



# **Introduction to Writing Modules for CMS Made Simple**

**For CMSMS 2.0.1+**

© 2015 – The CMSMS Dev Team

Author: Robert Campbell

# Table of Contents

To Begin.....	1
Requirements.....	1
A Module is More than a Plugin.....	1
The Simplest, Silliest Module.....	3
Tell CMSMS About Your Module.....	3
The Lang File.....	6
The Installation Routine.....	7
Uninstall.....	8
Status Report.....	9
The HolidayItem Class.....	10
Your First Admin Action.....	12
Add Holidays to the Database.....	16
Creating a List View.....	20
Status Report.....	24
Editing an Existing Record.....	24
Deleting a Record.....	25
Our First Front End Action.....	27
A Detail View.....	30
Status Report.....	32
Adding A Detail Page Parameter and a Page limit.....	33
SEO Friendly URLs and Routes.....	36
Registering the Module Name.....	38
Search module Integration.....	39
Lazy Loading.....	42
Asynchronous Content.....	44
Final Status Report.....	46
Taking this Module Further.....	47
Working with Other Modules.....	48
Translations.....	50
What's Changing.....	52
Scratched the Surface.....	52
References.....	53
Conclusion.....	53

## To Begin

This document describes the process of creating a simple module for the latest stable released version of CMSMS—2.0.1 at the time of this writing. The document will outline how to create most of the basic functionality for a database-based content module.

At the end of this tutorial you should have a firm understanding of the basic structure of a CMSMS module, how to create actions and templates for both the admin console and the frontend, how to enable translations, as well as creating and handling forms.

We will be creating a simple module to allow managing holidays. There will be the ability to add, edit, and delete holidays from within the CMSMS admin console, and to view a summary and detail view of holidays from within a CMSMS content page.

We will also describe how to work with pretty URLs, how to cooperate with other modules, and the general structure of a CMSMS module.

## Requirements

To get the most out of this document you will need a firm understanding of working with PHP and object oriented programming, including Exceptions and PHP's magic methods. You will also need a firm understanding of relational databases—particularly MySQL. A basic understanding of javascript and jQuery will be useful, although jQuery is only mentioned in a minor way in this document.

Additionally, you will need a firm understanding of how to work with CMSMS, including creating and managing pages and templates.

Ensure that you are using a recent, supported version of PHP. PHP 5.5+ is assumed. This tutorial was tested on an Ubuntu 14.04 virtual machine with PHP 5.5.9.

## A Module is More than a Plugin

CMS Made Simple has three different ways of adding PHP code to the system: Plugins, User Defined Tags, and Modules.

- Plugins are a single file located in the 'plugins' directory of CMSMS and usually perform a single task—either displaying data, or reformatting it. Plugins typically never write data anywhere, and, in a good design, should not.

These are true Smarty plugins, and can either be function plugins, block plugins, modifiers, or compiler plugins.

Examples of plugins are the {title} and {description} plugins that are shipped with CMSMS that output the title and description fields, respectively, of the current content page.

- User Defined Tags (UDT's) are Smarty plugins that are stored in the database. Again, they are normally small and perform a single task. CMSMS provides an interface in the admin console to create and manage UDTs.

UDTs are implemented as Smarty function plugins so that they can be called within a template. Within CMSMS, UDTs can also be used as event handlers.

Other than auditing, etcetera, UDTs typically should not store any data anywhere, and should be used for the exact same purpose as plugins. They also should not output complex HTML code, as that is the task of the template.

- Modules are larger sets of code that solve a single problem. They may create and manage one or more database tables, create numerous views including forms, process forms, and have a great deal of functionality.

Modules can also have functionality not available to plugins or UDTs. e.g: creating a new content page type, a new page block type, or interacting as a WYSIWYG or Syntax Highlighting module. This extended functionality will be discussed in subsequent documents.

Below is a brief chart that illustrates the primary differences modules, UDTs, and plugins. This chart should help you to understand why modules are usually the best answer when writing new code.

	<b>Plugin</b>	<b>UDT</b>	<b>Module</b>
A Single PHP function	YES	YES	
Can Have an admin Interface			YES
Can Handle Events		YES	YES
Has Install and Uninstall			YES
Handles Routing			YES
Can be shared on the forge			YES
Can Create Events			YES
Can create page types			YES
Can Create PseudoCron Tasks			YES
Can be Modifier / Compiler / Block	YES		YES*
Provide Numerous plugins			YES

	Plugin	UDT	Module
Provide functions and classes for other modules			YES

\* Modules can register multiple different plugins of different types, however, usually it is just a function plugin.

## The Simplest, Silliest Module

### STEPS:

1. Create a directory within the CMSMS modules directory. The name of this new directory must be the name of our module. In this case: 'Holidays'.

```
cd /home/website/public_html/modules
mkdir Holidays
```

2. Create a file within the new 'Holidays' directory entitled Holidays.module.php

```
cd Holidays
emacs Holidays.module.php1
```

3. Create a class within the Holidays.module.php file that extends CMSModule.

```
class Holidays extends CMSModule
{
}
```

#### Note:

- The directory name, the module file name, and the class name must be named consistently.
- CMSMS does not currently support namespaced module classes
- CMSMS is case sensitive, so always match the case when creating your module directory, module class file, and the class itself.

### Discussion

At this point we have our basic module skeleton with an empty class. The module is installable at this point from the ModuleManager module, but does not have any functionality of any sort.

## Tell CMSMS About Your Module

Now we have the basic structure of the module as CMSMS requires it, but CMSMS knows nothing about our module, its planned functionality, who the author is, or about this module's planned

---

<sup>1</sup> The author's environment of choice is linux, and editor of choice is emacs. Your's may differ.

functionality. To do this we need to tell CMSMS about our module. We do that by overriding a few virtual methods in our new module class.

The CMSMS Module API provides numerous virtual methods for interacting with the system, other modules, and performing different tasks. We will describe only a few of them here, and a few more as the document progresses.

- `GetName()`

This virtual method defines the name of your module. It must return the exact name of the directory of the module. Normally this method does not need to be overridden.

- `GetVersion()`

This virtual method returns the version number of your module. The version number is used to see if the upgrade routine must be executed.

- `GetFriendlyName()`

This method returns the string that will be used when placing this module in the admin navigation (if the module has an admin interface).

- `GetAdminDescription()`

This virtual method returns a string that describes your module. It is used from within the admin interface.

- `IsPluginModule()`

This method returns a boolean indicating whether or not your module will interact with the frontend of CMSMS.

- `HasAdmin()`

This method returns a boolean indicating whether or not your module will have a panel in the CMSMS admin console.

- `VisibleToAdminUser()`

This method returns a boolean indicating whether your module is visible within the admin console to the currently logged in admin user. The module must also return true for [HasAdmin\(\)](#)

- `GetDependencies()`

This method must return an associative array of module names, and minimum version numbers of modules that your module requires in order to properly operate. If your module does not require any other modules to operate completely then you can ignore this method.

- `GetHelp()`

This method returns a string representing the body portion of an HTML document containing help for your module.

There are two ways in which this help data can be displayed:

- By clicking on the “Module Help” link at the top of the CMSMS admin console when one of your module actions is displayed.
- By clicking on the 'Help' link next to your module while within the ModuleManager module. Note: your module's help can be displayed even if your module is not yet downloaded onto the website, so you cannot rely on relative links to images.

Like the `GetHelp()` method, the output of this function can be displayed from within the ModuleManager without even having downloaded the module into your website.

- `MinimumCMSVersion()`

This method returns a string that is the absolute minimum version of CMS Made Simple that your module requires to operate properly.

- `GetAuthor()`

This method returns a string containing the module author's full name.

- `GetAuthorEmail()`

This method returns a string containing the module author's email address.

## STEPS:

Modify your module class file now to look like this:

- `GetChangelog()`

This method returns a string representing the body portion of an HTML document that contains the history of changes for your module.

```
<?php
class Holidays extends CMSModule
{
    public function GetVersion() { return '0.1'; }
    public function GetFriendlyName() { return $this->Lang('friendlyname'); }
    public function GetAdminDescription() { return $this->Lang('admindescription'); }
    public function IsPluginModule() { return TRUE; }
    public function HasAdmin() { return TRUE; }
    public function VisibleToAdminUser() { return TRUE; }
    public function GetAuthor() { return 'Your Name'; }
    public function GetAuthorEmail() { return 'yourname@somedomain.com'; }
}
?>
```

## DISCUSSION:

Some details on our decisions here.

- CMSMS stores the currently 'installed' version number of a module in its database. When the version number changes, CMSMS compares that version number with the one in the file to tell if the module upgrade routine needs to be executed.

The version number string must be compatible with PHP's `version_compare()` function.

- We intend that this module will have a front end interface, therefore we have returned TRUE for the `IsPluginModule()` method.
- We also intend that this module will have a panel in the CMSMS admin console, so we have returned TRUE for the `HasAdmin()` method.
- At this time anybody with a valid login to the CMSMS admin console can interact with our module. Though we will change this later, for now we return TRUE for the `VisibleToAdminUser()` method.

## The Lang File

You will notice above that we introduced a new method: `$this->Lang('friendlyname');` This method is used to translate a string key into a human readable string in the current users selected language, or English as a default.

Language strings are stored in a PHP file within the lang folder of your CMSMS module directory. e.g:

```
modules/  
  \- Holidays/  
      \- Holidays.module.php  
      \- lang/  
          \- en_US.php
```

## STEPS:

Go ahead and create the lang directory now, and create a new file entitled en\_US.php within that directory.

```
mkdir lang  
emacs lang/en_US.php
```

The format of this file should look like this:

```
<?php  
$lang['friendlyname'] = 'Holidays';  
$lang['admindescription'] = 'A module for managing and displaying holidays';  
?>
```

## DISCUSSION:

We will be adding many more strings into this file as we develop our module. It is handy to keep this file open in your editor.



As described later in this document, the lang file is also the base document used to translate your module's strings into different languages.

## The Installation Routine

Most modules need an installation routine in order to create preferences and permissions, default settings, and any database tables and default data that is required for the module to operate properly. The method.install.php file handles this for CMSMS modules. This file must be placed in the module's directory.

Our module needs to create a permission so that only certain admin users will have access to manage holidays, and it needs a table for the holidays itself.

### STEPS:

1. Create a new file entitled method.install.php inside your modules/Holidays directory, with contents like this:

```
<?php
if( !defined('CMS_VERSION') ) exit;
$this->CreatePermission(Holidays::MANAGE_PERM, 'Manage Holidays');

$db = $this->GetDb();
$dict = NewDataDictionary($db);
$staboptarray = array('mysql' => 'TYPE=MyISAM');
$fields = "
    id I KEY AUTO,
    name C(255) KEY NOTNULL,
    description X,
    published I1,
    the_date I NOTNULL
";
$sqlarray = $dict->CreateTableSQL(CMS_DB_PREFIX.'mod_holidays',$fields,$staboptarray);
$dict->ExecuteSQLArray($sqlarray);
```

2. Add the following to your Holidays.module.php file

```
<?php
class Holidays extends CMSModule
{
    const MANAGE_PERM = 'manage_holidays';
    ...
}
```

3. Modify the VisibleToAdminUsers() method of your Holidays class as follows:

```
public function VisibleToAdminUser() { return $this->CheckPermission(self::MANAGE_PERM); }
```

### DISCUSSION:

The method.install.php file:

- The first thing we do is check if CMS\_VERSION is defined. If it is not, we simply exit. This is a security precaution and prevents your method.install.php file from being requested by a browser directly. Instead, the file can only be read and executed from within the scope of

CMSMS.

- Next we create a new permission into CMSMS using the module API's `CreatePermission()` convenience method. We provide the permission name (as a class constant for easy typing) and a human readable string for the permission that will appear in the CMSMS admin console. Note: CMSMS does not have a mechanism for localizing permission names, so your permission's human-readable string should usually be in English
  - Next we get a reference to the global database object provided by CMSMS by calling the module's `GetDb()` convenience method.
  - The following lines create the database table called `mod_holidays` that is prefixed with the same table prefix that is specified during the CMSMS installation process.
- Notes:**
- CMSMS uses a modified, trimmed down version of `adodb-lite` (a PHP database abstraction library), although we only support MySQL at this time.
  - A single connection to the CMSMS database is created on each request. The `GetDb()` method returns a reference to that single connection.
  - The “Data Dictionary” class provides a database-agnostic method of interacting with databases and tables. Though we will not detail the language and class itself in this document, you can find a reference to the `DataDictionary` language specification, and its methods [NEED LINK](#)

## The Holidays.module.php file

The new class constant minimizes the risk of an accidental typo preventing access to your module. Wherever a string is going to be reused throughout your module it may be a good idea to use a class constant.

The change to the `VisibleToAdminUsers()` method will ensure that only the admin users that have the permission we created in the `method.install.php` will have the ability to interact with this module's admin panel.

Note: Admin users who are members of the 'admin group' automatically have all permissions without the need to explicitly assign them.

## General Notes:

- The `method.install.php` file is executed within the scope of the `CMSModule`. Therefore you have access to all of your module's public and protected methods and members.
- `$this` refers to an instance of your module class.

## Uninstall

Modules must, when uninstalled permanently, remove all data specific to the module and restore everything to a state similar to before the module was installed. This allows a module to be installed, tested, and uninstalled multiple times on a website without polluting the database with unnecessary tables, templates, stylesheets, or other data that the website administrator would then need to clean.

## STEPS:

1. Create a new file in your modules/Holidays directory entitled method.uninstall.php and copy in this code:

```
<?php
if( !defined('CMS_VERSION') ) exit;
$this->RemovePermission(Holidays::MANAGE_PERM);

$db = $this->GetDb();
$d = NewDataDictionary( $db );
$sqlarray = $d->DropTableSQL( CMS_DB_PREFIX.'mod_holidays');
$d->ExecuteSQLArray($sqlarray);
```

2. Add the following method to your Holidays.module.php module class file:

```
public function UninstallPreMessage() { return $this->Lang('ask_uninstall'); }
```

3. Add the following line to your lang/en\_US.php file:

```
$lang['ask_uninstall'] = 'Are you sure you want to uninstall the Holidays module? All holiday data will be permanently deleted.';
```

## DISCUSSION:

### The method.uninstall.php file

Just like in the installation routine, we first check that CMS\_VERSION is defined as a security precaution.

Next, we remove the permission that was created in the method.install.php

Thirdly, we create a new Data Dictionary object using the global database connection instance, and execute the commands to drop the table from the database.

If necessary, we would also delete events, event handlers, and preferences that our module created in the install, or during its execution.

### The Holidays.module.php file

The `UninstallPreMessage()` virtual method of the CMSModule class returns a string that is displayed to the admin user to confirm that the user wants to uninstall the module. The lang/en\_US.php file contains the full text for the key 'ask\_uninstall'.

This will ensure that the admin user receives a presentable message to confirm the un-installation of the module. This string can also be translated to other languages.

## Status Report

We now have a working module named 'Holidays'. The module can be installed and uninstalled by using the ModuleManager module within the CMSMS admin console.

The module will create a permission and a database table upon installation, and remove them when the module is uninstalled.

We can use the CMSMS admin console to specify which admin users will have access to this module. If you install this module you will see the 'Holidays' item appear in the admin panel's navigation, and you can click on the item.

Our directory structure currently looks like this:

```
modules/Holidays/  
  \- Holidays.module.php  
  \- method.install.php  
  \- method.uninstall.php  
  \- lang  
     \- en_US.php
```

Our module still has no functionality of its own at this point, but that is coming soon.

## The HolidayItem Class

For the purpose of clean, extendable, re-usable, and readable code, we will create a basic PHP class that allows us to manage a single Holiday record. We will be using this class in almost all of our actions.

### STEPS:

1. Create a lib directory below your modules/Holidays directory.
2. Create a file called class.HolidayItem.php file within the newly created lib directory and paste in this code:

```
<?php  
class HolidayItem  
{  
    private $_data = array('id'=>null, 'name'=>null, 'description'=>null,  
                           'published'=>null, 'the_date'=>null);  
  
    public function __get($key)  
    {  
        switch( $key ) {  
            case 'id':  
            case 'name':  
            case 'description':  
            case 'published':  
            case 'the_date':  
                return $this->_data[$key];  
        }  
    }  
  
    public function __set($key,$val)  
    {  
        switch( $key ) {  
            case 'name':  
            case 'description':  
                $this->_data[$key] = trim($val);  
                break;  
            case 'published':  
                $this->_data[$key] = (bool) $val;
```

```

        break;
    case 'the_date':
        $this->_data[$key] = (int) $val;
        break;
    }
}

public function save()
{
    if( !$this->is_valid() ) return FALSE;
    if( $this->id > 0 ) {
        $this->update();
    } else {
        $this->insert();
    }
}

public function is_valid()
{
    if( !$this->name ) return false;
    if( !$this->the_date ) return false;
    return TRUE;
}

protected function insert()
{
    $db = \cms_utils::get_db();
    $sql = 'INSERT INTO '.CMS_DB_PREFIX.'mod_holidays
        (name,description,published,the_date)
        VALUES (?, ?, ?, ?)';
    $dbr = $db->Execute($sql,array($this->name,$this->description,$this->published,
        $this->the_date));

    if( !$dbr ) return FALSE;
    $this->_data['id'] = $db->Insert_ID();
    return TRUE;
}

protected function update()
{
    $db = \cms_utils::get_db();
    $sql = 'UPDATE '.CMS_DB_PREFIX.'mod_holidays SET name = ?, description = ?,
        published = ?, the_date = ? WHERE id = ?';
    $dbr = $db->Execute($sql,array($this->name,$this->description,$this->published,
        $this->the_date,$this->id));

    if( !$dbr ) return FALSE;
    return TRUE;
}

public function delete()
{
    if( !$this->id ) return FALSE;
    $db = \cms_utils::get_db();
    $sql = 'DELETE FROM '.CMS_DB_PREFIX.'mod_holidays WHERE id = ?';
    $dbr = $db->Execute($sql,array($this->id));
    if( !$dbr ) return FALSE;
    $this->_data['id'] = null;
    return TRUE;
}

/** internal */
public function fill_from_array($row)
{
    foreach( $row as $key => $val ) {
        if( array_key_exists($key,$this->_data) ) {
            $this->_data[$key] = $val;
        }
    }
}

public static function &load_by_id($id)
{

```

```

        $id = (int) $id;
        $db = \cms_utils::get_db();
        $sql = 'SELECT * FROM '.CMS_DB_PREFIX.'mod_holidays WHERE id = ?';
        $row = $db->GetRow($sql,array($id));
        if( is_array($row) ) {
            $obj = new self();
            $obj->fill_from_array($row);
            return $obj;
        }
    }
}
?>

```

## DISCUSSION:

CMS Made Simple does not provide or enforce a generic base model class for use in the Model/View/Controller paradigm, it is left to the module author to define these using their own standards.

Though it is not strictly necessary to create model classes such as this to work with your data items, it is considered good practice, allows for re-use of code, and makes the coding easier to read and understand. We will be using this class throughout the tutorial.

You can see a few things in this class file:

- We start to use the larger CMSMS API by calling methods like `cms_utils::get_db()`, `$db->GetRow()`, and `$db->Execute()` to get a reference to the global database connection object from within our data item class, and to work with the database. CMSMS has an extensive and documented API to allow for easily building different types of modules and interacting with the data in the application.
- We are using 'Parameterized Queries' which means that we are using the ? as a placeholder in our SQL statements, and then passing in an array of parameters. The database abstraction class can quote the parameters properly based on the input data type. This is a security mechanism to help minimize the chance of SQL injection attacks.

## Your First Admin Action

Okay, we are finally ready to make our new module do something! Our first action will be the beginning of our admin dashboard for holidays. Initially, it will display a link to a second action to allow us to add new holidays to the database. Later, this action will be expanded to allow us to list, edit, and delete holidays.

## STEPS:

1. Create a new file in your modules/Holidays directory entitled `action.defaultadmin.php` and copy in the following code:

```
<?php
```

```

if( !defined('CMS_VERSION') ) exit;
if( !$this->CheckPermission(Holidays::MANAGE_PERM) ) return;

$tpl = $smarty->CreateTemplate(
    $this->GetTemplateResource('defaultadmin.tpl'), null, null, $smarty);
$tpl->display();

```

2. Create a new directory named 'templates' in your modules/Holidays directory, and within that directory create a new file called 'defaultadmin.tpl' and copy the following code into it:

```

<div class="pageoptions">
  <a href="{cms_action_url action=edit_holiday}">{admin_icon icon='newobject.gif'} {$mod-
  >Lang('add_holiday')}</a>
</div>

```

3. Add the following line into your modules/Holidays/lang/en\_US.php file

```
$lang['add_holiday'] = 'Create a New Holiday';
```

## DISCUSSION:

Woah—there's a lot going on there in just a few lines of code! And what's this action file stuff? Don't worry, we'll go through it.

### About Actions:

- CMS Made Simple module controllers are called 'actions'. When an action for a module is requested the system will look in the module directory for a file entitled action.**actionname.php**.
- There are a few 'special' actions:
  - defaultadmin
 

This action is called when there is no action name specified, and the request is for an admin interface form. Typically, this is done from the admin navigation to display your modules primary admin panel.
  - default
 

This action is similar to the defaultadmin action but is only for frontend requests when an action name is not explicitly specified. i.e: {cms\_module module=Holidays}
  - defaulturl (*deprecated*)
 

This special action will be discussed will be discussed in further detail below, however it is used when a route is found for the module via an HTTP request but no action can be determined for the route.
- Module action requests can be made via HTTP requests, or via calling the module from within a Smarty template.
- Some variables are passed into the action file for general use:

- `$smarty` (*object*)

This variable is a Smarty template object representing the current Smarty scope. It is not necessarily the global `$smarty` object returned by `\Smarty\CMS::get_instance()`

- `$action` (*string*)

This is the name of the action that is being called, for convenience.

- `$id` (*string*)

This is the unique module-action id. Because on the CMSMS frontend the same module action may be called multiple times on the same request, this id is generated to allow the system to handle matching the proper data to the proper module call.

For admin requests this is always 'm1\_'.

- `$returnid` (*int|empty*)

This is the numeric page id that is currently being rendered, and that the module action is for. It is used when creating links or form elements.

There is no concept of a 'page' for admin requests, so when a module action is called for an admin request the returnid is always empty.

- `$params` (*array*)

This is the input parameters to the module. Parameters passed to the module either on the HTTP request, or in the module call will be in this array, with some limitations.

For front end requests, the `$params` array only contains the registered, and cleaned params. This is a security mechanism. See the section below on registering parameters.

- `$db` (*object*)

For convenience and compatibility reasons, a reference to the global database connection object is in scope.

- `$gCms` (*object*)

For convenience and compatibility reasons, a reference to the `CmsApp` object is in scope and is called `$gCms`.

**Note:** Unlike in early versions of CMSMS `$gCms` is not a global variable.

## About the PHP Code:

If you look at the `action.defaultadmin.php` file, at the top you will see the standard security check that we had on the installation and uninstall routines to ensure that your action files are not requested



directly, but only from within the context of CMSMS.

Next, we use the `CheckPermission()` method of the module API (from which our module is derived) to ensure that the currently logged in admin user has the proper permission to access this action. This is useful in the event that a user somehow gets a URL to this admin action from another location other than the standard admin navigation.

Thirdly, we call `$this->GetTemplateResource('defaultadmin.tpl')`. This module API function transforms the 'defaultadmin.tpl' string into a Smarty resource specification string. We pass the results of this call into `$smarty->CreateTemplate()`. This functional call is a Smarty method that creates a new template variable scope.

Lastly, we call `$tpl->display()` to actually process and display the 'defaultadmin.tpl' template file.

We do not pass any data into the new template object as of yet, but that will be added soon.

**Note:** It is not strictly necessary to create a new Smarty scope, and you can assign variables to the Smarty object passed in to your action. However, we recommend that in order to minimize memory requirements and avoid accidentally overwriting variables created by another action using the same scope that you create a new Smarty scope for all of your module actions.

## About the Smarty Template:

Inside a `<div class="pageoptions">` we create a link. The URL to that link is created by the `{cms_action_url}` Smarty plugin. The plugin automatically knows the module name, however we specify the action to link to as 'edit\_holiday'.

The visible portion of the link consists of an icon and some text. The icon is generated by the `{admin_icon}` Smarty plugin, and the text is generated by the `{$mod->Lang(...)}` call.

There are some relevant variables automatically provided to the Smarty template available in your module action calls

- `$actionid` (*string*)

This is the same as the `$id` variable in the module action file and is provided to Smarty for convenience in link building, etc.

- `$returnid` (*string*)

This is the same as the `$returnid` variable in the module action file and is provided to Smarty for convenience.

- `$actionmodule` (*string*)

This is the name of the module for which we are currently executing an action.

- `$actionparams` (*array*)

This is the same as the `$params` array in the module action file.

- `$mod` (*object*)

A reference to the module object, for convenience in calling the `Lang()` and other module methods from within your templates.

These variables will automatically be available to your templates, even if creating a new Smarty scope.

## Add Holidays to the Database

Okay, now we have a simple admin panel with a single link on it (neat, it has an icon too). But our module still does nothing. The next thing we need to do is to add holidays to the database. To do that, we need to display a form, and handle the results of said form.

### STEPS:

1. Create a new action file entitled `action.edit_holiday.php` in your `modules/Holidays` directory and in it, paste this code:

```
<?php
if( !defined('CMS_VERSION') ) exit;
if( !$this->CheckPermission(Holidays::MANAGE_PERM) ) return;

$holiday = new HolidayItem();
if( isset($params['cancel']) ) {
    $this->RedirectToAdminTab();
}
else if( isset($params['submit']) ) {
    $holiday->name = trim($params['name']);
    $holiday->published = cms_to_bool($params['published']);
    $holiday->the_date = strtotime($params['the_date']);
    $holiday->description = $params['description'];
    $holiday->save();
    $this->SetMessage($this->Lang('holiday_saved'));
    $this->RedirectToAdminTab();
}

$tpl = $smarty->CreateTemplate($this->GetTemplateResource('edit_holiday.tpl'),null,null,
$smarty);
$tpl->assign('holiday',$holiday);
$tpl->display();
?>
```

2. Create a new template file entitled `edit_holiday.tpl` in your `modules/Holidays/templates` directory and in it, paste this code:

```
<h3>{$mod->Lang('add_holiday')}</h3>
{form_start}
<div class="pageoverflow">
    <p class="pageinput">
        <input type="submit" name="{ $actionid }submit" value="{ $mod->Lang('submit') }"/>
        <input type="submit" name="{ $actionid }cancel" value="{ $mod->Lang('cancel') }"/>
    </p>
</div>
<div class="pageoverflow">
```

```

<p class="pagetext">{$mod->Lang('name')}:</p>
<p class="pageinput">
  <input type="text" name="{\$actionid}name" value="{\$holiday->name}"/>
</p>
</div>
<div class="pageoverflow">
  <p class="pagetext">{$mod->Lang('date')}:</p>
  <p class="pageinput">
    <input type="date" name="{\$actionid}the_date" value="{\$holiday->the_date|date_format:'%Y-
    %m-%d'}"/>
  </p>
</div>
<div class="pageoverflow">
  <p class="pagetext">{$mod->Lang('published')}:</p>
  <p class="pageinput">
    <select name="{\$actionid}published">
      {cms_yesno selected=\$holiday->published}
    </select>
  </p>
</div>
<div class="pageoverflow">
  <p class="pagetext">{$mod->Lang('description')}:</p>
  <p class="pageinput">
    {cms_textarea prefix=\$actionid name=description value=\$holiday->description
    enablewysiwyg=true}
  </p>
</div>
{form_end}

```

3. Add the following to your modules/Holidays/lang/en\_US.php file:

```

$lang['holiday_saved'] = 'This holiday is now saved';
$lang['submit'] = 'Submit';
$lang['cancel'] = 'Cancel';
$lang['name'] = 'Name';
$lang['date'] = 'Date';
$lang['published'] = 'Published';
$lang['description'] = 'Description';

```

## Appearance:

If you now visit your module's admin panel by navigating to “Extensions >> Holidays” within the CMSMS admin console, and then click on the “Add new Holiday” link, you will see a form like this:

The screenshot shows a web form titled "Create a New Holiday". At the top left, there are two buttons: "Submit" (with a checkmark icon) and "Cancel" (with an 'X' icon). Below these are four main sections:

- Name:** A simple text input field.
- Date:** A date picker widget showing "dd/mm/yyyy".
- Published:** A dropdown menu currently set to "No".
- Description:** A rich text editor (WYSIWYG) with a toolbar containing icons for undo, redo, bold, italic, underline, bulleted list, numbered list, link, unlink, insert image, and insert video. The text area below the toolbar is empty.

At the bottom right of the form, there is a small status indicator that says "Words: 0".

## DISCUSSION:

### The PHP Code

Here is a description of the code within the `action.edit_holiday.php` action file:

- The `action.edit_holiday.php` script begins with the two security checks we have previously described.
- The following line is interesting, it creates a new object of type `HolidayItem` using the class we have previously defined and placed in our `lib` directory. This class is automatically loaded for this request directly from the `lib` directory. There is no need to include it manually, thanks to the CMSMS autoload mechanism.
- Following that is a bit of logic that handles form processing when a form is submitted via the submit or cancel buttons.

If the form is submitted via the cancel button, we use the CMSMS module API function `CMSModule::RedirectToAdminTab()` to automatically redirect us back to the defaultadmin action. If our defaultadmin action were using a tabbed interface, we could have specified a tab name here.

If the form is submitted via the submit button, we fill the holiday object with the data from the form and save the holiday. You will notice that we convert the value of the 'the\_date' input parameter to a unix timestamp using `strtotime()` and that we convert the value of the 'published' input parameter to a boolean using the `cms_to_bool()` method, a utility function provided with CMSMS that is capable of interpreting a string in many formats and converting it to a boolean.

- Following that, we set a message using the `CMSModule::SetMessage()` method, and redirect back to the primary admin interface (the defaultadmin action) for our module.
- At the bottom of the file, we once again create a new Smarty template object for the 'edit\_holiday.tpl' template. We then assign the holiday object we created above to it, and then display the template.

### The Form Template

The form template looks much like a normal HTML form intermixed with a bunch of Smarty tags. Lets start from the top and work our way to the bottom:

- At the very top of the template is a header, using a language string. As we mentioned before, `$mod` is a reference to your module object that is automatically provided to your Smarty template by CMSMS.

The “add\_holiday” key is already in your lang file from a previous step, so here we show re-using language strings.

- On the second line of the form template is the `{form_start}` tag. This is a Smarty plugin provided by CMSMS that creates a `<form>` tag with all of the data needed to specify the destination php script, module action, and other required parameters so that when the form is submitted our action file will be executed, with the proper data.

This Smarty plugin automatically determines the module name, and module action that it should by default return to. But this, and many other things, are settable by adding arguments to this tag. Please read the documentation for this plugin by navigating to “Extensions >> Tags” in your CMSMS Installation.

- Next, we create an area with submit and cancel buttons. Note that we do not use special tags for this, instead we build them directly in HTML to allow the greatest flexibility in styling and functionality.

Each of the submit button names are prefixed with the `{ $actionid }` Smarty variable. As mentioned previously because CMSMS allows a module action to be called (on the front end) multiple times, the actionid is used to identify and collect the parameters for each call separately.

**Note:** Each and every parameter passed to a CMSMS module action must be prefixed with the actionid so that the data can be transmitted to the `$params` array in your action.

- We use the `{ $mod->Lang( ) }` method throughout the template for providing human readable strings that can also be translated to other languages.
- Following this, we create the name and date input fields for our form. Again, we prefix the name of the input field with the `{ $actionid }` Smarty variable. We also use the `$holiday` object variable that we provided to Smarty from within in our action.php file to provide the values for our input fields.

The date input field requires data in YYYY-MM-DD format, so we use the `date_format` Smarty modifier to convert the unix timestamp we store in the database to the proper format.

- Next, we create the published select box, it is a standard select box whose name is again prefixed with `{ $actionid }`, and then we use the `{ cms_yesno }` plugin to automatically provide the options for the dropdown. We pass the `$holiday->published` flag to this plugin to indicate which item should be selected.
- After this, we create an area for the description field. Again, we use the `{ $mod->Lang( ) }` method to provide a human readable string. We use the `{ cms_textarea }` plugin to create a WYSIWYG enabled text area for editing the description.

The `{cms_textarea}` plugin accepts a separate prefix argument to which we pass the `$_actionid` variable. We specify a name, and we provide the description stored within the holiday object (if any) for the default contents.

- You will see throughout this document that we commonly use a few special classes when formatting our admin template. These are `<div class="pageoverflow">` and within that a `<p class="pagetext">` and `<p class="pageinput">`. Earlier in this document we also used `<div class="pageoptions">`.

These are styles that the CMSMS admin theme understands and knows how to format. It is the beginning of a 'style guide' within CMSMS.

The admin theme also has a basic grid system for laying out forms using a 12 column grid.

## General Discussion

There is a lot to take away from this code, even though the code is relatively short:

- The Model/View/Controller Paradigm & the Separation of Concerns

Although you must provide your own models, CMSMS does support the Model/View/Controller paradigm. Using the action file as your controller, and the Smarty template as your view.

In our code, the controller is clean, simple, and functional—and there is no polluting of the controller code with view related issues.

- Flexibility in Form Building

Because the entire form is built as a Smarty template, to which we have provided the model, we are free to create our form in any style we want—using all of the HTML that we want—with very few constraints.

There are numerous Smarty plugins provided both by Smarty itself, and by CMSMS to aid in form building. As you see, there are plugins for creating the form itself, creating text areas, populating select boxes, manipulating dates, etc.

- Naming of input fields

We have learned that in order for the content of input field values to be routed back to your controller when a form is submitted, each and every input field must be prefixed with the `{$_actionid}`.

## Creating a List View

At this point we can add new holidays to the database, but we cannot yet do anything with them. The

next thing that we need to do is to modify our defaultadmin action to display our holidays in some kind of list, so that we can edit them and delete them.

## STEPS

1. Create a new file called class.HolidayQuery.php in your modules/Holidays/lib directory and paste in the following code that will act as the model for the data in our defaultadmin action:

```
<?php
class HolidayQuery extends CmsDbQueryBase
{
    public function execute()
    {
        if( !is_null($this->_rs) ) return;
        $sql = 'SELECT SQL_CALC_FOUND_ROWS H.*
              FROM '.CMS_DB_PREFIX.'mod_holidays H ORDER BY the_date DESC';
        $db = \cms_utils::get_db();
        $this->_rs = $db->SelectLimit($sql,$this->_limit,$this->_offset);
        IF( $db->ErrorMsg() ) throw new \CmsSQLException($db->sql.' -- '.$db->ErrorMsg());
        $this->_totalmatchingrows = $db->GetOne('SELECT FOUND_ROWS()');
    }

    public function &GetObject()
    {
        $obj = new HolidayItem;
        $obj->fill_from_array($this->fields);
        return $obj;
    }
}
?>
```

2. Modify your modules/Holidays/action.defaultadmin.php file to look like this:

```
<?php
if( !defined('CMS_VERSION') ) exit;
if( !$this->CheckPermission(Holidays::MANAGE_PERM) ) return;

$query = new HolidayQuery;
$holidays = $query->GetMatches();

$tpl = $smarty->CreateTemplate($this->GetTemplateResource('defaultadmin.tpl'),null,null,
$smarty);
$tpl->assign('holidays',$holidays);
$tpl->display();
```

3. Modify your modules/Holidays/templates/defaultadmin.tpl file to look like this:

```
<div class="pageoptions">
  <a href="{cms_action_url action=edit_holiday}">{admin_icon icon='newobject.gif'}
    {$mod->Lang('add_holiday')}
  </a>
</div>
{if !empty($holidays)}
<table class="pagetable">
  <thead>
    <tr>
      <th>{$mod->Lang('name')}</th>
      <th>{$mod->Lang('date')}</th>
      <th class="pageicon">{* edit icon *}</th>
      <th class="pageicon">{* delete icon *}</th>
    </tr>
  </thead>
  <tbody>
    {foreach $holidays as $holiday}
      {cms_action_url action=edit_holiday hid=$holiday->id assign='edit_url'}
```

```

<tr>
  <td><a href="{ $edit_url}" title="{ $mod->Lang('edit')}">{ $holiday->name}</a></td>
  <td>{ $holiday->the_date|date_format: '%x'}</td>
  <td><a href="{ $edit_url}" title="{ $mod->Lang('edit')}">
    {admin_icon icon='edit.gif'}</a>
  </td>
  <td>{* delete link will go here *}</td>
</tr>
</foreach>
</tbody>
</table>
{/if}

```

4. Add the following line to your modules/Holidays/lang/en\_US.php file:

```
$lang['edit'] = 'Edit this';
```

## DISCUSSION

### The Query Class

This class makes use of the `CmsDbQueryBase` class to provide a convenient way of querying the database and iterating through, or returning, results. It handles pagination, and can quite easily handle filtering etc., by modifying the code in the `execute()` method. We will use this class to return a set of `HolidayItem` objects for display.

By looking at this class definition you can see that we do a simple test to ensure that we do not execute the query more often than necessary. Then we get a reference to the global CMSMS database connection object, and query the database. We assign the results to `$this->_rs`. After this, we again query the database to find the total number of items that would have matched our query, not including any page limit.

The `GetObject()` method is a pure virtual method of this class, and is used to convert the data returned from the database into a usable object instance. In this case, `HolidayItem` classes. The `GetMatches()` method from the `CmsDbQueryBase` class is a simple wrapper method that iterates through the result set, and convert the results from database rows into an array of `HolidayItem` objects.

### The defaultadmin Action File

This file is still relatively simple, and very similar to what we had before. It would get more complex if we added pagination and filtering capabilities into the admin panel.

After the normal security checks that we have discussed previously, we create an object of type `HolidayQuery`. Again, the autoloader included with CMS Made Simple helps with this so that we do not have to include the `HolidayQuery` class manually. Then we call the `$query->GetMatches()` method to get our list of `HolidayItem` objects representing holidays we have created in the database.

Below that, after we have created our Smarty template object, we give our matching `HolidayItem` objects to Smarty for use in the template.



Our controller is clean and functional. Also, there is no view functionality polluting this code.

We are demonstrating code-reuse by reusing the `HolidayItem` class. Later we will reuse both classes in our front end summary view.

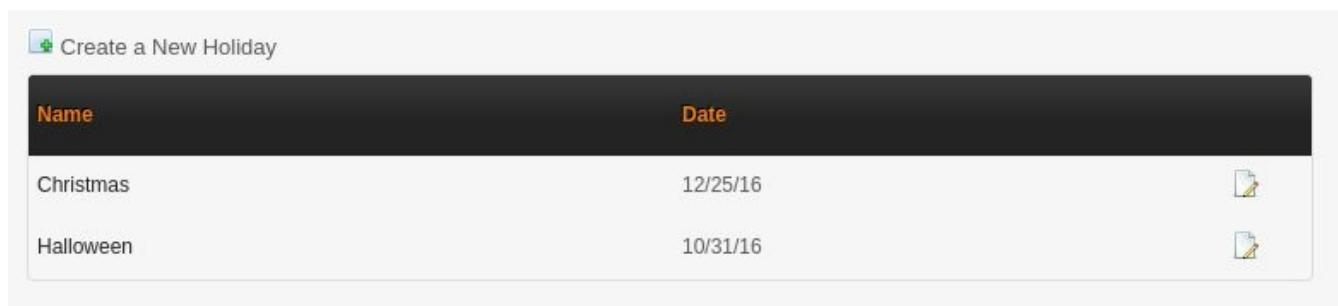
## The defaultadmin template

This is where the real magic of our list view happens, and there is a fair bit of stuff happening:



- At the top of the file is still the same link to the action for adding a new holiday entry.
- Next, there is a Smarty `{if}` expression to test if we have any holiday entries. If we do have holiday entries, then we define a table in which to display them.
- Inside the `<tbody>` definition for the table, we use the Smarty `{foreach}` plugin to iterate through all of the `HolidayItem` objects returned in our query. Each `HolidayItem` will be rendered as a single table row.
- Inside the `{foreach}` loop we use the `{cms_action_url}` plugin to create a URL to the `edit_holiday` action, and we pass in an additional parameter called 'hid' with the numeric, unique id of the holiday item. We assign the output of that plugin to a Smarty variable entitled `$edit_url` so that we can re-use that variable in a few places.
- Next, we build our row. The name column builds a link to the edit action, using the event name as the contents of the link. The date column uses the Smarty `date_format` modifier to format the date of the event in a locale acceptable manner. We then create another link to the edit action for this holiday, using an icon generated by the `{admin_icon}` plugin.
- If you also notice, we have made room on our table for another icon for the delete action, which we will fill in later.

## Appearance:

After you have created a few holiday items, your Holidays module admin panel should look something like this:



The screenshot shows a web interface for managing holidays. At the top left, there is a button labeled "Create a New Holiday" with a green plus icon. Below this is a table with a dark header. The table has two columns: "Name" and "Date". There are two rows of data: "Christmas" with date "12/25/16" and "Halloween" with date "10/31/16". To the right of each row, there is a small icon representing an edit or delete action.

Name	Date	
Christmas	12/25/16	
Halloween	10/31/16	

# Status Report

So, where are we at? Our module now can create holiday entries, and display them in a list in our admin panel. We have created links that will (after the next step) allow us to edit those holiday items.

Our module directory is now filling out, and now looks like this:

```
modules/Holidays
\ - Holidays.module.php
\ - action.defaultadmin.php
\ - action.edit_holiday.php
\ - method.install.php
\ - method.uninstall.php
\ - lib
  \ - class.HolidayItem.php
  \ - class.HolidayQuery.php
\ - templates
  \ - defaultadmin.tpl
  \ - edit_holiday.tpl
\ - lang
  \ - en_US.php
```

Next we will edit the code so that we can make alterations to a `HolidayItem` record.

## Editing an Existing Record

### STEPS:

1. Modify your `action.edit_holiday.php` file so that the top looks like this:

(new code is highlighted)

```
$holiday = new HolidayItem();
if( isset($params['hid']) && $params['hid'] > 1) {
    $holiday = HolidayItem::load_by_id((int)$params['hid']);
}
```

2. Change the `{form_start}` tag in your `templates/edit_holiday.tpl` file to look like this:

```
{form_start hid=$holiday->id}
```

### DISCUSSION:

That was quick and simple. Using our existing `HolidayItem` class we literally changed four lines of code. We can now edit an existing holiday and save it back to the database.

### The `action.edit_holiday.php` file

If you'll remember, our previous step was to create the links in the admin panel list view to edit the holiday item. We specified the id of the item in the 'hid' parameter.

With the three lines of code we just added, we check to see if that parameter was passed to the action,

and if it is a valid value. If it is, we load the holiday item from the database using the static `HolidayItem::load_by_id()` method that we previously created.

That is all we need to do in this file to enable editing.

## The edit\_holiday.tpl file

The one line we changed in the edit\_holiday.tpl file was to ensure that the 'hid' parameter was passed back to our action when the form is submitted, so that we can save edits to the proper holiday item. This plugin automatically handles adding the `{%actionid}` prefix to form variables.

## Deleting a Record

There is one last step to provide a complete CRUD interface in the admin panel of our module: we need to be able to delete records. Fortunately, that is simple too. But we'll get a little fancy and add in some javascript to make sure that administrators do not accidentally delete a record.

### STEPS:

1. Create a new file in your modules/Holidays directory entitled action.delete\_holiday.php and place this code within it:

```
<?php
if( !defined('CMS_VERSION') ) exit;
if( !$this->CheckPermission(Holidays::MANAGE_PERM) ) return;

if( isset($params['hid']) && $params['hid'] > 1) {
    $holiday = HolidayItem::load_by_id((int)$params['hid']);
    $holiday->delete();
    $this->SetMessage($this->Lang('holiday_deleted'));
    $this->RedirectToAdminTab();
}
```

2. Add this jQuery code to the top of your templates/defaultadmin.tpl file

```
<script type="text/javascript">
$(document).ready(function(){
    $('a.del_holiday').click(function(){
        return confirm('{%mod->Lang('confirm_delete')}');
    })
});
</script>
```

3. Find the Smarty comment `{* delete link will go here *}` in the defaultadmin.tpl file and replace it with:

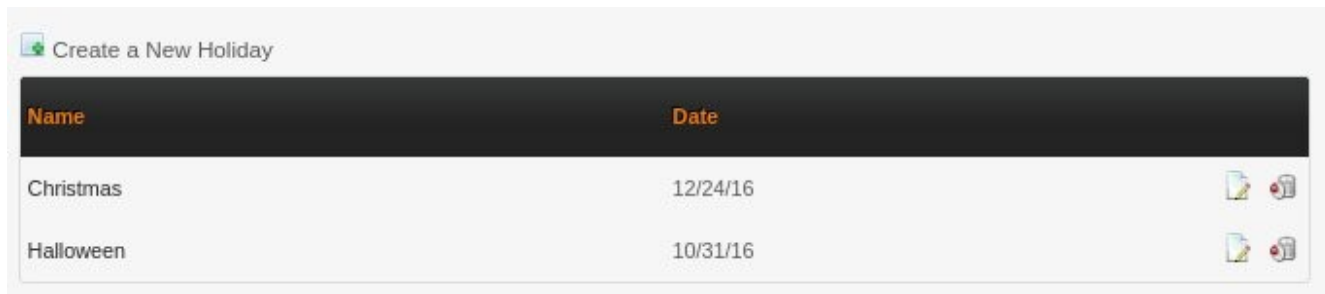
```
<a class="del_holiday" href="{cms_action_url action=delete_holiday hid=$holiday->id}"
title="{%mod->Lang('delete')}">{admin_icon icon='delete.gif'}</a>
```

4. Add the following to your lang/en\_US.php file:





```
$lang['delete'] = 'Delete this';
$lang['confirm_delete'] = 'Are you sure that you want to delete this holiday record?';
$lang['holiday_deleted'] = 'This holiday is now deleted';
```

## APPEARANCE:

After you have added a few holidays to the database, your module's admin panel should look something like this:



The screenshot shows a web interface titled "Create a New Holiday". Below the title is a table with two columns: "Name" and "Date". The table contains two rows of data: "Christmas" with date "12/24/16" and "Halloween" with date "10/31/16". To the right of each row, there are two small icons: a trash can and a document with a red 'X'.

Name	Date		
Christmas	12/24/16		
Halloween	10/31/16		

## DISCUSSION:

### The defaultadmin Template

The first thing you should notice is that we have replaced the empty column on the right side of the admin panel with a trashcan icon link. We created this link using the `{cms_action_url}` plugin, and specifying the `delete_holiday` action, and passing in the unique holiday id parameter. We then once again used the `{admin_icon}` plugin to generate the icon tag itself. We gave the link a class of `'del_holiday'` and also gave it a title attribute.

Next is our javascript. We created a simple event handler that is triggered when somebody clicks on a delete link. In that handler we confirm that they really want to delete it.

**Note:** The CMSMS admin console automatically includes the jQuery and jQuery-ui libraries, so jQuery can be used at any time. We also demonstrate in a trivial way how jQuery and Smarty code can be intermixed.

### The action.delete\_holiday.php File

When you look at this file you will see the two standard security checks at the top of this file. Followed by a check for the 'hid' parameter. If the hid parameter is set, and appears valid then the holiday is loaded from the database using the `HolidayItem::load_by_id()` method. Next it is deleted using the `HolidayItem::delete()` method that we created previously.

Lastly we set a message using `$this->SetMessage()` similar to what we did in the `action.edit_holiday.php` file, and we redirect back to the primary admin panel using `$this->RedirectToAdmin()`

This action works, but it is by no means production level code. There is absolutely no error checking

or handling in this code. In a production level module you would of course have to check to see if the holiday loaded properly, and if it deleted properly. If an error occurred you should display something to the user. You should also display a message to the user if for some reason the hid parameter was not passed to the action, or was an invalid value. This is left as an exercise to the programmer (Hint: throwing exceptions in the HolidayItem class is where I would start)

## Our First Front End Action

Woot! Our module is now functional. From within the CMSMS admin panel we can install and uninstall our module. We can also add, list, edit and delete holiday records, and we have a few classes to make interacting with holiday records simple and easy.

However, most CMSMS modules (though not all) are intended to display data on the website's front-end. A standard CMSMS module has a list view (typically called a summary view) on the frontend, and each item then creates a link to a view for a single record (typically called a detail view). This module also needs to display its data on the website's frontend.

We'll start by creating a simple summary view.

### STEPS:

1. Create a new file in your modules/Holidays directory entitled action.default.php and inside it place this code:

```
<?php
if( !defined('CMS_VERSION') ) exit;

$query = new HolidayQuery(array('published'=>1));
$holidays = $query->GetMatches();
$tpl = $smarty->CreateTemplate($this->GetTemplateResource('default.tpl'), null, null, $smarty);
$tpl->assign('holidays', $holidays);
$tpl->display();
?>
```

2. Modify the HolidayQuery::execute() method in your modules/Holidays/lib/class.HolidayQuery.php file like this:

```
public function execute()
{
    if( !is_null($this->_rs) ) return;
    $sql = 'SELECT SQL_CALC_FOUND_ROWS H.* FROM '.CMS_DB_PREFIX.'mod_holidays H';
    if( isset($this->_args['published']) ) {
        // store only draft or published items
        $tmp = $this->_args['published'];
        if( $tmp === 0 ) {
            $sql .= ' WHERE published = 0';
        } else if( $tmp === 1 ) {
            $sql .= ' WHERE published = 1';
        }
    }
    $sql .= ' ORDER BY the_date DESC';
    $db = \cms_utils::get_db();
    $this->_rs = $db->SelectLimit($sql, $this->_limit, $this->_offset);
    IF( $db->ErrorMsg() ) throw new \cmsSQLException($db->sql.' -- '.$db->ErrorMsg());
}
```

```
$this->_totalmatchingrows = $db->GetOne('SELECT FOUND_ROWS()');
}
```

3. Create a new file in your templates directory called default.tpl and inside it copy this Smarty/html code:

```
<div class="holidayWrapper">
  {foreach $holidays as $holiday}
    <div class="holiday">
      <div class="row">
        <div class="col-sm-6">
          <a href="{cms_action_url action='detail' hid=$holiday->id}">{$holiday->name}</a>
        </div>
        <div class="col-sm-6 text-right">{$holiday->the_date|date_format:'%x'}</div>
      </div>
    </div>
  {foreachelse}
    <div class="alert alert-danger">{$mod->Lang('sorry_noholidays')}</div>
  {/foreach}
</div>
```

4. Copy this code into your lang/en\_US.php file

```
$lang['sorry_noholidays'] = 'Sorry, we could not find any holidays that match the specified criteria';
```

5. Create a new content page

1. Turn off the WYSIWYG editor

On the 'options' tab click on the checkbox at the bottom with the label:  
**“Disable WYSIWYG editor on this page (regardless of template or user settings):”**

2. Paste this code into the content area

```
{cms_module module=Holidays}
```

## DISCUSSION:

### The Default Action

As mentioned previously in this action, the default action is a 'special' action that is called for front-end requests when no explicit action name is specified. It is required to be in the action.default.php file in your module's primary directory.

If you compare our action.default.php with the action.defaultadmin.php you will notice they are very similar. There are only three primary differences:

1. Because this is a front end action, we do not need to check if the currently logged in administrator has access to this action.
2. We only want to display published holidays in this view, so we pass in an array of parameters to the HolidayQuery class constructor to indicate this. The next step is to modify the HolidayQuery class to handle that parameter.

3. We are using a different view on our data. In this case we are using `default.tpl` instead of `defaultadmin.tpl`

## The `HolidayQuery::execute()` method

As you can see by the highlighted code, we have rewritten the query building code in this method to check if there is a 'published' parameter and to alter the SQL query accordingly.

## The `default.tpl` Template

This template uses some Smarty logic, and some sample styles to create a pseudo table of holidays. For convenience purposes we have used bootstrap3 styles.

We again use the `{cms_action_url}` plugin to create a link to a detail view, and pass the holiday item id as an `hid` parameter. Though the link will display in the summary view, clicking on the link will not work properly yet as we must register the `hid` parameter.

This is an important concept in all CMS Made Simple modules. There is no style-guide for front end displays. Website developers are free to use whatever toolkits they choose, or to build their own. Therefore, front-end module templates—particularly for modules that are intended for sharing—can only provide sample templates.

Though it is technically possible for a module to install CSS style sheets, images, and icons that are used with your module's front end templates, it is generally a bad idea to do this with a module you intend to release to the public. This is because you are then imposing your styling preferences on the website developer. A developer that wishes to not use these now has a lot of cleanup to do. It is best to provide basic and simple sample templates even though their default appearance may be sparse and unappealing.

## Calling the Module

The final thing we do is create a new content page, disable the WYSIWYG editor on that page, and then call our module.

We disable the WYSIWYG editor on that page because we are now using that page not for normal editor controlled content, but for module controlled content. Displaying holiday data on that page is part of our design.

The WYSIWYG editor will typically (and correctly) enforce a document structure, with `<p>` tags and other elements. Inserting module calls that generate `<div>` elements inside of those would result in invalid code.

When the page is used for displaying data, or other module generated content, it is best to disable the WYSIWYG editor and to ensure that normal admin users (those responsible for managing the site

content) cannot easily edit the page.

## A Detail View

The next thing we need to do is to add the ability to display a single record in its entirety on the front end of our website, and make the link in our summary template work.

### STEPS:

1. Add the following code to your Holidays.module.php file

```
public function InitializeFrontend() {
    $this->SetParameterType('hid', CLEAN_INT);
}

public function InitializeAdmin() {
    $this->CreateParameter('hid', null, $this->Lang('param_hid'));
}
```

2. Create a new file in your modules/Holidays directory entitled action.detail.php and paste in the following code:

```
<?php
if( !defined('CMS_VERSION') ) exit;
if( !isset($params['hid']) ) return;

$holiday = HolidayItem::load_by_id( (int) $params['hid'] );
$tpl = $smarty->CreateTemplate($this->GetTemplateResource('detail.tpl'), null, null, $smarty);
$tpl->assign('holiday', $holiday);
$tpl->display();
```

3. Create a new file in your modules/Holidays/templates directory entitled detail.tpl and paste in the following code

```
{if $holiday}
<fieldset>
  <legend>{$mod->Lang('holiday_detail')}</legend>
  <div class="row">
    <p class="col-sm-2 text-right">{$mod->Lang('name')}:</p>
    <p class="col-sm-10">{$holiday->name}</p>
  </div>
  <div class="row">
    <p class="col-sm-2 text-right">{$mod->Lang('date')}:</p>
    <p class="col-sm-10">{$holiday->the_date|date_format:'%X'}</p>
  </div>
  <div class="row">
    { $holiday->description }
  </div>
</fieldset>
{else}
  <div class="alert alert-danger">{$mod->Lang('error_notfound')}</div>
{/if}
```

4. Add the following to your lang/en\_US.php file:

```
$lang['holiday_detail'] = 'Holiday Detail';
$lang['error_notfound'] = 'The Holiday specified could not be displayed';
$lang['param_hid'] = 'Applicable only to the detail action, this parameter accepts the integer id of a holiday to display';
```



## APPEARANCE:

The appearance of the new default action depends on your styling as these frontend templates are only samples. However, if your website's page template is using bootstrap3, then the summary view might look something like this:

# Holidays - CMS2 SVN

Christmas  
Halloween

12/24/16  
10/31/16

## DISCUSSION:

- The first thing we do is add an `InitializeFrontend()` method to our module's class file.

This is a callback method that is called when your module is loaded for front-end requests. Within that method we call `$this->SetParameterType('hid', CLEAN_INT);`

If you recall the discussion earlier in this document we stated that for security purposes, all parameters that were to be passed to an action file for front-end requests must be registered.

And this is how it is done. In this example we are saying of course that the 'hid' parameter must be cleaned and that only integer values are accepted.

**Note:** A parameter needs only be registered once, for all front-end actions.

- Following this we add an `InitializeAdmin()` method to our module's class file.

This is a callback method that is called when your module is loaded for requests in the admin panel.

Here we create a parameter for use when displaying the module's help by calling `$this->CreateParameter('hid', null, $this->Lang('param_hid'));`. Here we specify the parameter name, its default value, and a string to display in the module's help.

- The next thing we do is add the code for our detail view to display a single article.

If you review the `action.edit_holiday.php` action, you will see that the two actions are very similar, with the exception that our `action.detail.php` has no need to process form parameters.

Again, this code is not production ready. There should be significant error checking in the code, and we should be double checking to ensure that the holiday is published before giving it to Smarty.

- Lastly, we create a detail template to display all of the information for the `HolidayItem` we loaded in the `action.detail.php`

This code should be very familiar to you by now as it is just a rehash of the other views that we have created. The only minor exception is that we are using a fieldset and a legend for formatting, though this is completely optional.

## Status Report

So, Where are we at? Our module is now fully functional:

- It has installation and uninstall routines
- Our module uses admin permissions
- It has a lang file to allow us to use human readable strings without embedding them directly into our PHP or HTML/Smarty code, which allows them to be translated to different languages
- It has a basic model class (HolidayItem)
- It has a basic query class (HolidayQuery)
- It has an admin panel. The admin panel will display a list of holidays, and provides a link to create new holidays. From within this list an authorized admin user can edit and delete holiday records.

Holiday records have a name, a date, a published flag, and a description

- It has a front-end summary view for displaying published holidays
- It has a front-end detail view for displaying all of the information about a single holiday

Our modules/Holidays module directory now looks like:

```
modules/Holidays/  
  \- Holidays.module.php  
  \- action.default.php  
  \- action.defaultadmin.php  
  \- action.delete_holiday.php  
  \- action.detail.php  
  \- action.edit_holiday.php  
  \- lang/  
    \- en_US.php  
  \- lib/  
    \- class.HolidayItem.php  
    \- class.HolidayQuery.php  
  \- templates/  
    \- default.tpl  
    \- defaultadmin.tpl  
    \- detail.tpl  
    \- edit_holiday.tpl
```

This tutorial could stop here. Our module is completely functional. Creating more controllers, views, and models largely involves repeating the processes we have done up until this point. However, there

are concerns about this module:

- There is no way to generate a detail view on a different content page. This may be useful, for example, if you are displaying a summary view in a sidebar and want to display the detail views using a different page template.
- As the number of holidays grows in the database this module will slow down and require more resources to display the entire list of holidays (this would probably require a few thousand holiday records though). And we don't necessarily want to display thousands of records.

Our next step will be to modify our summary view so that we can create detail views on different content pages, and also limit the output to a specified number of holidays.

## Adding A Detail Page Parameter and a Page limit

### STEPS:

1. Add the following two lines to your InitializeFrontend() method in your Holidays.module.php file:

```
$this->SetParameterType('pagelimit', CLEAN_INT);  
$this->SetParameterType('detailpage', CLEAN_STRING);
```

2. Add the following two lines to your InitializeAdmin() method in your Holidays.module.php

```
$this->CreateParameter('pagelimit', 1000, $this->Lang('param_pagelimit'));  
$this->CreateParameter('detailpage', null, $this->Lang('param_detailpage'));
```

3. Change your action.default.php file to look like this:

(changed code is highlighted)

```
<?php  
if( !defined('CMS_VERSION') ) exit;  
  
$limit = (isset($params['limit'])) ? (int) $params['limit'] : 1000;  
$limit = max(1, $limit);  
  
$detailpage = $returnid;  
if( isset($params['detailpage']) ) {  
    $hm = CmsApp::get_instance()->GetHierarchyManager();  
    $node = $hm->sureGetNodeByAlias($params['detailpage']);  
    if( is_object($node) ) $detailpage = $node->get_tag('id');  
}  
  
$query = new HolidayQuery(array('published'=>1, 'limit'=>$limit));  
$holidays = $query->GetMatches();  
$tpl = $smarty->CreateTemplate($this->GetTemplateResource('default.tpl'), null, null, $smarty);  
$tpl->assign('holidays', $holidays);  
$tpl->assign('detailpage', $detailpage);  
$tpl->display();  
?>
```

4. Add this code to your lib/class.HolidayQuery.php:

```
public function __construct($args = '')
```

```
{
    parent::__construct($args);
    if( isset($this->_args['limit']) ) $this->_limit = (int) $this->_args['limit'];
}
```

5. Change the {cms\_action\_url} tag in your templates/default.tpl file to this:

```
{cms_action_url action=detail hid=$holiday->id returnid=$detailpage}
```

6. Add these lines to your lang/en\_US.php file:

```
$lang['param_limit'] = 'Applicable only to the default action, this parameter limits the number of holidays displayed to the specified number.';
$lang['param_detailpage'] = 'Applicable only to the default action, this parameter allows specifying an alternate page id or alias on which to display the detail results.';
```

7. Edit the new content page you created above and change the call to the module to:

```
{cms_module module=Holidays limit=10 detailpage=news}
```

**Note:** The above instruction assumes that you are working with a default installation of CMSMS 2.0.1 that had the default content installed. For testing purposes, you may need to select an alias of a different existing content page that is not the default page.

## DISCUSSION:

Woah, there were a lot of steps there, but all of them were fairly minor changes. Don't worry, we'll discuss all of the important details.

- In steps one and two we just repeat the steps we did for registering the 'hid' parameter. This ensures that CMSMS knows about our limit and detailpage parameters, and that they can show up in the module's help (*the lang keys are strings added in step 6*).
- Step 3 is where the bulk of the work is done. All of the changed code is highlighted in yellow for easier reference.

You can see that we check for the 'limit' parameter, and if it is set we use its value to fill our \$limit variable. Otherwise we use 1000. Next we just ensure that a positive integer value was specified for \$limit. We then add the \$limit stuff into the array of parameters passed to the constructor when we create an instance of our [HolidayQuery](#) class.

Next is the interesting stuff: we test for the detailpage parameter, and, if it exists, we convert the value of that parameter (which could be a numeric page id or a string page alias) into a page id if possible.

We introduce a couple more CMSMS API Classes:

- CmsApp

This is perhaps one of the most important classes in the CMSMS API. It is the primary application skeleton. It contains numerous convenience and handy methods to access other

points in the API. This is a singleton object, and you receive a reference to the single CmsApp instance by calling `CmsApp::get_instance()`

There are a few convenience wrappers to this function call, including `cmsms()` which is a simple wrapper to the method above.

One of the methods of the CmsApp class is `GetHierarchyManager()`.

- The Hierarchy Manager

The name of this method is basically a misnomer. This method returns a reference to the root node of a tree of content nodes representing the page hierarchy of the entire website.

The nodes are all instances of the `cms_content_tree` class. The `cms_content_tree` class is in turn derived from the `cms_tree` class.

One of the methods of the `cms_content_tree` class is `sureGetNodeByAlias()` which will search through all of the nodes below the current point (*remember that `GetHierarchyManager()` returns a reference to the root node*) for a node with the alias you specify.

Our code uses the hierarchy manager to try to find a node given the value of the `detailpage` parameter that was passed in to our action. If a node can be found we call `$node->get_tag('id')` to retrieve the numeric page id for that alias, and assign the results to our `$detailpage` variable.

Lastly, we provide the `$detailpage` variable to Smarty for use in our template.

- In step 4 we modify our `HolidayQuery` class so that it can understand the 'limit' parameter passed into it.

If the 'limit' parameter is detected in the parameters provided to the constructor we merely set the `$this->_limit` protected member.

- In step 5 we modify the `{cms_action_url}` call to pass through the `detailpage` parameter that we created in step 3. This will ensure that the URL we are generating will request the proper CMSMS content page.

Remember from the code in the `action.default.php` that if no `detailpage` parameter is passed to the action that the current page id (`$returnid`) will be used as the `detailpage`. `$returnid` as mentioned earlier in this document is the id of the current page that we are rendering.

## General Discussion:

There is a lot to learn in this code. Though all in all the changes were relatively simple and straight forward PHP code, please review the code thoroughly. We introduced numerous new API functions

and classes, including the CmsApp class, and the Hierarchy manager for working with your page tree.

We also illustrated how \$returnid and different page id's work in CMSMS, which will be important concepts that you will need to understand in the next section.

**Note:** If search engine optimization is valuable to your site you should ensure that your usage of the detailpage parameter is consistent throughout the website. When developing a module it may be better to use a preference instead of allowing the detailpage to be specified in the call to the module. More about this when we talk about Search Engine Friendly URL's below.

## SEO Friendly URLs and Routes

Search Engine Friendly URLs (CMSMS calls them “Pretty URLs”) are usually important to modules that display data on the website front-end—particularly if they are modules that will be released for use by the general public.

In a CMSMS module there are two parts to the “Pretty URL” problem:

1. Generating the pretty URL when rendering our HTML code.
2. Routing an incoming pretty URL to the proper module action.

### STEPS:

1. Enable Pretty URLs on your website

If you have not done so already, follow the CMSMS configuration guide to enable pretty URLs on your website. Internal pretty URLs will be enough for this tutorial.

**Note:** Remember to clear the CMSMS cache every time you adjust the config.php file.

2. Add the following new method to your Holidays.module.php file

```
public function get_pretty_url($id,$action,$returnid='', $params = array(), $inline = false)
{
    if( $action != 'detail' || !isset($params['hid']) ) return;
    $holiday = HolidayItem::load_by_id((int)$params['hid']);
    if( !is_object($holiday) ) return;
    return "Holidays/$returnid/{$params['hid']}/".munge_string_to_url($holiday->name);
}
```

3. Add the following to your InitializeFrontend() method in your Holidays.module.php file

```
$this->SetParameterType('junk', CLEAN_STRING);
$this->RegisterRoute('/Holidays\/(?P<returnid>[0-9]+)\/(?P<hid>[0-9]+)\/(?P<junk>.*?)$/',
    array('action'=>'detail'));
```

4. Modify the templates/detail.tpl file like this:

(changed lines are highlighted)

```
{if $holiday}
{cms_action_url action=detail hid=$holiday->id assign='canonical'}
```

```
{$canonical=$canonical scope=global}
<fieldset>
...

```

## DISCUSSION:

1. As mentioned above, there are two parts to handling pretty URLs in CMSMS modules. The new method we have just added to the Holidays module class handles the generation of pretty URLs.

What we do here is check to see if we are creating a URL for the detail action, and we ensure that we have the “hid” parameter. If everything is good, we load the holiday object from the database. Lastly, we build a string containing the prefix “Holidays”, the page id that we want to render this detail view on, the holiday id, and a string representing the name of the holiday.

The `munge_string_to_url()` method will take a holiday name and translate the characters until they are all safe URL characters.

This method is automatically called any time `Holidays::create_url()` is called for a frontend request. `create_url()` is another virtual method in the module base class.

The `{cms_action_url}` plugin will call `Holidays::create_url()` for each URL we try to create. Therefore, every time we call `{cms_action_url}` to create a URL for the detail action, we will return a pretty URL.

If you now browse to the summary view of our Holidays module on your website’s front end, and then hover over one of the links to a detail view you should see that the generated URL looks something like:

```
<your root url>/index.php/Holidays/99/2/Christmas
```

**Note:** Your URL will look slightly different if you configured `mod_rewrite` pretty URLs, and because your `returnid`, `holiday id`, and `holiday name` may be different.

2. The modifications we made to the `InitializeFrontend()` method tells CMSMS how to handle incoming URL strings that match a certain pattern.

Firstly, we create a string parameter called “junk”. This is because the `get_pretty_url()` method will append the holiday name to the generated URL, and it must be handled.

Additionally, we already have the ability (via the `hid` parameter) to uniquely identify which article we want to display.

The `RegisterRoute()` method tells CMSMS that: For every incoming request where the URL matches this **regular expression** call the detail action for the Holidays module, and to parse out the `returnid` and `hid` parameters as well.

## GENERAL NOTES:

- The regular expression defined in the route must be able to uniquely identify your module, and all of the details to generate a view without conflicting with another module. To help with this uniqueness, module routes are usually prefixed with the module name. i.e: if they were not a route of 99/2/Christmas could refer to another module, or to a content page. The system would find the first route that matches, which may not necessarily be to your module.
- There must be some mechanism on every incoming route to determine a page id. This can either be encoded in the URL itself, or stored in some type of preference, but it must exist. Your decision on how to store the returnid depends on how your module will be used.
- Although it is possible to get rid of the hid in our routes, so that the URLs look like Holidays/<returnid>/<holiday name>, we would need to modify our code to ensure that the holiday name was unique for all holidays so that we could uniquely identify the holiday to display.
- If no action is specified as a parameter to the route, but a route is matched, then the magic module action “defaulturl” will be called.
- Routes are just additional ways to access the same data. The old 'un-pretty' url that was generated before this step will still work to generate a detail view of a holiday. That is why we modified our detail template so that we could generate a 'pretty' canonical URL.
- If you look at our get\_pretty\_url() method we are loading the HolidayItem specified by the hid. It is entirely probable that in a detail view that that Holiday item has already been loaded. This would result in the same data being loaded twice from the database.

For a production-ready module, the developer should take steps to ensure that this does not happen, and utilize caching to optimize performance.

## Registering the Module Name

You may have noticed that in our content page created above, we called the module like this:

```
{cms_module module=Holidays limit=10 detailpage=something}
```

For simplicity and brevity we want to be able to call our module like most other modules in CMS Made Simple. Like this:

```
{Holidays limit=10 detailpage=something}
```

## STEPS:

1. Add the following to the `InitializeFrontend()` callback method in your Holidays.module.php file:



```
$this->RegisterModulePlugin();
```

2. Edit your content page and change the call to our module to:

```
{Holidays limit=10 detailpage=something}
```

\* Remember to preserve the value of the detailpage parameter from the previous incarnation of this call.

## DISCUSSION:

The `RegisterModulePlugin()` method of the CMSMS module base class will create a **dynamic** shortcut so that the system recognizes `{Holidays}` in addition to `{cms_module module=Holidays}`

## Search module Integration

Since this is a content module, which means that we are managing content that will be visible on the website front end, we may want to add the ability for our module's data to be found via the CMSMS Search module. This is a purely optional step and depends on how your module will be used and distributed.

## STEPS:

1. Modify your `action.edit_holiday.php` file like this:

*(added lines are Highlighted)*

```
$holiday->save();

$search = \cms_utils::get_search_module();
if( is_object($module) ) {
    if( !$holiday->published ) {
        $search->Deletewords($this->GetName(), $holiday->id, 'holiday');
    } else {
        $search->Addwords($this->GetName(), $holiday->id, 'holiday', $holiday->name.'
        '.strip_tags($holiday->description));
    }
}

$this->SetMessage($this->Lang('holiday_saved'));
```

2. Modify your `action.delete_holiday.php` file like this:

*(added lines are highlighted)*

```
$holiday->delete();

$search = \cms_utils::get_search_module();
if( is_object($search) ) $search->Deletewords($this->GetName(), (int) $params['hid'],
'holiday');

$this->SetMessage($this->Lang('holiday_deleted'));
```

3. Add the following methods to your `Holidays.module.php` file

```
public function SearchReindex(&$search_module)
```

```

{
    $query = new HolidayQuery();
    $matches = $query->GetMatches();
    foreach( $matches as $holiday ) {
        $search_module->AddWords($this->GetName(), $holiday->id,
            'holiday', $holiday->name.' '.strip_tags($holiday->description));
    }
}

public function SearchResultWithParams($returnid, $articleid, $attr = '', $params = '')
{
    // this method returns an array with 3 elements. The module name, the article/item name,
    and the URL to display it.
    if( $attr != 'holiday' ) return;

    $holiday = HolidayItem::load_by_id((int)$articleid);
    if( !$holiday ) return; // could not find the item.

    $result = array();
    $result[0] = $this->GetFriendlyName();
    $result[1] = $holiday->name;
    $detailpage = $returnid;
    if( isset($params['detailpage']) ) {
        $hm = CmsApp::get_instance()->GetHierarchyManager();
        $node = $hm->sureGetNodeByAlias($params['detailpage']);
        if( is_object($node) ) $detailpage = $node->get_tag('id');
    }
    $result[2] = $this->create_url('cntnt01', 'detail', $detailpage,
        array('hid'=>(int)$articleid));
    return $result;
}

```

## DISCUSSION:

### The action.edit\_holiday.php file

The first thing of note in this code is the call to `\cms_utils::get_search_module()`. The `cms_utils` class contains a large number of convenience and utility functions to aid in writing code for CMSMS. We encourage you to review it. This method will return a reference to the instance of the Search module class if the module is installed and enabled. If the module is not yet loaded, it will be loaded into PHP memory.

Next, we check to ensure that we actually received a reference to the module, and, if so, proceed to do our work. Our work involves two paths:

a: The holiday we have just edited is not—or is no longer—published. In this case we need to ensure that this holiday is not indexed by the Search module.

To do this, we call the search module's `DeleteWords()` method. We pass in the module name, the holiday's unique id, and a keyword 'holidays' so that the search module can uniquely identify the item we are removing.

b: The holiday we have just edited is published. In this case we need to ensure that the words in this holiday object are indexed by the Search module.

To add words to the search module we call the module's `AddWords()` method. We again pass in the module name, the unique holiday id, and the 'holidays' keyword so that the search module can

uniquely identify this item. We also pass this method a single string of the words that should be indexed (without html tags).

## The action.delete\_holiday.php file

The code in this action is very similar to the related code in the action.edit\_holiday.php file, however we only need to be concerned about deleting words from the Search module when a holiday is deleted.

## The SearchReindex method

The search module has the ability to re-index all content. This happens when the module is upgraded, or the website administrator explicitly chooses that option from the Search module's admin panel.

The search module calls the SearchReindex method for each and every module that has that method so that each module can provide its contents to refresh the search index.

In our implementation of this method, we merely create a `HolidayQuery` object and iterate through its matches. You should be able to see from the definition of this query object that we will only (by default) be returning published holidays.

Once we have the results, we iterate through them and call the Search module's 'AddWords' method just as we did in the action.edit\_holiday.php file. The reference to the Search module is provided to us in this callback.

**Note:** In production level code, if there were a great deal of items in our holidays database we may have to extend this code to be a bit artistic and memory efficient. Additionally, our `HolidayQuery` class by default limits us to 1000 items.

## The SearchResultWithParams() method

The Search module, after a search has been performed, will call this method of your module when it has determined that one of the items from your module has matched the search criteria.

The Search module is asking you for the relevant information it needs to display this item in its output. It expects an array of 3 items to be returned. First, the module name, followed by the text to display to the user, and lastly the URL to create a link to.

The Search module is capable of passing arguments to your module such as the article id, or detailpage, templates or other parameters. Those parameters must already be registered by your module.

In our code we first do a few minor checks to ensure that we actually have to do work. We check that the Search module wants to display a 'holiday' (see the keyword we provided in the calls to `$search->AddWords()` and `$search->DeleteWords()`). Next we ensure that we can actually read the `HolidayItem` specified in the `$articleid` parameter. Then we get to work.

First, we create an empty array, and immediately add our module name to the first element. We use the holiday name for our second element, though in production environments we may want to display a portion of the description. And, lastly, we create a detail URL.

We test if a `detailpage` parameter was specified in the incoming parameters, and, if it was, we resolve that into a page id. This is the same code we use in our `action.default.php`.

Next, we call `$this->create_url()` and pass it a number of parameters. This is the first time we have used the `create_url()` module method directly. Until now we have used `{cms_action_url}` which is a wrapper to this method.

To the `create_url()` method we pass 'cntnt01' as a module action id. This is a special value for CMSMS front end requests, and it indicates that the output of the action should replace the contents of the default `{content}` block. Secondly, we pass in the name of the action we are creating a link to—in this case it is 'detail'—and the page that we wish to render the output on, in this case that page is identified by the `$detailpage` variable. Lastly, we pass in an array of parameters for that action. In this case, we only need the 'hid' parameter for our detail action.

## General Discussion

The above procedure demonstrates one way of interacting with other modules. There are a few ways in CMS Made Simple for module to interact with others.

Note that we have not created a tight dependency on the Search module. We are just adding data to it if the module is installed and enabled.

A tight dependency would be when your module absolutely must use the APIs of another module in order to function properly. In this case you would need to modify your `GetDependencies()` method to ensure that the dependent modules were loaded before your module was loaded.

**Note:** Each time you adjust the dependencies in your `GetDependencies()` method you must adjust the value of your module version in at least a minor way.

**Note:** The methods that we are using from the Search module are clearly documented and public (not internal) methods.

## Lazy Loading

Lazy loading is the concept of not loading your module into PHP memory until such time as it is needed. This can be useful when there are many modules installed and when your module may not be needed on each request.

Modules can be lazy loaded for front end or admin requests, however there are some caveats to using lazy loading, particularly on the front end. The table below indicates when you cannot use Lazy

Loading.

	<b>Can Lazy Load Front end Requests</b>	<b>Can Lazy Load admin requests</b>
Registers module name dynamically * See RegisterModulePlugin()	NO	YES
Registers routes dynamically * See cms_route_manager	NO	YES
Registers Additional Smarty plugins	NO	NO
Shares API's * Other modules may have to explicitly request your module instance, or list it as a dependency	YES*	YES*
Registers Content Types	NO	NO
Registers module block types	YES	YES

**Note:** Our current sample module cannot be lazy loaded for front end requests because it calls RegisterModulePlugin() from the InitializeFrontend() method. This callback is called after the module has already been loaded.

## STEPS:

1. Add this function to your Holidays.module.php file:

```
public function LazyLoadAdmin() { return TRUE; }
```

2. Modify your GetVersion() method as follows:

```
public function GetVersion() { return '0.1.1'; }
```

3. Visit “Site Admin >> Module Manager” in your CMSMS admin console, and click 'Upgrade' on the Holidays row.

## DISCUSSION:

Now our module will not be automatically loaded for admin requests with certain exceptions:

- When the admin navigation cache must be loaded for an admin user
- In the Module Manager
- When a dependent module requires it

- When an action from our module is requested

## Asynchronous Content

Many web sites, or web applications, these days implement some of their functionality by loading content asynchronously, or via AJAX. This involves using javascript to asynchronously load some data, or a small portion of content, from your website via a request that is sent AFTER the primary page is requested.

CMSMS makes making asynchronous requests to your module actions fairly simple. All that is needed is access to the proper URLs from within your javascript, and thankfully the `{cms_action_url}` plugin works great for this.

**Note:** For the following steps we will be assuming that jQuery is enabled in your front end template. You can do that by adding `{cms_jquery}` to the head portion of your page template, or by manually including the appropriate `<link>` tags.

### STEPS:

1. Add the following to your lang/en\_US.php file:

```
$lang['preview'] = 'Preview';
```

2. Add the following to your InitializeFrontend() method in your Holidays.module.php file

```
$this->SetParameterType('detailtemplate', CLEAN_STRING);
```

3. Modify your get\_pretty\_url() method in your Holidays.module.php file like this

```
public function get_pretty_url($id,$action,$returnid='', $params = array(), $inline = false)
{
    if( $action != 'detail' || !isset($params['hid']) ) return;
    if( isset($params['detailtemplate']) ) return; // can't make a pretty URL
    $holiday = HolidayItem::load_by_id((int)$params['hid']);
    if( !is_object($holiday) ) return;
    return "Holidays/$returnid/{ $params['hid'] }/".munge_string_to_url($holiday->name);
}
```

4. Modify your action.detail.php file to look like this:

```
<?php
if( !defined('CMS_VERSION') ) exit;
if( !isset($params['hid']) ) return;

$detailtemplate = (isset($params['detailtemplate'])) ? trim($params['detailtemplate']) :
'detail.tpl';
$holiday = HolidayItem::load_by_id( (int) $params['hid'] );
$tpl = $smarty->CreateTemplate($this->GetTemplateResource($detailtemplate), null, null, $smarty);
$tpl->assign('holiday', $holiday);
$tpl->display();
?>
```

5. Add a new file entitled 'ajax\_detail.tpl' in your modules/Holidays/templates directory and copy in this smarty code:

```

{if $holiday}
<div>
  <div class="row">{$holiday->name} @ {$holiday->the_date|date_format:'%x'}</div>
  <div class="row">
    {$holiday->description|strip_tags|summarize}
  </div>
</div>
{/if}

```

6. change your default.tpl file to look like this:

```

<script type="text/javascript">
$(document).ready(function(){
  $('a.preview').click(function(e){
    e.preventDefault();
    var url = $(this).attr('href')+'&showtemplate=false';
    $('#preview_area').load(url).show(0).delay(5000).hide(0);
  })
})
</script>

<div id="preview_area" class="well" style="display: none;"></div>

<div class="holidayWrapper">
  {foreach $holidays as $holiday}
  <div class="holiday">
    <div class="row">
      <div class="col-sm-6">
        <a href="{cms_action_url action=detail hid=$holiday->id
returnid=$detailpage}">{$holiday->name}</a>
        &dash; <a class="preview" href="{cms_action_url action=detail hid=$holiday->id
returnid=$detailpage forjs=1 detailtemplate='ajax_detail.tpl'}">{$mod->Lang('preview')}</a>
      </div>
      <div class="col-sm-6 text-right">{$holiday->the_date|date_format:'%x'}</div>
    </div>
  </div>
  {foreachelse}
  <div class="alert alert-danger">{$mod->Lang('sorry_noholidays')}</div>
  {/foreach}
</div>

```

## DISCUSSION:

### The Holidays.module.php file

- As should be evident by now, we register a new parameter called detailtemplate, which is of type string. Please add the help for this parameter as an exercise.
- Next, we modify our `get_pretty_url()` method so that if we are requesting a detail action, but we have specified a detailtemplate, we do not attempt to create a pretty URL.

This is because there really isn't any sense in “uglifying” a search engine friendly URL with a parameter that is not relevant to the URL.

**Note:** See the `{cms_module_hint}` plugin to allow specifying non-default parameters for module actions from within a parent page template.

### The action.detail.php file

Here we simply modify our action to accept the detailtemplate parameter if it is provided to our action.

If it is not, then we use the existing detail.tpl template.

## The ajax\_detail.tpl file

This template file is a detail template. It is intended to display the holiday information in a brief manner. Specifically it removes all of the HTML tags from the description.

## The default.tpl file

This is where the magic happens. First, you should notice the line beginning with `&dash`. Here we create another link. We give that link a class of preview, and for its href we again call `{cms_action_url}`, but this time passing it the `detailtemplate` parameter, and another parameter called 'forjs' that will trigger some URL processing for us.

At the top of the template is some jQuery code inside a `<script>` tag. We create a function to handle click events on links with the class preview.

Inside that handler we retrieve the href attribute for the link that we clicked on, and we add `&showtemplate=false` to it. `showtemplate=false` is a magic request variable for CMSMS. It indicates that only the default content block of the requested page will be processed, and none of the surrounding template. This means that we want only the results of our action to be returned, not the surrounding HTML from our page template. This parameter has a similar function for admin requests.

Once we have the URL, we simply load the HTML into the new div we created with the id of "preview\_area". Then we show the preview area as it is initially hidden, wait for 5 seconds and hide it again.

We now have a summary template that, if you click on a preview link will show some brief information about the holiday for a short period of time. The difficulty in this example was in modifying the detail view and the Holidays module class file to add yet another parameter. The javascript portion was relatively simple and straight forward.

## Final Status Report

That is the end—our module is functional. In approximately 490 lines of code we're done. It is time we consider all of the functionality we have already:

- Our module creates its database table and a permission on installation, and removes them on uninstall.
- It has an admin interface with basic CRUD capabilities
- It uses CMSMS permissions in the admin interface



- It has front end summary and detail actions, and two templates for the detail view.
- It supports a pagelimit parameter in the summary view.
- It knows how to work with a detailpage parameter so detail views can be on different pages
- It handles a detailtemplate parameter so that you can output different detail views of the same data
- It supports pretty URLs for requesting detail views
- It integrates with the Search module
- and it supports lazy loading on admin interface requests.
- We have demonstrated the model/view/controller paradigm within CMSMS
- We know how to create actions, and templates for actions
- We know how to modify our module class so that CMSMS knows about our module, its functionality, and its dependencies, among other things.
- We have demonstrated code re-use throughout the module with our HolidayItem and HolidayQuery classes.

Our file structure looks like this:

```

modules/Holidays/
  \- action.defaultadmin.php
  \- action.default.php
  \- action.delete_holiday.php
  \- action.detail.php
  \- action.edit_holiday.php
  \- Holidays.module.php
  \- lang/
    \- en_US.php
  \- lib/
    \- class.HolidayItem.php
    \- class.HolidayQuery.php
  \- method.install.php
  \- method.uninstall.php
  \- templates/
    \- ajax_detail.tpl
    \- default.tpl
    \- defaultadmin.tpl
    \- detail.tpl
    \- edit_holiday.tpl

```

## Taking this Module Further

Though this module has a lot of functionality, there is always functionality that could be added. A module never dies, it just grows. There is work to do—particularly if this module is going to be

redistributed or shared for production use. Some of the items worth adding are:

- **Error Handling**

For the sake of expediency this tutorial has not included much error handling. However this is a requirement in a production level module.

The author would recommend starting by modifying the `HolidayItem` and `HolidayQuery` classes to throw exceptions instead of returning true or false from the various methods.

Then each of the action files, and any of the methods using those classes can be modified with appropriate try/catch blocks, displaying of errors, and redirection. This would allow handling all sorts of errors with one set of code while keeping your source code readable.

- **Filtering and Pagination**

Particularly in the admin interface it would be useful to be able to paginate the results after there are more than 20 or 30 items in the database. The ability via a form to apply a filter to the results would aid in being able to find a specific item.

Similarly, filtering and pagination on the front end would be useful in a production-quality module.

- **Categories**

In a production-quality module, users typically require some sort of mechanism to categorize their items, and later filter them.

- **Integration with the Design Manager**

In a module intended to be shared on the Forge, users typically enjoy the capability of managing multiple templates for each view of a module from within the CMSMS Design Manager.

Integration with the Design Manager is the subject of another (though probably shorter) tutorial.

## **Working with Other Modules**

At this point, after we had introduced working with the Search module, We thought it prudent to briefly discuss how to interact with other modules, and the proper etiquette of doing so. Forgive us, as we briefly get a little bit preachy.

### **Creating a Dependency**

You can easily add another module to your module's dependency list by modifying the data returned from your `GetDependencies()` method. This will ensure that CMSMS first loads these modules

before loading yours. Your module will then be able to use the API functions of your dependent modules automatically.

You can also get a reference to a module instance by calling `cms_utils::get_module()` or a few other methods, and then checking to ensure the validity of the result of that function call. i.e.:

```
$captcha = \cms_utils::get_module('Captcha');  
if( $captcha ) {  
    $captcha->SomeMethod();  
}
```

You should never attempt to call a module instance, or use an API function provided by a module without first ensuring that the module is loaded via one of the two above methods.

## Finding Other module's API methods:

Sometimes you are fortunate and the module author has posted online somewhere a complete and well documented API interface to their module. Often though this is not the case—you will need to discover how to interact with another module by reading and understanding its source code.

## Etiquette

When writing your own module (or UDT or plugin) and interacting with another person's code there are a few things to remember:

### PLAY NICELY KIDS!

- Do not write directly to the other packages data, databases, tables or files

This is very important. A developer cannot be held responsible for bugs or issues caused by another third party application writing to his data without using his approved, public, documented APIs.

It happens all too often where package B is writing directly to package A's database, or deleting cache files etc. that are used for and interfere with the proper operation of module A. When a user encounters an issue they assume the problem is with module A. It all too often is not.

This causes the developer(s) of package A to waste time and get frustrated. It also causes bad feelings about the other developer of package B.

- Do not use the other packages undocumented, or 'internal' methods

It is also very important that you do not use API methods of another piece of source code that were not meant for sharing as this puts both your module and the dependency at risk. Internal or undocumented methods are sometimes intended for internal use, or may be temporary. Your dependent module may break if down the road this 'internal' method goes away or its

functionality changes. Although this is not the fault of the other third party developer, end users don't see it that way, and it leads to the same frustrations and bad feelings as described above.

If you need to interact with a module, and it has very limited documentation, or none at all, you have just a few choices:

- Ask the developer of the other module to nicely, and expediently write some documentation (sometimes you can help to adjust their priorities with cash)
- Ask to become involved with the other project

Many authors will jump at the offer of assistance. Some will not.

- Fork the project

All CMSMS modules are licensed under some sort of GPL compatible license. Therefore, it is possible to fork the module to a new name—and implement your changes. even if it is just documentation—yourself.

- Change your plans
- Do interact with the other package's developers

Most developers are eager to help you create quality code, and to improve their products. It pays, if possible to interact with other developers.

## Translations

As mentioned at the beginning of this document modules CMS Made Simple can be quite easily translated to other languages. This, and the ability to re-use and easily fix typos in human readable strings is the reason that we encourage people to use lang files for as much as possible.

CMSMS has the ability to set locales and languages for front end requests based on the users browser, what languages are installed on the server and CMSMS and on site settings. Additionally, for admin requests the language can be adjusted based on what languages are installed on the server and in the CMSMS install, and user preferences. For both front end and admin requests there is a concept of the 'current language' (the language for the request).

CMSMS will automatically read your en\_US.php file the first time that the Lang() method is called. Also, depending upon the 'current language' and if it exists, it will read a lang file from modules/<yourmodule>/lang/ext/xx\_XX.php where xx\_XX.php corresponds to an accepted language code.

For example: modules/Holidays/lang/ext/fr\_FR.php is a valid language file, and would be read if the users current language was set to fr\_FR.

## The Translation Center:

Most developers do not know more than one or two spoken and written languages, so they need help with translating their module into different languages. That is where the “Translation Center” comes in to play.

The Translation Center is a simple online application that allows authorized people to translate the human readable lang files from various modules into different languages. We encourage you to register your module for the translation center.

## Setting it Up:

In order to allow for your module to be translated into different languages via the translation center a few things need to occur:

1. Your module must be registered on the CMS Made Simple Forge ([dev.cmsmadesimple.org](http://dev.cmsmadesimple.org))

Though you do not need to keep your source code there, the forge provides a convenient mechanism for sharing releases, and managing bugs and feature requests. It is also integrated with the Translation Center.

2. You need to contact our translation team lead and let them know about your project

Our translation team lead currently needs to know a few things about the project:

- The project name
- The project's unix name on the forge
- The complete, public, web accessible URL to your up-to-date lang/en\_US.php file

3. You need to be able to pull translations from the translation center

Translations will change over time as people get to them, usually in their evenings. Before you issue a release of your module you will usually want to grab the latest transactions.

If using Subversion (SVN) as your source code management mechanism this is done by editing an external property. From the unix command line you would type something like this:

```
cd modules/Holidays/  
svn propedit svn:externals lang
```

That would instantiate a text editor into which you would type:

```
ext http://svn.cmsmadesimple.org/svn/translatecenter/modules/ModuleName/lang/ext
```

Lastly you would commit your change.

This would ensure that every time you 'updated' your current working directory from the repository that the latest translations would be pulled.

Instructions for using GIT are slightly different.

## What's Changing

CMSMS is constantly evolving and changing. Here is a brief list of some of the slated changes in the near future, particularly those changes that would effect this tutorial:

### Language Files:

It's ironic that just after we finished discussing language files and translations we are now stating that they are slated to change... However, it's a good change.

Our plans are to overcome certain difficulties and issues with lang files by changing the lang file format to .ini files. In the end, although the format of the en\_US.php file will change, the functionality will not dramatically change.

### Config files:

It is our intention to change the CMSMS config.php file to be in .ini format for security purposes. Though this does not directly affect the content of this tutorial.

### Name spaces

As of the writing of this document, CMSMS has limited support for name spaces in modules, and does not support in any way the actual module file being within a name space. This is slated for change to minimize the possibility that two module authors would write classes that collide with each other.

## Scratched the Surface

This tutorial has barely managed to introduce the CMSMS API. The APIs provided with a CMSMS install are extensive. They are—for the most part—well documented, stable and mature. There are a great deal of classes and functions for:

- Debugging
- Creating, Retrieving and managing site wide preferences
- Creating, Retrieving and managing admin user based preferences
- Requesting data from a remote URL
- Setting and retrieving cookies
- Working with CMSMS Admin users and groups

- Working with designs, templates, and css style sheets,
- Working with content pages and the content hierarchy
- more...

## References

Below you will find links to some important documentation references that are useful to understand how to use CMSMS and how to work with the various APIs when writing your modules.

- The CMS Made Simple primary documentation site:  
<http://docs.cmsmadesimple.org>
- CMS Made Simple API Documentation site:  
[http://www.cmsmadesimple.org/APIDOC2\\_0/](http://www.cmsmadesimple.org/APIDOC2_0/)
- The ADODB-Lite Data Dictionary Documentation:  
<http://adodblite.sourceforge.net/datadictionary.php>
- ADODB-Lite function Documentation:  
<http://adodblite.sourceforge.net/functions.php>
- CMSMS Skeleton module  
<http://dev.cmsmadesimple.org/projects/skeleton>

## Conclusion

Though we have learned a lot throughout this tutorial, the module that we have created is, in itself, only in its infancy. We have a basic CRUD interface for the admin, and a basic summary and detail view. Our module installs and uninstalls properly, handles pretty URLs and integrates with Search.

We have demonstrated some of the functions of the CMSMS module API, and introduced other API functions.

We have demonstrated using the model/view/controller paradigm within CMSMS, and a sample method of creating our models. We have also reused these model classes throughout our code.

At this point you should be able to add new models, views, and controllers (actions) to your module at will, and to create links between actions. You should have a good understanding of how to communicate and interact with other modules, and how pretty URLs work within CMSMS.

You should continue reading other sample modules for CMSMS, including the Skeleton module, and

learn about how to style things for the CMSMS admin theme.

Once you are comfortable writing basic modules for CMSMS, there are a few advanced topics worth learning about. You will find sample modules that you can review to understand how these work:

- Pretty URL Slugs and Static Routes
- Interacting with Other modules and the Core
  - Responding to and Sending Events
  - Working with Capabilities
- PseudoCron
- Working with the Design Manager
- Creating WYSIWYG and Syntax Highlighter modules
- Creating a brand new content type
- Creating content page block types
- and more...

Thank you for your time, and... Have fun with CMSMS!