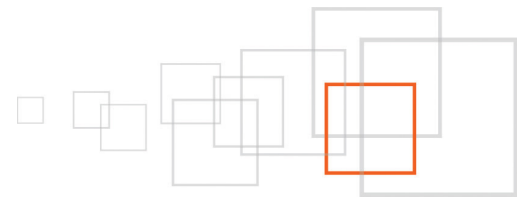


Creating Custom Admin Modules & Views

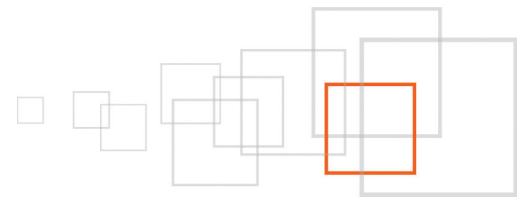
By David Linnard

<http://www.onequarterenglish.co.uk>



Index

1	Goal description.....	3
2	Introduction.....	3
3	Pre-requisites and target population.....	3
4	Setting up the extension.....	4
4.1	Activating modules within the extension.....	4
4.2	Activating template files within our extension.....	4
4.3	Activating the extension.....	5
5	Updating the User class.....	5
6	Creating our module.....	6
6.1	Creating the module.php file.....	7
6.2	Creating the view PHP file.....	8
6.3	Creating the view Template files.....	10
6.4	Providing Navigation.....	13
7	Passing parameters to Views.....	15
7.1	Accounting for Parameters.....	15
7.2	Allowing user input in our Template.....	17
7.3	Updating our PHP to allow user interaction.....	18
7.4	Providing user feedback.....	22
8	Creating a second view – handling POST data.....	24
8.1	Updating existing files.....	24
8.2	Creating our new view- Template file.....	26
8.3	Creating our new view – PHP file.....	27
8.4	The Finished View.....	30
9	Modules and Permissions.....	31
9.1	Updating the Role.....	31
10	Next Steps.....	32
10.1	Creating additional Views and Modules.....	32
10.2	Creating Modules for your main site access.....	32
11	Conclusion.....	33
12	Resources.....	33
13	About the author : David Linnard.....	33
14	License choice.....	33



1 Goal description

In this tutorial we will explore how to add custom modules and views to the CMS. By doing this you can extend the functionality of the back office to use your custom PHP to carry out various tasks. Once this tutorial is complete you should be comfortable with creating modules and views for any site access with the CMS and be able to appreciate situations when creating custom modules and views would be suitable.

2 Introduction

Adding your own tabs and functionality to the CMS is essential for some projects. This enables you to isolate functionality for authorised users and embed this functionality into the admin interface they are already familiar with. This tutorial will cover the basics for creating CMS based views and modules which you can use as a basis for your own admin based functionality.

We will setup several an extension for a simple mailing list. We will create the functionality to list the relevant users, remove them from the list and sending a simple, text based email to them.

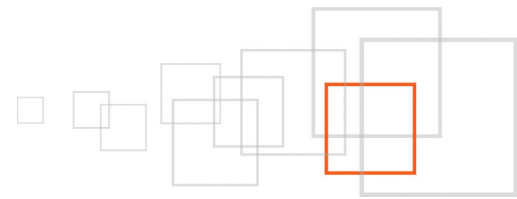
Please note there are already several extensions you can embed for setting up newsletters in eZ Publish so I'd recommend looking into these rather than building your own from scratch straight away.

There is already a tutorial available on the site if you want to explore creating modules for the front of your site. I would also recommend reading this to get a thorough understanding of how modules and views are created ([click here to view](#)).

3 Pre-requisites and target population

This tutorial is for users who are comfortable with the eZ Publish template language and PHP and wish to extend the CMS for their own projects. Users should also be familiar with the structure of ini files used by eZ Publish and the layout of the admin site access of a basic eZ Publish site.

This tutorial has been developed using eZ Publish 4.4 but some code examples require slightly different code to work on all versions of 4.x. Where this is the case alternative code is supplied and its use noted.



4 Setting up the extension

We will create an extension to house our code in this tutorial so the code is isolated and can be easily used on multiple projects. Let's create an extension called mynewsletter. If you call your extension something else be sure to rename all code samples which use the extension name.

This extension will store the custom views we create and also the custom module we build.

4.1 Activating modules within the extension

We will firstly ensure eZ Publish looks within our extension for the custom modules we create. At the same time, we also need to tell it the name of the modules which exist in our extension This offers us the potential to create multiple modules within the same extension however in our case we will just need to create one, which I have named as "newsletter". The module name we provide here will be used in the URL whenever we are dealing with the module.

Create a file named extension/mynewsletter/settings/module.ini.append.php and add the following content:

```
<?php /*

[ModuleSettings]
ExtensionRepositories[]=mynewsletter
ModuleList[]=newsletter

*/ ?>
```

There are a couple of lines of code here and each are equally important.

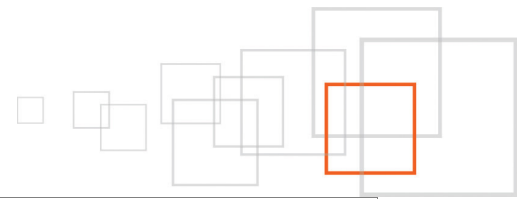
- The `ExtensionRepositories[]` setting provides an array of all extensions which contain modules. If you want to see which other extensions contain modules go to the Setup tab within the admin siteaccess and click on the "Ini settings" link within the left hand column. Once the two dropdowns appear, select `module.ini.append.php` and your siteaccess and you should see a complete list (this can be useful for the other settings files if you are unsure why a certain setting is being used over a different one).
- The `ModuleList[]` ensures we can find the specific modules within this extension (this will be explained further when we create our first module). Multiple Modules can be created within a single extension by duplicating this line for each module you create.

4.2 Activating template files within our extension

To ensure the templates we create in our extension (as well as any specific CSS or images you are storing in the extension) are retrieved we need to ensure the design folder is included when templates are searched for. We do this by creating another settings file in the settings directory called `design.ini.append.php` (extension/mynewsletter/settings/design.ini.append.php). Create the file and add the following code:

```
<?php /*

[ExtensionSettings]
DesignExtensions[]=mynewsletter
```



*/ ?>

4.3 Activating the extension

The final part of setting up the extension is ensuring the extension itself is active. Navigate to `/settings/override` from the root of your eZ Publish site.

In this directory there should be a file called `site.ini.append.php`. If no file exists, create it now and then open it.

Enter the following settings underneath the settings which already exist, if any :

```
<?php /* #?ini charset="iso-8859-1"?  
...  
[ExtensionSettings]  
ActiveExtensions[]=mynewsletter  
  
*/ ?>
```

Note: If you already have an `ExtensionSettings` element in the file, simply add the `ActiveExtensions` line rather than duplicating the whole block.

Now that we have setup the extension we are ready to create our module.

5 Updating the User class

We can now prepare the CMS for our module. Since we are building a simple mailing list module, we need to be able to differentiate between users who wish to be part of the mailing and those who do not. To do this we need to add an additional attribute to the User content class. Updating a content class is a straightforward process. All classes are split into content groups and these in turn can be restricted according to location in the CMS and by user by using the comprehensive roles and policies available in eZ Publish (these can also be limited in other ways). Go to the settings tab and click on "Classes" in the left hand navigation. You will now be presented by two sections of grouped content. The top section contains the content groups all of the content classes are grouped into. Below that we have the classes which have most recently been updated.

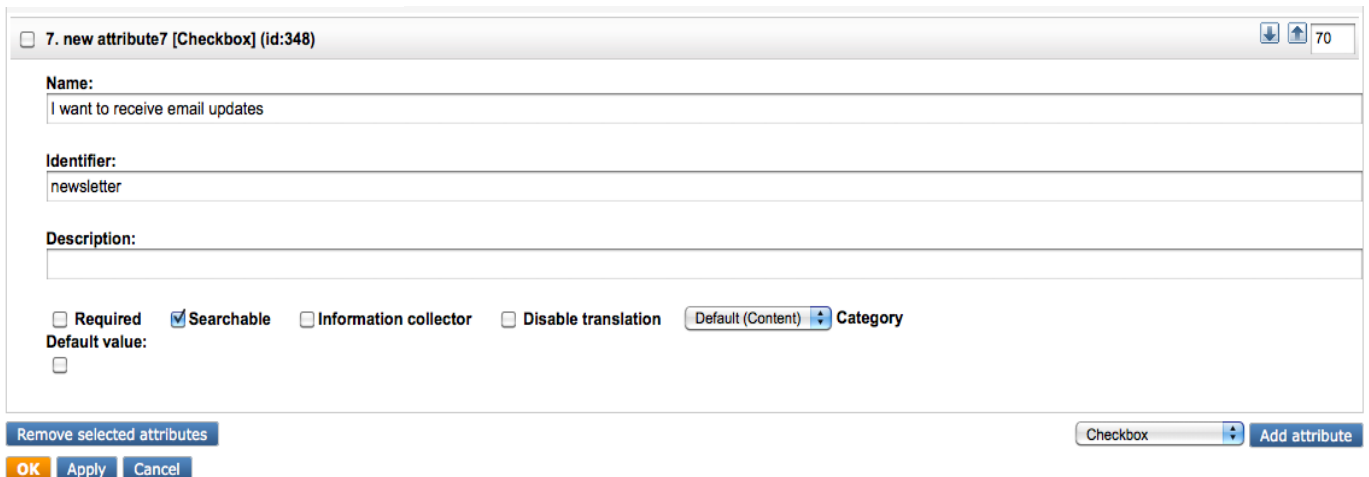
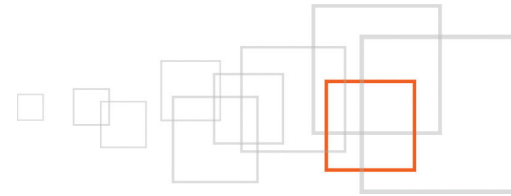
If you do not see the User class within the bottom section of the page, click on the "Users" Content Group in the top section. Once you see the User class listed, click on the pencil icon to the right of the name to edit the class.

To add an attribute, go to the bottom of the page and you will see an option to "Add attribute". Please note the location of the link varies according to which version of eZ Publish you are using. In 4.3 onwards, it is on the right but in previous versions it is on the left.

We want to add a checkbox with the following settings:

Name: I want to receive email updates
Identifier: newsletter

The final result should look as follows:



The screenshot shows a dialog box titled "7. new attribute7 [Checkbox] (id:348)". It contains the following fields and options:

- Name:** I want to receive email updates
- Identifier:** newsletter
- Description:** (empty)
- Options:** Required, Searchable, Information collector, Disable translation
- Default value:**
- Category:** Default (Content)

At the bottom, there are buttons for "Remove selected attributes", "OK", "Apply", "Cancel", "Checkbox", and "Add attribute".

Next, click OK and our class is ready.

I would recommend at this point creating a couple of test users and make sure the checkbox is checked for each of them, just to make sure we have content in place for the remainder of this tutorial.

6 Creating our module

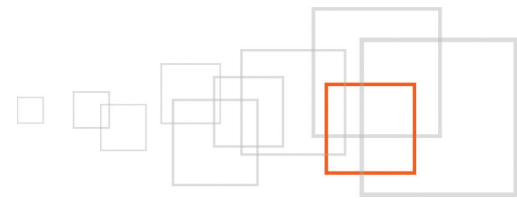
We can now finally create our module. We set the name of our module in our `module.ini.append.php` file and so eZ Publish will know to look for a module called "newsletter". Within this module we can create many views which will be linked to a central module file so that eZ Publish can find all of them. Each view is comprised of two main files:

- A PHP file – this is used for carrying out any custom PHP logic required by the view. For instance, you can carry out database queries or process a specific user action.
- A TPL file – this is specified within the PHP file and is used to render what the user will see (template).

This approach provides a high level of flexibility as different TPL files can be used according to what happens within your custom PHP logic. The PHP file can also pass custom variables into views for use in the template. We will use this functionality later in the tutorial to pass feedback based on user actions.

To create our module, firstly create a folder named "modules" in the root of your extension. Within this create a folder with the name of the module. In our case the module is called "newsletter". All of our custom PHP code will be stored in this directory but first we need to create the central module file, called `module.php`, file so eZ Publish can find all of the views.

Please note that in this tutorial you will need to be logged in as a user with Admin rights. We will cover how to access the module for other users at the end of the tutorial.



6.1 Creating the module.php file

The module.php file is used to store details about the module. In particular it stores a list of views within the module and the permissions required within the module. The file itself is quite straightforward but it is important to include all necessary details when you create the file.

Within the “newsletter” folder you have just created (/extension/mynewsletter/modules/newsletter/), create a file called module.php and enter the following code:

```
<?php
$module = array( 'name' => 'newsletter' ); //the name of our module, this ties in with
the name specified in module.ini.append.php and the name of the parent folder

$ViewList = array(); //add as many views as you want here:
$ViewList['userlist'] = array( 'script' => 'userlist.php',
                               'functions' => array( 'read' ) );

//the script used to setup the template plus the user permissions required for it (also
see below). The default navigation part is optional but can be used to default what
template will be loaded for the left navigation (this can also be defined within the
view's PHP file)

//setting user permissions required by our module:
$FunctionList = array();
$FunctionList['read'] = array();
?>
```

The code is split into three parts:

6.1.1 Module Name

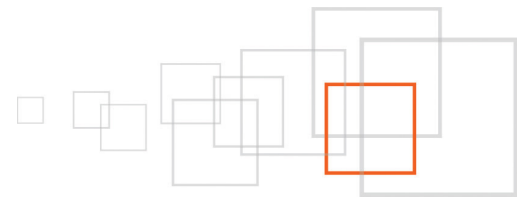
Firstly, we define the name of the module we are in. This will tie in with the folder we just created and also the module.ini.append.php file we created at the start of this tutorial. This will also be the name of the module used in the URL. For example, for this module all of the views in the module will be located as follows: <http://www.mysite.com/admin/newsletter/xxx> where xxx represents a view name.

6.1.2 Defining the Views within the Module

After this comes the definition of the PHP files associated with the views in the module. You can include as many views as you want here. We only declare the PHP file associated with the view as the template used to render the view may vary according to what happens in the PHP. Because of this the main template to render the view is specified in the PHP file associated with the view rather than in the module.php file.

In our case we are initially making only a single view called “userlist”. As with the name of the module, this will be used in the URL, therefore the path to our first view will be:

<http://www.yoursite.com/admin/newsletter/userlist>



6.1.3 Defining the Security Policies of the module

After the views we define the security policies associated with the module. If you want to read more about this then I would recommend reading the article Adding custom security policy limitations to your modules, we will only look it at in terms of basic usage in this tutorial.

6.2 Creating the view PHP file

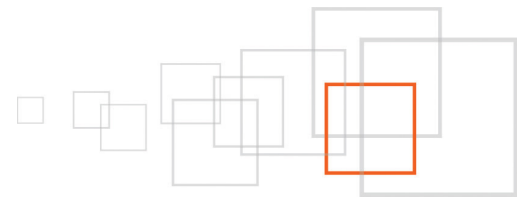
We will now create the custom PHP file for the View. Initially, our PHP will not require any custom code above what is needed to setup the template.

```
<?php
/**
 * File containing the eZ Publish view implementation.
 *
 * @copyright Your Name here
 * @license licence details
 * @version 1.0.0
 * @package mynewsletter
 */
//setting up the eZ template object:
$tpl = eZTemplate::factory(); //this line of code is for ez publish 4.3, replace it with
the following lines for versions prior to that
//version <4.3 of eZ Publish should use these lines of code instead:
//include_once( 'kernel/common/template.php' );
//$tpl = templateInit();

//carry out internal processing here, none required in this case.

// setting up what to render to the user:
$result = array();
$result['content'] = $tpl->fetch( 'design:newsletter/userlist.tpl' ); //main tpl file to
display the output
$result['left_menu'] = "design:newsletter/leftmenu.tpl"; //uncomment this line if you
want to use a custom left navigation for this view.

$result['path'] = array( array( 'url' => 'newsletter/users',
                                'text' => 'User List' ) ); //what to show in the Title
bar for this URL
?>
```

The code can again be split into three parts:

6.2.1 Setting up Variables

In this case we are not doing any custom PHP and so the only variable we need is an instance of eZTemplate so that we can retrieve the template files.

```
//setting up the eZ template object:
$tpl = eZTemplate::factory();//this line of code is for ez publish 4.3, replace it with
the following lines for versions prior to that
//version <4.3 of eZ Publish should use these lines of code instead:
//include_once( 'kernel/common/template.php' );
//$tpl = templateInit();
```

Please note that if your eZ Publish build is before 4.3, you will need to comment out the second line of code above and uncomment the other lines.

6.2.2 Carrying out custom PHP

For our simple example there is not any custom PHP but any required can be added straight after the variables have been setup (we will do this later in this tutorial)

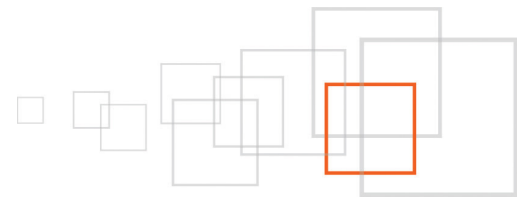
```
//carry out internal processing here, none required in this case.
```

6.2.3 Establishing what to render

The final part of the code is required to setup what template(s) will be used to render the view and other view specific details we need to know before the page can be rendered for the user. In this section we can also set custom parameters to send to the template file (again, we will do this later in the tutorial).

```
// setting up what to render to the user:
$result = array();
$result['content'] = $tpl->fetch( 'design:newsletter/userlist.tpl' );//main tpl file to
display the output
$result['left_menu'] = "design:newsletter/leftmenu.tpl";//uncomment this line if you want
to use a custom left navigation for this view.

$result['path'] = array( array( 'url' => 'newsletter/users',
                               'text' => 'User List' ) );//what to show in the Title bar
for this URL
```



There's a couple of things to note here:

- The Result array is used by eZ Publish to establish what needs to be displayed to the user, based on what templates are stored within it
- Although the menu.ini file provides options for setting up a left navigation you must provide an implementation of a left menu to display to the user. In this tutorial we are going to provide a flat implementation but see this post for more details and an alternative implementation using ini file contents.

Now that we have a basic PHP file linking to our View template lets setup the templates for the main content and also the left navigation.

Please note that if you want to see the current progress, you can now visit <http://www.yoursite.com/admin/newsletter/userlist> and you should currently be returned a blank page (since we have not created our templates yet). As we create the templates refer to this address to view what we have done so far. In the URL above, replace 'admin' by the name of your administration interface siteaccess, if different. You may also be using a different access method than URI-based, in which case you can simply remove the siteaccess name.

6.3 Creating the view Template files

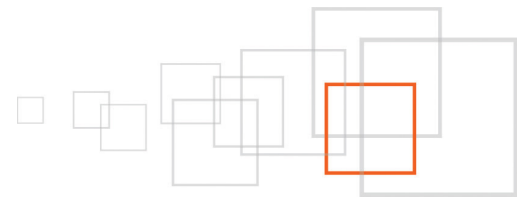
Move back into the root of your extension and create a folder called "design". In our case we are just storing template files but any specific CSS, Javascript or images for your extension should be stored in here, if required.

Create a directory called "admin" and within this a folder called "templates" (the templates folder will have the following path /extension/mynewsletter/design/admin/templates).

Our templates have been placed in a subdirectory of templates called "newsletter" so create a folder called newsletter within templates (the path will be /extension/mynewsletter/design/admin/templates/newsletter).

We are now ready to create our templates. Firstly, we will create an empty implementation of the left navigation. Create a file called "leftmenu.tpl" within newsletter (/extension/mynewsletter/design/admin/templates/newsletter/leftmenu.tpl). In the template, do nothing but enter some static text for now (for example "MENU HERE". This will act as a placeholder while we create the main template for the page.

Next, create a template called "userlist.tpl" in the newsletter design folder (/extension/mynewsletter/design/admin/templates/newsletter/userlist.tpl). In this template we need to pull out all users who have said they wish to receive the newsletter. For now, we will carry out a fetch statement limited to which users want to receive the newsletter and then print the name and email address of each user to screen. After we have done this we will extend the functionality to allow admin users to remove users from this list.

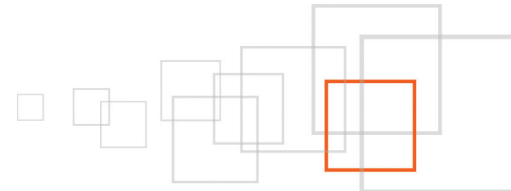


Enter the following code in your `userlist.tpl` template:

```
{def $users_folder = fetch( 'content', 'node', hash( 'node_path', 'Users/Members' ))
    $newsletter_users = fetch( 'content', 'list',
        hash( 'parent_node_id', $users_folder.node_id,
            'attribute_filter', array( array( 'user/newsletter', '=', 1 ))) )
}
<div class="box-header">
    <div class="button-left">
        <h2 class="context-title">Newsletter Users ({$newsletter_users|
count()})</h2>
    </div>
    <div class="float-break"></div>
</div>

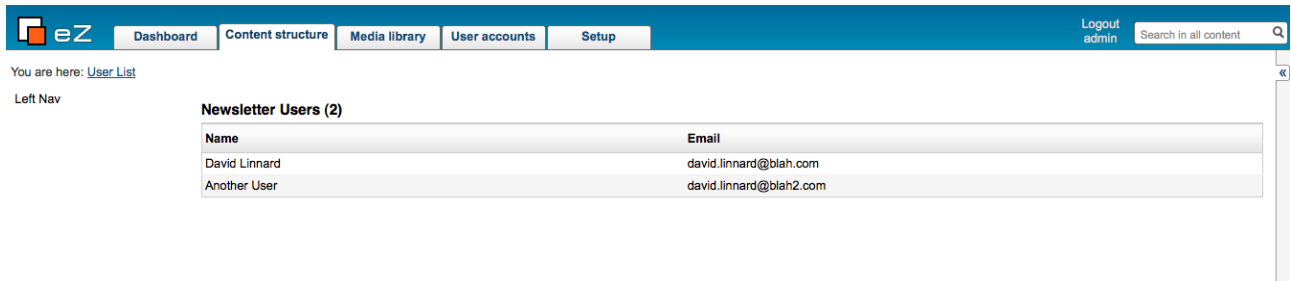
<div class="box-content">
    <div class="content-navigation-childlist">
        {if $newsletter_users|count()}
            <table cellspacing="0" class="list" id="ezasi-subitems-list">
                <thead>
                    <tr>
                        <th>Name</th>
                        <th>Email</th>
                    </tr>
                </thead>
                <tbody>
                    {for 0 to $newsletter_users|count()|dec() as $counter}
                        <tr class="{if eq($counter|
mod(2),0)}bglight{else}bgdark{/if}">
                            <td>{$newsletter_users[$counter].name}</td>
                            <td>{$newsletter_users[$counter].data_map.user_account.content.email}</td>
                        </tr>
                    {/for}
                </tbody>
            </table>
        {else}
            <p>No users are signed up!</p>
        </div>
</div>
{undef}
```

The code does look quite long but this is just to ensure our template follows the same style as the rest of the admin site. All we are doing is carrying out a simple fetch at the start and then looping through the data. If we strip the styling the code is simply the below:



```
{def $users_folder = fetch( 'content', 'node', hash( 'node_path', 'Users/Members'))
  $newsletter_users = fetch( 'content', 'list',
    hash( 'parent_node_id', $users_folder.node_id,
    'attribute_filter', array(array('user/newsletter', '=' ,1))))
}
<h2>Newsletter Users ({$newsletter_users|count()})</h2>
{if $newsletter_users|count()}
<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
{for 0 to $newsletter_users|count()|dec() as $counter}
<tr><td>{$newsletter_users[$counter].name}</td>
<td>{$newsletter_users[$counter].data_map.user_account.content.email}</td>
{/for}
</tbody>
</table>
{else}
<p>No users are signed up!</p>
{undef}
```

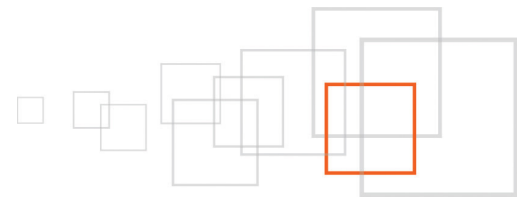
If you visit the page now you should see the list of users which looks similar to the below (<http://www.yoursite.com/admin/newsletter/userlist>) :



The screenshot shows the eZ CMS admin interface. At the top, there is a navigation bar with tabs for Dashboard, Content structure, Media library, User accounts, and Setup. On the right, there are links for Logout and admin, and a search box. Below the navigation bar, the breadcrumb path is "You are here: [User List](#)". On the left, there is a "Left Nav" section. The main content area displays a table titled "Newsletter Users (2)".

Name	Email
David Linnard	david.linnard@blah.com
Another User	david.linnard@blah2.com

The only task left to make our view usable is to make sure the users can access it in the first place. Once we have done this you will have a simple working view which you should be able to extend however you want.



6.4 Providing Navigation

There are two things we now need to do to ensure so that the user can easily navigate to and around our module:

- Add a tab to the options running along the top of the site access
- Provide a menu within the module to allow the user to navigate around the module

Since we currently only have one view the more important of the two is definitely the first but we will do both of these now.

6.4.1 Adding a menu tab

Adding a menu tab will do two things. As well as providing an easy way to navigate to your module it will also ensure the new tab is highlighted rather than the default “Content Structure” highlight. We need to do this in two steps.

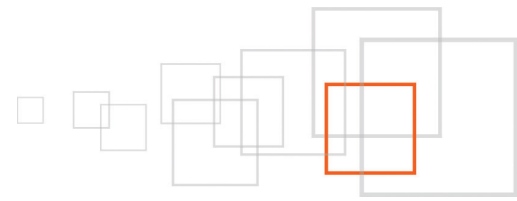
Creating the tab requires you to create a settings file within our extension and add the details to the new module we have created in the new file. eZ Publish will then pick this up automatically and our tab is ready to use.

The settings file we need to create is “menu.ini.append.php”. Navigate to the root of your extension and then navigate to your settings directory (/extension/mynewsletter/settings/). Once you are in there create a file called menu.ini.append.php and enter the following content:

```
[NavigationPart]
Part[newsletternavigationpart]=Newsletters

[TopAdminMenu]
Tabs[]=newsletters

[Topmenu_newsletters]
NavigationPartIdentifier=newslettersnavigationpart
Name=Newsletters
Tooltip=Managing the Newsletter list and sending emails
URL[]
URL[default]=newsletter/userlist
Enabled[]
Enabled[default]=true
Enabled[browse]=false
Enabled[edit]=false
Shown[]
Shown[default]=true
Shown[edit]=true
Shown[navigation]=true
Shown[browse]=true
```



The code is very straightforward. The NavigationPart section of the file allows us to target the tab as our current one when we are in our module. The tab is added using the Tabs[] array. The other settings specify a name, tool tip and behaviours associated with this tab. For instance, our tab does not allow you to navigate to our tab while you are editing an object (although the tab is still shown).

With the tab now showing we now need to ensure that the Newsletter tab is highlighted whilst we are in our module. Go to the module.php file you created earlier (/extension/mynewsletter/modules/newsletter/module.php) and add the line of code in **bold**,

```
<?php
$module = array( 'name' => 'newsletter' ); //the name of our module

$ViewList = array();//add as many views as you want here
$ViewList['userlist'] = array( 'script' => 'userlist.php',
                               'default_navigation_part' => 'newslettersnavigationpart',
                               'functions' => array( 'read' ));//the script used to setup
the template plus the user permissions required for it (also see below)

//setting user permissions required by our module:
$FunctionList = array();
$FunctionList['read'] = array();
?>
```

The extra line will links our view with the extra tab we have just created.

6.4.2 Adding a left menu

For the left menu, we just need to update the code in the template file we have already created (/extension/mynewsletter/design/admin/templates/newsletter/leftmenu.tpl). At the current time we only have the one link but we will add to this later. The styling used on other parts of the admin site is used here to ensure the module is consistent with the rest of the CMS:

```
{def $has_read_permission = fetch( 'user', 'has_access_to',
                                   hash( 'module', 'newsletter',
                                           'function', 'read',
                                           'user_id', $current_user.contentobject_id ) )}

<div class="box-header"><div class="box-tc"><div class="box-ml"><div class="box-mr"><div
class="box-tl"><div class="box-tr">

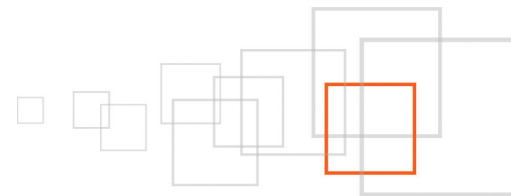
<h4>Newsletters</h4>

</div></div></div></div></div></div>

<div class="box-bc"><div class="box-ml"><div class="box-mr"><div class="box-bl"><div
class="box-br"><div class="box-content">

<ul>

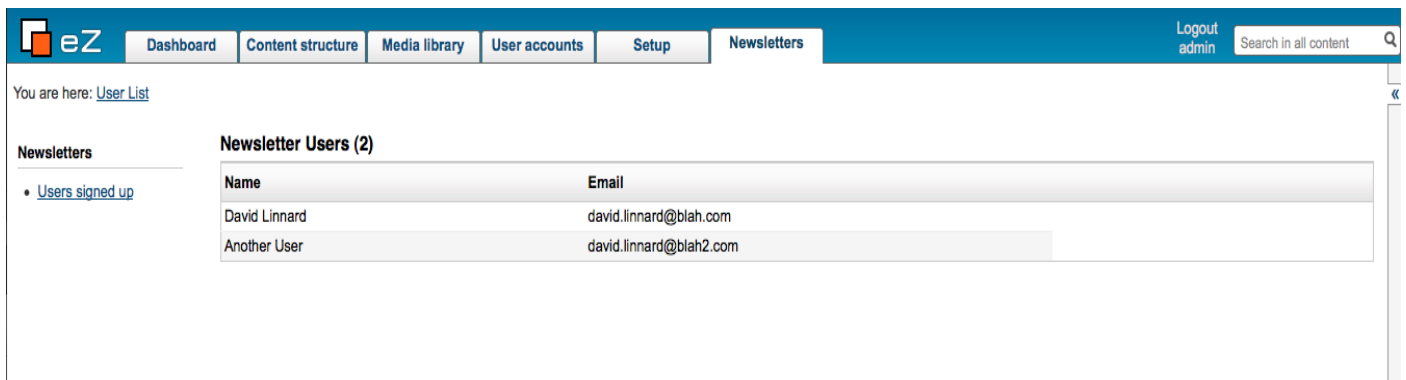
{if $has_read_permission}
    <li><div><a href={'/newsletter/userlist'|ezurl()}>Users signed up</a></div></li>
```



```
{/if}  
</ul>  
</div></div></div></div></div></div>
```

Although the majority of the code is static HTML, it is worth noting how permissions can be used to restrict which menu items are shown. This is carried out on the first couple of lines with a Fetch built in to eZ Publish. We then use the variable we set to store the value of this when we display the menu item. As the module currently only has one view, this is actually not needed as the only time the menu will show is when the user is on this particular page anyway. We are adding the code here as we will soon be adding another view with a different permission set and so it will then be needed.

Your view should now look like this:



The screenshot shows the eZ Publish dashboard with a blue header. The navigation menu includes Dashboard, Content structure, Media library, User accounts, Setup, and Newsletters. The Newsletters menu item is active. On the right, there is a 'Logout admin' link and a search box labeled 'Search in all content'. The main content area shows 'You are here: [User List](#)'. On the left, there is a sidebar with 'Newsletters' and a sub-item 'Users signed up'. The main content area displays a table titled 'Newsletter Users (2)' with two columns: 'Name' and 'Email'. The table contains two rows: 'David Linnard' with email 'david.linnard@blah.com' and 'Another User' with email 'david.linnard@blah2.com'.

Although our example does not provide much functionality it demonstrates the basics for creating modules and views. Let's extend the view we've created to allow some user interaction to show how additional functionality can be added to your code.

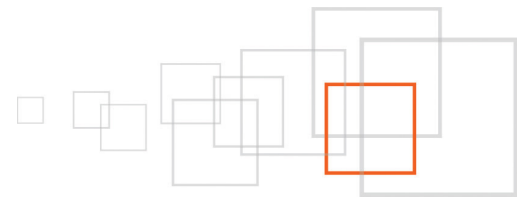
7 Passing parameters to Views

A simple requirement asked when dealing with newsletters is the ability to remove users from the mailing list. Let's add that functionality to our view to allow admin users to remove a user from the mailing list by clicking on a link alongside the email address of each user. This will allow us to demonstrate how we can pass and handle parameters with Views.

7.1 Accounting for Parameters

Parameters being sent to the page need to be specified within the module.php file so our first step is to return to ours and add the parameters to the file (/extension/mynewsletter/modules/newsletter/module.php).

There are two types of parameters within eZ Publish and both are accessed in the same way. The only difference is how they appear in the URL and where they are defined in module.php. To remove a user from the mailing list we just need to supply one parameter into our view which will be the user ID of the user to remove from the mailing list. Let's see how we would do this with both types of parameter.



7.1.1 Ordered Parameters

These always come first out of the two types. When declaring ordered parameters the name of the parameter does not appear in the URL. For example, if the user ID of the user we were removing from our mailing list was 42 the parameter would appear as follows if it was ordered:

<http://www.yoursite.com/admin/newsletter/userlist/42>

If we were to declare the parameter in the module.php file this is how our view declaration would be written:

```
$ViewList['userlist'] = array( 'script' => 'userlist.php',  
                             'default_navigation_part' => 'newslettersnavigationpart',  
                             'params' => array( 'remove' ),  
                             'functions' => array( 'read' ) );
```

You can then access the parameter within your View PHP file as follows:

```
$Params['remove']
```

If we want to add more parameters into the module.php file just add more array elements to the “params” array.

7.1.2 Unordered Parameters

These always come last out of the two types and they can be written in any order. This is because both the name and the value are defined in the URL. Following on from our previous example, if the user ID of the user we were removing from our mailing list was 42 the parameter would appear as follows if it was unordered:

[http://www.yoursite.com/admin/newsletter/userlist/\(remove\)/42](http://www.yoursite.com/admin/newsletter/userlist/(remove)/42)

If we were to declare the parameter in the module.php file this is how our view declaration would be written:

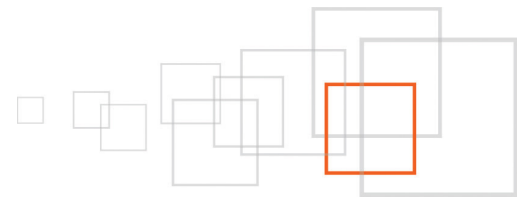
```
$ViewList['userlist'] = array( 'script' => 'userlist.php',  
                             'default_navigation_part' => 'newslettersnavigationpart',  
                             'unordered_params' => array( 'remove' => 'remove' ),  
                             'functions' => array( 'read' ) );
```

You can then access the parameter in the same way within your View PHP file as follows:

```
$Params['remove']
```

As with the ordered parameters, adding new parameters is as easy as adding more array items to an array, in this case the “unordered_params” array.

The key in the unordered_params array is used to specify what is used within the URL. The value is then used within the \$Params array in your view.



7.1.3 Implementing Parameters

We are going to use unordered parameters in our example. This is mainly because out of habit I tend to only use ordered items when it makes as much semantic sense as possible.

For instance, if we were to extend our newsletters functionality with the ability to have different newsletters we would want to display the mailing list for each newsletter. If we had a newsletter called news and another called sport the URLs for the mailing list could be:

- <http://www.yoursite.com/admin/newsletter/userlist/news>
- <http://www.yoursite.com/admin/newsletter/userlist/sport>

This infers that our url is for a list of users of a news/sports newsletter and so it makes sense in terms of the page we are viewing.

In the current view we are implementing, the user ID we are removing has no semantic meaning attached. By having it as an unordered parameter so it has some meaning applied as the name of the parameter is also supplied:

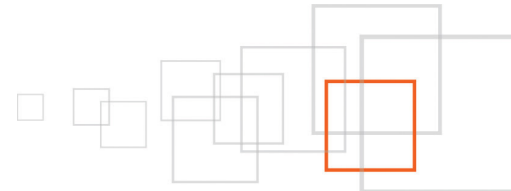
- Ordered: <http://www.yoursite.com/admin/newsletter/userlist/42>
- Unordered: <http://www.yoursite.com/admin/newsletter/userlist/remove/42>

In the code examples that follows (and the code which you can download at the end of the tutorial) we will cover the code for implementing both ordered and unordered parameters so in future you can use both when necessary.

7.2 *Allowing user input in our Template*

We can make the required changes to the page by adding a link in each row to the same page, whilst adding the additional "remove" parameter. Open the main template file for the page again (/extension/mynewsletter/design/admin/templates/newsletter/userlist.tpl) and update the div with the class "box-content" with the following code (the changes are highlighted in **bold**):

```
...
<div class="box-content">
    <div class="content-navigation-childlist">
        {if $newsletter_users|count()}
            <table cellpadding="0" class="list" id="ezasi-subitems-
list">
                <thead>
                    <tr>
                        <th>Name</th>
                        <th>Email</th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
```



```

                                {for 0 to $newsletter_users|count()|dec()
as $counter}
                                <tr class="{if eq($counter|
mod(2),0)}bglight{else}bgdark{/if}">
                                <td>{$newsletter_users[$counter].name}</td>
                                <td>{$newsletter_users[$counter].data_map.user_account.content.email}</td>
                                <td>
                                <a
                                href={concat($module_details['module_name'],'/',
                                $module_details['function_name'],'/remove/',
                                $newsletter_users[$counter].contentobject_id)|ezurl()}>
                                Remove from list
                                </a>
                                </td>
                                </tr>
                                {/for}
                                </tbody>
                                </table>
                                {else}
                                <p>No users are signed up!</p>
                                {/if}
                                </div>
</div>
...

```

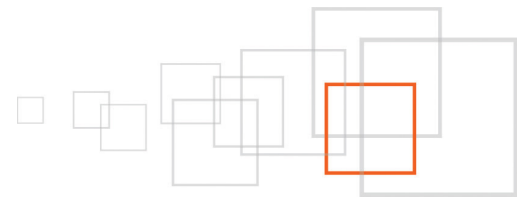
When clicked, the buttons next to each set of user details will bring the operator back to the same page of the site so we now need to take additional steps in our PHP file to make sure something is done about the input.

7.3 Updating our PHP to allow user interaction

The PHP we create needs to update the content object of the user so that the newsletter attribute is set to false. eZ Publish 4.3 launched the perfect functionality to do this easily with the introduction of the function `eZContentFunctions::updateAndPublishObject`. Unfortunately though, the code is not as straightforward if your eZ Publish build is pre 4.3.

Both unordered and ordered parameters can be accessed in your PHP using the `$Params` array. This is a simple associated array so in our case `remove` can be accessed using `$Params['remove']`. We can check whether this value has been set and if it has, carry out our additional

I have split the code for this example into two parts to deal with the two versions of the code. These are the complete files. Make sure to update all of the lines in **bold** text:



7.3.1 eZ Publish 4.2 and earlier

```
<?php
/**
 * File containing the eZ Publish view implementation.
 *
 * @copyright Your Name here
 * @license licence details
 * @version 1.0.0
 * @package mynewsletter
 */
function ezUpdate42($contentObject)/*adapted from updateAndPublishObject in version 4.3*/
{
    $db = eZDB::instance();
    $db->begin();
    $newVersion = $contentObject->createNewVersion( false, true, $languageCode );

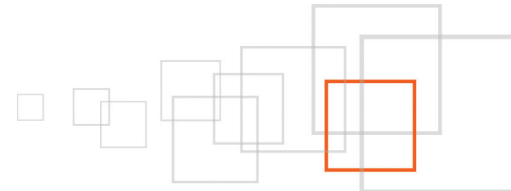
    if ( !$newVersion instanceof eZContentObjectVersion )
    {
        eZDebug::writeError( 'Unable to create a new version for object ' .
                            $contentObject->attribute( 'id' ), 'userlist.php' );
        $db->rollback();
        return false;
    }

    $newVersion->setAttribute( 'modified', time() );
    $newVersion->store();

    $attributeList = $newVersion->attribute( 'data_map' );
    $newsletter_attribute = $attributeList['newsletter'];
    $newsletter_attribute->fromString( 0 );
    $newsletter_attribute->store();

    $db->commit();

    $operationResult = eZOperationHandler::execute( 'content', 'publish',
        array( 'object_id' => $newVersion->attribute( 'contentobject_id' ),
              'version'   => $newVersion->attribute( 'version' ) ) );
    return $operationResult;
}
```



```
//version <4.3 of eZ Publish:
include_once( 'kernel/common/template.php' );
$tpl = templateInit();

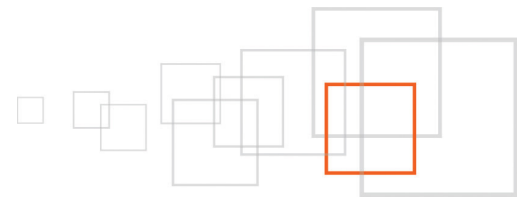
if ( $Params['remove'] != null )
{
    $result = 0; //used to measure whether the update has been made successfully or not
    $contentObject = eZContentObject::fetch( $Params['remove'] );
    if( $contentObject instanceof eZContentObject )
    {
        /*code for eZ Publish 4.2 and earlier - see additional function for full code*/
        $result = ezUpdate42($contentObject);
    }
    if ($result==0)
    {
        $tpl->setVariable( 'error', "There was a problem updating the user" );
    }
    else
    {
        $tpl->setVariable( 'feedback', "The user has been removed successfully" );
    }
}

// Process template and set path data:
$result = array();
$result['content'] = $tpl->fetch( 'design:newsletter/userlist.tpl' );//main tpl file to
display the output

$result['left_menu'] = "design:newsletter/leftmenu.tpl";

$result['path'] = array( array( 'url' => 'newsletter/userlist',
                                'text' => 'User List' ) );

?>
```



7.3.2 eZ Publish 4.3 onwards

```
<?php
$tpl = eZTemplate::factory();//this line of code is for ez publish 4.3, replace it with
the following lines for versions prior to that

if ($Params['remove']!=null)
{
    $result = 0;//used to measure whether the update has been made successfully or not
    $contentObject = eZContentObject::fetch( $Params['remove'] );
    if( $contentObject instanceof eZContentObject )
    {
        /*code for eZ Publish 4.3 onwards*/
        $params = array();
        $attributes = array("newsletter"=>0);
        $params['attributes'] = $attributes;
        $result = eZContentFunctions::updateAndPublishObject( $contentObject, $params )

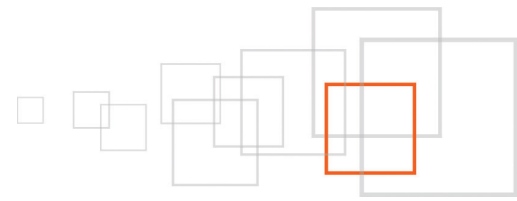
    }
    if ($result==0)
    {
        $tpl->setVariable( 'error', "There was a problem updating the user" );
    }
    else
    {
        $tpl->setVariable( 'feedback', "The user has been removed successfully" );
    }
}

// Process template and set path data:
$Result = array();
$Result['content'] = $tpl->fetch( 'design:newsletter/userlist.tpl' );//main tpl file to
display the output

$Result['left_menu'] = "design:newsletter/leftmenu.tpl";

$Result['path'] = array( array( 'url' => 'newsletter/userlist',
                                'text' => 'User List' ) );

?>
```



7-3-3 How it works

The difference between the two versions is primarily due to the update process (the only other difference is how the template object is initialised). The logic makes use of built in eZ Publish functionality to update the content, if we establish that a content object update is required.

Once we have established the user has requested to remove a mailing list entry, we extract the Content Object ID of the mailing list and then ensure that this is valid. If it is we update the object so that the newsletter field is turned off.

In order for us to provide feedback to the user we then need to pass details of what we have done to the template. This is carried out through the template object we create at the start of the code. In our initial code this only fetched the template we were using. This time, before we fetch the template we pass in one of two variables to the page:

```
if ( $result == 0 ) //setting a template variable to return feedback to the user
{
    $tpl->setVariable( 'error', "There was a problem updating the user" );
}
else
{
    $tpl->setVariable( 'feedback', "The user has been removed successfully" );
}
```

If the user could not be updated we pass an error into the template and if not we pass the message that the user was removed from the list successfully. We now need to make sure this message is displayed in our template.

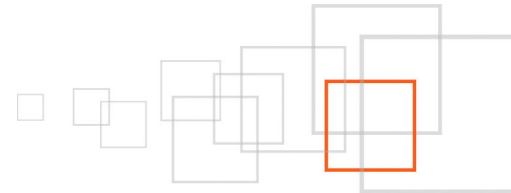
7.4 **Providing user feedback**

To provide the feedback to the user, we are going to use the message box used elsewhere in the admin site access to display the variable we have just set in our PHP . Accessing these variables is easy and is identical to if you would of declared these variables in your template. To access the error variable we use `{ $error }` and for the feedback variable we use `{ $feedback }`.

We are going to store the feedback message in a new template so we can easily reuse the functionality in our other views.

7.4.1 Creating the error message template

Go into your newsletter template directory (`/extension/mynewsletter/design/admin/templates/newsletter/`). Within the directory create a file called "newsletter_feedback_box.tpl". We are going to use a simplified version of the code used to provide feedback elsewhere. In our case we can display errors or warnings and so the following code is sufficient. Add this to the template and then save and exit (this code will not need to be changed again) :



```
{if is_set($error)}
    <div class="message-error">
        <h2><span class="time">[{{currentdate()|l10n( shortdatetime )}}</span> Problem
encountered</h2>
        <p>{$error}</p>
    </div>
{elseif is_set($feedback)}
    <div class="message-feedback">
        <h2><span class="time">[{{currentdate()|
l10n( shortdatetime )}}</span> Success!</h2>
        <p>{$feedback}</p>
    </div>
{/if}
```

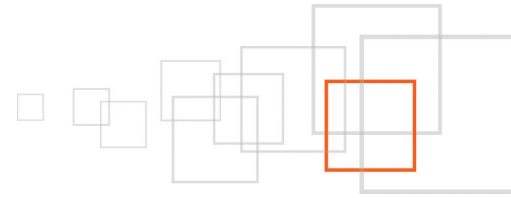
All the code does is check to see if an error or feedback is passed into the template and if it has this is displayed in the usual style for the admin site access. Now we need to include this in our main view template.

7.4.2 Updating our main view template

Go back into the `userlist.tpl` file (`/extension/mynewsletter/design/admin/templates/newsletter/userlist`). Add the code in bold (the code block is taken from the start of the template):

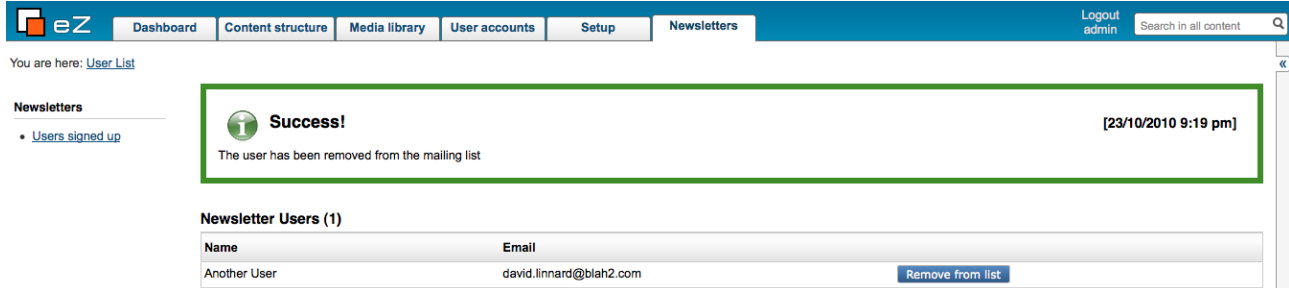
```
{def $users_folder = fetch( 'content', 'node', hash( 'node_path', 'Users/Members' ))
    $newsletter_users = fetch( 'content', 'list',
        hash( 'parent_node_id', $users_folder.node_id,
            'attribute_filter', array( array( 'user/newsletter', '=', 1 ) ) )
    )
}
{*Include message box if necessary:*
{if or( is_set( $feedback ), is_set( $error ) )}
    {include uri="design:newsletter/newsletter_feedback_box.tpl" feedback=$feedback
error=$error}
{/if}
<div class="box-header">
    <div class="button-left">
        <h2 class="context-title">Newsletter Users ({{newsletter_users|
count()}}</h2>
    </div>
        <div class="float-break"></div>
</div>
...
```

The rest of the template is left out for brevity.



7-4-3 The finished view

You will now receive feedback messages when you remove users from the newsletter list. If you open the page in your browser (<http://www.yoursite.com/admin/newsletter/userlist>) when you click on a user to remove them, you will receive notification they have been removed, similar to the below:



We have now covered how to implement custom PHP based on user input and how you can return information back to the user through the Template object. Let's now look at how to implement a second view within the same module. We'll create a view which will send out an email to all users on the newsletter list we have just set up. This will allow us to see how POST and GET parameters can be used within your views.

8 Creating a second view – handling POST data

Once you have created the module and the first view, setting up the second view is much, much quicker as a lot of the initial steps we took only need to be carried out at module rather than view level. Because of this we do not need to create or update nearly as many files.

To show how we can handle form data, let's setup a basic form which takes email content from the user and then sends it to the users who have signed up to receive newsletters. We will start by updating the files we already have and then we will build the additional TPL and PHP file we need to create to implement the logic and display code for the new view.

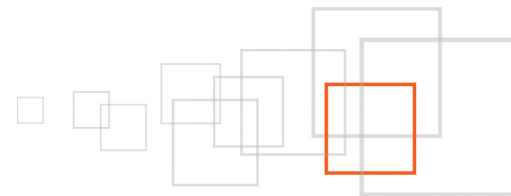
8.1 Updating existing files

The only files we need to update are those which hold view specific information. The key file to update is the module.php we created in our modules directory (`/extension/mynewsletter/modules/newsletter/module.php`) which you may recall details where each view is located within the module. Apart from this the only other file to update is the static left navigation template file we created (`/extension/mynewsletter/design/admin/templates/newsletter/leftmenu.tpl`).

Let's update both of these now with the details of our new view.

8.1.1 Updating the Module.php file for multiple views

Open your module.php file (`/extension/mynewsletter/modules/newsletter/module.php`). To add another view to our module we need to add another array to `$ViewList` with the details for the new module. Let's call our new view "sendemail" and give it a different permission so that only certain users can send newsletters (additions to our previous module.php file are in **bold**) :



```
<?php
$module = array( 'name' => 'newsletter' ); //the name of our module

$ViewList = array();//add as many views as you want here
$ViewList['userlist'] = array( 'script' => 'userlist.php',
                                'default_navigation_part' => 'newslettersnavigationpart',
                                'functions' => array( 'read' )); //the script used to
                                setup the template plus the user permissions required for it (also see below)
$ViewList['sendemail'] = array( 'script' => 'sendemail.php',
                                'default_navigation_part' =>
                                'newslettersnavigationpart',
                                'functions' => array( 'send' ));

//setting user permissions required by our module:
$FunctionList = array();
$FunctionList['read'] = array();
$FunctionList['send'] = array();
?>
```

As you can see it is pretty much identical to our previous code. We have separated the permissions between the views so it is possible for a user to view the user list without sending emails and vice versa. This is a great way of limiting who can do what in your modules and ensure you maintain a good level of security.

As with the 'userlist' view we can get the URL from the array key used in the \$ViewList array. Once we have created the templates we will be able to access the page at the url:

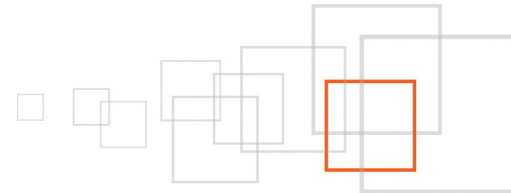
<http://www.yoursite.com/admin/newsletter/sendemail>

8.1.2 Updating the left navigation

Again, our left menu is going to be a simple addition to the code we already have. Open the menu template (/extension/mynewsletter/design/admin/templates/newsletter/leftmenu.tpl). Since we are using a different permission we need to check the user's access level to both "read" and now "send" but this is based on the code we have already created. Our new code is shown in **bold**:

```
{def $has_read_permission = fetch( 'user', 'has_access_to',
                                hash( 'module', 'newsletter',
                                      'function', 'read',
                                      'user_id', $current_user.contentobject_id ) )}

{def $has_send_permission = fetch( 'user', 'has_access_to',
                                hash( 'module', 'newsletter',
                                      function, 'send',
                                      'user_id', $current_user.contentobject_id ) )}}
```



```
<div class="box-header"><div class="box-tc"><div class="box-ml"><div class="box-mr"><div
class="box-tl"><div class="box-tr">
<h4>Newsletters</h4>
</div></div></div></div></div></div>

<div class="box-bc"><div class="box-ml"><div class="box-mr"><div class="box-bl"><div
class="box-br"><div class="box-content">
<ul>
                {if $has_read_permission}
                <li><div><a href={'/newsletter/userlist'|ezurl()}>Users signed
up</a></div></li>
                {/if}
                {if $has_send_permission}
                <li><div><a href={'/newsletter/sendemail'|ezurl()}>Send email</a></div></li>
                {/if}
</ul>
</div></div></div></div></div></div>
```

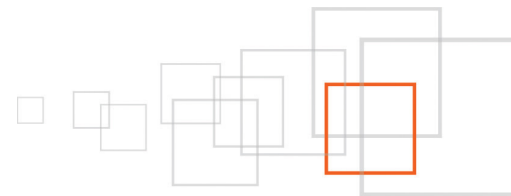
8.2 Creating our new view- Template file

Now, we need to make sure the user has a template to see. We are going to create a simple POST based form. We will then use eZHTTPTool to access the text the user has entered send it out to the mailing list. We will make use of the feedback box we used in our previous view to show feedback to the user. For simplicity, our form will only allow users to send text based rather than HTML emails. Create a file called "sendemail.tpl" in the same directory as our other templates (/extension/mynewsletter/design/admin/templates/newsletter/). Add the following code to the new file:

```
{*Include message box if necessary:*}
{if or(is_set($feedback),is_set($error))}
                {include uri="design:newsletter/newsletter_feedback_box.tpl"
feedback=$feedback error=$error}
{/if}

<div class="box-header">
    <div class="button-left">
        <h2 class="context-title">Send Email</h2>
    </div>
        <div class="float-break"></div>
</div>

<div class="box-content">
    <form method="POST" >
```



```
<div class="block">
    <label>Enter your email</label>
    <textarea name="email" cols="60" rows="20"></textarea>
</div>
<div class="controlbar">
    <div class="block">
        <input type="submit" title="Send the email to all users
in your mailing list" value="Send email" name="sendButton" class="defaultbutton">
    </div>
</div>
</form>
</div>
```

The key part is the simple form we have created within the bottom div (with the class "box-content") which will pass the email back to the view where it will be sent out. The other content within the template is to ensure the view is tailored for the admin site access.

8.3 Creating our new view – PHP file

We now need to create the PHP file which will setup the view. The same file will also be sending the email. Create a new PHP file called `sendemail.php` and store it alongside our existing PHP files (`/extension/mynewsletter/modules/newsletter/`).

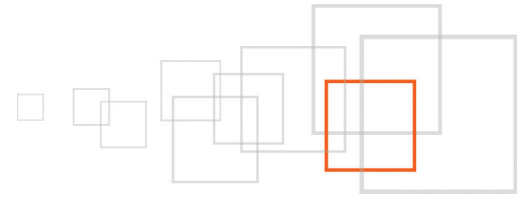
Our code is split into three parts, code each of the following code blocks in the order they appear:

8.3.1 Initialising the eZ Publish based variables

As well as the Template Object we initialised in our previous view, in our new view we need to process form fields. In eZ Publish we do this using the `eZHTTPTool` which provides a wrapper to GET, POST and Session data.

Be mindful of the different code for eZ Publish 4.2 and earlier.

```
<?php
$http = eZHTTPTool::instance();
$tpl = eZTemplate::factory();//this lines of code is for ez publish 4.3, replace it with
the following line for versions prior to that
//version <4.3 of eZ Publish should use these lines of code instead:
//include_once( 'kernel/common/template.php' );
//$tpl = templateInit();
```



8.3.2 Processing the form

Straight after the variables are set we need to carry out the form processing and send the email to the mailing list, if the user has submitted the form. We are carrying this out now to ensure that when the main view template is fetched, the feedback from the pending request is available if necessary:

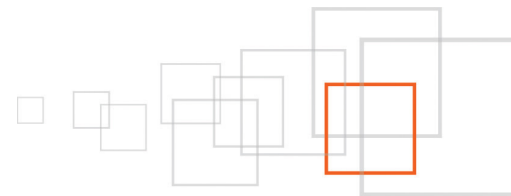
```
if ( $http->hasPostVariable( 'email' ) )
{
    //setting up the eZMail object:
    $mail = new eZMail();
    $ini = eZINI::instance();
    $emailSender = $ini->variable( 'MailSettings', 'AdminEmail' );
    $mail->setSender( $emailSender );
    $mail->setReceiver( $emailSender, "MySite Newsletter");
    $mail->setSubject( "Newsletter from MySite" );
    $mail->setBody( $http->postVariable('email') );

    /* fetching all users who wish to receive emails: */
    $attributeFilter = array( array( 'user/newsletter', '=', 1 ) );
    $params = array('AttributeFilter'=>$attributeFilter);
    //getting the ID of where the users sit in the cms (limiting the
area eZ has to search for the objects):
    $parent_node =
eZContentObjectTreeNode::fetchByURLPath( 'users/members' ); //note the lowercase. Use
underscores rather than hyphens if spaces are included in the path
    $parent_node_id = $parent_node->NodeID;

    //putting all of the above together into a fetch query:
    $newsletter_users =
eZContentObjectTreeNode::subTreeByNodeID( $params, $parent_node_id );

    //adding all users to the receiver list:
    foreach( $newsletter_users as $user )
    {
        $userFields = $user->attribute( 'data_map' );
        $mail->addBcc( $userFields['user_account']-
>attribute('content')->attribute('email'), $userFields['user_account']-
>attribute('content')->attribute('login'));
    }

    if ( eZMailTransport::send( $mail ) )
    {
        $tpl->setVariable( 'feedback', "The email has been sent to " .
count( $mail->bccElements() ) . " users" );
    }
}
```



```
        }
    else
    {
        $tpl->setVariable( 'error', "There was a problem sending the
email" );
    }
}
```

The eZHTTP tool is very straightforward to use and can be used in exactly the same way to pull out GET parameters if necessary. Once we have established an email needs to be sent, we initialise an eZMail object and add all users within the mailing list to the BCC list the email is being sent to. We then send out the email and then supply an appropriate message to the user, based on the result. It should be noticed that the following PHP:

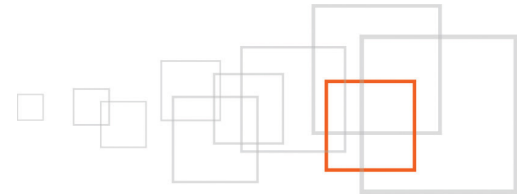
```
/*fetching all users who wish to receive emails:*/
$attributeFilter = array( array( 'user/newsletter', '=', 1 ) );
$params = array('AttributeFilter'=>$attributeFilter);
//getting the ID of where the users sit in the cms (limiting the area eZ has to search
for the objects):
$parent_node = eZContentObjectTreeNode::fetchByURLPath( 'users/members' ); //note the
lowercase. Use underscores rather than hyphens if spaces are included in the path
$parent_node_id = $parent_node->NodeID;

// putting all of the above together into a fetch query:
$newsletter_users = eZContentObjectTreeNode::subTreeByNodeID( $params,$parent_node_id );
```

Is equivalent to the Fetch we created previously in our template for the list of users (userlist.tpl):

```
{def $users_folder = fetch( 'content', 'node', hash( 'node_path', 'Users/Members' ) )
$newsletter_users = fetch( 'content', 'list',
    hash( 'parent_node_id', $users_folder.node_id,
        'attribute_filter', array( array( 'user/newsletter', '=', 1 )))
}
```

Please note you can also use the ezcMail class (from the Apache Zeta Components) for emails when sending emails via eZ Publish.



8.3.3 Process the view

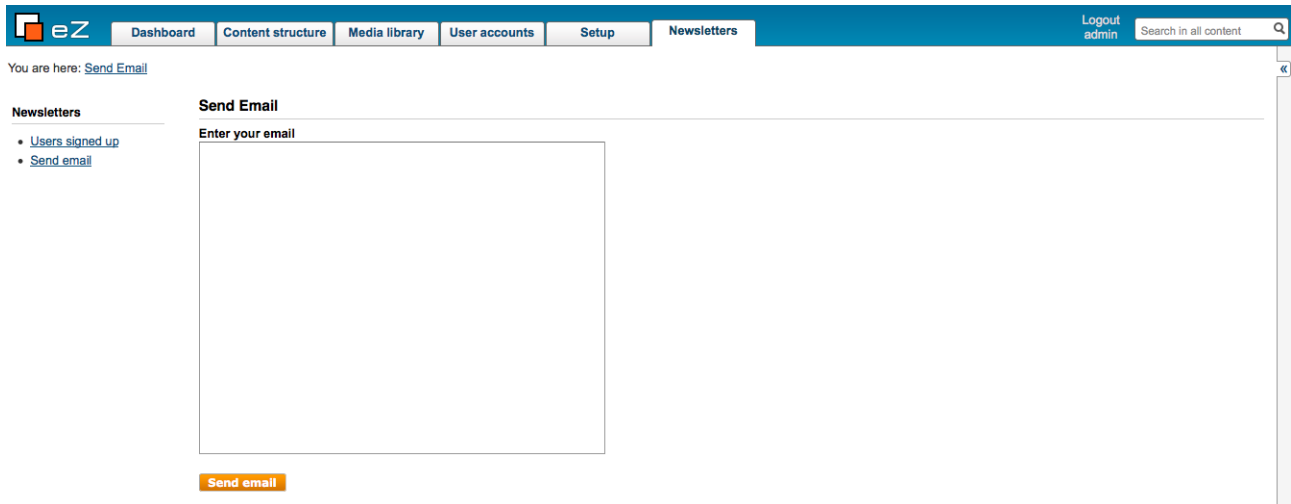
Finally, we need to setup the content the user sees. This code works in an identical manner to the code we created for our previous view:

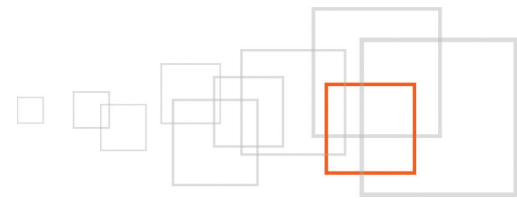
```
// Process template and set path data:
$Result = array();
$Result['content'] = $tpl->fetch( 'design:newsletter/userlist.tpl' );//main tpl file to
display the output

$Result['left_menu'] = "design:newsletter/leftmenu.tpl";
$Result['path'] = array( array( 'url' => 'newsletter/userlist',
                               'text' => 'User List' ) );
?>
```

8.4 The Finished View

Our second view is now complete, The final view should look something like the following:





9 Modules and Permissions

So far, we have assumed the only users with access to our module, are admin users with rights to all of the eZ Publish system. We will now cover the stages to make our module available for another type of user within the CMS.

Please note the Roles and Permissions system is extremely flexible and powerful in eZ Publish and so we will concentrate on what we need to do to make the module accessible. I am doing the first changes from a fresh install.

9.1 Updating the Role

Click on the "User Accounts" tab in the CMS. If you know the name of the role you need to update, you can click on the Roles and Policies link in the left hand column. If you do not, or if you want to see limitations associated with the User Group roles, click on the User Group you want to give permission. These are found in the main content towards the bottom of the page. From the User Group page, click on the Role associated with the user.

In our example, let's update the role associated with the User Group "Editors". Once the User Group is displayed for editors, click on the "Roles" tab and you should see a list of roles and limitations associated with Editors. If you are building from a fresh build the roles will look similar to the below.

Name	Limitation
 Editor	Subtree (/1/2/) 
 Editor	Subtree (/1/43/) 


For our module this is an issue as Editor's only have access to the Content and Media tabs (as they should). However, we need to make sure that when the User List is displayed, the editors also have permission to see the users in the mailing list (currently they will be shown an empty list).

From this though we know the Role we need to update the "Editor" role. Click on Editor in the Roles tab and the Editor role will be displayed, along with associated policies. There are several additions we need to make:

- Allow Editors to see the site members' details
- Allow editors access to the Users List page
- Allow editors access to the Send Email page

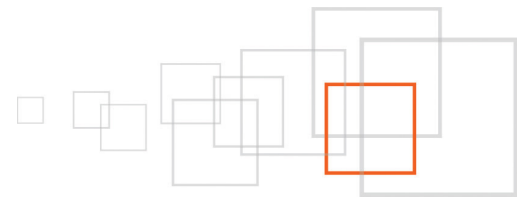
To do this, click on the edit button below the list of policies. To allow Editors to see the site members' details we need to do two things. Firstly we need to make sure the policy allows them to read from the "User Accounts" tab. Secondly we need to restrict their view of this to site members only.

To allow Editors to read from the "User Accounts" tab, find the following policy and click the edit button in the right hand side of the row:

content read Section(Standard , Media , Restricted) 

You will now see a set of drop downs. In the "Section" drop down, add the "Users" section to the list of sections already available and then press the OK button at the bottom of the page.

Now let's ensure that the Editor is restricted to seeing only Members. We do this when we assign the role so go to the bottom of the "Edit <Editor> Role" page and click on the save button. You should now be back at the



main page for the Editor role. Go to the bottom of the page and you will be presented with a list of User Groups who have this role and you will notice Editors are in here twice for different areas of the site. We need to add them for a third time. While the dropdown next to "Assign with limitation" is set to subtree, press the "Assign with limitation" button. Browse to the User Accounts tab and select the radio button for "Members". Now press OK and then check the "Editors" checkbox before clicking on OK again. Your Editors will now have read access to all users with the Members usergroup. If you wish to allow Editors to remove users from the mailing list repeat the same process but this time for the content edit function.

We now need to give the users access to our new module. the User Role "Editor" again by clicking on the "Edit" button. This time go to the bottom of the existing policies and click the button "New Policy". Select our module from the "Modules" dropdown and then assign the function you want to provide access to (we want to do both so do this for each) and then click on "Grant Full Access".

All of our policies should now be setup for Editors to use our Module.

10 Next Steps

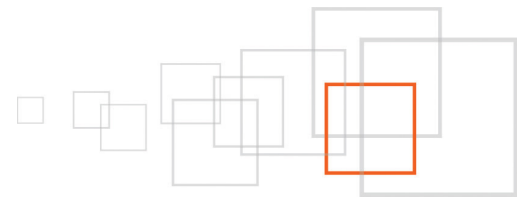
10.1 *Creating additional Views and Modules*

You should now be comfortable with setting up additional views if you want to within the same module. If you want to create a new module to work on your own set of functionality, update your module.ini.append.php file as you did at the start of the tutorial to add your new module (there's an example of how this will look below) or alternatively run through what we have covered in this tutorial to start afresh in a new extension.

```
<?php /*
[ModuleSettings]
ExtensionRepositories[]=mynewsletter
ModuleList[]=newsletter
ModuleList[]=yourmodule
*/?>
```

10.2 *Creating Modules for your main site access*

If you want to enable your modules on the front of your site, you can do so by updating the permissions for the default user type within the Users tab. Update the user to get the relevant permissions for your modules and they will become available to them (as we have just done in the admin site access for Editorial Users).



11 Conclusion

You should now know all you need to to start creating your own modules and views. We have covered how to do the initial set up and how to handle user feedback. You should also be comfortable with setting up basic permissions within your roles to enable select user groups to access your custom functionality (or for everybody to view it if necessary). Modules are a great way of extending eZ Publish and they provide a massive amount of flexibility and power to carry out complex tasks. The code example for the module we have created in this tutorial is available for download so please do download and play around with it.

12 Resources

There are several excellent tutorials already on the site which are of great use when creating custom extensions. Below is a list of these and other resources I found useful while compiling this tutorial:

- An introduction to developing eZ Publish extensions
- Adding custom security policy limitations to your modules
- Using menu.ini for your tabs and menus.
- eZMail Class Reference
- ezcMail documentation

13 About the author : David Linnard



David is a London based web developer with a wide variety of skills who has spent the past several years developing for commercial eZ Publish sites. He is also experienced at handling a variety of other content management systems and frameworks, in particular the Zend Framework.

14 License choice

Available under the Creative Commons AttributionNon-Commercial License

