl": "http://imbo/s/{id}",
    "metadata": "http://imbo/users/{publicKey}/images/{imageIdentifier}/metadata"
  }
}
```

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

Stats resource - `/stats`

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

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

results in:

```
{
  "users": {
    "someuser": {
      "numImages": 11,
      "numBytes": 3817197
    },
    "someotheruser": {
      "numImages": 1,
      "numBytes": 81097
    }
  },
  "total": {
    "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
<?php
return array(
    // ...

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

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

    // ...
);
```

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

```json
{
  "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 then 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.

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:

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

where `publicKey` is the public key of 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 Public key 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.

```
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",
      "extension": "png",
      "height": 77,
      "imageIdentifier": "<image>",
      "metadata": {
        "key": "value",
        "foo": "bar"
      },
      "mime": "image/png",
      "publicKey": "<user>",
      "size": 6791,
      "updated": "Mon, 10 Dec 2012 11:57:51 GMT",
      "width": 1306
    }
  ]
}
```

The `search` object is data related to pagination, where `hits` is the number of images found by your 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, where `added` is a formatted date of when the image was added to Imbo, `extension` is the original image extension, `height` is the height of the image in pixels, `imageIdentifier` is the image identifier (MD5 checksum of the file itself), `metadata` is a JSON object containing metadata attached to the image, `mime` is the mime type of the image, `publicKey` is the public key of the user who owns the image, `size` is the size of the image in bytes, `updated` is a 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), and `width` is the width of the image in pixels. The fact that the image identifier is the MD5 checksum of the image is an implementation detail and might change in the future.

The `metadata` field is only available if you used the `metadata` query parameter described above.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 Public key 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 (checksum) 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
X-Imbo-ShortUrl: http://imbo/s/w7CiqDM
```

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

ShortURL resource - `/s/<id>`

Images in Imbo have short URLs associated with them, which are generated on request when you access an image (with or without image transformations) for the first time. These URLs do not take any query parameters and can be used in place for original image URLs. To fetch these URLs you can request an image using `HTTP HEAD`, then look for the `X-Imbo-ShortUrl` header in the response:

```
curl -Ig "http://imbo/users/<user>/images/<image>?t[]=thumbnail&t[]=desaturate&t[]=border&accessToke
```

results in (some headers omitted):

```
X-Imbo-OriginalMimeType: image/gif
X-Imbo-OriginalWidth: 771
X-Imbo-OriginalHeight: 771
X-Imbo-OriginalFileSize: 152066
X-Imbo-OriginalExtension: gif
X-Imbo-ShortUrl: http://imbo/s/3VEFrpB
X-Imbo-ImageIdentifier: 4492acb937a1f056ae43509bc7f85d21
```

The value of the `X-Imbo-ShortUrl` can be used to request the image with the applied transformations, and does not require an access token query parameter.

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 URL 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 URL 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 append `.jpg` to the URL when generating the short URL.

Note: In Imbo only images have short URL's

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.

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

results in:

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

where `<image>` is the image that just got updated.

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.

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

results in:

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

where `<image>` is the image that just got updated.

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

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:

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

where `<image>` is the image identifier of the image that just got all its metadata deleted.

Typical response codes:

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

2.1.2 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:

```php
<?php
$publicKey  = "<user>";         // The public key of the user
$privateKey = "<secret value>"; // The private key of the user
$image      = "<image>";        // The image identifier

// Image transformations
$query = implode("&", array(
    "t[]=thumbnail:width=40,height=40,fit=outbound",
    "t[]=border:width=3,height=3,color=000",
    "t[]=canvas:width=100,height=100,mode=center"
));

// The URI
$uri = sprintf("http://imbo/users/%s/images/%s?%s", $publicKey, $image, $query);

// Generate the token
$accessToken = hash_hmac("sha256", $uri, $privateKey);

// Output the URI with the access token
echo sprintf("%s&accessToken=%s", $uri, $accessToken);
```

and Python:

```python
from hashlib import sha256
import hmac

publicKey  = "<user>"         # The public key of the user
privateKey = "<secret value>" # The private key of the user
image      = "<image>"        # The image identifier

# Image transformations
query = "&".join([
    "t[]=thumbnail:width=40,height=40,fit=outbound",
    "t[]=border:width=3,height=3,color=000",
    "t[]=canvas:width=100,height=100,mode=center"
])

# The URI
uri = "http://imbo/users/%s/images/%s?%s" % (publicKey, image, query)

# Generate the token
accessToken = hmac.new(privateKey, uri, sha256)

# Output the URI with the access token
print "%s&accessToken=%s" % (uri, accessToken.hexdigest())
```

and Ruby:

```ruby
require "digest"

publicKey  = "<user>"         # The public key of the user
privateKey = "<secret value>" # The private key of the user
image      = "<image>"        # The image identifier

# Image transformations
query = [
    "t[]=thumbnail:width=40,height=40,fit=outbound",
    "t[]=border:width=3,height=3,color=000",
    "t[]=canvas:width=100,height=100,mode=center"
].join("&")

# The URI
uri = "http://imbo/users/#{publicKey}/images/#{image}?#{query}"

# Generate the token
accessToken = Digest::HMAC.hexdigest(uri, privateKey, Digest::SHA256)

# Output the URI with the access token
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).

2.1.3 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 of the user
- 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:

```
<?php
$publicKey  = "<user>";                 // The public key of the user
$privateKey = "<secret value>";         // The private key of the user
$timestamp  = gmdate("Y-m-d\TH:i:s\Z"); // Current timestamp (UTC)
$image      = "<image>";                // The image identifier
$method     = "DELETE";                 // HTTP method to use

// The URI
$uri = sprintf("http://imbo/users/%s/images/%s", $publicKey, $image);

// Data for the hash
$data = implode("|", array($method, $uri, $publicKey, $timestamp));

// Generate the token
$signature = hash_hmac("sha256", $data, $privateKey);

// Request the URI
$response = file_get_contents($uri, false, stream_context_create(array(
    "http" => array(
        "method" => $method,
        "header" => array(
            "X-Imbo-Authenticate-Signature: " . $signature,
            "X-Imbo-Authenticate-Timestamp: " . $timestamp,
        ),
    ),
)));
```

and Python (using the Requests library):

```
import hmac, requests
from hashlib import sha256
from datetime import datetime

publicKey  = "<user>"                                        # The public key of the user
privateKey = "<secret value>"                                # The private key of the user
timestamp  = datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ") # Current timestamp (UTC)
image      = "<image>"                                       # The image identifier
method     = "DELETE"                                        # HTTP method to use

# The URI
uri = "http://imbo/users/%s/images/%s" % (publicKey, image)

# Data for the hash
data = "|".join([method, uri, publicKey, timestamp])

# Generate the token
signature = hmac.new(privateKey, data, sha256).hexdigest()

# Request the URI
response = requests.delete(uri, headers = {
    "X-Imbo-Authenticate-Signature": signature,
    "X-Imbo-Authenticate-Timestamp": timestamp
})
```

and Ruby (using the httpclient gem):

```
require "digest"
require "httpclient"

publicKey  = "<user>"                                   # The public key of the user
privateKey = "<secret value>"                           # The private key of the user
timestamp  = Time.now.utc.strftime("%Y-%m-%dT%H:%M:%SZ") # Current timestamp (UTC)
image      = "<image>"                                  # The image identifier
method     = "DELETE"                                   # HTTP method to use

# The URI
uri = "http://imbo/users/#{publicKey}/images/#{image}"

# Data for the hash
data = [method, uri, publicKey, timestamp].join("|")

# Generate the token
signature = Digest::HMAC.hexdigest(data, privateKey, Digest::SHA256)

# Request the URI
client = HTTPClient.new
response = client.delete(uri, {}, {
    "X-Imbo-Authenticate-Signature" => signature,
    "X-Imbo-Authenticate-Timestamp" => timestamp
})
```

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.

2.1.4 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).

2.1.5 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.

2.2 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.

2.2.1 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`

2.2.2 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`

2.2.3 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`

2.2.4 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`

2.2.5 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`

2.2.6 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.

Examples:

- `t[]=crop:x=10,y=25,width=250,height=150`

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

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

Examples:

- `t[]=desaturate`

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

This transformation flips the image horizontally.

Examples:

- `t[]=flipHorizontally`

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

This transformation flips the image vertically.

Examples:

- `t[]=flipVertically`

2.2.10 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`

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

This transformation makes the image progressive.

Examples:

- `t[]=progressive`

2.2.12 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`

2.2.13 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`

2.2.14 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`

2.2.15 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`

2.2.16 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`

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

This transformation transposes the image.

Examples:

- `t[]=transpose`

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

This transformation transverses the image.

Examples:

- `t[]=transverse`

2.2.19 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 `<img>` will be used.

`height` Height of the watermark image in pixels. If omitted the height of `<img>` 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`

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`

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

```
<?php
return array(
    // ...

    'eventListeners' => array(
        '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.

CHAPTER 3

Extending/customizing Imbo

3.1 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.

3.1.1 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*
- *shorturl*
- *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, lowercased 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.

3.1.2 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:

```php
<?php
/**
 * This file is part of the Imbo package
 *
 * (c) Christer Edvartsen <cogo@starzinger.net>
 *
 * For the full copyright and license information, please view the LICENSE file that was
 * distributed with this source code.
 */
```

```

namespace Imbo\EventListener;

/**
 * Event listener interface
 *
 * @author Christer Edvartsen <cogo@starzinger.net>
 * @package Event\Listeners
 */
interface ListenerInterface {
    /**
     * Return an array with events to subscribe to
     *
     * Single callbacks can use the simplest method, defaulting to a priority of 0
     *
     * return array(
     *     'event' => 'someMethod',
     *     'event2' => 'someOtherMethod',
     * );
     *
     * If you want to specify multiple callbacks and/or a priority for the callback(s):
     *
     * return array(
     *     'event' => array(
     *         'someMethod', // Defaults to priority 0, same as 'someMethod' => 0
     *         'someOtherMethod' => 10, // Will trigger before "someMethod"
     *         'someThirdMethod' => -10, // Will trigger after "someMethod"
     *     ),
     *     'event2' => 'someOtherMethod',
     * );
     *
     * @return array
     */
    static function getSubscribedEvents();
}
```

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 = array();
    $events = array(
        'images.post',
        'image.delete',
        'metadata.put',
        'metadata.post',
        'metadata.delete'
    );

    foreach ($events as $event) {
        $callbacks[$event] = array('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 invokable 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 array(
    // ...

    'eventListeners' => array(
        'customListener' => array(
            'callback' => function(Imbo\EventManager\EventInterface $event) {
                // Custom code
            },
            'events' => array('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.

3.1.3 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.

3.2 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 confguration* 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.

3.3 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
<?php
class My\Custom\Transformation implements Imbo\EventListener\ListenerInterface {
    public static function getSubscribedEvents() {
        return array('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 array(
    // ..

    'eventListeners' => array(
        '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:

```
array(
    'width' => '100',
    'height' => '200',
)
```

Take a look at the existing transformations included with Imbo for more information.

3.4 Cache adapters

If you want to leverage caching in a custom event listener, Imbo ships with some different solutions:

3.4.1 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"
```

3.4.2 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
<?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"
```

3.4.3 Implement a custom cache adapter

If you want to use some other cache mechanism an interface exists (`Imbo\Cache\CacheInterface`) for you to implement:

```php
<?php
/**
 * This file is part of the Imbo package
 *
 * (c) Christer Edvartsen <cogo@starzinger.net>
 *
 * For the full copyright and license information, please view the LICENSE file that was
 * distributed with this source code.
 */

namespace Imbo\Cache;

/**
 * Cache adapter interface
 *
 * An interface for cache adapters.
 *
 * @author Christer Edvartsen <cogo@starzinger.net>
 * @package Cache
 */
interface CacheInterface {
    /**
     * Get a cached value by a key
     *
     * @param string $key The key to get
     * @return mixed Returns the cached value or null if key does not exist
     */
    function get($key);

    /**
     * Store a value in the cache
     *
     * @param string $key The key to associate with the item
     * @param mixed $value The value to store
     * @param int $expire Number of seconds to keep the item in the cache
     * @return boolean True on success, false otherwise
     */
    function set($key, $value, $expire = 0);

    /**
     * Delete an item from the cache
     *
     * @param string $key The key to remove
     * @return boolean True on success, false otherwise
     */
    function delete($key);

    /**
     * Increment a value
     *
     * @param string $key The key to use
     * @param int $amount The amount to increment with
     * @return int|boolean Returns new value on success or false on failure
     */
    function increment($key, $amount = 1);

    /**
     * Decrement a value
     *
     * @param string $key The key to use
     * @param int $amount The amount to decrement with
     * @return int|boolean Returns new value on success or false on failure
     */
    function decrement($key, $amount = 1);
}
```

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.

3.5 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 features 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.

3.5.1 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.

3.5.2 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
```

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 --profile travis
```

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.