CakePHP-Upload Documentation

Release 1.3.0

Jose Diaz-Gonzalez

September 13, 2016

Contents:

CHAPTER 1

Introduction

1.1 Upload Plugin 2.0

The Upload Plugin is an attempt to sanely upload files using techniques garnered from packages such as MeioUpload , UploadPack and PHP documentation.

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 2.x
- Imagick/GD PHP Extension (for thumbnail creation)
- PHP 5
- 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 primaryKey field.
- It can also move files around programatically. Such as from the filesystem.
- The path to which the files are saved can be customised.
- It can also create thumbnails for image files if the `thumbnails` option is set in the behaviour options.
- The plugin can also upload multiple files at the same time to different fields.
- Each upload field can be configured independantly of each other, such as changing the upload path or thumbnail options.
- Uploaded file information can be stored in a data store, such as a MySQL database.

- A variety of validation rules are provided to help validate against common rules.

1.5 This plugin does not do

- 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

2.1 Using Composer

View on Packagist, and copy the json snippet for the latest version into your project's `composer.json`. Eg, v. 2.x-dev would look like this:

```
{
        "require": {
                "josegonzalez/cakephp-upload": "2.x-dev"
        }
}
```

This plugin has the type `cakephp-plugin` set in its own `composer.json`, composer knows to install it inside your `/Plugins` directory, rather than in the usual vendors file. It is recommended that you add `/Plugins/Upload` to your .gitignore file. (Why? read this.)

2.2 Manual

- Download this: https://github.com/josegonzalez/cakephp-upload/archive/2.x.zip
- Unzip that download.
- Copy the resulting folder to `app/Plugin`
- Rename the folder you just copied to `Upload`

2.3 GIT Submodule

In your *app directory* type:

```
git submodule add -b 2.x git://github.com/josegonzalez/cakephp-upload.git Plugin/Upload
git submodule init
git submodule update
```

2.4 GIT Clone

In your `Plugin` directory type:

```
git clone -b 2.x git://github.com/josegonzalez/cakephp-upload.git Upload
```

CHAPTER 3

Imagick Support

To enable Imagick support, you need to have Imagick installed:

```
# Debian systems
sudo apt-get install php5-imagick

# OS X Homebrew
brew tap homebrew/dupes
brew tap josegonzalez/homebrew-php
brew install php54-imagick

# From pecl
pecl install imagick
```

If you cannot install Imagick, instead configure the plugin with `'thumbnailMethod' => 'php'` in the files options.

CHAPTER 4

Enable plugin

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

```
<?php
CakePlugin::load('Upload');
```

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

CHAPTER 5

Examples

5.1 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
class User extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'photo'
        )
    );
}
?>
```

```
<?php echo $this->Form->create('User', array('type' => 'file')); ?>
<?php echo $this->Form->input('User.username'); ?>
<?php echo $this->Form->input('User.photo', array('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
class User extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'photo' => array(
                'fields' => array(
                    'dir' => 'photo_dir'
                )
            )
        )
    );
}
?>
```

In the above example, photo can be a file upload via a file input within a form, a file grabber (from a url) via a text input, OR programatically used on the controller to file grab via a url.

5.1.1 File Upload Example

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

5.1.2 File Grabbing via Form Example

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

5.1.3 Programmatic File Retrieval without a Form

```
<?php
$this->User->set(array('photo' => $image_url));
$this->User->save();
?>
```

5.1.4 Thumbnail Creation

Thumbnails are not automatically created. To do so, thumbnail sizes must be defined: Note: by default thumbnails will be generated by imagick, if you want to use GD you need to set the thumbnailMethod attribute. Example: `'thumbnailMethod' => 'php'`.

```
<?php
class User extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'photo' => array(
                'thumbnailSizes' => array(
                    'xvga' => '1024x768',
                    'vga' => '640x480',
                    'thumb' => '80x80'
                )
            )
        )
    );
}
?>
```

5.2 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
$exampleData = [
    'Example' => [
        'image' => 'imageFile.jpg',
        'dir' => '7'
    ]
];

echo $this->Html->link('../files/example/image/' . $exampleData['Example']['dir'] . '/' . $exampleDat
?>
```

If we have configured a thumbnail in our application. We can simply prefix our file with the name of that thumbnail.

```
<?php
echo $this->Html->link('../files/example/image/' . $exampleData['Example']['dir'] . '/thumb_' . $exan
?>
```

5.3 Uploading Multiple files

Multiple files can also be attached to a single record:

```
<?php
class User extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'resume',
            'photo' => array(
                'fields' => array(
                    'dir' => 'profile_dir'
                )
            )
        )
    );
}
?>
```

Each key in the `Upload.Upload` array is a field name, and can **contain it's own configuration**. For example, you might want to set different fields for storing file paths:

```
<?php
class User extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'resume' => array(
                'fields' => array(
                    'dir' => 'resume_dir',
                    'type' => 'resume_type',
                    'size' => 'resume_size',
                )
            ),
            'photo' => array(
                'fields' => array(
                    'dir' => 'photo_dir',
                    'type' => 'photo_type',
                    'size' => 'photo_size',
                )
            )
        )
    );
}
?>
```

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.

If you are looking to add an unknown or high number of uploads to a model it's worth considering using a polymorphic attachment.

5.4 Remove a current file without deleting the entire record

In some cases you might want to remove a photo or uploaded file without having to remove the entire record from the Model. In this case you would use the following code:

```
<?php
echo $this->Form->create('Model', array('type' => 'file'));
echo $this->Form->input('Model.file.remove', array('type' => 'checkbox', 'label' => 'Remove existing
?>
```

5.5 Saving two uploads into different folders

Sometimes you might want to upload more than one file, but upload each file into a different folder. This is actually very simple. By simply using the behavior configuration for *each file* you can change the path. Don't forget to make sure the plugin is installed first.

Let's assume for this example that we want to upload a picture of a user, and say, a picture of their car. For the sake of simplicity we'll also assume that these files are just stored in the `User` model.

Note: It's important to notice that each field can have it's own configuration.

```
<?php
// app/Model/User.php
public $actsAs = array(
    'Upload.Upload' => array(
        'avatar' => array( // The name of the field in our database, so this is `users.avatar`
            'rootDir' => ROOT, // Here we can define the rootDir, which is the root of the applicatio
            'path' => '{ROOT}{DS}webroot{DS}files{DS}{model}{DS}{field}{DS}', // The path pattern tha
            'fields' => array(
                'dir' => 'image_dir' // It's always helpful to save the directory our files are in, j
            )
        ),
        'car' => array(
            'path' => '{ROOT}{DS}webroot{DS}files{DS}cars{DS}' // Here we have changed the path, so c
        )
    )
)
```

5.6 Changing the upload path dynamically

If you need to change the path of the upload dynamically you can do that by changing the behavior settings in your model. Perhaps in a model callback such as `beforeSave()`.

```
<?php
// app/Model/User.php
$this->Behaviors->Upload->settings['field']['path'] = $newPath;
?>
```

CHAPTER 6

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.

- `pathMethod`: The method to use for file paths. This is appended to the `path` option below
  - Default: (string) `primaryKey`
  - Options:
    - `flat`: Does not create a path for each record. Files are moved to the value of the 'path' option.
    - `primaryKey`: Path based upon the record's primaryKey is generated. Persists across a record update.
    - `random`: Random path is generated for each file upload in the form `nn/nn/nn` where `nn` are random numbers. Does not persist across a record update.
    - `randomCombined`: Random path - with model id - is generated for each file upload in the form `ID/nn/nn/nn` where `ID` is the current model's ID and `nn` are random numbers. Does not persist across a record update.
- `path`: A path relative to the `rootDir`. Should end in `{DS}`
  - Default: (string) `'{ROOT}webroot{DS}files{DS}{model}{DS}{field}{DS}'`
  - Tokens:
    - {ROOT}: Replaced by a `rootDir` option
    - {DS}: Replaced by a `DIRECTORY_SEPARATOR`
    - {model}: Replaced by the Model Alias.
    - {field}: Replaced by the field name.
    - {primaryKey}: Replaced by the record primary key, when available. If used on a new record being created, will have undefined behavior.
    - {size}: Replaced by a zero-length string (the empty string) when used for the regular file upload path. Only available for resized thumbnails.
    - {geometry}: Replaced by a zero-length string (the empty string) when used for the regular file upload path. Only available for resized thumbnails.
- `fields`: An array of fields to use when uploading files
  - Default: (array) `array('dir' => 'dir', 'type' => 'type', 'size' => 'size')`
  - Options:

    * dir: Field to use for storing the directory
    * type: Field to use for storing the filetype
    * size: Field to use for storing the filesize

- `rootDir`: Root directory for moving images. Auto-prepended to `path` and `thumbnailPath` where necessary
  - Default (string) `ROOT . DS . APP_DIR . DS`
- `mimetypes`: Array of mimetypes to use for validation
  - Default: (array) empty
- `extensions`: Array of extensions to use for validation
  - Default: (array) empty
- `maxSize`: Max filesize in bytes for validation
  - Default: (int) `2097152`
- `minSize`: Minimum filesize in bytes for validation
  - Default: (int) `8`
- `maxHeight`: Maximum image height for validation
  - Default: (int) `0`
- `minHeight`: Minimum image height for validation
  - Default: (int) `0`
- `maxWidth`: Maximum image width for validation
  - Default: (int) `0`
- `minWidth`: Minimum image width for validation
  - Default: (int) `0`
- `deleteOnUpdate`: Whether to delete files when uploading new versions (potentially dangerous due to naming conflicts)
  - Default: (boolean) `false`
- `thumbnails`: Whether to create thumbnails or not
  - Default: (boolean) `true`
- `thumbnailMethod`: The method to use for resizing thumbnails
  - Default: (string) `imagick`
  - Options:
    * imagick: Uses the PHP `imagick` extension to generate thumbnails
    * php: Uses the built-in PHP methods (`GD` extension) to generate thumbnails. Does not support BMP images.
- `thumbnailName`: Naming style for a thumbnail
  - Default: `NULL`
  - Note: The tokens `{size}`, `{geometry}` and `{filename}` are valid for naming and will be auto-replaced with the actual terms.

  - Note: As well, the extension of the file will be automatically added.
  - Note: When left unspecified, will be set to `{size}_{filename}` or `{filename}_{size}` depending upon the value of `thumbnailPrefixStyle`

- `thumbnailPath`: A path relative to the `rootDir` where thumbnails will be saved. Should end in `{DS}`. If not set, thumbnails will be saved at `path`.
  - Default: `NULL`
  - Tokens:
    - {ROOT}: Replaced by a `rootDir` option
    - {DS}: Replaced by a `DIRECTORY_SEPARATOR`
    - {model}: Replaced by the Model Alias
    - {field}: Replaced by the field name
    - {size}: Replaced by the size key specified by a given `thumbnailSize`
    - {geometry}: Replaced by the geometry value specified by a given `thumbnailSize`
- `thumbnailPrefixStyle`: Whether to prefix or suffix the style onto thumbnails
  - Default: (boolean) `true` prefix the thumbnail
  - Note that this overrides `thumbnailName` when `thumbnailName` is not specified in your config
- `thumbnailQuality`: Quality of thumbnails that will be generated, on a scale of 0-100. Not supported gif images when using GD for image manipulation.
  - Default: (int) `75`
- `thumbnailSizes`: Array of thumbnail sizes, with the size-name mapping to a geometry
  - Default: (array) empty
- `thumbnailType`: Override the type of the generated thumbnail
  - Default: (mixed) `false` or `png` when the upload is a Media file
  - Options:
    - Any valid image type
- `mediaThumbnailType`: Override the type of the generated thumbnail for a non-image media (`pdfs`). Overrides `thumbnailType`
  - Default: (mixed) `png`
  - Options:
    - Any valid image type
- `saveDir`: Can be used to turn off saving the directory
  - Default: (boolean) `true`
  - Note: Because of the way in which the directory is saved, if you are using a `pathMethod` other than flat and you set `saveDir` to false, you may end up in situations where the file is in a location that you cannot predict. This is more of an issue for a `pathMethod` of `random` and `randomCombined` than `primaryKey`, but keep this in mind when fiddling with this option
- `deleteFolderOnDelete`: Delete folder related to current record on record delete
  - Default: (boolean) `false`

  - Note: Because of the way in which the directory is saved, if you are using a `pathMethod` of flat, turning this setting on will delete all your images. As such, setting this to true can be potentially dangerous.

- `keepFilesOnDelete`: Keep *all* files when uploading/deleting a record.
  - Default: (boolean) `false`
  - Note: This does not override `deleteFolderOnDelete`. If you set that setting to true, your images may still be deleted. This is so that existing uploads are not deleted - unless overwritten.
- `mode`: The UNIX permissions to set on the created upload directories.
  - Default: (integer) `0777`
- `handleUploadedFileCallback`: If set to a method name available on your model, this model method will handle the movement of the original file on disk. Can be used in conjunction with `thumbnailMethod` to store your files in alternative locations, such as S3.
  - Default: `NULL`
  - Available arguments:
    - `string $field`: Field being manipulated
    - `string $filename`: The filename of the uploaded file
    - `string $destination`: The configured destination of the moved file
- `nameCallback`: A callback that can be used to rename a file. Currently only handles original file naming.
  - Default: `NULL`
  - Available arguments:
    - `string $field`: Field being manipulated
    - `string $currentName`
    - `array $data`
    - `array options`:
      - `isThumbnail` - a boolean field that is on when we are trying to infer a thumbnail path
      - `rootDir` - root directory to replace `{ROOT}`
      - `geometry`
      - `size`
      - `thumbnailType`
      - `thumbnailName`
      - `thumbnailMethod`
      - `mediaThumbnailType`
      - `dir` field name
      - `saveType` - create, update, delete
  - Return: String - returns the new name for the file

Using a polymorphic attachment model for file storage

In some cases you will want to store multiple file uploads for multiple models, but will not want to use multiple tables because your database is normalized. For example, we might have a `Post` model that can have many images for a gallery, and a `Message` model that has many videos. In this case, we would use an `Attachment` model:

Post hasMany Attachment

We could use the following database schema for the `Attachment` model:

```
CREATE table attachments (
    `id` int(10) unsigned NOT NULL auto_increment,
    `model` varchar(20) NOT NULL,
    `foreign_key` int(11) NOT NULL,
    `name` varchar(32) NOT NULL,
    `attachment` varchar(255) NOT NULL,
    `dir` varchar(255) DEFAULT NULL,
    `type` varchar(255) DEFAULT NULL,
    `size` int(11) DEFAULT 0,
    `active` tinyint(1) DEFAULT 1,
    PRIMARY KEY (`id`)
);
```

Our attachment records would thus be able to have a name and be activated or deactivated on the fly. The schema is simply an example, and such functionality would need to be implemented within your application.

Once the `attachments` table has been created, we would create the following model:

```
<?php
class Attachment extends AppModel {
    public $actsAs = array(
        'Upload.Upload' => array(
            'attachment' => array(
                'thumbnailSizes' => array(
                    'xvga' => '1024x768',
                    'vga' => '640x480',
                    'thumb' => '80x80',
                ),
            ),
        ),
    );

    public $belongsTo = array(
        'Post' => array(
            'className' => 'Post',
            'foreignKey' => 'foreign_key',
        ),
        'Message' => array(
            'className' => 'Message',
            'foreignKey' => 'foreign_key',
        ),
    );
}
?>
```

We would also need to create a valid inverse relationship in the `Post` model:

```
<?php
class Post extends AppModel {
    public $hasMany = array(
        'Image' => array(
            'className' => 'Attachment',
            'foreignKey' => 'foreign_key',
            'conditions' => array(
                'Image.model' => 'Post',
            ),
        ),
    );
}
?>
```

The key thing to note here is the `Post` model has some conditions on the relationship to the `Attachment` model, where the `Image.model` has to be `Post`. Remember to set the `model` field to `Post`, or whatever model it is you'd like to attach it to, otherwise you may get incorrect relationship results when performing find queries.

We would also need a similar relationship in our `Message` model:

```
<?php
class Message extends AppModel {
    public $hasMany = array(
        'Video' => array(
            'className' => 'Attachment',
            'foreignKey' => 'foreign_key',
            'conditions' => array(
                'Video.model' => 'Message',
            ),
        ),
    );
}
?>
```

Now that we have our models setup, we should create the proper actions in our controllers. To keep this short, we shall only document the Post model:

```
<?php
class PostsController extends AppController {
    /* the rest of your controller here */
    public function add() {
        if ($this->request->is('post')) {
            try {
                $this->Post->createWithAttachments($this->request->data);
                $this->Session->setFlash(__('The message has been saved'));
            } catch (Exception $e) {
                $this->Session->setFlash($e->getMessage());
            }
        }
    }
}
?>
```

In the above example, we are calling our custom `createWithAttachments` method on the `Post` model. This will allow us to unify the Post creation logic together in one place. That method is outlined below:

```
<?php
class Post extends AppModel {
    /* the rest of your model here */

    public function createWithAttachments($data) {
        // Sanitize your images before adding them
        $images = array();
        if (!empty($data['Image'][0])) {
            foreach ($data['Image'] as $i => $image) {
                if (is_array($data['Image'][$i])) {
                    // Force setting the `model` field to this model
                    $image['model'] = 'Post';

                    // Unset the foreign_key if the user tries to specify it
                    if (isset($image['foreign_key'])) {
                        unset($image['foreign_key']);
                    }

                    $images[] = $image;
                }
            }
        }
        $data['Image'] = $images;

        // Try to save the data using Model::saveAll()
        $this->create();
        if ($this->saveAll($data)) {
            return true;
        }

        // Throw an exception for the controller
        throw new Exception(__("This post could not be saved. Please try again"));
    }
}
?>
```

The above model method will:

- Ensure we only try to save valid images
- Force the foreign_key to be unspecified. This will allow saveAll to properly associate it
- Force the model field to `Post`

Now that this is set, we just need a view for our controller. A sample view for `View/Posts/add.ctp` is as follows (fields not necessary for the example are omitted):

```
<?php
    echo $this->Form->create('Post', array('type' => 'file'));
    echo $this->Form->input('Image.0.attachment', array('type' => 'file', 'label' => 'Image'));
    echo $this->Form->input('Image.0.model', array('type' => 'hidden', 'value' => 'Post'));
    echo $this->Form->end(__('Add'));
?>
```

The one important thing you'll notice is that I am not referring to the `Attachment` model as `Attachment`, but rather as `Image`; when I initially specified the `$hasMany` relationship between an `Attachment` and a `Post`, I aliased `Attachment` to `Image`. This is necessary for cases where many of your Polymorphic models may be related to each other, as a type of *hint* to the CakePHP ORM to properly reference model data.

I'm also using `Model.{n}.field` notation, which would allow you to add multiple attachment records to the Post. This is necessary for `$hasMany` relationships, which we are using for this example.

Once you have all the above in place, you'll have a working Polymorphic upload!

Please note that this is not the only way to represent file uploads, but it is documented here for reference.

Thumbnail Sizes and Styles

The Upload plugin can automatically generate various thumbnails at different sizes for you when uploading files. The thumbnails must be configured in order for thumbnails to be generated.

To generate thumbnails you will need to configure the `thumbnailSizes` option under the field you are configuring.

```
<?php
class User extends AppModel {
    public $name = 'User';
    public $actsAs = array(
        'Upload.Upload' => array(
            'photo' => array( // The field we are configuring for
                'thumbnailSizes' => array( // Various sizes of thumbnail to generate
                    'big' => '200x200', // Resize for best fit to 200px by 200px, cropped from the ce
                    'small' => '120x120',
                    'thumb' => '80x80'
                )
            )
        )
    );
}
?>
```

Once this configuration is set when uploading a file a thumbnail will automatically be generated with the prefix defined in the options. For example (using default configuration) `app/webroot/files/Example/photo/1/big_example.jpg`. Where `Example` is the model, `photo` is the field, `1` is the model primaryKey value and finally `big_` is the thumbnail size prefix to the filename.

Thumbnail sizes only apply to images of the following types:

- image/bmp
- image/gif
- image/jpeg
- image/pjpeg
- image/png
- image/vnd.microsoft.icon
- image/x-icon

You can specify any of the following resize modes for your sizes:

- `100x80` - resize for best fit into these dimensions, with overlapping edges trimmed if original aspect ratio differs

- `[100x80]` - resize to fit these dimensions, with white banding if original aspect ratio differs
- `100w` - maintain original aspect ratio, resize to 100 pixels wide
- `80h` - maintain original aspect ratio, resize to 80 pixels high
- `80l` - maintain original aspect ratio, resize so that longest side is 80 pixels
- `600mw` - maintain original aspect ratio, resize to max 600 pixels wide, or copy the original image if it is less than 600 pixels wide
- `800mh` - maintain original aspect ratio, resize to max 800 pixels high, or copy the original image if it is less than 800 pixels high
- `960ml` - maintain original aspect ratio, resize so that longest side is max 960 pixels, or copy the original image if the thumbnail would be bigger than the original.

8.1 PDF Support

It is now possible to generate a thumbnail for the first page of a PDF file. (Only works with the `imagick` `thumbnailMethod`.) Please read about the Behavior options for more details as to how to configure this plugin.

CHAPTER 9

Validation rules

By default, no validation rules are attached to the model. You must explicitly attach each rule if needed. Rules not referring to PHP upload errors are configurable but fallback to the behavior configuration.

9.1 isUnderPhpSizeLimit

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isUnderPhpSizeLimit',
        'message' => 'File exceeds upload filesize limit'
    )
);
?>
```

9.2 isUnderFormSizeLimit

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isUnderFormSizeLimit',
        'message' => 'File exceeds form upload filesize limit'
    )
);
?>
```

9.3 isCompletedUpload

Check that the file was completely uploaded

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isCompletedUpload',
        'message' => 'File was not successfully uploaded'
    )
);
?>
```

9.4 isFileUpload

Check that a file was uploaded

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isFileUpload',
        'message' => 'File was missing from submission'
    )
);
?>
```

9.5 isFileUploadOrHasExistingValue

Check that either a file was uploaded, or the existing value in the database is not blank

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isFileUploadOrHasExistingValue',
        'message' => 'File was missing from submission'
    )
);
?>
```

9.6 tempDirExists

Check that the PHP temporary directory is missing

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'tempDirExists',
        'message' => 'The system temporary directory is missing'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('tempDirExists', false),
        'message' => 'The system temporary directory is missing'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.7 isSuccessfulWrite

Check that the file was successfully written to the server

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'isSuccessfulWrite',
        'message' => 'File was unsuccessfully written to the server'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isSuccessfulWrite', false),
        'message' => 'File was unsuccessfully written to the server'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.8 noPhpExtensionErrors

Check that a PHP extension did not cause an error

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => 'noPhpExtensionErrors',
        'message' => 'File was not uploaded because of a faulty PHP extension'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('noPhpExtensionErrors', false),
        'message' => 'File was not uploaded because of a faulty PHP extension'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.9 isValidMimeType

Check that the file is of a valid mimetype

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidMimeType', array('application/pdf', 'image/png')),
        'message' => 'File is not a pdf or png'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidMimeType', array('application/pdf', 'image/png'), false),
        'message' => 'File is not a pdf or png'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.10 isWritable

Check that the upload directory is writable

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isWritable'),
        'message' => 'File upload directory was not writable'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isWritable', false),
        'message' => 'File upload directory was not writable'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.11 isValidDir

Check that the upload directory exists

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidDir'),
        'message' => 'File upload directory does not exist'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidDir', false),
        'message' => 'File upload directory does not exist'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.12 isBelowMaxSize

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxSize', 1024),
        'message' => 'File is larger than the maximum filesize'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxSize', 1024, false),
        'message' => 'File is larger than the maximum filesize'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.13 isAboveMinSize

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinSize', 1024),
        'message' => 'File is below the mimimum filesize'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinSize', 1024, false),
        'message' => 'File is below the mimimum filesize'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.14 isValidExtension

Check that the file has a valid extension

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidExtension', array('pdf', 'png', 'txt')),
        'message' => 'File does not have a pdf, png, or txt extension'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isValidExtension', array('pdf', 'png', 'txt'), false),
        'message' => 'File does not have a pdf, png, or txt extension'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.15 isAboveMinHeight

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinHeight', 150),
        'message' => 'File is below the minimum height'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinHeight', 150, false),
        'message' => 'File is below the minimum height'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.16 isBelowMaxHeight

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxHeight', 150),
        'message' => 'File is above the maximum height'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxHeight', 150, false),
        'message' => 'File is above the maximum height'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.17 isAboveMinWidth

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinWidth', 150),
        'message' => 'File is below the minimum width'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isAboveMinWidth', 150, false),
        'message' => 'File is below the minimum width'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

9.18 isBelowMaxWidth

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

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxWidth', 150),
        'message' => 'File is above the maximum width'
    )
);
?>
```

If the argument `$requireUpload` is passed, we can skip this check when a file is not uploaded:

```
<?php
public $validate = array(
    'photo' => array(
        'rule' => array('isBelowMaxWidth', 150, false),
        'message' => 'File is above the maximum width'
    )
);
?>
```

In the above, the variable `$requireUpload` has a value of false. By default, `requireUpload` is set to true.

CHAPTER 10

FileImportBehavior

`FileImportBehavior` may be used to import files directly from the disk. This is useful in importing from a directory already on the filesystem.

CHAPTER 11

Thumbnail shell

11.1 What it does

The shell will look through your database for images and regenerate the thumbnails based on your models Upload behaviour configuration. This allows you to change your thumbnail configuration and run the shell to update your images without having to re-upload the image.

11.2 How it works

The shell takes the model you provide and checks that the Upload plugin is present and configured. Then it will loop though all the images checking that the configured upload field is populated in the database and also ensuring that the file exists on the file system. Then it will regenerate the thumbnails using the current model configuration.

11.3 Running the shell

You can run the shell from the command line as you would any cake shell.

```
Console/cake upload.thumbnail generate
```

You will then be asked which model you want to process, and the shell will then process your images.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search