Aloe CMS Documentation

[Introduction 4](#_Toc355701661)

[Database Access 5](#_Toc355701662)

[Selecting 5](#_Toc355701663)

[Inserting, Updating and Deleting 5](#_Toc355701664)

[Executing 6](#_Toc355701665)

[Transactions 6](#_Toc355701666)

[Where Strings 6](#_Toc355701667)

[Friendly URLs 8](#_Toc355701668)

[Table Structure 8](#_Toc355701669)

[The Request Process 8](#_Toc355701670)

[Finding the Friendly URL 8](#_Toc355701671)

[Implementing Friendly URLs in Modules 9](#_Toc355701672)

[Resources 11](#_Toc355701673)

[Using the Resource Manager 11](#_Toc355701674)

[Settings 14](#_Toc355701675)

[Global Settings 14](#_Toc355701676)

[Page Settings 15](#_Toc355701677)

[Layout Settings 16](#_Toc355701678)

[Core Content 17](#_Toc355701679)

[Menus 17](#_Toc355701680)

[Pages 17](#_Toc355701681)

[Layouts 18](#_Toc355701682)

[Defining Layouts 18](#_Toc355701683)

[Managing Layouts 20](#_Toc355701684)

[Columns 20](#_Toc355701685)

[Content Blocks 20](#_Toc355701686)

[The structure of a Content Block 21](#_Toc355701687)

[The Class File 21](#_Toc355701688)

[The Config File 23](#_Toc355701689)

[The Editor 23](#_Toc355701690)

[Specific Content Blocks 26](#_Toc355701691)

[CMRawHTML 26](#_Toc355701692)

[CMStandard 26](#_Toc355701693)

[CMListItem 26](#_Toc355701694)

[CMFeature 27](#_Toc355701695)

[CMLinkList 27](#_Toc355701696)

[CMIndex 27](#_Toc355701697)

[CMNewsFeed 27](#_Toc355701698)

[CMRSSFeed 27](#_Toc355701699)

[CMPhotoGallery 27](#_Toc355701700)

[CMLightboxGallery 28](#_Toc355701701)

[CMSlideShow 28](#_Toc355701702)

[CMContact 28](#_Toc355701703)

[CMEnquiry 29](#_Toc355701704)

[CMSpacer 29](#_Toc355701705)

[CMTable 29](#_Toc355701706)

[Skins 30](#_Toc355701707)

[The Skin Extension Class 30](#_Toc355701708)

[Skin Templates 32](#_Toc355701709)

[Skin Files 32](#_Toc355701710)

[Customising Modules and Content Blocks 32](#_Toc355701711)

[Preparing a Skin for the Content Module 33](#_Toc355701712)

[Aloe Templating Engine 34](#_Toc355701713)

[Replacing Variables 34](#_Toc355701714)

[Additional Tags 34](#_Toc355701715)

[Settings 34](#_Toc355701716)

[Pageobject 34](#_Toc355701717)

[SEOTags 35](#_Toc355701718)

[Content 35](#_Toc355701719)

[If elseif else 35](#_Toc355701720)

[Call 36](#_Toc355701721)

[Contentcontainer 36](#_Toc355701722)

[Menu 36](#_Toc355701723)

[Processing Templates 36](#_Toc355701724)

[Lists 38](#_Toc355701725)

[Modules 38](#_Toc355701726)

[News 38](#_Toc355701727)

[Request – Response Path 38](#_Toc355701728)

[Controls 38](#_Toc355701729)

[Calendar 38](#_Toc355701730)

[Context Menu 38](#_Toc355701731)

[Gallery View 38](#_Toc355701732)

[Input Control 38](#_Toc355701733)

[Lightbox 38](#_Toc355701734)

[List View 38](#_Toc355701735)

[Menu 38](#_Toc355701736)

[Multi List 38](#_Toc355701737)

[Tiny MCE 38](#_Toc355701738)

[Tree View 38](#_Toc355701739)

[Popups 39](#_Toc355701740)

[Full Screen Popups 39](#_Toc355701741)

[Invisible 39](#_Toc355701742)

[Disabled 39](#_Toc355701743)

[Loading 40](#_Toc355701744)

[Completed 40](#_Toc355701745)

[Pre-defined Popups 40](#_Toc355701746)

[Error 40](#_Toc355701747)

[Message 40](#_Toc355701748)

[Resource Manager 40](#_Toc355701749)

[Image Selector 41](#_Toc355701750)

[Document Selector 41](#_Toc355701751)

[Link Selector 41](#_Toc355701752)

[Colour Selector 41](#_Toc355701753)

[Custom Popups 42](#_Toc355701754)

[Creating a popup 42](#_Toc355701755)

[Displaying a popup 43](#_Toc355701756)

[Sizing, positioning and preparing the popup 43](#_Toc355701757)

[Interacting with a popup 44](#_Toc355701758)

[Error Pages 45](#_Toc355701759)

[Sending Emails 45](#_Toc355701760)

[Images 45](#_Toc355701761)

[AJAX 45](#_Toc355701762)

[Users 45](#_Toc355701763)

[Common Functions 45](#_Toc355701764)

[Installing 45](#_Toc355701765)

[Events 46](#_Toc355701766)

[Handling events 46](#_Toc355701767)

[Firing events 46](#_Toc355701768)

[Standard Events 46](#_Toc355701769)

[linkSelectorLoading 46](#_Toc355701770)

[sitemapLoading 47](#_Toc355701771)

[htmpathResetting 47](#_Toc355701772)

[Extending the Core Modules 48](#_Toc355701773)

# Introduction

# Database Access

Database access is provided by the DataAccess class. An instance of this class is created for every request in the Global.lib.php file, and added to the Global scope as $GLOBALS[‘db’], making it available from anywhere in the system. This instance connects to the primary database for the site using connection settings set in Settings.

The DataAccess object connects to a MySQL server using supplied server name, database (schema) name, username and password. Once connected, the connection is maintained for the duration of the request.

The DataAccess class currently uses the deprecated mysql\_ methods. For the most part, these are safe – they have been designed to be as safe to use as possible, with all the necessary checks. However, SELECT statements are all run as supplied, so it is the responsibility of the developer to ensure that all parameters passed with a SELECT statement are cleaned. We plan to update the class to using mysqli\_ library in the near future.

The DataAccess class assumes that all tables in the database include, as their primary key, an id field. This is used extensively to make inserting, updating and deleting data much simpler.

## Selecting

There are a number of methods for selecting data from the database. All of them require that the developer construct the entire select statement themselves, including any security protection. The most basic method is simply select, which takes a query and returns the entire dataset as an array of rows (also arrays) of data. The rest of the select functions are designed for more specific purposes, and basically package the results in slightly different ways:

|  |  |
| --- | --- |
| Method | Returns |
| select | A dataset (array of arrays) |
| selectsingle | The first value in the first row of the returned dataset |
| selectindex | An associative array of rows, with the supplied column used for the keys. If the values in the column are not unique, the last row to contain that key will be returned. It is expected that you use the primary key for the column. |
| selectrow | The first row in the resultset as an array. Often used to collect the details of a single item. |
| selectcolumn | An array containing the values of the supplied field from each row in the resultset. For example, this method is often used to collect the IDs from a table. |

## Inserting, Updating and Deleting

Specific methods are provided to perform the most common insert, update and delete actions. For more complex actions, use the execute method.

Insert data into a table with the insert method, passing it the name of the table and an associative array of values, with the keys representing the field names, and the values, the values to be inserted. The $handlenull parameter tells the method whether to handle null values as NULL, or as empty strings. The method returns the id field value of the inserted row.

Use the insertifnotexists method to insert values into a table if the record does not already exist. In addition to the parameters passed to the insert method, this method also takes an associative array of primary keys. These are used to identify whether or not the record already exists.

Update an existing row in a table using the update method. Pass this the name of the table, an associative array of the new values for the row, with the field names as keys, and an associative array of primary keys to identify the rows to be updated, again with field names as keys. If any row has the same values in its fields that the primary keys array has (it must correspond exactly with all values, but data type is not checked), then all the fields in the values array will be updated. No other fields will be touched. The $handlenull parameter tells the method whether to handle null values as NULL, or as empty strings.

The insertupdate method is provided to handle the situation where you want to update a row if it exists, or add it if it does not. This method takes pretty much the same parameters as the update method, but also allows you to choose which field you want returned, rather than just returning the id.

Delete rows from a table based on the value of a single field with the delete method. Pass the name of the table, the value of the field that identifies rows to be deleted and, if the field is not the id field, name the field as well. Most often used to delete a row by passing the id of that row.

## Executing

If more complex queries need to be processed, use the execute method. Pass the query to be executed, and receive the number of rows affected in return. It is your responsibility to ensure that your SQL is secure. This method cannot be used to select data, but it can execute any other query at all.

## Transactions

You can wrap a series of queries in a transaction, which allows you to roll back the whole set if a single error occurs. Only one transaction may be active at any one time – you cannot nest transactions. Start a transaction with the begintransaction method. If you need to roll back the transaction, call the rollbacktransaction method. Otherwise, always close your transaction with the committransaction method. If you do not, the system assumes an error, and the whole set of queries will be rolled back.

## Where Strings

It is often necessary to construct complex WHERE clauses for queries, and the DataAccess class has several methods to help you with this.

|  |  |
| --- | --- |
| Method | When to use it |
| getSearchStringWhere | Check a field against a string. Will match the string against any part of the value in the field using LIKE |
| getSearchStringExactWhere | Check a field against a string. The match must be exact. |
| getSearchIntWhere | Check a field against an integer |
| getSearchDateWhere | Check a field against a date. As with all dates in Aloe, the dates must be unix timestamps. The date can either be a start date (matching any date on or after the given) or an end date (matching any date on or before the given) |
| getSearchStringArrayWhere | Check a field against the strings in an array. Matches if the value in the field matches any one of the strings in the array. |
| getSearchIntArrayWhere | Check a field against the integers in an array. Matches if the value in the field matches any one of the integer in the array. |

Each method takes as one of its parameters the current WHERE clause, and returns that clause with the new statement added. If the value being checked is empty, the statement is not added, and the original WHERE clause is returned.

# Friendly URLs

Aloe includes a Friendly URL system which translates system URLs (eg: <http://yoursite.com/page.php?id=123>) into SEO and human friendly URLs (eg: <http://yoursite.com/information/interesting_stuff.htm>). It achieves this by maintaining a table in the database which maps the friendly URL to the corresponding processing file. This feature can be switched on or off with the usefriendlyurls global setting.

## Table Structure

The database table maps a path, relative to the site root, to a processing file and a set of three possible parameters. Each parameter is recorded with a name and a value. It is not required that all three parameters be filled. Some examples:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Path | Filename | Att1 | Val1 | Att2 | Val2 | Att3 | Val3 |
| information/interesting\_stuff.htm | index.php | id | 34 |  |  |  |  |
| news/2013/january/a\_story.htm | news.php | id | 234 |  |  |  |  |
| za/shoes/ladies.htm | products.php | loc | za | cat | shoes | sub | ladies |

## The Request Process

An incoming request makes its way to the correct processing files as follows:

The incoming request is translated by a .htaccess file, and transferred to redirect.php for processing. This file receives the original requested path as a variable (eg: information/interesting\_stuff.htm).

If the redirect.php file determines that there exists a php version of the requested file (eg: information/interesting\_stuff.php), then that file will be included, and the process stopped.

If not, the redirect.php will check the database to find the corresponding processing file, and the relevant parameters for it. The parameters will be set as global variables, and the processing file included. This means that the processing file should check for required parameters in both $\_REQUEST (or $\_POST) and $\_GLOBAL.

If no processing file can be found, a 404 error page is included, and the request closed.

## Finding the Friendly URL

Use the HTMPaths library to work with friendly URLs. To identify the friendly URL for a page, use the getHTMPath method. Pass this method the name of the processing module, or null for the core content module, the filename of the processing file and up to three parameters for this file, as a parameter name followed by the value for that parameter.

If the usefriendlyurls global setting is false, this method returns the URL for the processing file with the parameters appended as a query (eg: index.php?id=123).

Otherwise, if no parameters are passed, the processing file is simply translated into an .htm form (eg: index.php into index.htm), and returned.

Otherwise, if an entry exists in the database for this file already, that path is returned. The order of parameters does not matter when the database is searched.

Otherwise, if a processing module is defined, that module will be loaded, and the getHTMPath method of the Module class for that module is called.

Otherwise, the Content module will construct the friendly URL by concatenating the path stubs of all the pages in the tree down to the current page, and adding a .htm.

If all else fails, the method returns index.htm.

If the method had to construct the friendly URL, the result is saved into the database. The current global webroot variable is prepended to the URL before it is returned, allowing the path to be used as-is in HTML code. If you need the path relative to the site root, you need to strip the webroot from the URL.

If you are very sure the friendly URL has already been created, you can also find it using the getExistingHTMPath method. This does everything the getHTMPath method does, except that if no friendly URL is found in the database, none is generated, and index.htm is returned.

## Implementing Friendly URLs in Modules

When you create a new module, you must create a new class that extends the Module class. If your module provides pages for the site, and you want these pages to use the Friendly URL system, you need to override the getHTMPath method of the Module class to construct useful and unique friendly URLs for each of the pages managed by your module. Your method will receive a processing file (your module may have more than one processing file – for example, you may have one for product categories and another for product details) and the three possible parameters as parameters to the method. The HTMPaths library also has a couple of methods you can use to make your job easier:

|  |  |
| --- | --- |
| Method | Description |
| getHTMPathFromTitle | Constructs a URL from a given name and path, and then checks the database to see if that path already exists for another page. If so, it appends a number to the URL, and checks again, until the URL is unique. |
| createHTMPath | Adds the constructed URL to the database, with the processing file and associated parameters. |
| getCleanRoot | Recently moved to the Text library, this method will translate a sentence into a string that can be used for a URL. All spaces and other problematic characters will be removed or replaced with “\_”. |

Remember to call the createHTMPath method to ensure that your URLs are saved to the database, and to prepend the global webroot variable to the URL before you return it (but only after saving it, of course). An example implementation of the getHTMPath method, from the News module, is below:

public function getHTMPath(**$filename**, **$att1=**null, **$val1=**null, **$att2=**null, **$val2=**null, **$att3=**null, **$val3=**null){

require\_once('HTMPaths.lib.php');

try{

switch(**$att1**){

case 'id'**:**

**$itm** **=** getNewsItem(**$val1**);

**$htmpath** **=** getHTMPathFromTitle(**$itm**['title'], 'news/'**.**date('Y',**$itm**['date'])**.**'/'**.**date('F',**$itm**['date'])**.**'/', **$filename**, **$att1**, **$val1**, **$att2**, **$val2**, **$att3**, **$val3**);

break;

case 'cat'**:**

**$listmanager** **=** ListManager**::**getListManager();

**$category** **=** **$listmanager->**getListItem(**$val1**); if(empty(**$category**)) **$htmpath** **=** 'news.htm';

else{

**$htmpath** **=** getHTMPathFromTitle(**$category**['name'], 'news/', **$filename**, **$att1**, **$val1**, **$att2**, **$val2**, **$att3**, **$val3**);

}

break;

default**:**

**$htmpath** **=** 'news.htm';

break;

}

}catch(exception **$err**){

**$htmpath** **=** 'news.htm';

}

if(**!**empty(**$htmpath**)) {

createHTMPath(**$htmpath**,**$filename**,**$att1**,**$val1**,**$att2**,**$val2**,**$att3**,**$val3**,**$error**);

}

if(substr(**$htmpath**,0,4) **!=** 'http' **&&** substr(**$htmpath**,0,1) **!=** '/'){

**$htmpath** **=** **$GLOBALS**['webroot']**.$htmpath**;

**}**

return **$htmpath**;

}

# Resources

The Resource Manager is designed to manage additional files for the site (eg: images, pdf downloads, etc). It works by representing the file system below the /resources folder exactly, in the manner of Windows Explorer. It allows an admin to view and select a single file, or upload a file to use. In manage mode, the admin can also rename, delete, resize, optimise, refresh, move or copy one or more files as needed.

The Resource Manager is made up of a series of files:

|  |  |
| --- | --- |
| File | Description |
| /class/ResourceManager.class.php | The server-side entry point for the Resource Manager. Use the ResourceManager class to build and render the Resource Manager. |
| /class/ResourceTreeView.class.php | An extension of the TreeView control, customised for use with the Resource Manager |
| /ajax/ResourceManager.aim.php | Handles actual file uploads to the Resource Manager |
| /ajax/ResourceManager.ajax.php | AJAX library for all client-server interactions except actual file uploads. |
| /css/resourcemanager.css | The default CSS file. Extensions to the Resource Manager should either provide custom CSS in addition to this file or make sure that all the classes in this file are reproduced. |
| /js/ResourceManager.js | The Javascript library handling all client-side action and display. |

The Resource Manager is designed as an AJAX application, so once the initial page has rendered, all interaction is managed by Javascript, using AJAX to communicate with the server.

## Using the Resource Manager

A typical use of the Resource Manager might look something like this:

<link rel="StyleSheet" href="../css/resourcemanager.css" type="text/css" />

<script language="javascript" src="../js/ResourceManager.js"></script>

<?

**$resman** **=** new ResourceManager('all','icons',true);

**$resman->**prefix **=** 'rm';

**$resman->**mode **=** 'section';

function drawResourceManager(**$sourceid**, **$owner=**'', **$path=**'', **$type=**'all', **$display=**'details', **$mode=**'select'){

**$rm** **=** new ResourceManager(**$type**,**$display**);

**$rm->**prefix **=** 'rm';

**$rm->**mode **=** **$mode**;

return **$rm->**\_drawFiles(**$path**);

}

echo **$resman->**drawResourceManager();

?>

<script language="javascript">

var currenttype = 'all';

var currentdisplay = 'icons';

var currentmode = 'manage';

var currentpath = 'resources/';

var currentuid = 1;

var currentrights = 'rwt';

var rsv\_ajaxpath = '../ajax/ResourceManager.ajax.php';

var rsv\_webroot = '<?=$GLOBALS['webroot']?>';

var rsv\_prefix = 'rm';

rs\_init();

</script>

This is quite a complicated setup. It requires several elements to be set up:

* You must include the default CSS file in your code explicitly. You may also include any custom CSS files you may need.
* You must include the ResourceManager.js file explicitly.
* You must instantiate the ResourceManager class, set any settings you might need, and render the drawResourceManager method.
* You must also provide an implementation of the drawResourceManager method in the current page. This is used by the AJAX library to re-render the file list.
* Then you must set any javascript settings you need, and finally call the rs\_init function.

NOTE: this is a very complicated system, and will definitely be revised and simplified in the future.

There are a number of settings you can set, both on the server side with the initial class, and on the client side with the javascript:

|  |  |
| --- | --- |
| Class Variable | Description |
| root | The path to the root folder for the whole resources system, relative to the root folder of the site. By default, this is “resources/”. The trailing slash is required. |
| iconpath | The path to the folder containing the icons for the manager. This path will be prepended to the icon file and used on the front-end, so it should be a relative path from the current page. |
| selectedpath | The path to the currently selected file. The path is relative to the root above. |
| user | The user using the Resource Manager. Used for calculating permissions. By default, this is the current user. |
| display | Whether the initial display should be details or icons |
| prefix | The prefix to use for this Resource Manager, in case you need two managers on a single page |
| type | Whether to show images only (“images”), documents only (“docs”) or all files. Images are defined as files with extensions found in the “displayimagetypes” Global setting, and documents are those with extensions in the “displaydoctypes” setting. |
| mode | Whether the manager should be focussed on selecting a file or managing the files. May be “select”, ”manage” or “session” (which works like “manage”, but does not allow the user to switch between modes) |
| \_extensions | An array of file extensions allowed in the Resource Manager. Uses the Global setting “validfiletypes” by default |
| \_thumbextensions | An array of file extensions identifying files that can use thumbnails. By default, these are: 'jpg','jpeg','gif','png' |
| linkformat | In “select” mode, clicking on a file selects it. Use this setting to define the URL that is used when the item is clicked. By default, this is, “javascript:fbd\_insert('%s')”, and assumes the existence of a javascript function called “fbd\_insert” |
| selectedlinkformat | As above, but the URL to use for the currently selected file. By default, this is the same as the linkformat above. |
| folderlinkformat | The format of the link to use when a folder is clicked. By default, this is, “javascript:rs\_selectFolder('%s','%s')”, but you may want to add other actions to this click as well. |

|  |  |
| --- | --- |
| Javascript Variable | Description |
| currenttype | The current type |
| currentdisplay | The current display mode |
| currentmode | The current management mode. This is either “manage” or “select”. |
| currentpath | The currently selected path |
| currentuid | The current unique ID. This is an incremental integer, usually starting at 1. It is used to ensure files and folders are uniquely identifiable. |
| currentrights | The rights set on the current path for the current user |
| rsv\_ajaxpath | The relative path to the AJAX file |
| rsv\_webroot | The relative path to the root folder of the site |
| rsv\_prefix | The prefix used by this Resource Manager |

# Settings

Aloe offers three levels of settings – global, page and layout settings.

## Global Settings

Global settings are managed by the Settings class. An instance of this class is created in the Globals.lib.php file, and is available from the $GLOBALS[‘settings’] variable. This object will construct the set of global settings from several sources. The class itself has defaults for several settings. These are:

|  |  |  |
| --- | --- | --- |
| Setting | Default | Description |
| siteroot |  | The basic URL of the site, without the trailing slash (eg: <http://www.mysite.com>) |
| errorlevel | E\_ALL ^ E\_NOTICE | a value valid for the [error\_reporting](http://www.php.net/manual/en/function.error-reporting.php) function |
| timezone | Africa/Johannesburg | a value valid for the [date\_default\_timezone\_set](http://www.php.net/manual/en/function.date-default-timezone-set.php) function |
| defaulttitle |  | The default title to use on all pages of the site |
| dbserver |  | The server address for the database. Often localhost |
| dbdatabase |  | The schema name for the database |
| dbuser |  | The username to use to connect to the database |
| dbpassword |  | The password to use to connect to the database |
| sourceemail | gareth@blacksquare.co.za | The email address from which all emails in the system are sent. This should be a valid email address that is checked, because it will also be the default reply-to address. Some servers require that this address have the same domain as the website, so the default will be invalid. |
| adminemail | gareth@blacksquare.co.za | The email address of the system administrator |
| skin | Standard | The name of the current Skin. |

On instantiation, the class will load the config.php file, which may include any global settings the system administrator wants to define. These should be included in that file in this format:

**$config[**'dbserver'**]** **=** 'localhost';

At the very least, this file should include the following settings:

|  |  |
| --- | --- |
| Setting | Description |
| siteroot | The basic URL of the site, without the trailing slash (eg: <http://www.mysite.com>) |
| usefriendlyurls | Boolean, whether or not to use friendly URLs for this site |
| islivesite | Boolean, whether or not this is the live site. Dev sites have debug javascript tools and all emails sent through a dev site are redirected to the admin email address. Skins may also change other aspects of the front end depending on this setting. |
| skin | The name of the current Skin |
| dbserver | The server address for the database. Often localhost |
| dbdatabase | The schema name for the database |
| dbusername | The username to use to connect to the database |
| dbpassword | The password to use to connect to the database |
| sourceemail | The email address from which all emails in the system are sent. This should be a valid email address that is checked, because it will also be the default reply-to address. Some servers require that this address have the same domain as the website, so the default will be invalid. |

As a special case, path variables can be included with the setting name prefixed with a “@”, and the value being a path relative to the site root. This will create two settings; the first, named for the setting name without the @ will hold the absolute physical path, and the second will append “web” to the setting name, and be the path relative to the current page. For example, a setting:

**$config[**'@contentpath'**]** **=** '/content/';

Might create:

|  |  |
| --- | --- |
| setting | value |
| contentpath | /home/yoursite/public\_html/content/ |
| contentpathweb | ../content/ |

The config.php file can also be used to define global constants.

Once the config.php file is loaded and processed, the Settings object will create a new instance of the DataAccess class, and make that available through the $GLOBALS[‘db’] variable. The Settings object will then fetch all the settings in the settings table in the database. These are loaded in the same way as those from the config.php file, and will overwrite those if there is any overlap.

All settings are available as properties of the Settings object, and can be accessed in a number of ways, the most common being: **$GLOBALS[**'settings'**]->**skin.

Global settings that will be set once when the site is set up, like the database access settings, should be included in the config.php file. All other settings should be included in the database. These settings can be modified by the admin in the Global Settings tab of the System module in the CMS. All settings except those set as “hidden” in the database will be displayed and may be modified here, each according to its type.

Settings stored in the database can be one of thirteen types. These define the data type of the setting, and affect the way the data is offered to the admin for editing. These correspond exactly to the Input Control types, as this control is designed to manage the information in settings (global, page and layout)., with one exception; the ‘hidden’ data type holds settings that should not be available to the admin for modification.

Global settings can be added to the database by and Module, Content Block or Skin during installation using the addGlobalSetting method of the Install.lib.php library, and can be removed during uninstallation with the removeGlobalSetting method.

## Page Settings

Page settings allow each Page in the content system to have a different value for one or more custom settings. This allows the Skin, Module or Content Block that defined the setting to customise its output accordingly. Page settings are created during an installation using the addPageSetting method of the Install.lib.php library, and remove on uninstallation with the removePageSettings method.

Page settings are available from the custom property of a Page object, eg: **$mypage->**custom**[**'mysetting'**]**. The PopulateCustom method must previously have been called, but this is called automatically by the PopulateContent method.

## Layout Settings

Layout settings allow each individual Layout in the content system to have different values for one or more custom settings. This allows the Skin, Module or Content Block that defined the setting to customise its output accordingly. Layout settings are created during an installation using the add LayoutSetting method of the Install.lib.php library, and remove on uninstallation with the remove LayoutSettings method.

Layout settings are available through the custom property of the Layout object, eg: **$mylayout->**custom**[**'mysetting'**]**. The PopulateCustom method must previously have been called, but this is called automatically if the Layout has been added to the parent Page collection already.

# Core Content

The Aloe CMS system is structured around the core Content Module. This defines the menu structure of the site, and provides entry points to extended modules, as well as managing the primary content for the site.

## Menus

The primary component in the content structure of Aloe is the Page. Each page is assigned a parent page, and this link creates an hierarchical menu structure. Not all pages in the menu system represent actual pages on the site. There are in fact five types of page, and only one represents an actual page:

|  |  |  |
| --- | --- | --- |
| Type | type parameter | Description |
| Menu | special | Pages in the system that have null recorded as their parent are considered to be menu containers rather than pages. |
| Page | content | Represents an actual page, with content |
| Link | link | a link to an external resource |
| Label | label | A containing page – often used with more complex menus to allow a menu item to have text without linking anywhere. |
| Special | special | Special pages are usually the entry points for additional modules. Special pages can define different processing pages from the default used for content pages. |

Aloe comes with two menus by default; the Main Menu contains all the pages intended to appear in the menu on the site, and the Widows and Orphans menu contains pages that are never intended to appear in any menu. These include pages that will be directly linked from a footer (Privacy Policy, for example) and test pages (like the Style Test page which is used to demonstrate the standard styles available on the site).

Skins and Modules can create new menus during installation by creating a new page with the addPage method of the Install.lib.php library, and passing null as the parent parameter. New menus can be created at any other time using the Page::CreatePage static method, again passing null as the parent parameter.

A Skin or Module can retrieve all the pages in a menu (or a branch of a menu) using the getSubMenu method of the Content.lib.php library. The returned array is hierarchically structured, with all child pages included in a “children” property of the parent. This method returns an array of data from the database, not a set of Page objects. Skins can also use the Menu Control, or make use of the aloe:menu tag from the Aloe Templating System to automatically include a menu.

If a Skin or Module provides a custom implementation of a menu, it should not include pages that are not published.

## Pages

Pages are managed using the Page class. This class provides methods for creating, updating and deleting pages, as well as moving and managing them, and creating and managing layouts and content blocks.

Create a new Page with the Page::CreatePage static method. The Page::GetNewPage method will return a Page object for the given ID. This method create a cache for the duration of the request, so this method can be called as many times as needed from as many disparate code blocks as necessary without making any additional database calls. This will return a reference to the same object every time it is called.

When a new page is created, it is populated only with the details available in the page table of the database. No content is compiled for a new Page. This is to ensure that overheads are kept to a minimum when a number of Pages need to be created. If more details are needed for a particular page, one of the Populate methods will be needed:

|  |  |
| --- | --- |
| Method | Description |
| PopulateFriendlyUrl | Populate the Friendly URL field. If the HTM path has previously been populated, it will be fetched from the cache. Otherwise, it will be built from the path and pathstub properties. |
| PopulateLayouts | Populate the layouts array with Layout objects, and build the column array, which is a list of the IDs of the available columns on the Page. |
| PopulateBlockTypes | Populate the blocktypes property with a list of all the Content Blocks allowed for all columns currently available in this page. |
| PopulateChildren | Populate the children property with a collection of Page objects representing the children of the Page. If the recursive parameter is true, the PopulateChildren method will be called each of those child Pages as well. |
| PopulateContent | Populate the pagecontent property with the complete HTML content for the Page. This calls PopulateCustom, PopulateLayouts automatically and PopulateBlockTypes as well if the Page is not in edit mode. |
| PopulateCustom | Populates the custom property with a full set of Page custom settings, and the values of those settings for this Page. |

Update the Page using the update method, and publish or unpublish it using the PublishPage and UnPublishPage methods. This will not prevent the page from being viewed – it will only prevent the page from appearing in the menu (assuming the Skin’s implementation of the menu honours this property). Pages can be moved up one space or down one space within their parent using the movePageUp and movePageDown methods, or moved anywhere within the entire menu structure (including into another menu) using the movePageTo method. Children of a Page can be ordered alphabetically with the OrderPagesAlphabetically method. Modify the Friendly URL by calling the resetURL method with a new pathstub. This will update any references to the old URL that can be found in content.

Delete a Page using the DeletePage method. This method will delete all child pages as well. Before deleting each page, this method will manually delete each content block on the page, allowing the content block to run any clean up code it may have defined. All references to the page are also removed from the HTMpath table. If this method is called as part of a larger process that uses a transaction, the usetransaction parameter may be set to false to prevent this method from prematurely committing the transaction. This should only be done if this method is being called from within an existing transaction.

## Layouts

Layouts are used to provide variety to the way content can be laid out on a page. If you imagine the page as a single column vertical container, then layouts are strips that are stacked from the top down. Each Layout contains one or more columns.

### Defining Layouts

The skin defines what Layouts are available by providing extensions to the abstract Layout class. Each extension must define several properties, and provide at least two methods:

|  |  |
| --- | --- |
| Property / Method | Description |
| name | The display name of the Layout |
| icon | The icon to use when displaying this Layout for selection. Only the name of the file should be included. |
| priority | An integer which defines the order in which Layouts are presented for selection – lower numbers first. |
| columnIds | An array containing the IDs of each column in the Layout. |
| getEditableContent | Supplies the code to be used by the CMS. |
| getDisplayContent | Supplies the code to use for display on the front end. This need not be the same, or even similar, to the Editable Content. |

The system will look for the icon in the images/layouts/ folder using the Skin files system. The core folder holds a small set of default layout icons, but custom images should be saved in the skin version of that folder. Icons should be 100px x 50px, and use transparent backgrounds for empty space. This allows the selected item to be easily visible.

The requirements of the code supplied by the getEditableContent method are very specific:

* The code must deliver HTML structured with a TABLE inside a containing DIV.
* Each cell in the TABLE must represent one column.
* The containing DIV must have an ID of “contentlayout\_*lid*”, where the *lid* represents the numeric ID of the current Layout.
* The TABLE must have the class of “maincontent”.
* Each cell of the TABLE must have a class of “contentcolumn\_*cid*” where the *cid* represents the numeric ID of the Column.
* Each cell of the TABLE must have an ID of “contentcolumn\_*lid*\_*cid*”, where the *lid* represents the numeric ID of the current Layout and the *cid* represents the numeric ID of the Column.
* Each cell of the TABLE must be empty

For Example:

protected function getEditableContent**(){**

**$res** **.=** '<div id="contentlayout\_'**.$this->**id**.**'">';

**$res** **.=** '<table cellpadding="0" cellspacing="10" class="maincontent">';

**$res** **.=** '<tr>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_LEFT**.**'" id="contentcolumn\_'**.$this->**id**.**'\_'**.**CONTENTCOLUMN\_LEFT**.**'"></td>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_CENTRE**.**'" id="contentcolumn\_'**.$this->**id**.**'\_'**.**CONTENTCOLUMN\_CENTRE**.**'"></td>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_RIGHT**.**'" id="contentcolumn\_'**.$this->**id**.**'\_'**.**CONTENTCOLUMN\_RIGHT**.**'"></td>

</tr>';

**$res** **.=** '</table>';

**$res** **.=** '</div>';

return **$res**;

**}**

It may seem that this requirement is a little pointless considering how easy it would be to automate this code. This is, however, only true for Layouts in which all the columns align next to each other in a single row. We have provided this level of flexibility in the Layout to provide for the possibility of Layouts with multiple rows. These might be sued, for example, to provide a single right column to a page with a complex layout on the left of the page (eg: The CommonRight Layout of the Standard skin).

The requirements of the getDisplayContent method, however, are far more relaxed. In fact, there are none. Use the getColumnContent method to fetch the HTML content of each column, and then display it in a manner that fits with the style of the site. You may use a table layout, as we have below, or any other style you choose. For example:

protected function getDisplayContent**(){**

**$left** **=** **$this->**getColumnContent**(**CONTENTCOLUMN\_LEFT**,$this->**foredit**)**;

**$centre** **=** **$this->**getColumnContent**(**CONTENTCOLUMN\_CENTRE**,$this->**foredit**)**;

**$right** **=** **$this->**getColumnContent**(**CONTENTCOLUMN\_RIGHT**,$this->**foredit**)**;

**$res** **.=** '<div id="contentlayout\_'**.$this->**id**.**'">';

**$res** **.=** '<table cellpadding="0" cellspacing="0" class="maincontent">';

**$res** **.=** '<tr>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_LEFT**.**'">'**.$left.**'</td>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_CENTRE**.**'">'**.$centre.**'</td>

<td valign="top" class="contentcolumn\_'**.**CONTENTCOLUMN\_RIGHT**.**'">'**.$right.**'</td>

</tr>';

**$res** **.=** '</table>';

**$res** **.=** '</div>';

return **$res**;

**}**

### Managing Layouts

The best way to manage Layouts assigned to a Page is through the Page object. Once the PopulateLayouts method has been called on the Page object, its layouts property will contain an array of Layout objects representing the Layouts on the Page, in the order that they appear.

Use the createLayout, updateLayout and deleteLayout methods of the Page object to create, update or delete Layouts respectively. These methods are part of the Page rather than the Layout because the creation, updating or deletion of a Layout affects more of the Page than just the Layout itself – at the very least the Page needs to update its Last Modified property.

You can move the Layout to a new position within the Page with the Layout’s moveTo method. Custom fields of the Layout are not populated until the PopulateCustom method is called. This method is called automatically for Layouts populated through the Page object’s populateLayouts method.

## Columns

Each Layout in the system contains a set of Columns. The main purpose of the Columns is to provide containers into which Content Blocks can be vertically stacked. The Content Blocks will fill the width of the Column, so the most important property of a Column is its width. Each column within a Layout must be unique, even if they have the same widths, but different Layouts can include the same Column.

Prepare Columns for your Skin using the prepareColumn method (see [Skins](#_The_Skin_Extension) for more details)

Column details are available through the columns property of the current skin. This property is an array of objects of the ContentColumn class. This is a simple class, containing just a few simple properties: id, width, adminwidth, and a list of applicable Content Blocks. The class also contains a static method getSimilarColumns which will return an array of the IDs of all columns that have the same width as that of the column with the supplied ID.

## Content Blocks

Content Blocks are the heart of the Content system. These collect, manage and display the actual content of each page on the site. Each type of Content Block collects different information, and displays it differently. Some are very simple, some collect very specific information and display it in visually impressive ways, while others build and manage complex widgets, like photo galleries.

Content Blocks allow us to walk the fine line between providing a WYSIWYG interface, while not requiring the content editor to know the complex HTML required to display many of the more challenging content widgets. Layouts and Columns provide the WYSIWYG element, but Content Blocks generally separate the information from the display. The site architect designs the output of the Content Block when the site is built, and the Content Editor simply plugs in the information as needed.

Content Blocks are all extensions of the ContentModule class. The class does not, however, provide methods for managing the Content Blocks. These are all accessed through the containing [Page](#_Pages) class. These allow you to create, update, delete, copy and move the Content Block as needed. This is done because creating, moving, copying and deleting a Content Block will affect other Content Blocks as well. All Content Blocks share the following properties:

|  |  |
| --- | --- |
| Property | Description |
| id | The database ID of this Content Block |
| foredit | Boolean value indicating whether or not the Content Block is currently being used in the editor |
| pageid | The database ID of the containing Page |
| layoutid | The database ID of the containing Layout |
| columned | The ID of the containing Column |
| position | The position of this Content Block within the containing column, with position 1 being the top. |
| modulename | The class name of this Content Block |
| pagename | The menuname property of the containing Page |
| column | A reference to a Column object representing the containing column. |
| content | The primary HTML content for the Content Block |
| properties | A container for editable properties of this Content Block |

Most of these are defined by the context of the Content Block. The content and properties properties, however, allow individual Content Blocks to be customised. The content property is intended to hold the primary HTML content of the Content Block. This has meaning in some cases, while in others it may serve simply as a way to hold one of many useful pieces of data. We generally try to include a title or descriptive piece of data if there is no obvious content item, but this is just convention. The properties property is an array of editable properties for the Content Block. The exact elements of this are defined by each type of Content Block. Simple Content Blocks may have to additional properties at all, while more complex ones may store very complex arrays. Arrays are stored in the database as base64 encoded, serialised strings.

### The structure of a Content Block

Each Content Block is stored in a folder under the /content folder. The folder must be named the same as the class name for that Content Block. By convention, all Content Block class names begin with CM. At minimum, every Content Block must have three files:

#### The Class File

The class file for the Content Block must contain the class definition for that class, and be named *ClassName*.class.php, where ClassName is the name of the class. This class must be an extension of the ContentModule class. A sample Content Block class might look something like this:

class CMStandard extends ContentModule**{**

// Define all the public properties of this Content Block

public **$content**;

// Constructor

public function \_\_construct**($row,$foredit=**false**){**

// Call the parent constructor to prepare the common properties

parent**::**\_\_construct**($row,$foredit)**;

// Set all the properties of this Content Block

**$this->**content **=** **$row[**'content'**]**;

**}**

// This function must be included - Display the content

public function drawContentBlock**(){**

// Create the containing DIV

**$res** **.=** '<div id="cbl\_' **.** **$this->**id **.** '"';

// Include the properties for the editor - the CMS won't work without these

if**($this->**foredit**)** **$res** **.=** ' prop="' **.** **$this->**getBlockProperties**()** **.** '"';

**$res** **.=** ' class="CMStandard\_container">';

// Create the content for the Content Block

**$res** **.=** **$this->**content;

// Close the containing DIV

**$res** **.=** '</div>';

return **$res**;

**}**

**}**

The public properties of the Content Block will be available to the drawContentBlock method and to the editor for this Content Block. The parent constructor will set all the properties common to all Content Blocks (id, pageid, layoutid, columnid, pagename, position, and modulename, as well as a reference to the containing ContentColumn object, and a Boolean foredit, indicating whether or not this particular instance is being used in the editor). It will also populate the properties property with the values of all the editable properties of this Content Block. The content property can be set from the **$row[**'content'**]** variable in the constructor, while other editable properties may be populated from the properties property, **$this->**myval **=** **$this->**properties**[**'myval'**]**.

The drawContentBlock method must be included, and it should return the full HTML of the Content Block. If the Content Block is displaying on the website itself (foredit=false), the convension is to contain the Content Block in a containing DIV. This is not absolutely required, however – you can display it just as you like. However, if the Content Block is being displayed in the CMS, it must at minimum be contained in a DIV tag with the ID of “cbl\_*id*”, where *id* is the numeric ID if the Content Block. That DIV tag must also include a non-standard props attribute, which must include a modified JSON string (all inverted commas are replaced with apostrophes, so the string won’t break the HTML parser) containing details of the Content Block for the editor. This string can be fetched using the getBlockProperties method of the Content Block.

As an alternative to creating the HTML for the Content Block in the drawContentBlock method, you can use a template file, and simply return the processed contents of that template instead. For example:

public function drawContentBlock**(){**

// Set any extra properties that might be needed by the template

**$this->**target **=** getLinkTarget**($this->**link**,**false**)**;

// Process the template through the Skin, passing the Content Block as context

**$res** **=** **$GLOBALS[**'skin'**]->**getFragment**(**'/content/CMListItem/CMListItem.tmp.html'**,$this)**;

return **$res**;

**}**

The HTML in a template file for a Content Block should always be contained within a aloe:contentcontainer tag. This will be replaced with the containing DIV tag as required for the Content Block, and the props will be populated if necessary. A very simple example:

<aloe:contentcontainer class="CMListItem\_container">

<h3 class="CMListItem\_heading">##$this->heading##</h3>

<h4 class="CMListItem\_subtitle">##$this->subtitle##</h4>

<div class="CMListItem\_content">##{nl2br($this->content)}##</div>

<aloe:if condition="$this->drawline"><hr style="clear: both;" /></aloe:if>

</aloe:contentcontainer>

The extended class can provide additional logic for creating, updating, deleting or copying by overriding the appropriate method:

|  |  |
| --- | --- |
| Task | Method |
| Create | public static function CreateBlock**($id,** **$content,** **$properties,** **&$error)** |
| Update\* | public function UpdateBlock**($content,** **$properties,** **&$error)** |
| Delete | public function DeleteBlock**(&$error)** |
| Copy | public function CopyBlock**($newblockid,&$error)** |

The default implementation of each of these simply returns true, except for the Update method, which provides all the logic for updating the Content Block. Remember to call the parent::UpdateBlock method, or provide your own implementation of all the update logic, if you override this method. All methods must return either true or false, and populate the $error variable with a string error message if appropriate.

#### The Config File

The config file is a very simple file used to provide the editor with basic info about the Content Block. This is stored in a separate file so that the editor doesn’t need to instantiate one of every type of Content Block in order to compile the selection popups. Instead, it populates a $ContentModules array by including the config file for each Content Block. Config files must be named for the class name of the Content Block, ie: *ClassName*.config.php. Config files must provide only four pieces of information: classname, name (a display name), description and group. The group should be one of: basic, list, gallery or feature, and defines which list of Content Blocks this one falls under in the selector. An example config file might look like this:

**$ContentModules[**'CMStandard'**][**'classname'**]** **=** 'CMStandard';

**$ContentModules[**'CMStandard'**][**'name'**]** **=** 'Standard Content';

**$ContentModules[**'CMStandard'**][**'description'**]** **=** 'Add text, images, and other media.';

**$ContentModules[**'CMStandard'**][**'group'**]** **=** 'basic';

#### The Editor

The editor file is a complete HTML page that is loaded into an IFRAME when a Content Block is created or edited. The same file is loaded in each case, but no blockid is sent in the case of creating. The editor may use this fact to provide different editors for the two cases. The purpose of the editor is to allow the Content Editor to modify editable properties of the Content Block. To do this, the file will standardly consist of a form to collect the information and some PHP code to handle the submission of the form.

An example of an editor file might be something like this:

<?php

**$GLOBALS[**'authcode'**]** **=** 'Acojc5ttj 24t0qtqv#';

require\_once**(**'../../lib/Global.lib.php'**)**;

// set global values

**$module** **=** 'CMRawHTML';

**$pageid** **=** **$\_REQUEST[**'pageid'**]**;

**$blockid** **=** **$\_REQUEST[**'blockid'**]** **|** 0;

**$layout** **=** **$\_REQUEST[**'layout'**]**;

**$col** **=** **$\_REQUEST[**'col'**]**;

**$pos** **=** **$\_REQUEST[**'pos'**]** **|** 0;

// handle a submitted form

if**(!**empty**($\_REQUEST[**'submit'**])){**

**$content** **=** **$\_REQUEST[**'content'**]**;

**$properties** **=** array**()**;

**$properties[**'somevalue'**]** **=** **$\_REQUEST[**'somevalue'**]**;

**$page** **=** Page**::**GetNewPage**($pageid,**true**)**;

if**($blockid){**

**$res** **=** **@$page->**updateContentBlock**($blockid,** **$content,** **$properties,** **$error)**;

**}**else**{**

**$res** **=** **@$page->**createContentBlock**($col,** **$layout,** **$module,** **$content,** **$properties,** **$pos,** **$error)**;

**}**

**$error** **=** str\_replace**(**"'"**,**"&apos;"**,$error)**;

**$res** **=** **$res?$res:**'false';

echo "<script>parent.ContentBlockCompleteEdit('**$res**', **$blockid**, **$col**, **$layout**, '**$module**', **$pos**, '**$error**');</script>";

exit;

**}**

// populate variables if we're editing an existing block

if**($blockid** **&&** is\_numeric**($blockid)){**

**$block** **=** ContentModule**::**getContentBlock**($blockid,**true**)**;

**$content** **=** **$block->**content;

**}**

?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<link rel="StyleSheet" href="/css/editors.css" type="text/css" />

<link rel="StyleSheet" href="/css/contentframe.css" type="text/css" />

<link rel="StyleSheet" href="<?=**$module**?>.css" type="text/css" />

</head>

<body>

<form method="post">

<!-- Ensure these variables are passd with the form -->

<input type="hidden" name="layout" value="<?=**$layout**?>">

<input type="hidden" name="col" value="<?=**$col**?>">

<input type="hidden" name="pos" value="<?=**$pos**?>">

<input type="hidden" name="blockid" value="<?=**$blockid**?>">

<input type="hidden" name="pageid" value="<?=**$pageid**?>">

<table class="edt\_table" cellpadding="0" cellspacing="0" border="0">

<tr>

<td class="field">

<textarea name="content" id="content" style="width: 664px; height: 450px;"><?=**$content**?></textarea>

</td>

</tr>

<tr>

<td><input type="submit" name="submit" class="edt\_button" value="Submit"></td>

</tr>

</table>

</form>

<!-- Use this script to automatically size and position the popup -->

<script>if(parent && parent.PopupManager) parent.PopupManager.prepare('<?=**$module**?>',0,0,'Raw HTML Editor');</script>

</body>

</html>

When the initial popup is loaded into its IFRAME, it will be passed five GET variables:

|  |  |
| --- | --- |
| Variable | Description |
| pageid | The database ID of the containing Page |
| layout | The database ID of the containing Layout |
| col | The ID of the containing Column |
| blockid | The ID of this Content Block, if editing. Not included when creating. |
| pos | When creating, the ID of the Content Block after which the new Content Block should be inserted within the column, or null to be inserted at the top. When editing, the current position within the column. |

The form must recycle these five variables so that the processing script can use them. Apart from that, the form can pretty much contain any fields, controls or tools that are needed to collect the required information. The /css/editors.css file contains CSS classes for form fields to standardise the look and feel of the form.

When the popup is first opened (either for creating a new Content Block or for editing an existing one) the size is automatically calculated, and the default title used. In some cases, however, the size of the form, and therefore the popup, will change depending on whether this popup is used to create or to edit a Content Block, or on the content of the block itself. In this case, you can use the following script to resize, reposition and re-title the popup as you need it:

<script>if(parent && parent.PopupManager) parent.PopupManager.prepare('<?=**$module**?>',0,0,'My new title');</script>

Notice that we call on the parent, because this is running from within an IFRAME, and it is very important to check for the existence of the parent and its [PopupManager](#_Popups) first. You can supply the width and height specifically, or provide 0 and 0 to force the PopupManager to calculate them for you.

When handling the submitted form the following steps should be taken:

* Prepare the content and properties variables. The content variable should be a string, and the properties an associative array.
* Fetch the current Page object with Page**::**GetNewPage**($pageid,**true**)**;. Remember to pass true for the foredit parameter.
* Call the updateContentBlock method on the Page object to update an existing Content Block, or the createContentBlock method to create a new one.
* Clean the result and the error message for inclusion in the HTML SCRIPT tag
* Complete the process by writing a SCRIPT tag to the output stream, then closing the request. This tag should run a script calling the ContentBlockCompleteEdit method of the parent window. This HTML will be sent back to the IFRAME, and run. The ContentBlockCompleteEdit method will add the new block, or update the existing block, in the editor, then close the popup.

In some cases, a Content Block needs to have been created before it can collect information. For example, the CMPhotoGallery Content Block needs to save the photos in a folder named for the Content Block ID. It therefore needs to have been created before the photos can be saved. This means that when the editor is opened for creation, it simply contains a submit button. The user must create the block, and then edit it to add photos, which isn’t very user friendly. To overcome this, you could expand the form handling script with the following code:

if(!empty($\_REQUEST['submit'])){

$properties = array();

$content = '';

$page = Page::GetNewPage($pageid,true);

//and so on

exit;

}elseif(empty($blockid)){

$page = Page::GetNewPage($pageid,true);

$blockid = @$page->createContentBlock($col, $layout, $module, '', null, $pos, $error);

$error = str\_replace("'","&apos;",$error);

if(!$blockid){

echo "<script>parent.ContentBlockCompleteEdit('false', 0, $col, $layout, '$module', $pos, '$error');</script>";

exit;

}else{

$script = "<script>parent.contentBlockManager.layouts[$layout].columns[$col].AppendContentBlock($blockid,$pos);</script>";

}

}

And then add the line: <?=**$script**?> into the HEAD of your HTML. This will automatically create an empty Content Block, and supply an ID for it. It will also tell the editor to insert the Content Block into the page in the correct position, as if it had been created already. Now the editor will behave as if we’re editing the block.

### Specific Content Blocks

#### CMRawHTML

This Content Block allows the Content Editor to supply any HTML code they want for inclusion on the site. This code may include Flash objects which, without careful configuration, will interfere with the editor when displayed. For this reason, this Content Block will check the HTML code for either an OBJECT or IFRAME tag and, if found, will display the code itself in the editor, rather than the rendering.

#### CMStandard

This Content Block uses the [TinyMCE](#_Tiny_MCE) editor from Moxicode to collect formatted HTML content. It identifies the width of the containing column, and resizes itself, and the TinyMCE editor, to ensure that the space provided for the content in the editor is the same as what will be available on the site. This provides a more WYSIWYG experience for the Content Editor. If the width of the column is smaller than the minimum space available for the editor, as defined by the width of the button menu, the space within the editor is limited. The style of the editor (the arrangement of the button menu) is also modified according to the width of the content.

#### CMListItem

This Content Block uses a template to display the various pieces of information it collects, allowing the Site Designer to customise the layout completely. The orientation property defines the width of the image and may also be used in the template to modify the layout. The image will be resized and cached if it is not the correct size originally, which means that any changes to the image in Resources after this Content Block has been created may not filter down to the block. The Content Block overrides the CopyBlock and DeleteBlock methods to copy or delete the image from the cache, if it exists.

#### CMFeature

This Content Block uses a template to display the various pieces of information it collects, allowing the Site Designer to customise the layout completely. The width of the image is set to the width of the containing column. The image will be resized and cached if it is not the correct size originally, which means that any changes to the image in Resources after this Content Block has been created may not filter down to the block. The Content Block overrides the CopyBlock and DeleteBlock methods to copy or delete the image from the cache, if it exists.

#### CMLinkList

The editor for this Content Block uses a custom JavaScript system to allow the user to add an unlimited number of text/url pairs to the block. These are stored in two separate arrays, with identical indexes, and are stored as base64 encoded, serialised strings in the database. The Content Block uses a template to lay out the basic block, but calls the drawLinks method to output the basic list of links. Each link is simply an A tag with a set of classes around the text. Use the following ATE tag to include the links: <aloe:call object="this" method="drawLinks" />

#### CMIndex

This Content Block will collect all published child or sibling (depending on the relationship property) Pages that contain the given Content Block filter. If no filter is supplied, the search will still only return Pages that contain at least one Content Block. This Content Block uses two templates to display its content. The CMIndex.tmp.html template id called from the main drawContentBlock method. This, in turn, calls the drawItems method, which calls the CMIndex\_item.tmp.html template for each item in the list.

#### CMNewsFeed

This Content Module requires the News module installed. It uses two templates to display, in the manner of other list blocks. The main CMNewsFeed.tmp.html template calls the drawArticles method, which in turn calls the CMNewsFeed\_item.tmp.html template for each article in the list.

#### CMRSSFeed

This Content Block uses the FeedReader.class.php class to parse the supplied RSS feed. This can parse RSS2 or Atom feeds. It was found that this block would slow a page down by as much as 1000%, so it now caches the feed for an hour. This means that only one request an hour is affected by the lag, rather than all of them, but does introduce an hour delay. This Content Block uses two templates to display, in the manner of other list blocks. The main CMRSSFeed.tmp.html template calls the drawFeedItems method, which in turn calls the CMRSSFeed\_item.tmp.html template for each article in the list. This Content Block overrides the UpdateBlock method to delete the cache file whenever the block is updated.

#### CMPhotoGallery

This Content block uses the [GalleryView](#_Gallery_View) control to collect and manage the photos in the gallery. It collects a single caption property of each photo, and saves each set of photos and thumbnails in a subfolder named for the ID of the block, under the /galleries subfolder. The editor is sized to fit five thumbnails wide, and two deep. The size of the photo and thumbnail is defined by the sizematrix [Skin variable](#_Customising_Modules_and). The sizematrix is a multidimensional array. Each primary element represents one column, and contains an array with four elements; width, height, thumbwidth and thumbheight. You might set this variable in the Skin like this:

**$this->**prepareSetting**(**'CMPhotoGallery'**,**'sizematrix'**,**array**(**

CONTENTCOLUMN\_FULL=>array**(**'width'=>720**,**'height'=>480**,**'thumbwidth'=>80**,**'thumbheight'=>53**),**

CONTENTCOLUMN\_MAIN=>array**(**'width'=>470**,**'height'=>313**,**'thumbwidth'=>80**,**'thumbheight'=>53**),**

**))**;

The GalleryView control saved updates to the photos directly, and doesn’t require the submit button to be clicked. This Content Block overrides the copyBlock and deleteBlock methods to copy or delete the photos as needed.

#### CMLightboxGallery

This Content block uses the [GalleryView](#_Gallery_View) control to collect and manage the photos in the gallery. It collects a single caption property of each photo, and saves each set of photos and thumbnails in a subfolder named for the ID of the block, under the /galleries subfolder. The editor is sized to fit five thumbnails wide, and two deep. The size of the thumbnail is defined by the sizematrix [Skin variable](#_Customising_Modules_and). The sizematrix is a multidimensional array. Each primary element represents one column, and contains an array with two elements; width and heightYou might set this variable in the Skin like this:

**$this->**prepareSetting**(**'CMLightboxGallery'**,**'sizematrix'**,**array**(**

CONTENTCOLUMN\_FULL=>array**(**'width'=>80**,**'height'=>53**),**

CONTENTCOLUMN\_MAIN=>array**(**'width'=>50**,**'height'=>33**),**

**))**;

The GalleryView control saved updates to the photos directly, and doesn’t require the submit button to be clicked. This Content Block overrides the copyBlock and deleteBlock methods to copy or delete the photos as needed.

#### CMSlideShow

This Content block uses the [GalleryView](#_Gallery_View) control to collect and manage the photos in the gallery. It collects a url and a title property of each slide, and saves each set of slides and thumbnails in a subfolder named for the ID of the block, under the /galleries subfolder. The size of the is defined by the sizematrix [Skin variable](#_Customising_Modules_and). The sizematrix is a multidimensional array. Each primary element represents one column, and contains an array with two elements; width and an array of height values. You might set this variable in the Skin like this:

**$this->**prepareSetting**(**'CMSlideShow'**,**'sizematrix'**,**array**(**

CONTENTCOLUMN\_FULL=>array**(**'width'=>720**,**'height'=>array**(**'short'=>330**,**'medium'=>480**,**'long'=>640**)),** CONTENTCOLUMN\_MAIN=>array**(**'width'=>470**,**'height'=>array**(**'short'=>216**,**'medium'=>313**,**'long'=>420**)),**

**))**;

The GalleryView control saved updates to the slides directly, and doesn’t require the submit button to be clicked. This Content Block overrides the copyBlock and deleteBlock methods to copy or delete the photos as needed.

#### CMContact

This Content Block uses a template file to provide complete control over the way it is rendered. The email address is obfuscated. It provides a drawMap method which can be called to render the map property according to its content:

* If the map property contains HTML code (ie: it starts with a <), it is included as-is.
* if it starts with ‘http://’, it is included as a link.
* If it is an image in resources (it is a file with an extension in the validimagetypes global setting), it is rendered as an image
* It is included as the query in a URL which searches Google Maps, which is rendered as a link. An address can be used in this case, but it should always be checked first – some addresses work on Google Maps, but others really don’t.

#### CMEnquiry

This Content Block uses a template file to provide complete control over the way it is rendered, but will manually append the required JavaScript code, if necessary. It is not necessary to include the JavaScript in the template file. The form uses AJAX to send the message to the supplied recipient, or the admin email address (set in global settings) if that is invalid. The AJAX uses the sendEnquiryForm method of the Email.lib.php library. It includes a simple CAPTCHA sum to prevent spam.

#### CMSpacer

This Content Block will display with a visible background in the CMS, but will be invisible on the site. There is nothing else interesting about this Content Block at all.

#### CMTable

# Skins

Most of the effort put into Aloe is about managing the content, or perhaps more accurately, getting the content into the system. This is only half the job, however, and the other half is done by the skins – displaying the content to the visitors. The Aloe CMS is designed to look the same for all users, but the skin allows the sites using Aloe to look different and unique. Skins are not intended to manage content at all. They are responsible only for collating and presenting it properly.

All skins are stored in a folder under the /skins folder. The root folder of the skin must be named the same as the skin itself, case sensitive. You can have as many skins as you like in the skins folder, but only one skin will be active at a time. You can manually change the current skin by changing the skin global variable, but it is recommended that you use the Installation tab of the System module to install or remove skins, as they may have install or remove scripts that need to be run.

Only two files are technically required to make a skin work, but most will have many more supporting files as well. The basic skin can have the following files:

|  |  |  |
| --- | --- | --- |
| Filename | Required? | Description |
| *SkinName*.skin.php | yes | An extension of the base Skin class. |
| *SkinName*.tmp.html | yes | The basic template for the site, defining all the global HTML, and setting a space for the content. |
| *SkinName*.install.php | no | An install script run when the skin is installed in the CMS |
| *SkinName*.remove.php | no | A remove script run when the skin is replaced by another in the CMS |

A skin can also have any number of other supporting files, including images, stylesheets, javascripts and custom template files.

## The Skin Extension Class

The primary task of the extended Skin class is to define the parameters of the content. For the Content Module, the extension needs to define which Layouts are available, which Columns are used in those layouts, which Content Blocks are available in each of those Columns, and provide settings for all these.

To extend the Skin class, overload the prepareSkin method. In this, you should set the contentwidth property of the skin to the width of the editable content area, in pixels. This should be the sum of the width of the columns plus their spacings.

Call the prepareLayout method once for each Layout the skin uses, passing the name of the Layout class. Layouts should be stored in the layouts subdirectory of the skin. See Layouts for more details.

Call the prepareColumn method to define a column to be used in one or more Layouts. The prepareColumn method takes the following parameters:

|  |  |
| --- | --- |
| *Field* | *Description* |
| Id | An integer representing this Column. |
| Name | The name of the Column. A constant will be created by appending the name to “CONTENTCOLUMN\_”. Standardly, the name is in uppercase form. |
| Width | The width, in pixels, of the content area of this column in the front end. |
| Adminwidth | The width of the column when used in the admin. This should start at the width + 8, and increment by 5s until all the layouts in the CMS look the same width. This is not necessary, but will make the CMS look better. |
| contentblocks | An array of the names of all Content Blocks available in this Column. |

An alternative way to make a content block available to one or more columns is to use the prepareContentBlock method. Pass it the name of the Content Block and an array of the IDs of pre-prepared columns.

Some Content Blocks and Modules might allow the skin to define certain parameters. For example, the Slide Show Content Block will allow the skin to define the width and a set of heights for the slide show in a range of columns, allowing the skin designer to define exactly how the slideshow can fit into the skin. Use the prepareSetting method to define these details, passing the name of the Module (or Content Block), the name of the setting and the value of the setting. To use these settings in a Module or Content Block, call the getSetting method of the current skin, passing the name of the Module, the name of the setting, and a default value in case the skin hasn’t set the setting explicitely.

If the display name of the skin is different from the class name (eg: it contains spaces), set the name property of the Skin to the correct name. By default the display name and the class name are the same.

An example of an extended Skin class might look like this:

class Testing extends Skin**{**

public **$name** **=** 'Testing';

public function prepareSkin**(){**

**$this->**contentwidth **=** 720;

**$this->**prepareLayout**(**'OneColumn'**)**;

**$this->**prepareLayout**(**'TwoColumnLeft'**)**;

**$this->**prepareLayout**(**'TwoColumnRight'**)**;

**$this->**prepareLayout**(**'TwoColumnEqual'**)**;

**$this->**prepareLayout**(**'ThreeColumn'**)**;

**$this->**prepareColumn**(**1**,**'FULL'**,**720**,**728**,**array**(**'CMStandard'**,**'CMContact'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMListItem'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMPhotoGallery'**,**'CMLightboxGallery'**,**'CMSlideShow'**))**;

**$this->**prepareColumn**(**2**,**'MAIN'**,**470**,**483**,**array**(**'CMStandard'**,**'CMContact'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMListItem'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMLightboxGallery'**,**'CMPhotoGallery'**))**;

**$this->**prepareColumn**(**3**,**'LEFT'**,**220**,**233**,**array**(**'CMStandard'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMFeature'**,**'CMLightboxGallery'**))**;

**$this->**prepareColumn**(**4**,**'CENTRE'**,**220**,**238**,**array**(**'CMStandard'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMFeature'**,**'CMLightboxGallery'**))**;

**$this->**prepareColumn**(**5**,**'RIGHT'**,**220**,**233**,**array**(**'CMStandard'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMFeature'**,**'CMLightboxGallery'**))**;

**$this->**prepareColumn**(**6**,**'SINISTER'**,**345**,**358**,**array**(**'CMStandard'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMFeature'**,**'CMLightboxGallery'**))**;

**$this->**prepareColumn**(**7**,**'DEXTER'**,**345**,**358**,**array**(**'CMStandard'**,**'CMEnquiry'**,**'CMRawHTML'**,**'CMSpacer'**,**'CMLinkList'**,**'CMIndex'**,**'CMTable'**,**'CMFeature'**,**'CMLightboxGallery'**))**;

**$this->**prepareContentBlock**(**'CMNewsFeed'**,**array**(**CONTENTCOLUMN\_LEFT**,**CONTENTCOLUMN\_CENTRE**,**CONTENTCOLUMN\_RIGHT**))**;

**$this->**prepareContentBlock**(**'CMRSSFeed'**,**array**(**CONTENTCOLUMN\_LEFT**,**CONTENTCOLUMN\_CENTRE**,**CONTENTCOLUMN\_RIGHT**))**;

**$this->**prepareSetting**(**'CMLightboxGallery'**,**'sizematrix'**,**array**(**

CONTENTCOLUMN\_FULL=>array**(**'width'=>80**,**'height'=>53**),**

CONTENTCOLUMN\_MAIN=>array**(**'width'=>80**,**'height'=>53**),**

CONTENTCOLUMN\_LEFT=>array**(**'width'=>50**,**'height'=>33**),**

CONTENTCOLUMN\_CENTRE=>array**(**'width'=>50**,**'height'=>33**),**

CONTENTCOLUMN\_RIGHT=>array**(**'width'=>50**,**'height'=>33**),**

CONTENTCOLUMN\_SINISTER=>array**(**'width'=>75**,**'height'=>50**),**

CONTENTCOLUMN\_DEXTER=>array**(**'width'=>75**,**'height'=>50**),**

**))**;

**}**

**}**

## Skin Templates

The global template file for the skin provides all the basic HTML for the site. Essentially, it creates an HTML container for the content managed by the CMS. This file should include a DOCTYPE, HEAD and BODY tags, and everything else required to produce good, standards compliant HTML. The template file uses the Aloe Templating Engine to produce the final HTML (see Aloe Templating Engine for details on how this works). The most important tag to include in the template is aloe:content. This will be replaced by the content from the CMS, no matter which module or page is being processed. Without this tag, no content can be included in the site.

## Skin Files

Skin files allow the skin to provide a customised version of a file. The skin should copy the folder structure of the original file from the skin root. So, a skin might customise the /modules/News/news.css file by creating a /skins/MySkin/modules/News/news.css file. This doesn’t work just by magic – the Module or Content Block must specifically request the Skin file rather than simply using their default. This is done by calling the getFile method on the current skin. This method takes the following parameters:

|  |  |
| --- | --- |
| *Parameter* | *Description* |
| Path | The path to the original file. If the skin version exists, that will be returned. Otherwise this actual file will be returned. |
| Default | If neither the skin version nor the actual file exists, you can supply a default. Otherwise, an empty string is returned. |
| Type | By default, the path is returned relative to the site root. This parameter can have the following other values:   * site: the path includes the site root (<http://www.mysite.com/>...) * doc: the path is the complete physical file path * web: the path is relative to the current URL. |

## Customising Modules and Content Blocks

There are several ways that Modules and Content Blocks might be customised by a skin. The exact details are defined by the Module or Content Block.

The most obvious method to customise an element is to supply a skin version of the CSS. Modules and Content Blocks should use the skin version of the file rather than their default, although some instances may include the default, then also the skin version if it exists.

Some Modules and Content Blocks will use the Aloe Templating Engine with HTML snippets to allow the skin to customise the layout of the content. Whether or not a template is used, and the manner of its use is determined by the needs of the Module or Content Block.

The Module or Content Block may also define any other Skin File for use, allowing the skin to provide a custom version. Images might be handled like this, for example.

Some examples of the way Modules or Content Blocks might be extended:

|  |  |
| --- | --- |
| *Module* | *Description* |
| CMStandard | No template files are used here, because there is no additional formatting to provide. A custom CSS file is included, but is not generally used. |
| CMPhotoGallery | No templates are used because the scripting and layout of the content is very specific. A custom CSS file is provided. |
| CMListItem | A single layout template is used, as well as a custom CSS |
| CMLinkList | A custom CSS file is used, and a template file for each link in the list, but not for the list itself. |
| News | Three template files are used. One for the whole page, one for an individual article and one for each entry in a listing. A custom CSS file is also used. |

## Preparing a Skin for the Content Module

Preparing a Skin for the Content Module involves deciding what Layouts, Columns and Content Blocks to include in your Skin. It is beyond this manual to tell you how to make these decisions, but there is a process we follow to arrive at a successful Skin.

First, start with the content. If you don’t know the actual content that the site will hold (starting content, obviously), you must at least know the type of content you’ll need to be dealing with. A website is nothing more than a tool to distribute or communicate content; a conduit for the content. Without understanding the content, or at least the types of content, how can anyone possibly plan a website?

Once you know what content you’ll be dealing with, you can decide on the best techniques to use to present that content. Is your content best delivered in article form? Do you need photo galleries? Are there lots of data tables, or lists of references or citations? Identify all the techniques you’ll need, then choose or develop a Content Block for each of them.

…

# Aloe Templating Engine

The Aloe Templating Engine is designed to allow front-end developers to provide HTML layouts for various uses within Aloe. The engine is built into the Skin.class.php class. Templates themselves are extended HTML code, saved by convention in files with a .tmp.html extension. Templates are used in two main ways; The Skin itself supplies a global template for the site, which contains HTML code for the global areas of the site. Content Blocks and Modules may use templates for HTML fragments which are intended for inclusion into the main Skin template in specific places, as needed by the Content Block or Module.

The engine extends standard HTML in two ways; by replacing variables, and providing a small set of additional tags.

## Replacing Variables

The value of a PHP variable may be inserted into the template by surrounding it by pairs of hashtags (##). Variables are accessed in the global context, but do not require the $GLOBAL container. For example, the variable ##myvariable## (or ##$myvariable## - the $ is optional) will be replaced with a value if the PHP variable $GLOBALS[‘myvariable’] exists with a value. You can also reference a variable associated with the object supplied as context for this template with the keyword $this (in this case, the $ is required), for example: ##$this->title##.

If the variable is also surrounded by curly brackets, it will not be processed as a variable, and instead run as a PHP command. Use this feature to run PHP code directly. Be careful – this uses the dreaded eval method. For example: ##{date(‘d F Y’,$this->date)}## or ##{nl2br($this->description)}##. The $this keyword will still reference the current context object.

The engine replaces all variables before it begins processing the additional tags. This includes the aloe:if tag, so variables in sections of the template that are not actually going to be used in this case (that do not fulfil the conditions of an if condition) will still be replaced.

Some common uses for replacement variables are:

|  |  |
| --- | --- |
| variable | result |
| ##$skin->webpath## | The path to the root of the current skin |
| ##$webroot## | The relative path to the root of the site (eg: ../../) |
| ##settings->islivesite## | Returns the value of the given global setting. |

## Additional Tags

Additional tags follow all the standard rules for XHTML tags. Tags must be closed, either internally (with a trailing /) or with a separate closing tag. All tags have the aloe prefix. The following additional tags are available in the Aloe Templating Engine:

### Settings

Attributes: var.

This tag will be completely replaced with the value of the global setting described in the var attribute. This tag is the same as using the replacement variable: ##$settings->var##.

Example: <aloe:settings var="adminemail" />

### Pageobject

Arrtibutes: var.

This tag will be completely replaced with the value of the property of the PageObject object described in the var attribute. This tag is the same as using the replacement variable: ##$pageobject->var##.

Example: <aloe:pageobject var="title" />

### SEOTags

Attributes: none.

This tag will be completely replaced by a set of SEO tags. It is intended to be used in the HEAD of a main template. This tag will create the standard title, description and keywords meta tags. If the site is live, it will also add a set of Open Graph tags and, if the correct Site Settings exist, Google or Bing site verification tags. It will also add a robots tag, inclusive if the site is live, but with nofollow and noindex flags if not.

Example: <aloe:seotags />

### Content

Attributes: none.

This special tag is intended only for use in the Skin’s main template. It will be completely replaced by the content of the current page (the $pageobject->pagecontent property), be that part of the Content Module, or provided by another module.

Example: <aloe:content />

### If elseif else

Attributes: condition.

Possibly the most powerful and useful tag, the if/elseif/else set of tags allows the developer to include a block of HTML only if a supplied condition resolves true. The condition will be evaluated by PHP (using the eval method) exactly as entered. The documentroot, webroot, settings, pageobject and metatags global variables are available directly to this tag (ie: without the $GLOBALS[] container). The $this keyword refers to the current context object for the template.

The tag will be completely replaced by its contents if the condition resolves to true, and with nothing if not. Elseif tags can be used to provide additional conditions, and additional HTML fragments, and the else tag can be used to provide an alternative. The elseif and else tags should be closed internally.

A limitation of the current parsing engine requires that any elseif and else tags to be on the same hierarchical level as the if tag. This means that any block tags opened in the first conditional block must be closed before the elseif or else tag is used. So, this is good:

<aloe:if condition="$this->width > 800">

<div class="wide"><!-- some code --></div>

<aloe:else />

<div class="normal"><!-- some code --></div>

</aloe:if>

But this will never display the default, even if the condition is false:

<aloe:if condition="$this->width > 800">

<div class="wide">

<aloe:else />

<div class="normal">

</aloe:if>

This issue is known, and we’re working on a new parsing engine to fix it.

Example:

<aloe:if condition="$this->width > 800">

<!-- some code -->

<aloe:elseif condition="$this->width > 400" />

<!-- some other code -->

<aloe:else />

<!-- default code -->

</aloe:if>

### Call

Attributes: object, class, method, params.

This rather powerful and tricky tag allows the developer to call PHP methods, passing whatever parameters are necessary. The most important attribute is method, which is the name of the function or method to call. Standard library functions are called by including the name of the function in the method attribute. Static class methods are called by naming the class in the class attribute, and the method in the method attribute. Similarly, existing object’s methods may be called by naming the object and the method. In all cases, the params attribute should contain the exact set of parameters that should be passed to the method, including quotes and commas if appropriate.

The methods are called using the eval method. This means that the PHP code is constructed into a string before being processed. If any of the parameters contain apostrophes, which may happen if replaced variables are used in the parameter attribute, this tag may throw a fatal error.

Example: <aloe:call object="agent" method="init" params="false" />

### Contentcontainer

Attributes: any.

This tag is intended as a container for Content Blocks. It will automatically create the containing DIV tag required for the block in the editor, with its required ID and props attributes. Content Blocks should use this tag as the outermost tag in their primary template. Any attributes added to this tag will be transferred unaltered to the containing DIV tag.

Example:

<aloe:contentcontainer class="CMIndex\_container">

<aloe:call object="this" method="drawItems" />

</aloe:contentcontainer>

### Menu

Attributes: id, orientation, levels

This tag will be completely replaced by the HTML produced by the [Menu Control](#_Menu). The three attributes will be passed directly to the menu.

Example: <aloe:menu levels="2" />

## Processing Templates

There are two ways to process templates.

In a processing file for a module, use the getContent method of the current Skin to construct the HTML for the page. This will process the Skin’s main template file, and populate it from the pagecontent property of the pageobject object. It is the responsibility of each processing page to populate this property before calling the getContent method.

Content Blocks and Modules can call the getFragment method of the current Skin to fetch the HTML that results from specific template files. Pass the path to the template and an object to use for context. This object will be used whenever the $this keyword is used. If an array (or any other non-object) is passed as context, it will be cast into an object.

A Module’s processing file might implement these like this:

**$pageobject[**'pagecontent'**]** **=** **$GLOBALS[**'skin'**]->**getFragment**(**'/modules/News/news.tmp.html'**, $context)**;

echo **$GLOBALS[**'skin'**]->**getContent**()**;

# Lists

# Modules

## News

# Request – Response Path

# Controls

## Calendar

## Context Menu

## Gallery View

## Input Control

## Lightbox

## List View

## Menu

## Multi List

## Tiny MCE

## Tree View

# Popups

Popup windows in Aloe are managed using the Popup Manager (js/Popups.js). This file is included in the template for the CMS, and the Popup Manager is instantiated, by default. The Popup Manager can be used to open disabling popups (eg: the loading screen), a number of preset standard popups (eg: the Resource Selector, or an error message) or custom popups create either from hidden DIVs or as full pages opened inside an IFRAME.

The Popup Manager has four optional settings. In most cases these will not be needed, but if they are, they should be set as variables before the Popup Manager Javascript file is included.

|  |  |  |
| --- | --- | --- |
| Variable Name | Description | Default |
| popupcsspath | The path to the CSS file to use for popups | /css/popups.css |
| popupcontainerpaddingwidth | The total width of the padding around the popup content. This should include the padding on both left and right. Note: this does not set the CSS – it merely reports what is already in the CSS for positioningcalculations. | 18 |
| popupcontainerpaddingheight | The total height of the padding and header around the actual popup content. This should include the padding and header size above and below the content. Note: this does not set the height; it is used in positioning calculations only. | 49 |
| popupborderwidth | The width of the border around the popup. This should be the sum of the width of the borders on both the left and right sides. Note: this does not set the border width; it is used in positioning calculations only. | 2 |

The Popup Manager automatically calls the init() when it is included. It inserts the correct CSS file into the document, however, so it must be included in the BODY of the HTML file.

## Full Screen Popups

The Popup Manager has three preset types of full screen popup. All full screen popups cover the entire screen and ensure that the user cannot interact with anything below them. They are generally used in conjunction with other popups, which are positioned above the full screen popup, and are therefore the only elements on the screen that a user may interact with. Essentially, these create modal popups.

The following full screen popups are available:

### Invisible

Show: showInvisible()

Hide: hideInvisible()

This popup creates an invisible layer covering the whole screen, ensuring that the user can see the content underneath, but cannot interact with it. The user cannot click on links, and hover or mouseover events do not fire. This might be used with a drag-and-drop system.

### Disabled

Show: showDisabled()

Hide: hideDisabled()

This popup covers the entire screen with a semi-transparent cover. This also prevents the user from interacting with the content beneath it. This is usually used to make other popups modal; first open this popup, then your popup on top of it.

The Disabled popup works slightly differently from others; when other popups are shown twice without being hidden in between, the popup is simply moved up the z-index stack. This popup, however, will remember the old position before being moved up. Then, when it is hidden, if there are previous positions recorded, the popup is instead moved back down to the previous position. Only if there are no previous positions is this popup actually hidden. This allows a single Disabled popup to be used repeatedly on the same page

### Loading

Show: showLoading()

Hide: hideLoading()

This popup incorporates the Disabled popup, but also shows a loading image in the center of the screen. Because it incorporates the Disabled popup rather than replicating the functionality, this popup and the Disabled popup may be used interchangeably. This popup actually just shows the loading image in the center of the screen so, if you show the Loading popup, then hide the Disabled popup, you’ll be left with a loading image floating in the center of the screen.

### Completed

Show: showCompleted()

Hide: hideCompleted()

The Completed popup is designed to show that an action has been successfully completed without making the user click a button to make a message go away. A big green tick appears in the middle of the Disabled popup, then 700ms later the whole popup hides itself.

## Pre-defined Popups

A small number of commonly used popups have been defined.

### Error

Show: showError(error,title)

Hide: hideError()

The Error popup is used (surprisingly) to display an error to the user. The popup first opens the Disabled popup, then its own DIV containing the error provided.

### Message

Show: showMessage(message,title)

Hide: hideMessage()

The Message popup show a positive message to the user. The popup first opens the Disabled popup, then its own DIV containing the message provided. Use this popup to provide the user with necessary information; if you want to simply let the user know something completed successfully, use the Completed popup.

### Resource Manager

Show: showResourceManager(selected,sourceid,owner,type)

Hide: hideResourceManager()

Opens a popup to allow the user to select a resource from the Resources module. This popup is usually opened from an icon next to the text box into which the selected path will be placed. This is an IFRAME popup, and uses the popups/ResourceManager.pop.php file.

This method takes four parameters:

|  |  |
| --- | --- |
| Field | Description |
| selected | The path to the currently selected value. If this is empty, the current value of the source element is used. |
| sourceid | The ID of the source element – usually the text box into which the selected path will be placed |
| owner | If this popup is being opened from within another IFRAME popup, this should be a reference to the IFRAME document object. This reference must be a string, because it will be passed inside a URL, but it may be a JavaScript statement that will evaluate to the reference (eg: a List Item Content Block will use this form in its edit popup: “PopupManager.popups.CMListItem.GetOwnerDocument()”) |
| type | The Type filter for the Resource Manager – show only images, only documents or all files (“images”,”docs” or null). |

### Image Selector

Show: showImageSelector (selected,sourceid,owner)

Hide: hideImageSelector()

This method is simply a shorthand way to display the Resource Manager popup with “images” type preselected.

### Document Selector

Show: showDocSelector (selected,sourceid,owner)

Hide: hideDocSelector ()

This method is simply a shorthand way to display the Resource Manager popup with “docs” type preselected.

### Link Selector

Show: showLinkSelector (selected,sourceid,owner)

Hide: hideLinkSelector()

Opens a popup to allow the user to select a URL. This popup is usually opened from an icon next to the text box into which the selected URL will be placed. The popup allows users to enter an external URL, construct an email link, select a resource or select a link from the Content module. This is an IFRAME popup, and uses the popups/LinkSelector.pop.php file.

Extensions can add additional tabs to the Link Selector popup by creating an event handler for the [linkSelectorLoading](#_linkSelectorLoading) event.

This method takes three parameters:

|  |  |
| --- | --- |
| Field | Description |
| selected | The currently selected URL. If this is empty, the current value of the source element is used. |
| sourceid | The ID of the source element – usually the text box into which the selected path will be placed |
| owner | If this popup is being opened from within another IFRAME popup, this should be a reference to the IFRAME document object. This reference must be a string, because it will be passed inside a URL, but it may be a JavaScript statement that will evaluate to the reference (eg: a List Item Content Block will use this form in its edit popup: “PopupManager.popups.CMListItem.GetOwnerDocument()”) |

### Colour Selector

Show: showColourSelector(selected,sourceid,owner,pageid)

Hide: hideColourSelector()

Opens a popup to allow the user to select a colour . This popup is usually opened from an icon next to the text box into which the selected colour will be placed. Users can select from a preset range of colours, supply the hex values themselves, or choose from a sliding colour gradient. This is an IFRAME popup, and uses the popups/ColourPicker.pop.php file.

This method takes four parameters:

|  |  |
| --- | --- |
| Field | Description |
| selected | The currently selected colour. If this is empty, the current value of the source element is used. |
| sourceid | The ID of the source element – usually the text box into which the selected colour will be placed |
| owner | If this popup is being opened from within another IFRAME popup, this should be a reference to the IFRAME document object. This reference must be a string, because it will be passed inside a URL, but it may be a JavaScript statement that will evaluate to the reference (eg: a List Item Content Block will use this form in its edit popup: “PopupManager.popups.CMListItem.GetOwnerDocument()”) |
| pageid | The ID of the current page, if opened within the Content module. The colour picker uses this to better identify recently used colours. |

## Custom Popups

A custom popup consists of a container, which includes a border, some padding and a heading bar, which contains the popup content. The entire container is positioned fixed, and centred both vertically and horizontally on the screen. Custom popups do not automatically show the Disabled popup – you must do this yourself, in either its Disabled or Loading version.

### Creating a popup

The preferred way to create a popup is to use the createOrFetchPopup method of the PopupManager object. This method will create a new popup if it does not exist already, but simply return a reference to the popup if it does. This method takes the following parameters:

|  |  |
| --- | --- |
| Parameter | Description |
| name | The name of the popup, used as a reference. The popup will be available later using PopupManager.popups[*name*]. |
| title | The title to use in the header bar of the popup. |
| width | The width of the popup in pixels, or 100% for full screen. |
| height | The height of the popup in pixels, or 100% for full screen. |
| type | Whether the content of the popup is in an IFRAME or not. May be “iframe” or “div” |
| path | For IFRAME popups, the URL of the src of the IFRAME. For DIV popups, the ID of the containing DIV |
| hide | The type of full screen popup to close with the close button in the header bar. Can be “disabled”, “loading” or “invisible” |

The PopupManager uses the popup object to create and manage popups. When each popup object is created, it will build and prepare the popup, depending on the type. Full screen popups are simply expanded to the full screen size, positioned to cover the screen, and moved to be the first child of the BODY tag in the DOM. For DIV and IFRAME popups, the container is prepared, including the header bar, and the popup moved to the top of the BODY tag in the DOM. The popup is also sized and positioned centred on the screen.

Popups are positioned fixed, so the screen cannot scroll to show very long popups. The popup is therefore sized to be no higher than the visible height of the screen. Higher popups will scroll within the popup.

### Displaying a popup

The primary method for displaying a popup is the Show(replacements,zindex) method (or the PopupManager.showPopup(name,replacements) wrapper). The important parameter for this method is the replacements parameter. This should be an associative array representing field names and their values. When the method is called, the popup is searched for all SPAN, DIV, INPUT, SELECT and TEXTAREA elements with the ID of each key in the replacements array, and update the innerHTML, value, checked or selected property as applicable. It will also update TinyMCE controls. Note that this replacement will have no effect on IFRAME popups unless the IFRAME has been previously loaded. If the zindex parameter is supplied, the popup will be positioned with that z-index, but otherwise it be positioned one higher than the last popup (starting at 1000000).

IFRAME popups often need to be reloaded whenever they are displayed, and there are two methods to accomplish this. ShowRefreshWithLoad(replacements) will open the popup immediately, and let the IFRAME content itself handle any loading indicator required. ShowRefresh (replacements) will load the IFRAME content fully before showing the popup. This is used with the Loading popup to handle a loading indicator (So, when the user clicks the icon, the Loading popup opens immediately, but the IFRAME popup only appears when it has fully loaded, covering up the loading image).

The standard replacement system doesn’t work with IFRAME popups before they’ve loaded, and they will often need the replacement information to build their content so, instead, these two methods will build a query string of key=value parameters from the replacements array, and append that to the IFRAME URL.

The PopupManager also contains a method to show the popup asynchronously. This only affects Internet Explorer – for all other browsers you can show a popup and then interact with it assuming it has shown. In IE (or at least older versions), the display would be delayed until the entire block of code had been run. To overcome this, use the PopupManager.showAsync(name) method. This is primarily used for full screen popups.

You can hide a popup with the Hide method of the popup object (or the PopupManager.hidePopup(name) wrapper) or, for asynchronous hiding, the PopupManager.hideAsync(name) method. These simply set the display of the popup to none.

### Sizing, positioning and preparing the popup

You can change the size of a popup by calling the SetSize(width,height) method of the popup object. The width and height parameters should be in pixels, and represent the size of the popup DIV or IFRAME, without consideration for the popup container. The method will size the popup, and reposition it to the center of the screen. This method is called automatically when the popup is first created.

If the exact width or height of the popup are not known, you can pass 0 as the width and height. If either the width or height is 0, the popup will attempt to measure itself. To do this, it creates a clone of the DIV, or the BODY of the IFRAME page, displays it off-screen, and measures that. The clone is then destroyed. This can sometimes result in slightly inaccurate sizes if the popup attempts to size itself before all images have loaded, or if the CSS in an IFRAME isn’t available in the main page.

The title of the popup can be changed by calling the SetTitle(title) method. The Prepare(width,height,title) method is shorthand for SetSize(width,height) and SetTitle(title).

### Interacting with a popup

The GetOwnerDocument() method provides access to the document object for the popup. For DIV popups, this is simply document, but for IFRAME popups, this is the document object for the page loaded within the IFRAME, giving you access to all the elements inside it.

You can use the PopupManager.isElementInPopup(elem) method to get a reference to the popup that contains the supplied element. If you pass the element ID as a string, this method will work only on DIV popups. For IFRAME popups, a reference to the actual element is required. This can be useful when references to elements are passed between popups.

# Error Pages

# Sending Emails

# Images

# AJAX

# Users

# Common Functions

# Installing

# Events

The events system allows modules, content blocks and skins to react to certain events within the core system, or in other modules or content blocks. The events system is managed by the Events.lib.php library. There are two parts to events.

## Handling events

The handlers table in the database contains a list of all event handlers that have been registered with the system. Any component of Aloe that can be installed can register (or remove) an event handler. There are three types of event handler, defined by the way the system accesses it:

|  |  |
| --- | --- |
| Type | Description |
| function | The function is defined in a library file. The path to the file is recorded in the path parameter, and this file is automatically included before the function is called. |
| class | The function is a static method of a class. The classname parameter includes the name of the class, which is autoloaded using the standard autoloader. |
| instance | The function is a method of an object. The classname is the name of the variable representing the object. This variable must be available in the global scope when the event is fired. This is not a recommended method. |

The function handling the event will be passed two parameters: the name of the event and an associative array containing any data that might be relevant to the event. The data parameter should be passed by reference, so that any changes to that object or child objects will be available to the method that fired the event. The event should return true of false for success or failure. This result of the handler may or may not be taken into account by the firing method. For example, this method is used by the News module to handle the htmpathResetting event:

function resetHTMPathHandler(**$event**,**&$data**){

resetHTMPathsInTable('news','content',**$data**['oldpath'],**$data**['newpath']);

return true;

}

When a method fires an event, all handlers registered for that event will be called, one at a time, in no particular order.

## Firing events

To fire an event, simply call the fireEvent function in the Events library. Pass it the name of the event, and an associative array of information about the event. This data parameter is passed by reference, so if a handler modifies any of the fields in this array, those changes will be available to your code thereafter.

Events are standardly named in the present tense, describing exactly what is happening – see the list of standard events for examples.

## Standard Events

The following events are fired by the core Aloe system:

### linkSelectorLoading

This event is fired by the Link Selector when it is loaded, and allows additional modules to add extra tabs to the link selector, providing any links that module manages for selection as well. The data object for this event contains the following fields:

|  |  |
| --- | --- |
| Field | Description |
| selected | The currently selected URL. This may be used in the link list produced by the handler to highlight the current selection |
| sourceid | The ID of the source element, used in the javascript for the URL. |
| owner | The owner document, used by the javascript to reference elements in different iframes. |
| tabs | An array of the additional tabs for the selector. |

A handler should add a new array to the tabs field which should contain two fields: description should contain the description of the tab – a single line. The html field should contain the complete HTML code for the selector tab.

### sitemapLoading

This event is fired when the XML sitemap is loaded. It allows additional modules to provide the URLs of any pages that module manages. The data object is simply a list of URLs to be added to the sitemap by all the handlers of the event. To add new URLs to the sitemap, simply add them to the data array.

### htmpathResetting

This event is fired from the HTMPaths library when a URL is reset, and by the Files library when the path or filename of a resource is changed. This event provides modules with the opportunity to update any references they might have recorded. The data parameter includes the following fields:

|  |  |
| --- | --- |
| Field | Description |
| oldpath | The original URL, before it was changed |
| newpath | The updated URL |

# Extending the Core Modules