Imbo Documentation

Release 0.3.3-beta

Christer Edvartsen

January 30, 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. See *Image transformations* for a complete list of the supported transformations.

Imbo is an open source 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.

Feel free to join the #imbo channel on the Freenode IRC network (chat.freenode.net).

CHAPTER 1

Documentation

1.1 Requirements

Imbo requires a web server 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.

Optional requirements are Doctrine Database Abstraction Layer (for some storage and database drivers), and Memcached and/or APC for caching.

1.2 Installation

The easiest way to install Imbo is to *clone the repository*, and then use Composer to install the dependencies:

```
$ git clone git@github.com:imbo/imbo.git
$ cd imbo
$ curl -s https://getcomposer.org/installer | php
$ php composer.phar install
```

After installing the PHP files you will need to configure your web server.

1.2.1 Web server configuration

Imbo ships with sample configuration files for Apache and Nginx that can be used with a few minor adjustments. Both configuration files assumes you run your httpd 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.

```
<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/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/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.

Nginx

The sample Nginx configuration 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/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/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.

1.2.2 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 you have Imbo installed in `example.com/imbo`.

1.3 Configuration

Imbo ships with a default configuration file named `config/config.default.php` that Imbo will load. You can specify your own configuration file, `config/config.php`, that Imbo will merge with the default. You should never update `config/config.default.php`.

1.3.1 User key pairs

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 the configuration file:

```php
<?php
namespace Imbo;

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.

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 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 drivers use it when storing images and metadata.

1.3.2 Database configuration

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

In the default configuration file the *MongoDB* storage driver is used, and it is returned via a Closure. You can choose to override this in your `config.php` file by specifying a closure that returns a different value, or you can specify an implementation of the `Imbo\Database\DatabaseInterface` interface directly. Which database driver to use is specified in the `database` key in the configuration array:

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // or

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

    // ...
);
```

Available database drivers

The following database drivers are shipped with Imbo:

- Doctrine
- MongoDB

Doctrine

This driver uses the Doctrine Database Abstraction Layer. The options you pass to the constructor of this driver is passed to the underlying classes, so have a look at the Doctrine-DBAL documentation over at doctrine-project.org.

Database schema When using this driver you need to create a couple of tables in the DBMS you choose to use. Below you will find statements to create the necessary tables for SQLite and MySQL.

SQLite

```sql
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
)
```

MySQL

```sql
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 ;
```

Note: Imbo will not create these tables automatically.

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

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

```
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

2. Connect to a MySQL database using PDO:

```
<?php
namespace Imbo;

return array(
    // ...

    'database' => function() {
        return new Database\Doctrine(array(
            'dbname'   => 'database',
            'user'     => 'username',
            'password' => 'password',
            'host'     => 'hostname',
            'driver'   => 'pdo_mysql',
        ));
    },

    // ...
);
```

MongoDB

This driver 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`.

collectionName Name of the collection to use. Defaults to `images`.

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

options Options passed to the underlying driver. Defaults to `array('connect' => true, 'timeout' => 1000)`. See the manual for the Mongo constructor at php.net for available options.

Examples

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

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

2. Connect to a replica set:

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

1.3.3 Storage configuration

Storage drivers are responsible for storing the original images you put into imbo. Like with the database driver it is not possible to simply switch a driver without having migration scripts available to move the stored images. Choose a driver with care.

In the default configuration file the *GridFS* storage driver is used, and it is returned via a Closure. You can choose to override this in your `config.php` file by specifying a closure that returns a different value, or you can specify an implementation of the `Imbo\Storage\StorageInterface` interface directly. Which storage driver to use is specified in the `storage` key in the configuration array:

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

Available storage drivers

The following storage drivers are shipped with Imbo:

- Doctrine
- Filesystem
- GridFS

Doctrine

This driver uses the Doctrine Database Abstraction Layer. The options you pass to the constructor of this driver is passed to the underlying classes, so have a look at the Doctrine-DBAL documentation over at doctrine-project.org.

Database schema When using this driver you need to create a table in the DBMS you choose to use. Below you will find a statement to create this table in SQLite and MySQL.

SQLite

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

MySQL

```
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;
```

Note: Imbo will not create the table automatically.

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

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

```
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

2. Connect to a MySQL database using PDO:

```
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

Filesystem

This driver simply stores all images on the file system. This driver only has one parameter, and that is the directory where you want your images stored:

dataDir The base path where the images are stored.

This driver 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. If you have configured this driver with `/path/to/images` as `dataDir` and issue the following command:

```
$ curl -XPUT http://imbo/users/username/images/bbd9ae7bbfcefb0cc9a52f03f89dd3f9 --data-binary @someIr
```

the image will be stored in:

`/path/to/images/u/s/e/username/b/b/d/bbd9ae7bbfcefb0cc9a52f03f89dd3f9`

The algorithm that generates the path simply takes the three first characters of `<user>` and creates directories for each of them, then the full public key, then a directory of each of the first characters in `<image>` and lastly it stores the image in a file with a filename equal to `<image>`.

Read more about the API in the *RESTful API* topic.

Examples Default configuration:

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

GridFS

The GridFS driver is used to store the images in MongoDB using the GridFS specification. This driver 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 driver. Defaults to `array('connect' => true, 'timeout' => 1000)`. See the manual for the Mongo constructor at php.net for available options.

Examples

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

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

2. Connect to a replica set:

```
<?php
namespace Imbo;

return array(
    // ...

    'storage' => function() {
        return new Storage\GridFS(array(
            'server' => 'mongodb://server1,server2,server3',
            'options' => array(
                'replicaSet' => 'nameOfReplicaSet',
            ),
        ));
    },

    // ...
);
```

1.3.4 Event listeners

Imbo also supports 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. The keys are short names used to identify the listeners, and are not really used for anything in the Imbo application, but exists so you can override/disable event listeners specified in `config.default.php`. If you want to disable the default event listeners simply specify the same key in the `config.php` file and set the value to `null` or `false`.

Event listeners can be added in the following ways:

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

```
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

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

```
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

3. Use an instance of a class implementing the `Imbo\EventListener\ListenerInterface` interface together with a public key filter:

```
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'maxImageSize' => array(
            'listener' => new EventListener\MaxImageSize(1024, 768),
            'publicKeys' => array(
                'include' => array('user'),
                // 'exclude' => array('someotheruser'),
            ),
        ),
    ),

    // ...
);
```

where `listener` is an instance of the `Imbo\EventListener\ListenerInterface` interface, and `publicKeys` 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 one of two keys: `include` or `exclude` where `include` is an array you want your listener to trigger for, and `exclude` is an array of users you don't want your listener to trigger for. `publicKeys` is optional, and per default the listener will trigger for all users.

4. Use a closure:

```
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'customListener' => array(
            'callback' => function(EventManager\EventInterface $event) {
                // Custom code
            },
            'events' => array('image.get'),
            'priority' => 1,
            'publicKeys' => array(
                'include' => array('user'),
                // 'exclude' => 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 1. 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 1 and 100. `publicKeys` uses the same format as described above.

Events

When configuring 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 five resources:

- *status*
- *user*
- *images*
- *image*
- *metadata*

Examples of events that is triggered:

- `image.get`
- `image.put`
- `image.delete`

As you can see from the above examples the events are built up by the resource name and the HTTP method, 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`
- `route`
- `response.send`

Below you will see the different event listeners that Imbo ships with and the events they subscribe to.

Event listeners

Imbo ships with a collection of event listeners for you to use. Some of them are enabled in the default configuration file.

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 works in the *Access tokens* topic in the *RESTful API* section.

To enforce the access token check for all read requests this event listener subscribes to the following events:

- `user.get`
- `images.get`
- `image.get`
- `metadata.get`
- `user.head`
- `images.head`
- `image.head`
- `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 *Image transformations* 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 filters (which means that the access token will be enforced for all requests):

```php
<?php
namespace Imbo;

return array(
    // ...

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

    // ...
);
```

Disable this event listener with care. Clients can easily DDoS your installation if you let them specify image transformations without limitations.

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* topic in the *RESTful API* section.

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

- `image.put`
- `image.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
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'authenticate' => function() {
            return new EventListener\Authenticate();
        },
    ),

    // ...
);
```

Disable this event listener with care. Clients can delete all your images and metadata when this listener is not enabled.

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
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'autoRotate' => function() {
            return new EventListener\AutoRotateImage();
        },
    ),

    // ...
);
```

If you enable this listener all new images added to Imbo will be auto rotated based on the EXIF data.

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.

To enable the listener, use the following:

```php
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'cors' => function() {
            return new EventListener\Cors(array(
                'allowedOrigins' => array('http://some.origin'),
                'allowedMethods' => array(
                    'image'  => array('GET', 'HEAD', 'PUT'),
                    'images' => array('GET', 'HEAD'),
                ),
                'maxAge' => 3600,
            ));
        },
    ),

    // ...
);
```

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 (image, images, metadata, status and user). The value is an array of HTTP methods you wish to open up.

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

Exif metadata

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

The event listener subscribes to the following events:

- image.put
- db.image.insert

and has the following parameters:

$allowedTags The tags you want to be populated as metadata, if present. Optional - by default all tags are added.

and is enabled like this:

```php
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'exifMetadata' => function() {
            return new EventListener\ExifMetadata(array(
                'exif:Make',
                'exif:Model',
            ));
        },
    ),

    // ...
);
```

which would allow only exif:Make and exif:Model as metadata tags. Not passing an array to the constructor will allow all tags.

Image transformation cache

This event listener enables caching of image transformations. Read more about image transformations in the *Image transformations* topic in the *RESTful API* section.

To achieve this the listener subscribes to the following events:

- image.get (both before and after the main application logic)
- image.delete

The event listener has one parameter:

`$path` Root path where the cached images will be stored.

and is enabled like this:

```php
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'imageTransformationCache' => function() {
            return new EventListener\ImageTransformationCache('/path/to/cache');
        },
    ),

    // ...
);
```

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

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.

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:

- `image.put`

and has the following parameters:

`$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
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'maxImageSize' => function() {
            return new EventListener\MaxImageSize(1024, 768);
        },
    ),

    // ...
);
```

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

Metadata cache

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

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

and has the following parameters:

`Imbo\Cache\CacheInterface $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. Here is an example that uses the APC adapter for caching:

```
<?php
namespace Imbo;

return array(
    // ...

    'eventListeners' => array(
        'metadataCache' => function() {
            return new EventListener\MetadataCache(new Cache\APC('imbo'));
        },
    ),

    // ...
);
```

The event object

The object passed to the event listeners (and closures) 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`.

`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`)

1.3.5 Image transformations

Imbo supports a set of image transformations out of the box using the Imagick PHP extension. All supported image transformations are included in the configuration, and you can easily add your own custom transformations or create presets using a combination of existing transformations.

Transformations are triggered using the `t[]` query parameter together with the image resource (read more about the image resource and the included transformations and their parameters in the *Image resource* section). This parameter should be used as an array so that multiple transformations can be made. The transformations are applied in the order they are specified in the URL.

All transformations are registered in the configuration array under the `imageTransformations` key:

```php
<?php
namespace Imbo;

return array(
    // ...

    'imageTransformations' => array(
        'border' => function (array $params) {
            return new Image\Transformation\Border($params);
        },
        'canvas' => function (array $params) {
            return new Image\Transformation\Canvas($params);
        },
        // ...
    ),

    // ...
);
```

where the keys are the names of the transformations as specified in the URL, and the values are closures which all receive a single argument. This argument is an array that matches the parameters for the transformation as specified in the URL. If you use the following query parameter:

`t[]=border:width=1,height=2,color=f00`

the `$params` array given to the closure will look like this:

```php
<?php
array(
    'width' => '1',
    'height' => '1',
    'color' => 'f00'
)
```

The return value of the closure must either be an instance of the `Imbo\Image\Transformation\TransformationInterface` interface, or code that is callable (for instance another closure, or a class that includes an `__invoke` method). If the return value is a callable piece of code it will receive a single parameter which is an instance of `Imbo\Model\Image`, which is the image you want your transformation to modify. See some examples in the *Custom transformations* section below.

Presets

Imbo supports the notion of transformation presets by using the `Imbo\Image\Transformation\Collection` transformation. The constructor of this transformation takes an array containing other transformations.

```php
<?php
namespace Imbo;

return array(
    // ...

    'imageTransformations' => array(
        'graythumb' => function ($params) {
            return new Image\Transformation\Collection(array(
                new Image\Transformation\Desaturate(),
                new Image\Transformation\Thumbnail($params),
            ));
        },
    ),

    // ...
);
```

which can be triggered using the following query parameter:

```
t[]=graythumb
```

Custom transformations

You can also implement your own transformations by implementing the `Imbo\Image\Transformation\TransformationInterface` interface, or by specifying a callable piece of code. An implementation of the border transformation as a callable piece of code could for instance look like this:

```php
<?php
namespace Imbo;

return array(
    // ...

    'imageTransformations' => array(
        'border' => function (array $params) {
            return function (Model\Image $image) use ($params) {
                $color = !empty($params['color']) ? $params['color'] : '#000';
                $width = !empty($params['width']) ? $params['width'] : 1;
                $height = !empty($params['height']) ? $params['height'] : 1;

                try {
                    $imagick = new \Imagick();
                    $imagick->readImageBlob($image->getBlob());
                    $imagick->borderImage($color, $width, $height);

                    $size = $imagick->getImageGeometry();

                    $image->setBlob($imagick->getImageBlob())
                          ->setWidth($size['width'])
                          ->setHeight($size['height']);
                } catch (\ImagickException $e) {
                    throw new Image\Transformation\TransformationException($e->getMessage(), 400, $e
                }
            };
        },
    ),

    // ...
);
```

It's not recommended to use this method for big complicated transformations. It's better to implement the interface mentioned above, and refer to your class in the configuration array instead:

```php
<?php
namespace Imbo;

return array(
    // ..

    'imageTransformations' => array(
        'border' => function (array $params) {
            return new My\Custom\BorderTransformation($params);
        },
    ),

    // ...
);
```

where `My\Custom\BorderTransformation` implements `Imbo\Image\Transformation\TransformationInterfac`

1.4 RESTful API

Imbo uses a RESTful API to manage the stored images and metadata. Each image is identified by a public key (the "username") and an MD5 checksum of the file itself. The public key and the image identifier will be referred to as `<user>` and `<image>` respectively for the remainder of this document. For all cURL examples `imbo` will be used as a host name. The examples will also omit access tokens and authentication signatures.

1.4.1 Content types

Currently Imbo 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, and for most examples in this document you will see the `.json` extension being used. Change that to `.xml` to get XML data. 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
  }
)
```

1.4.2 Resources

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

Available resources

- Status resource
- User resource
- Images resource
- Image resource
- Metadata resource

Status resource

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 drivers respectively. If both are `true` the HTTP status code is `200 OK`, and if one or both are `false` the status code is `500`. When the status code is `500` the status message will inform you whether it's the database or the storage driver (or both) that is having issues.

Typical response codes:

- 200 OK
- 500 Internal Server Error

User resource

The user resource represents a single user on the current Imbo installation.

GET /users/<user>

Fetch information about a specific user. 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, `numImages` is the number of images the user has stored in Imbo and `lastModified` is when the user last uploaded an image or updated metadata of an image.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 Not found

Images resource

The images resource represents a collection of images owned by a specific user.

GET /users/<user>/images

Get information about the images stored in Imbo for a specific user. Supported query parameters are:

`page` The page number. Defaults to `1`.

`limit` Number of images pr. 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.

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

results in:

```
[
  {
    "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
  }
]
```

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

Images in the array are ordered on the `added` field in a descending fashion.

Typical response codes:

- 200 OK
- 304 Not modified
- 404 Not found

Image resource

The image resource represents specific images owned by a user.

GET /users/<user>/images/<image>

Fetch the image identified by `<image>` owned by `<user>`. Without any query parameters this will return the original image.

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

results in:

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

Typical response codes:

- 200 OK
- 304 Not modified
- 400 Bad Request
- 404 Not found

Image transformations

Below you can find information on the transformations shipped with Imbo along with their parameters.

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`

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

Parameters:

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

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

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

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

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

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

Examples:

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

compress This transformation compresses images on the fly resulting in a smaller payload.

Parameters:

`quality` Quality of the resulting image. 100 is maximum quality (lowest compression rate).

Examples:

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

Warning: This transformation currently only works as expected for `image/jpeg` images.

convert 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:

- `jpg`
- `png`
- `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`

It is not possible to explicitly trigger this transformation via the `t[]` query parameter.

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`

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

Examples:

- `t[]=desaturate`

flipHorizontally This transformation flips the image horizontally.

Examples:

- `t[]=flipHorizontally`

flipVertically This transformation flips the image vertically.

Examples:

- `t[]=flipVertically`

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`

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

Parameters:

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

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

Examples:

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

rotate 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`

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`

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`

transpose This transformation transposes the image.

Examples:

- `t[]=transpose`

transverse This transformation transverses the image.

Examples:

- `t[]=transverse`

PUT /users/<user>/images/<image>

Store a new image on the server.

The body of the response contains a JSON object containing the image identifier of the resulting image:

```
$ curl -XPUT http://imbo/users/<user>/images/<checksum of file to add> --data-binary @<file to add>
```

results in:

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

where `<image>` can be used to fetch the added image and apply transformations to it. The output from this method is important as the `<image>` in the response might not be the same as `<checksum of file to add>` in the URI in the above example (which might occur if for instance event listeners transform the image in some way before Imbo stores it).

Typical response codes:

- 200 OK
- 201 Created
- 400 Bad Request

DELETE /users/<user>/images/<image>

Delete the image identified by `<image>` owned by `<user>` along with all metadata attached to the image.

```
$ 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
- 404 Not found

Metadata resource

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`

Metadata is handled via the `meta` resource in the URI, which is a sub-resource of `<image>`.

GET /users/<user>/images/<image>/meta

Get all metadata attached to `<image>` owned by `<user>`. The output from Imbo is an empty list if the image has no metadata attached, or a JSON object with keys and values if metadata exists:

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

results in:

```
[]
```

when there is not metadata, 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 Not found

PUT /users/<user>/images/<image>/meta

Replace all existing metadata attached to `<image>` owned by `<user>` with the metadata contained in a JSON object in the request body. The response body contains a JSON object with the image identifier:

```
$ curl -XPUT http://imbo/users/<user>/images/<image>/meta.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.

Typical response codes:

- 200 OK
- 400 Bad Request
- 404 Not found

POST /users/<user>/images/<image>/meta

Edit existing metadata and/or add new keys/values to `<image>` owned by `<user>` with the metadata contained in a JSON object in the request body. The response body contains a JSON object with the image identifier:

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

results in:

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

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

Typical response codes:

- 200 OK
- 400 Bad Request
- 404 Not found

DELETE /users/<user>/images/<image>/meta

Delete all existing metadata attached to `<image>` owner by `<user>`. The response body contains a JSON object with the image identifier:

```
$ curl -XDELETE http://imbo/users/<user>/images/<image>/meta.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 Not found

1.4.3 Authentication

Imbo uses two types of authentication mechanisms out of the box. It requires access tokens for all `GET` and `HEAD` requests made against all resources (with the exception of the status resource), and a valid request signature for all `PUT`, `POST` and `DELETE` requests made against all resources that support these methods. Both mechanisms are enforced by event listeners that is enabled in the default configuration file.

Access tokens

Access tokens for all read requests are enforced by an event listener that is enabled per default. The access tokens are used to prevent DoS attacks so think twice (or maybe even some more) before you remove the listener. More about how to remove the listener in *Event listeners*.

The access token, when enforced, must be supplied in the URI using the `accessToken` query parameter and without it all `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 hash of the URI itself (hashed with the SHA-256 algorithm) using the private key of the user as the secret key. 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

// The URI
$url = sprintf('http://example.com/users/%s/images/%s', $publicKey, $image);

// Add some transformations
$transformations = 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'
);
$query = implode('&', $transformations);

// Data for the HMAC
$url .= '?' . $query;

// Generate the token
$accessToken = hash_hmac('sha256', $url, $privateKey);

// Output the URI with the access token
echo $url . '&accessToken=' . $accessToken;
```

If you request a resource from Imbo without a valid access token it will respond with a `400 Bad Request`. 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 *Custom event listeners* for more information).

Signing write requests

Imbo uses a similar method when authenticating write operations. 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`, or two query parameters: `signature` and `timestamp`. `X-Imbo-Authenticate-Signature/signature` is, like the access token, an HMAC (also using SHA-256 and the private key of the user), and 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
<?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
$image      = '<image>';                   // The image identifier

// The URI
$url = sprintf('http://example.com/users/%s/images/%s', $publicKey, $image);

// The method to request with
$method = 'DELETE';

// Data for the hash
$data = $method . '|' . $url . '|' . $publicKey . '|' . $timestamp;

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

// Request using request headers

$context = stream_context_create(array(
    'http' => array(
        'method' => $method,
        'header' => array(
            'X-Imbo-Authenticate-Signature: ' . $signature,
            'X-Imbo-Authenticate-Timestamp: ' . $timestamp,

        ),
    ),
));
file_get_contents($url, false, $context);

// or, request using query parameters

$context = stream_context_create(array(
    'http' => array(
        'method' => $method,
    ),
));
$url = sprintf('%s?signature=%s&timestamp=%s',
               $url,
               rawurlencode($signature),
               rawurlencode($timestamp));

file_get_contents($url, false, $context);
```

Imbo requires that `X-Imbo-Authenticate-Timestamp`/`timestamp` is within ± 120 seconds of the current time on the server. Both the signature and the timestamp must be URL-encoded when used as query parameters.

As with the access token the signature check is enforced by an event listener that can also be disabled. If you want to implement your own authentication paradigm you can do this by creating a custom event listener.

1.4.4 Errors

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

```
$ curl "http://imbo/users/<user>/images/<image>.jpg?t\[\]=foobar"
```

results in:

```
{
  "error": {
    "code": 400,
    "message": "Unknown transformation: foobar",
    "date": "Wed, 12 Dec 2012 21:15:01 GMT",
    "imboErrorCode":0
  },
  "imageIdentifier": "<image>"
}
```

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.

If the user agent specifies a nonexistent username the following occurs:

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

results in:

```
{
  "error": {
    "code": 404,
    "message": "Unknown public key",
    "date": "Mon, 13 Aug 2012 17:22:37 GMT",
    "imboErrorCode": 100
  }
}
```

if `<user>` does not exist.

CHAPTER 2

Extending Imbo

2.1 Cache adapters

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

2.1.1 APC

This adapter uses the APC 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"
```

2.1.2 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
$memcached = new Memcached();
$memcached->addServer('hostname', 11211);

$adapter = new Imbo\Cache\Memcached($memcached, 'imbo');
$adapter->set('key', 'value');
```

```php

echo $adapter->get('key'); // outputs "value"
```

2.1.3 Custom 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 driver interface
 *
 * This is an interface for different database drivers.
 *
 * @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 to the project over at GitHub.

2.2 Custom database drivers

If you wish to implement your own database driver you are free to do so. The only requirement is that you implement the `Imbo\Database\DatabaseInterface` interface that comes with Imbo. Below is the complete interface with comments:

```
<?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\Database;

use Imbo\Model\Image,
    Imbo\Resource\Images\Query,
    Imbo\Exception\DatabaseException,
    DateTime;

/**
 * Database driver interface
 *
 * This is an interface for different database drivers.
 *
 * @author Christer Edvartsen <cogo@starzinger.net>
 * @package Database
 */
interface DatabaseInterface {
    /**
     * Insert a new image
     *
     * This method will insert a new image into the database. If the same image already exists,
     * just update the "updated" information.
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @param Image $image The image to insert
     * @return boolean Returns true on success or false on failure
     * @throws DatabaseException
     */
    function insertImage($publicKey, $imageIdentifier, Image $image);

    /**
     * Delete an image from the database
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return boolean Returns true on success or false on failure
     * @throws DatabaseException
     */
    function deleteImage($publicKey, $imageIdentifier);

    /**
     * Edit metadata
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @param array $metadata An array with metadata
     * @return boolean Returns true on success or false on failure
     * @throws DatabaseException
     */
    function updateMetadata($publicKey, $imageIdentifier, array $metadata);

    /**
     * Get all metadata associated with an image
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return array Returns the metadata as an array
     * @throws DatabaseException
     */
    function getMetadata($publicKey, $imageIdentifier);

    /**
     * Delete all metadata associated with an image
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return boolean Returns true on success or false on failure
     * @throws DatabaseException
     */
    function deleteMetadata($publicKey, $imageIdentifier);

    /**
     * Get images based on some query parameters
     *
     * @param string $publicKey The public key of the user
     * @param Query $query A query instance
     * @return array
     * @throws DatabaseException
     */
    function getImages($publicKey, Query $query);

    /**
     * Load information from database into the image object
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier The image identifier
     * @param Image $image The image object to populate
     * @return boolean
     * @throws DatabaseException
     */
    function load($publicKey, $imageIdentifier, Image $image);

    /**
     * Get the last modified timestamp of a user
     *
     * If the $imageIdentifier parameter is set, return when that image was last updated. If not
     * set, return when the user last updated any image. If the user does not have any images
     * stored, return the current timestamp.
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier The image identifier
     * @return DateTime Returns an instance of DateTime
     * @throws DatabaseException
     */
    function getLastModified($publicKey, $imageIdentifier = null);

    /**
     * Fetch the number of images owned by a given user
     *
     * @param string $publicKey The public key of the user
     * @return int Returns the number of images
     * @throws DatabaseException
     */
    function getNumImages($publicKey);

    /**
     * Get the current status of the database connection
     *
     * This method is used with the status resource.
     *
     * @return boolean
     */
    function getStatus();

    /**
     * Get the mime type of an image
     *
     * @param string $publicKey The public key of the user who owns the image
     * @param string $imageIdentifier The image identifier
     * @return string Returns the mime type of the image
     * @throws DatabaseException
     */
    function getImageMimeType($publicKey, $imageIdentifier);
```

```php
    /**
     * Check if an image already exists
     *
     * @param string $publicKey The public key of the user who owns the image
     * @param string $imageIdentifier The image identifier
     * @return boolean Returns true of the image exists, false otherwise
     * @throws DatabaseException
     */
    function imageExists($publicKey, $imageIdentifier);
}
```

Have a look at the existing implementations of this interface for more details. If you implement a driver that you think should be a part of Imbo feel free to send a pull request to the project over at GitHub.

2.3 Custom storage drivers

If you wish to implement your own storage driver you are free to do so. The only requirement is that you implement the `Imbo\Storage\StorageInterface` interface that comes with Imbo. 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\Storage;

use Imbo\Model\Image,
    Imbo\Exception\StorageException;

/**
 * Storage driver interface
 *
 * This is an interface for different storage drivers for Imbo.
 *
 * @author Christer Edvartsen <cogo@starzinger.net>
 * @package Storage
 */
interface StorageInterface {
    /**
     * Store an image
     *
     * This method will receive the binary data of the image and store it somewhere suited for the
     * actual storage driver. If an error occurs the driver should throw an
     * Imbo\Exception\StorageException exception.
     *
     * If the image already exists, simply overwrite it.
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier The image identifier
     * @param string $imageData The image data to store
     * @return boolean Returns true on success or false on failure
     * @throws StorageException
     */
    function store($publicKey, $imageIdentifier, $imageData);

    /**
     * Delete an image
     *
     * This method will delete the file associated with $imageIdentifier from the storage medium
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return boolean Returns true on success or false on failure
     * @throws StorageException
     */
    function delete($publicKey, $imageIdentifier);

    /**
     * Get image content
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return string The binary content of the image
     * @throws StorageException
     */
    function getImage($publicKey, $imageIdentifier);

    /**
     * Get the last modified timestamp
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return DateTime Returns an instance of DateTime
     * @throws StorageException
     */
    function getLastModified($publicKey, $imageIdentifier);

    /**
     * Get the current status of the storage
     *
     * This method is used with the status resource.
     *
     * @return boolean
     */
    function getStatus();

    /**
     * See if the image already exists
     *
     * @param string $publicKey The public key of the user
     * @param string $imageIdentifier Image identifier
     * @return DateTime Returns an instance of DateTime
     * @throws StorageException
     */
    function imageExists($publicKey, $imageIdentifier);
}
```

Have a look at the existing implementations of this interface for more details. If you implement a driver that you think should be a part of Imbo feel free to send a pull request to the project over at GitHub.

2.4 Custom event listeners

If you wish to implement your own event listeners you are free to do so. The only requirement is that you implement the `Imbo\EventListener\ListenerInterface` interface that comes with Imbo. 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 a list of listener definitions
     *
     * @return ListenerDefinition[]
     */
    function getDefinition();
}
```

Have a look at the existing implementations of this interface for more details. If you implement a listener that you think should be a part of Imbo feel free to send a pull request to the project over at GitHub.