CakePHP-Upload Documentation

Release 3.0.0

Jose Diaz-Gonzalez

July 23, 2016

Contents

Contents:

CHAPTER 1

Introduction

1.1 Upload Plugin 3.0

The Upload Plugin is an attempt to sanely upload files using techniques garnered from packages such as MeioUpload , UploadPack and PHP documentation. It uses the excellent *Flysystem <http://flysystem.thephpleague.com/>* library to handle file uploads, and can be easily integrated with any image library to handle thumbnail extraction to your exact specifications.

1.2 Background

Media Plugin is too complicated, and it was a PITA to merge the latest updates into MeioUpload, so here I am, building yet another upload plugin. I'll build another in a month and call it "YAUP".

1.3 Requirements

- CakePHP 3.x
- PHP 5.4+
- Patience

1.4 What does this plugin do?

- The Upload plugin will transfer files from a form in your application to (by default) the `webroot/files` directory organised by the model name and upload field name.
- It can also move files around programatically. Such as from the filesystem.
- The path to which the files are saved can be customised.
- The plugin can also upload multiple files at the same time to different fields.
- Each upload field can be configured independently of each other, such as changing the upload path etc.
- Uploaded file information can be stored in a data store, such as a MySQL database.
- You can upload files to both disk as well as distributed datastores such as S3 or Dropbox.
- It can optionally delete the files on record deletion

- It offers multiple validation providers but doesn't validate automatically

1.5 This plugin does not do

- Create thumbnails. You can use a custom Transformer to create modified versions of file uploads.
- It will not convert files between file types. You cannot use it convert a JPG to a PNG
- It will not add watermarks to images for you.

CHAPTER 2

Installation

The only officialy supported method of installing this plugin is via composer.

2.1 Using Composer

View on Packagist, and copy the json snippet for the latest version into your project's `composer.json`.

```
composer require josegonzalez/cakephp-upload
```

2.2 Enable plugin

You need to enable the plugin your `config/bootstrap.php` file:

```
<?php
Plugin::load('Josegonzalez/Upload');
```

If you are already using `Plugin::loadAll();`, then this is not necessary.

CHAPTER 3

Examples

3.1 Basic example

Note: You may want to define the Upload behavior *before* the core Translate Behavior as they have been known to conflict with each other.

```
CREATE table users (
    id int(10) unsigned NOT NULL auto_increment,
    username varchar(20) NOT NULL,
    photo varchar(255)
);
```

```
<?php
namespace App\Model\Table;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function initialize(array $config)
    {
        $this->table('users');
        $this->displayField('name');
        $this->primaryKey('id');
        $this->addBehavior('Josegonzalez/Upload.Upload', [
            // You can configure as many upload fields as possible,
            // where the pattern is `field` => `config`
            //
            // Keep in mind that while this plugin does not have any limits in terms of
            // number of files uploaded per request, you should keep this down in order
            // to decrease the ability of your users to block other requests.
            'photo' => []
        ]);
    }
}
?>
```

```
<?php echo $this->Form->create('User', ['type' => 'file']); ?>
<?php echo $this->Form->input('User.username'); ?>
<?php echo $this->Form->input('User.photo', ['type' => 'file']); ?>
<?php echo $this->Form->end(); ?>
```

Using the above setup, uploaded files cannot be deleted. To do so, a field must be added to store the directory of the file as follows:

```
CREATE table users (
    `id` int(10) unsigned NOT NULL auto_increment,
    `username` varchar(20) NOT NULL,
    `photo` varchar(255) DEFAULT NULL,
    `photo_dir` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`id`)
);
```

```
<?php

<?php
namespace App\Model\Table;
use Cake\ORM\Table;

class UsersTable extends Table
{
    public function initialize(array $config)
    {
        $this->table('users');
        $this->displayField('name');
        $this->primaryKey('id');
        $this->addBehavior('Josegonzalez/Upload.Upload', [
            'photo' => [
                'fields' => [
                    // if these fields or their defaults exist
                    // the values will be set.
                    'dir' => 'photo_dir', // defaults to `dir`
                    'size' => 'photo_size', // defaults to `size`
                    'type' => 'photo_type', // defaults to `type`
                ],
            ],
        ]);
    }
}
?>
```

```
<?php echo $this->Form->create('User', ['type' => 'file']); ?>
    <?php echo $this->Form->input('User.username'); ?>
    <?php echo $this->Form->input('User.photo', ['type' => 'file']); ?>
    <?php echo $this->Form->input('User.photo_dir', ['type' => 'hidden']); ?>
<?php echo $this->Form->end(); ?>
```

3.2 Displaying links to files in your view

Once your files have been uploaded you can link to them using the `HtmlHelper` by specifying the path and using the file information from the database.

This example uses the default behaviour configuration using the model `Example`.

```
<?php
// assuming an entity that has the following
// data that was set from your controller to your view
$entity = new Entity([
    'photo' => 'imageFile.jpg',
    'photo_dir' => '7'
]);
```

```
$this->set('entity', $entity);

// You could use the following to create a link to
// the image (with default settings in place of course)
echo $this->Html->link('../files/example/image/' . $entity->photo_dir . '/' . $entity->photo);
?>
```

You can optionally create a custom helper to handle url generation, or contain that within your entity. As it is impossible to detect what the actual url for a file should be, such functionality will *never* be made available via this plugin.

CHAPTER 4

Behavior configuration options

This is a list of all the available configuration options which can be passed in under each field in your behavior configuration.

- `pathProcessor`: Returns a ProcessorInterface class name.
  - Default: (string) `Josegonzalez\Upload\File\Path\DefaultProcessor`
- `writer`: Returns a WriterInterface class name.
  - Default: (string) `Josegonzalez\Upload\File\Writer\DefaultWriter`
- `transformer`: Returns a TransformerInterface class name. Can also be a PHP *callable*.
  - Default: (string) `Josegonzalez\Upload\File\Transformer\DefaultTransformer`
- `path`: A path relative to the `filesystem.root`.
  - Default: (string) `'webroot{DS}files{DS}{model}{DS}{field}{DS}'`
  - Tokens:
    - {DS}: Replaced by a `DIRECTORY_SEPARATOR`
    - {model}: Replaced by the Table-alias() method.
    - {table}: Replaced by the Table->table() method.
    - {field}: Replaced by the field name.
    - {primaryKey}: Replaced by the entity primary key, when available. If used on a new record being created, will have undefined behavior.
    - {year}: Replaced by `date('Y')`
    - {month}: Replaced by `date('m')`
    - {day}: Replaced by `date('d')`
    - {time}: Replaced by `time()`
    - {microtime}: Replaced by `microtime()`
- `fields`: An array of fields to use when uploading files
  - Options:
    - `fields.dir`: (default `dir`) Field to use for storing the directory
    - `fields.type`: (default `type`) Field to use for storing the filetype
    - `fields.size`: (default `size`) Field to use for storing the filesize

- `filesystem`: An array of configuration info for configuring the writer

  If using the DefaultWriter, the following options are available:

  - Options:
    - `filesystem.root`: (default `ROOT . DS`) Directory where files should be written to by default
    - `filesystem.adapter`: (default Local Flysystem Adapter) A Flysystem-compatible adapter. Can also be a callable that returns an adapter.
- `nameCallback`: A callable that can be used by the default pathProcessor to rename a file. Only handles original file naming.
  - Default: `NULL`
  - Available arguments:
    - `array $data`: The upload data
    - `array $settings`: UploadBehavior settings for the current field
  - Return: (string) the new name for the file
- `keepFilesOnDelete`: Keep *all* files when deleting a record.
  - Default: (boolean) `true`
- `restoreValueOnFailure`: Restores original value of the current field when uploaded file has error
  - Defaults: (boolean) `true`

CHAPTER 5

Validation

By default, no validation rules are loaded or attached to the table. You must explicitly load the validation provider(s) and attach each rule if needed.

5.1 Installation

This plugin allows you to only load the validation rules that cover your needs. At this point there are 3 validation providers:

- UploadValidation (validation rules useful for any upload)
- ImageValidation (validation rules specifically for images)
- DefaultValidation (loads all of the above)

Since by default, no validation rules are loaded, you should start with that:

```
<?php

    $validator->provider('upload', \Josegonzalez\Upload\Validation\UploadValidation::class);
    // OR
    $validator->provider('upload', \Josegonzalez\Upload\Validation\ImageValidation::class);
    // OR
    $validator->provider('upload', \Josegonzalez\Upload\Validation\DefaultValidation::class);

?>
```

Afterwards, you can use its rules like:

```
<?php

    $validator->add('file', 'customName', [
        'rule' => 'nameOfTheRule',
        'message' => 'yourErrorMessage',
        'provider' => 'upload'
    ]);

?>
```

It might come in handy to only use a validation rule when there actually is an uploaded file:

```
<?php

    $validator->add('file', 'customName', [
```

```
        'rule' => 'nameOfTheRule',
        'message' => 'yourErrorMessage',
        'provider' => 'upload',
        'on' => function($context) {
            return !empty($context['data']['file']) && $context['data']['file']['error'] == UPLOAD_ER
        }
    ]);

?>
```

More information on conditional validation can be found here.

5.2 UploadValidation

isUnderPhpSizeLimit

Check that the file does not exceed the max file size specified by PHP

```
<?php

    $validator->add('file', 'fileUnderPhpSizeLimit', [
        'rule' => 'isUnderPhpSizeLimit',
        'message' => 'This file is too large',
        'provider' => 'upload'
    ]);

?>
```

isUnderFormSizeLimit

Check that the file does not exceed the max file size specified in the HTML Form

```
<?php

    $validator->add('file', 'fileUnderFormSizeLimit', [
        'rule' => 'isUnderFormSizeLimit',
        'message' => 'This file is too large',
        'provider' => 'upload'
    ]);

?>
```

isCompletedUpload

Check that the file was completely uploaded

```
<?php

    $validator->add('file', 'fileCompletedUpload', [
        'rule' => 'isCompletedUpload',
        'message' => 'This file could not be uploaded completely',
        'provider' => 'upload'
    ]);

?>
```

isFileUpload

Check that a file was uploaded

```
<?php

    $validator->add('file', 'fileFileUpload', [
        'rule' => 'isFileUpload',
        'message' => 'There was no file found to upload',
        'provider' => 'upload'
    ]);

?>
```

isSuccessfulWrite

Check that the file was successfully written to the server

```
<?php

    $validator->add('file', 'fileSuccessfulWrite', [
        'rule' => 'isSuccessfulWrite',
        'message' => 'This upload failed',
        'provider' => 'upload'
    ]);

?>
```

isBelowMaxSize

Check that the file is below the maximum file upload size (checked in bytes)

```
<?php

    $validator->add('file', 'fileBelowMaxSize', [
        'rule' => ['isBelowMaxSize', 1024],
        'message' => 'This file is too large',
        'provider' => 'upload'
    ]);

?>
```

isAboveMinSize

Check that the file is above the minimum file upload size (checked in bytes)

```
<?php

    $validator->add('file', 'fileAboveMinSize', [
        'rule' => ['isAboveMinSize', 1024],
        'message' => 'This file is too small',
        'provider' => 'upload'
    ]);

?>
```

5.3 ImageValidation

isAboveMinHeight

Check that the file is above the minimum height requirement (checked in pixels)

```
<?php

    $validator->add('file', 'fileAboveMinHeight', [
        'rule' => ['isAboveMinHeight', 200],
        'message' => 'This image should at least be 200px high',
        'provider' => 'upload'
    ]);

?>
```

isBelowMaxHeight

Check that the file is below the maximum height requirement (checked in pixels)

```
<?php

    $validator->add('file', 'fileBelowMaxHeight', [
        'rule' => ['isBelowMaxHeight', 200],
        'message' => 'This image should not be higher than 200px',
        'provider' => 'upload'
    ]);

?>
```

isAboveMinWidth

Check that the file is above the minimum width requirement (checked in pixels)

```
<?php

    $validator->add('file', 'fileAboveMinWidth', [
        'rule' => ['isAboveMinWidth', 200],
        'message' => 'This image should at least be 200px wide',
        'provider' => 'upload'
    ]);

?>
```

isBelowMaxWidth

Check that the file is below the maximum width requirement (checked in pixels)

```
<?php

    $validator->add('file', 'fileBelowMaxWidth', [
        'rule' => ['isBelowMaxWidth', 200],
        'message' => 'This image should not be wider than 200px',
        'provider' => 'upload'
    ]);

?>
```

CHAPTER 6

Upload Plugin Interfaces

For advanced usage of the upload plugin, you will need to implement one or more of the followng interfaces.

6.1 ProcessorInterface

Fully-namespaced class name: `Josegonzalez\Upload\File\Path\ProcessorInterface`

This interface is used to create a class that knows how to build paths for a given file upload. Other than the constructor, it contains two methods:

- `basepath`: Returns the basepath for the current field/data combination
- `filename`: Returns the filename for the current field/data combination

Refer to `Josegonzalez\Upload\File\Path\DefaultProcessor` for more details.

6.2 TransformerInterface

Fully-namespaced class name: `Josegonzalez\Upload\File\Transformer\TransformerInterface`

This interface is used to transform the uploaded file into one or more files that will be written somewhere to disk. This can be useful in cases where you may wish to use an external library to extract thumbnails or create PDF previews. The previous image manipulation functionality should be created at this layer.

Other than the constructor, it contains one method:

- `transform`: Returns an array of key/value pairs, where the key is a file on disk and the value is the name of the output file. This can be used for properly naming uploaded/created files.

Refer to `Josegonzalez\Upload\File\Transformer\DefaultTransformer` for more details. You may **also** wish to look at `Josegonzalez\Upload\File\Transformer\SlugTransformer` as an alternative.

6.3 WriterInterface

Fully-namespaced class name: `Josegonzalez\Upload\File\Writer\WriterInterface`

This interface is used to actually write files to disk. It writes files to disk using the `Flysystem` library, and defaults to local storage by default. Implement this interface if you want to customize the file writing process.

Other than the constructor, it contains one methods:

- `write`: Writes a set of files to an output.

Refer to `Josegonzalez\Upload\File\Writer\DefaultWriter` for more details.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search