Introduction

phpFox is based on a modular system, which runs all of the common features we as users see when we visit a phpFox website. It also contains libraries or libs as we will call it, which is the engine that runs the skeleton of phpFox and its modules. The products core is designed to be a CMS (Content Management System). If you were only to include the core engine and the few required Core modules to run the product it would simply by a CMS. We bring in modules to introduce other features, which once packed with a heap of them turn the CMS into a Social Networking Solution. With this in mind we leave it up to the client to decide what sort of a site they would like to run and what sort of features they really need to get their crowd interested in their site and what it has to offer. Since the core can be stripped away from modules this leaves a lot of freedom for the script to evolve into other products such as a Software Licensing System or a Bug Tracker.

The modules are based on a MVC (Model View Controller) architectural pattern. phpFox is developed under a OOP (Object-oriented programming) environment, which will give future developers an easier time to enhance the product to their benefits.

Requirements for the product have become more flexible when dealing with PHP related requirements such as safe_mode or open_base_dir. Using those as an example the product can work with whatever those are set to allowing clients not to worry about if their server has a specific setting enabled or not. Our goal is to have a product that is as flexible as possible to make sure it works on all of the popular hosts today. There is one strict requirement and that in order to run the product you must have PHP5. Past requirements have been PHP 4.3.3, however since PHP5 introduced a wave of improvements when it comes down to OOP we decided it was time we move forward. As for MySQL the same 4.1 is required, although phpFox will support other database drivers and not just MySQL.

When reading this manual it is wise to read everything in order and not skip ahead as items that we cover later on will require knowledge of something you have learned earlier in the manual.

Development Environment

While you are developing there are certain constants to make things a lot easier for all of us. Many aspects of the script is cached into flat files. This can be from data we pull out from the database, templates files that are converted into PHP code or JavaScript files that are gzipped. While we are developing we always enable debug mode to make sure we don't have any errors in our code. The constants we are about to show you is a list of constants that do not all have to be set. You can choose which constants you want. If you do not want a certain constant simply remove it from the file. Note then when certain constants are enabled your site will run much slower. You have to consider that the site is not caching any data and this is very risky on a live community. Remember never have any of these settings on a live community.

To enable this settings first create a file:
`include/setting/dev.sett.php`
You can then add the following:

PHP:
```php
<?php

// Enable debug
define('PHPFOX_DEBUG', true);

// Debug level
define('PHPFOX_DEBUG_LEVEL', 1);

// Force templates re-cache on each page refresh
define('PHPFOX_NO_TEMPLATE_CACHE', true);

// Force browsers to re-cache static files on each page refresh
define('PHPFOX_NO_CSS_CACHE', true);

// Override default email
define('PHPFOX_DEFAULT_OUT_EMAIL', 'your_email@email.com');

// Skip sending out of emails
define('PHPFOX_SKIP_MAIL', true);

// Use live templates and not those from the database
define('PHPFOX_LIVE_TEMPLATES', true);

// Add user_name in the title of each page. Great for when working with man
y browsers open
define('PHPFOX_ADD_USER_TITLE', true);

// Cache emails to flat files
define('PHPFOX_CACHE_MAIL', true);

// Log error messages to XML flat file within the cache folder
define('PHPFOX_LOG_ERROR', true);

// Skip the storing of cache files in the DB
define('PHPFOX_CACHE_SKIP_DB_STORE', true);

?>
```

Additionally you can wrap the defines (all in one block if you want) into an if statement so debug mode is enabled only for your IP. You can check what your IP is by going here and the code would look like:

PHP:
```php
<?php
if ($_SERVER['REMOTE_ADDR'] == 'My ip goes here')
{
// Enable debug
define('PHPFOX_DEBUG', true);

// Debug level
define('PHPFOX_DEBUG_LEVEL', 1);
//...
}
?>
```

Creating a Product

When you develop a module or plug-in it must belong to a product. Since you would be releasing your module or plug-in publicly it needs to be part of your unique product. Everything you create within the AdminCP requires that it be part of a product. All the menus, phrases, settings etc... that is provided with the script when you first install it is part of our product, which has the unique ID phpfox. If you add menus or settings for example to your product and once you export your product it will be included in an XML file. From there you can release this as your product for others to install on their site.

Steps to Create a New Product

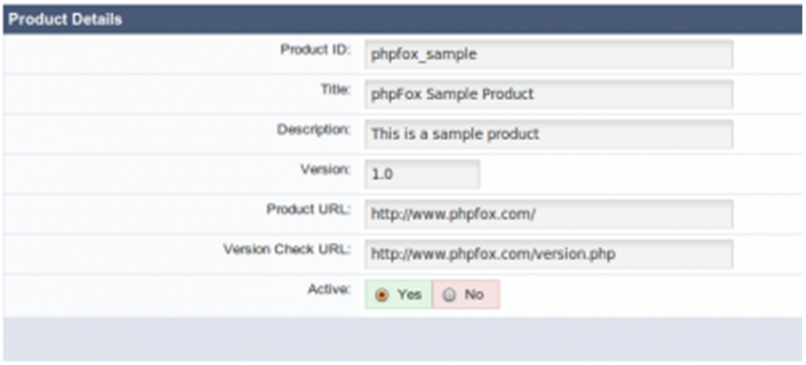
Follow the guide below to create your first product. When creating your product add the values we added as we plan on using our sample product throughout this manual.

Log into your AdminCP and navigate to:

```
Extensions >> Product >> Create New Product
```

Next we need to fill out the form.

Your form should look like the following:



Below we have provided detailed information about each form field.

Product ID

This is the unique product ID you must create. It can have alphanumeric characters and the underscore (_). It must be lowercase. If you plan to release your product publicly it is vital you create a unique ID for your product. If you were to create a product ID that another developer has created this will cause problems for the person installing both products. In order to create a unique ID we ask that you use your company or developer alias as a prefix. For example our product name is**phpfox**. We also include another product **phpfox_installer**, which is used during the install or upgrade of the script. As you can see we used **phpfox** as the prefix.

For this field add the following product ID:

```
phpfox_sample
```

Title

Here you can add the title of your product, which will be displayed when listing all the products.

Add the following:

```
phpFox Sample Product
```

Description

Short description about your product.

Add the following:

```
This is a sample product
```

Version

Current version of your product.

Add the following:
```
1.0
```

Product URL

URL of your website. Usually the company URL or developers home page can be added here.

Add the following:
```
http://www.phpfox.com/
```

Version Check URL

URL of where the phpFox2 should check if a new version of your product has been released.

Add the following:
```
http://www.phpfox.com/version.php
```

Active

Whether the product is active or inactive. Select:
```
Yes
```

Submit the form. Now you have created your first product!

Creating a Module

Introduction in Modules

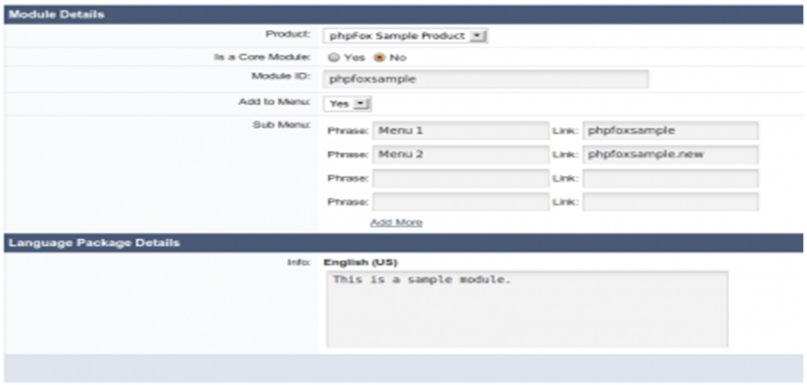
All of the features you see and interact with on phpFox2 is controlled by a module or in many cases several modules. When viewing a blog this is controlled by the **blog** module. There are certain features that are part of the blog module which are a part of other modules such as tagging. Tagging a blog is part of the **tag** module. The reason we split up features into modules is because it allows us to easily remove or build onto the core engine. Since phpFox2 is built as an engine that can support other types of products and not just a Social Networking Solution this will allow users to simply remove or uninstall features they don't like. They can also install a whole different set of modules which will allow them to start a site that is just focused on being a personal blog or web page. Since all our features are split up you will only need to install the required phrases, settings, blocks, menus etc... for that module thus saving the space in the database to not install unnecessary data. There are over 2,000 phrases at the time of this writing and if you were to load the entire language package at once this would cause a major overhead. With the usage of modules it only loads the required phrases for that module and caches it so we don't have to pull such information from the database all the time.

Another advantage is it allows other developers the ability to easily build onto the package without having to modify any of the other existing modules. Especially since the product is equipped with an advanced plug-in system which allows you to virtually access all the methods provided by other modules, developers will no need to modify other modules to accomplish what they need for their specific product.

Steps to Create a Module

To create a new module log into your AdminCP and navigate to:
```
Extension >> Module >> Create New Module
```
Next, we need to fill out the form.

Your form should look like the following:



Below we have provided detailed information about each form field.

Product

Select the product:
```
phpFox Sample Product
```

Is a Core Module

If a module is part of the "core" then the module cannot be deactivated. It is mainly intended for phpFox modules that must always be enabled in order for the product to run. If you are creating a module that can be disabled and the site will still function it does not have to be part of the "core", however if the module you are creating modifies the function of the system and must be on to run the site; then its best to set it up as a "core" module.

Module ID

Unique module ID. Similar to how you create a product ID your module ID must be unique, however your module ID can only contain lowercase characters from a-z. Adding your alias or company name is a good idea. For this example lets add:
```
phpfoxsample
```

Add to Menu

In the AdminCP your module can be added to the Modules menu in case your module has controllers that need access only for Admins. Lets select Yes as we plan on adding some sample menus.

Sub Menu

Since we selected **Yes** for **Add to Menu** we must now add these menus. As you may have noticed we provide by default 4 links for your AdminCP menu. The phrase for a link is found on the left side and the URL connection is on the right. For the first Phrase
we add:
```
Menu 1
```
and for the first Link we add:
```
phpfoxsample
```
For the 2nd Phrase we add:
```
Menu 2
```
and for the 2nd Link we add:
```
phpfoxsample.new
```

Info

Short description about your module. Here we can add:

```
This is a sample module.
```
Submit the form and you should now have created your very first module. Before we go on we need to create the module file structure. Navigate to the folder **/module/** on your server where you have your phpFox2 source files. Within that folder since we used the module name **phpfoxsample** we must create a new folder with the same name. Within the **/module/** create the folder:
```
phpfoxsample
```
Within the new folder you created create the following folder structure:
```
/include/
/include/component/
/include/component/ajax/
/include/component/block/
/include/component/controller/
/include/plugin/
/include/service/
/static/
/static/css/default/default/
/static/image/
/static/jscript/
/template/
/template/default/
/template/default/block/
/template/default/controller/
```

Note that the folder structure we have asked you to create is not really needed for every single add-on, however its important to learn what each folder holds and in the future you only need to create the folder structure based on your add-on.

Now we can move on to creating our first controller.

Creating a Controller - (Hello World!)

Controllers are PHP classes that control all the pages on phpFox. It is the first PHP class we connect to in order to control the page we are viewing. The controller then decides how the page should behave and what sort of information we should display. To start things off we will create a small "Hello World!" page.

First, lets create the PHP class file for our controller. To do that create the following file:
```
/module/phpfoxsample/include/component/controller/index.class.php
```
In that file add the following PHP code:
PHP:
```
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {

    }
}

?>
```

This will contain the PHP code for this specific page. Now lets create the HTML file for our controller, which will hold all the HTML. To do that create the file:
```
/module/phpfoxsample/template/default/controller/index.html.php
```
In that file add:
HTML:
```
Hello World!
```
Your first page is about ready. Now we just need to access via your web browser. Since the name of our recently created module is **phpfoxsample** anytime we want to access a controller in this module we will have to identify the module name in the URL. For example to connect to the index controller of the **phpfoxsample** module we would use the URL:
```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```

If you visit that URL (substituting www.yoursite.com with the path of your site) you should see something similar:

Congratulations! You have created your first controller.

Understanding Controllers in Detail

Before we go on we feel its best to further explain in detail how controllers work.

If you recall the PHP class name of our controller is:
PHP:
```
Phpfoxsample_Component_Controller_Index
```
The underscore (_) signifies a folders slash (/). The class name above means it would connect to the module file:
```
/module/phpfoxsample/component/controller/index.class.php
```
Each controller must extend the parent class:
```
Phpfox_Component
```
In order for us to run a controller we need to provide the public method **process** which is this part:
PHP:
```
public function process()
```

In example so far we have covered how to create an index controller, however what if we wanted to create a new page for our module. In our next example we are going to create the following page:

```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```

In order to create this page you will have to create the file:

```
/module/phpfoxsample/include/component/controller/add.class.php
```

Within that file we add:

PHP:

```php
<?php

class Phpfoxsample_Component_Controller_Add extends Phpfox_Component
{
    public function process()
    {

    }
}

?>
```

Next, we need to create a new template:

```
/module/phpfoxsample/template/default/controller/add.html.php
```

In that file lets add:

HTML:

```
We will be adding a form here...
```

Save that file. The page is complete and you should be able to visit it at:

```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```

You will notice the name of the PHP file we created was **add.class.php**. The suffix for this file is always **.class.php** and the controller name comes before it and is how you connect to a controller via the URL. Since we used "**add**" as our controller name the HTML template file must be named **add.html.php**, where the suffix now would be **.html.php**. In the PHP class file we created we have the following in that file:

PHP:

```php
class Phpfoxsample_Component_Controller_Add extends Phpfox_Component
```

Notice where we have "**Add**" as the ending of the class name.

This should now give you a better idea of how to create and connect to controllers easily.

Working with Controllers

Let us know look into common ways we work with controllers. This mostly deals with working with our core Phpfox_Templateclass, which allows us to set a page title, meta tags as well as assign variables that we can use within the HTML template.

We are going to be working with the page:

```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```

Open that page up on your browser. For this chapter lets open both these files:

```
/module/phpfoxsample/include/component/controller/index.class.php
/module/phpfoxsample/template/default/controller/index.html.php
```

You may recall we created these files earlier in this article and the first file we will refer to as the PHP file and the 2nd will be referred to as the HTML file.

Set a Title & Breadcrumb

Our index controller class should look similar to this:

PHP:
```php
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {

    }
}

?>
```

Since all controllers extends the class Phpfox_Component we can take advantage of all the protected methods it provides like the template() method, which creates an object for the class Phpfox_Template.

Now let us assign a title for this page. Within the **process** method lets add:

PHP:
```php
$this->template()->setTitle('My Sample Title');
```

Visit the page:
```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```
You should now see **My Sample Title** as the title for your page.

Most templates will be designed to have breadcrumb support, in order to add a breadcrumb add the following:

PHP:
```php
$this->template()->setBreadcrumb('My Sample Breadcrumb');
```

Your page should now look similar to the following image:

Set Meta Keywords & Description

To set any sort of meta tags you can use the setMeta() method. Lets add the following within the **process()** method:

PHP:
```php
$this->template()->setMeta('keywords', 'phrase1, phrase2, phrase3');
$this->template()->setMeta('description', 'This is a sample page.');
```

Once you save that and visit the page we are working on and view the HTML source code you will see something like this:

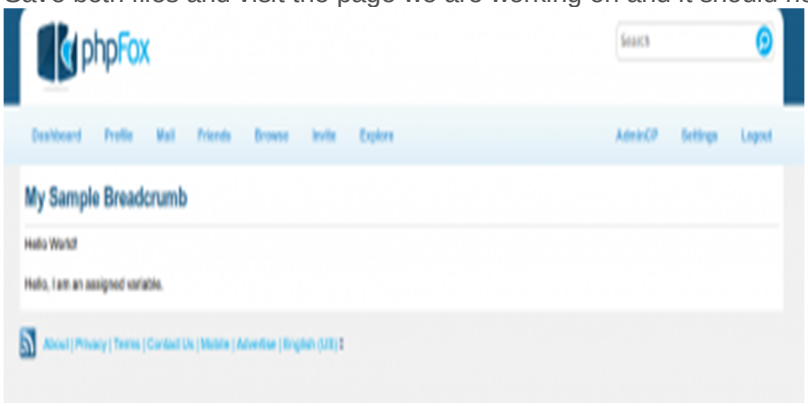
```
<html xmlns="http://www.w3.org/1999/xhtml" dir="ltr" lang="en">
    <head>
        <title>My Sample Title &raquo; phpFox - Social Networking Script</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="generator" content="cGhwRm94" />
        <meta name="version" content="Mi4wLjY=" />
        <meta name="package" content="W1BIUEZPWF9QQUNLQUdFX05BTUVd" />
        <meta name="keywords" content="phrase1, phrase2, phrase3" />
        <meta name="description" content="This is a sample page." />
        <meta name="robots" content="index,follow" />
        <meta http-equiv="imagetoolbar" content="no" />
        <meta http-equiv="cache-control" content="no-cache" />
        <meta http-equiv="expires" content="-1" />
        <meta http-equiv="pragma" content="no-cache" />
```

Assigning Variables to HTML Files

Since our script is based on a MVC (Model-View-Controller) architecture we have separated PHP logic from HTML design. In order to access information created within the PHP file to the HTML file you need to create template variables. To assign template variables we will be using the assign() method. Lets add the following to the PHP file within the **process** method:

PHP:
```
$this->template()->assign('sSampleVariable', 'Hello, I am an assigned variable.');
```
Now lets get the HTML file and put this at the bottom of the file:

HTML:
```
<br />
<br />

{$sSampleVariable}
```
Save both files and visit the page we are working on and it should now look like this:



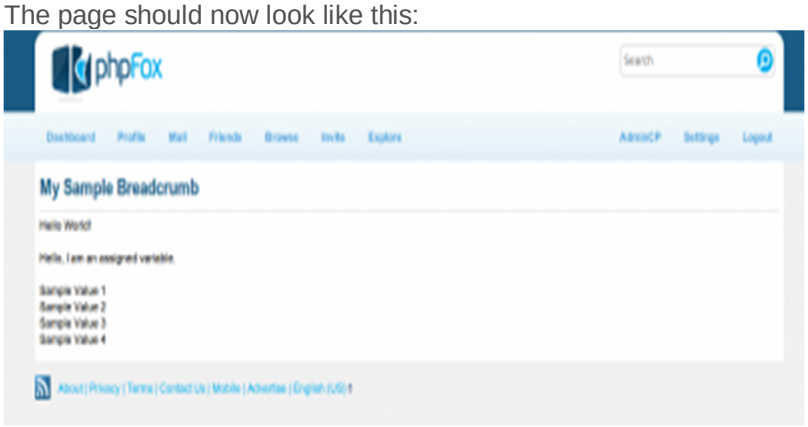
In most cases you will probably assigning more then just one variable and instead of calling the assign() method several times you can put everything into an associative array. So lets add this to our PHP file:

PHP:
```
$this->template()->assign(array(
        'sSampleKey1' => 'Sample Value 1',
        'sSampleKey2' => 'Sample Value 2',
        'sSampleKey3' => 'Sample Value 3',
        'sSampleKey4' => 'Sample Value 4'
    )
);
```
Add the following to the bottom of your HTML file:

HTML:
```
<br />
<br />
```

```
<ul>
        <li>{$sSampleKey1}</li>
        <li>{$sSampleKey2}</li>
        <li>{$sSampleKey3}</li>

        <li>{$sSampleKey4}</li>
</ul>
```

The page should now look like this:



Connect with CSS & JavaScript

The goal with modules is to keep as much of what we use for that module within our root module folder. Each module can connect to CSS or JavaScript files that are part of the core product or part of a specific module.

To include CSS & JavaScript within the templates **<head></head>** we are going to use the setHeader() method. In your PHP file add the following:

PHP:
```
$this->template()->setHeader('sample.css', 'module_phpfoxsample');
```

Before we go on into creating the CSS file lets look at what we have just added. The first argument we have **sample.css** and this is the name of the CSS file we are about to create and from now we will refer to this as the **CSS file**. For the 2nd argument we have **module_phpfoxsample** and the prefix **module_** identifies that we are connecting to a CSS file that is located within a module and in this specific case the module is the suffix, which is **phpfoxsample**. For the controller to successfully connect to this CSS file we need to create the file:

```
/module/phpfoxsample/static/css/default/default/sample.css
```

In that file add:

CSS:
```
body
{
        background:#000;
}
```

Now the page we are working on will have a black background and should look like this:

Now let us look into connecting to a JavaScript file. To save space let us replace what we added earlier in our PHP file:

PHP:
```
$this->template()->setHeader('sample.css', 'module_phpfoxsample');
```
Replace that with:

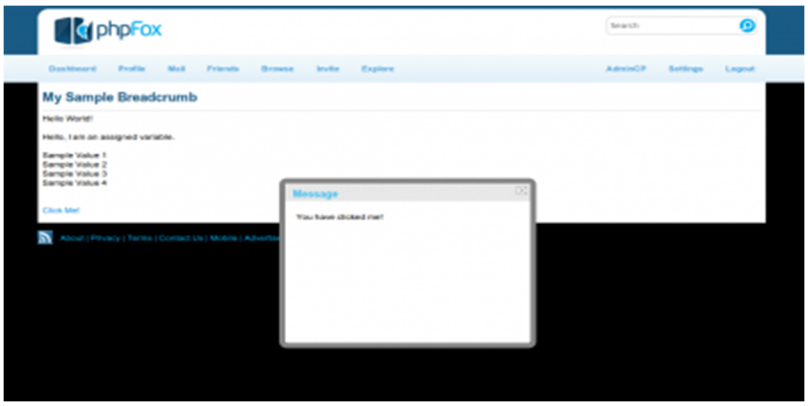
PHP:
```
$this->template()->setHeader(array(
        'sample.css' => 'module_phpfoxsample',
        'sample.js' => 'module_phpfoxsample'
    )
);
```
You will notice that the setHeader() method works similar to that of the assign() method where we can group all the calls into an associative array. Notice in the array we have added:

PHP:
```
'sample.js' => 'module_phpfoxsample'
```
The key is the name of the JavaScript file and the value is the connection to the module it belongs to. We will refer to the file**sample.js** as the **JavaScript file**. Create the JavaScript file:
```
/module/phpfoxsample/static/jscript/sample.js
```

In that file add the following:
```
$(function()
{
        $('#sample_id').click(function()
        {
                alert('You have clicked me!');

                return false;
        });
});
```

In your HTML file at this to the bottom of the file:

HTML:
```
<br />
<br />
<a href="#" id="sample_id">Click Me!</a>
```
Visit the page we are working on and look for and click on the link **Click Me!** and a pop-up should appear.



Note that the default JavaScript library we provide is jQuery, however you can use plain JavaScript.

So far we have looked into connecting to JavaScript files within our own module. Using the setHeader() method you can actually place anything within the HTML **<head></head>** element. In your PHP file add the following:

PHP:
```
$this->template()->setHeader('<!-- Add me in the header -->');
```

Now view the source of the page we are working on and you will find that within the **<head></head>** element.

Method Chaining

At this point your PHP file should look similar to this:

PHP:
```
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {
        $this->template()->setTitle('My Sample Title');
        $this->template()->setBreadcrumb('My Sample Breadcrumb');
        $this->template()->
>setMeta('keywords', 'phrase1, phrase2, phrase3');
        $this->template()->
>setMeta('description', 'This is a sample page.');
        $this->template()->
>assign('sSampleVariable', 'Hello, I am an assigned variable.');
        $this->template()->assign(array(
                'sSampleKey1' => 'Sample Value 1',
                'sSampleKey2' => 'Sample Value 2',
                'sSampleKey3' => 'Sample Value 3',
                'sSampleKey4' => 'Sample Value 4'
            )
        );
        $this->template()->setHeader(array(
                'sample.css' => 'module_phpfoxsample',
                'sample.js' => 'module_phpfoxsample'
            )
        );
        $this->template()->setHeader('<!-- Add me in the header -->');
    }
}

?>
```

Once you start to look around in other areas of the script you will see we combine all methods we are executing into what is commonly referred to as Method Chaining. So another way this PHP file could look is like:

PHP:
```
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {
        $this->template()->setTitle('My Sample Title')
            ->setBreadcrumb('My Sample Breadcrumb')
            ->setMeta('keywords', 'phrase1, phrase2, phrase3')
            ->setMeta('description', 'This is a sample page.')
            -
>assign('sSampleVariable', 'Hello, I am an assigned variable.')
            ->assign(array(
                    'sSampleKey1' => 'Sample Value 1',
                    'sSampleKey2' => 'Sample Value 2',
                    'sSampleKey3' => 'Sample Value 3',
                    'sSampleKey4' => 'Sample Value 4'
                )
            )
            ->setHeader(array(
                    'sample.css' => 'module_phpfoxsample',
                    'sample.js' => 'module_phpfoxsample'
                )
            )
            ->setHeader('<!-- Add me in the header -->');
    }
}

?>
```

Either way is fine, however for now we will be working with the first example.

We have now looked into working with controllers. Next, we are going to be focusing on how we can work with services and controllers.

Creating a Service

A phpFox Service is what we use as a model for our modules. We use services to interact with database tables, cache modular data or extend the PHP logic for a module. It is best not to use a component (controllers or blocks) for such actions as this should all be handled by your module services. To use a module service you need to call it from a module controller.

At this point we are still working with the following page:
`http://www.yoursite.com/index.php?do=/phpfoxsample...`

We are also going to continue working with our PHP & HTML file, which are:
```
/module/phpfoxsample/include/component/controller/index.class.php
/module/phpfoxsample/template/default/controller/index.html.php
```

Open both those files in preparation for this chapter.

At this point your PHP file should look similar to this:

PHP:
```php
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {
        $this->template()->setTitle('My Sample Title');
        $this->template()->setBreadcrumb('My Sample Breadcrumb');
        $this->template()-
>setMeta('keywords', 'phrase1, phrase2, phrase3');
        $this->template()-
>setMeta('description', 'This is a sample page.');
        $this->template()-
>assign('sSampleVariable', 'Hello, I am an assigned variable.');
        $this->template()->assign(array(
                'sSampleKey1' => 'Sample Value 1',
                'sSampleKey2' => 'Sample Value 2',
                'sSampleKey3' => 'Sample Value 3',
                'sSampleKey4' => 'Sample Value 4'
            )
        );
        $this->template()->setHeader(array(
                'sample.css' => 'module_phpfoxsample',
                'sample.js' => 'module_phpfoxsample'
            )
        );
        $this->template()->setHeader('<!-- Add me in the header -->');
    }
}

?>
```

Creating the Service File

Lets create our first service class by creating the PHP file:

`/module/phpfoxsample/include/service/phpfoxsample.class.php`

We are going to refer to this file as the **Service class**. In your newly created service file add the following:

PHP:
```php
<?php

class Phpfoxsample_Service_Phpfoxsample extends Phpfox_Service
{
    public function getUsers($iTotal)
    {
        return $this->database()->select('u.full_name')
            ->from(Phpfox::getT('user'), 'u')
            ->limit($iTotal)
            ->execute('getRows');
    }
}

?>
```

You will notice the name of our class is **Phpfoxsample_Service_Phpfoxsample**, similar to how other classes work we replace the underscore with a folder slash. So the name of a class always

corresponds with the location of the file.

All service classes must extend our Phpfox_Service class.

For our Service class created a method **getUsers()**. This method is just an example and simply gets a specific number of members from the user database table. The number is specified by the variable **$iTotal**, which is defined when calling this specific method. In order for us to use this class and call this method we need to do this from a module controller. Open your PHP file and add the following within the **process()** method:

PHP:
```
$aUsers = Phpfox::getService('phpfoxsample')->getUsers(10);
$this->template()->assign('aUsers', $aUsers);
```

Before we go on and output this information in the HTML file let us look over how we called the **getUsers()** method. To call a Service class we use the static method getService() and this takes care of requiring the file, creating the object and then calling the method. So the conventional method would be something like:

PHP:
```
require('/module/phpfoxsample/include/service/phpfoxsample.class.php');
$oObject = new Phpfoxsample_Service_Phpfoxsample();
$oObject->getUsers(10);
```

Just a quick note before we go on. In the example we have provided we called the **getUsers()** method like:

PHP:
```
$aUsers = Phpfox::getService('phpfoxsample')->getUsers(10);
```

You may need to call more methods with the same service class and you may want to create a variable to hold the object for the Service class. So another way to accomplish this would be like:

PHP:
```
$oObject = Phpfox::getService('phpfoxsample');
$aUsers = $oObject->getUsers(10);
```

Now that we have looked into how to connect to a Service class let us get back to our PHP file where we added the following to the file earlier:

PHP:
```
$aUsers = Phpfox::getService('phpfoxsample')->getUsers(10);
$this->template()->assign('aUsers', $aUsers);
```

Based on the code we added above we created a template variable **aUsers** that we can now use in our HTML file. Save your PHP file and open your HTML file and add the following to the bottom of the file:

HTML:
```
<br />
<br />
<b>Members:</b>
<ul class="p_4" style="list-style-type:square; margin:0px 0px 0px 15px;">
{foreach from=$aUsers item=aUser}
        <li>{$aUser.full_name}</li>
{/foreach}
</ul>
```

Save that file and visit the page we are working on. Near the bottom of the page you should find a list of members. It should look similar to this:

Creating a Block

Blocks are used to expand controllers and easily allows the ability to interact with other modules. Later in this article we will look over the importance of block placements and how to easily add them direct from your AdminCP, however for now we will focus on connecting to blocks via HTML controller templates.

Let us create our Block file by creating the file:
`/module/phpfoxsample/include/component/block/display.class.php`
In that file add:

PHP:
```php
<?php

class Phpfoxsample_Component_Block_Display extends Phpfox_Component
{
    public function process()
    {

    }
}

?>
```

Similar to our controller class the block class has the same layout. The main difference is that since this class is within the**block** folder the class name uses _**Block**_ instead of _**Controller**_.

Much like a controller a block needs a HTML template file. So let's create the file:
`/module/phpfoxsample/template/default/block/display.html.php`
In that file add:

HTML:
```
<br />
<br />
Hello I am a block...
```

Next we need to connect this block with one of our controllers. Let us open the controller template we created earlier:
`/module/phpfoxsample/template/default/controller/index.html.php`
At the bottom of the file add the following:

HTML:

```
{module name='phpfoxsample.display'}
```
Notice the value **phpfoxsample.display**. This is the connection to the block. This value is translated to:
```
/module/phpfoxsample/include/component/block/display.class.php
```
Save the files and look over at the page we are working on. At the bottom of the page you should see where we added "**Hello I am a block...**".

Since blocks also extend the class Phpfox_Component you can also use methods that belong to Phpfox_Template much like controllers. However, you cannot set the pages title, breadcrumb, meta tags or anything related to be placed within the pages**<head>** element since that is the job of the controller. You can use the assign() method to assign variables for your HTML Block file.

Controllers are considered parents of blocks and in the example we have showed you so far we have covered how to use blocks to interact with other module controllers and to also save on repeating the same code over and over again.

In our next chapter we are going to show you how to easily place blocks anywhere on your site direct from your AdminCP

Registering Blocks with Controllers

So far we have looked into how to work with controllers, blocks and services. In this chapter we are going to connect blocks to controllers and not by adding a call in the controller template file like we showed you in the last chapter, instead we are going to be placing it from your AdminCP. By registering and placing blocks via the AdminCP this gives you and others the ability to connect blocks with virtually any controller and place it in up to 10 locations (depending on the theme installed).

In this chapter we are still going to be working with the following page:
```
http://www.yoursite.com/index.php?do=/phpfoxsample...
```

We are also going to continue working with our PHP file, which is:
```
/module/phpfoxsample/include/component/controller/index.class.php
```
Just to make sure we are on the same page your PHP file should look similar to this:

PHP:
```php
<?php

class Phpfoxsample_Component_Controller_Index extends Phpfox_Component
{
    public function process()
    {
        $this->template()->setTitle('My Sample Title');
        $this->template()->setBreadcrumb('My Sample Breadcrumb');
        $this->template()->
>setMeta('keywords', 'phrase1, phrase2, phrase3');
        $this->template()->
>setMeta('description', 'This is a sample page.');
        $this->template()->
>assign('sSampleVariable', 'Hello, I am an assigned variable.');
        $this->template()->assign(array(
                'sSampleKey1' => 'Sample Value 1',
                'sSampleKey2' => 'Sample Value 2',
                'sSampleKey3' => 'Sample Value 3',
```

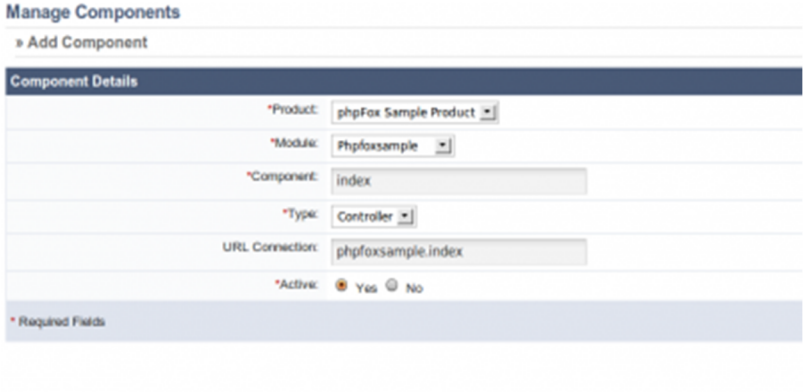
```
                'sSampleKey4' => 'Sample Value 4'
            )
        );
        $this->template()->setHeader(array(
                'sample.css' => 'module_phpfoxsample',
                'sample.js' => 'module_phpfoxsample'
            )
        );
        $this->template()->setHeader('<!-- Add me in the header -->');
        $aUsers = Phpfox::getService('phpfoxsample')->getUsers(10);
        $this->template()->assign('aUsers', $aUsers);
    }
}

?>
```

Registering a Controller

Before we can connect a block to a controller we have worked on we must register it in the AdminCP. Log into your AdminCP and go to:

```
Extensions >> Module >> Add Component
```

Now we need to fill out the form. Here is how the form should look:



Here is a summary of what each field means:

- **Product** - *Name of your product.*
- **Module** - *Name of the module this controller is a part of.*

- **Component** - *Name of the component. This is the name of the PHP file without the suffix (.class.php).*
- **Type** - *Identifies if this is a block or controller.*
- **URL Connection** - *Assign how we are going to connect to this via a web browser. It is always**module_name.controller_file_name***
- **Active** - *Controls if the controller or block is active.*

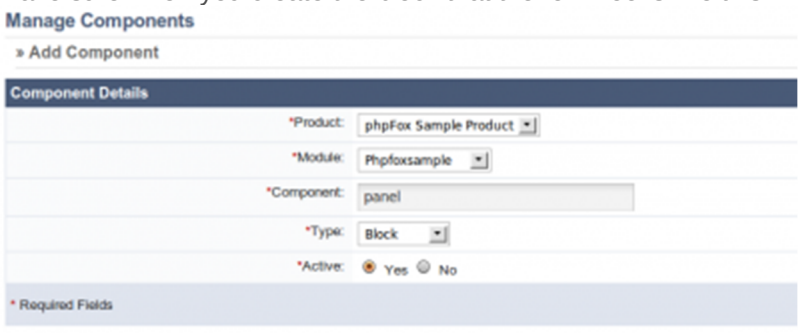
Once you have submitted and successfully added your controller you can now connect any blocks direct from your AdminCP.

Registering a Block

At this point you can connect any registered block to your controller, however we want to work with our own block so we will need to register our block as well. This works very similar to how we register controllers. Before we do that let's create a new block file so we don't work with the file we worked with earlier as that is already connected to our index controller via the HTML template file.

Create the following block file:
`/module/phpfoxsample/include/component/block/panel.class.php`

In that file add the following:
PHP:
```
<?php

class Phpfoxsample_Component_Block_Panel extends Phpfox_Component
{
    public function process()
    {
        $this->template()->assign(array(
                'sHeader' => 'Panel Block',
                'aFooter' => array(
                    'Panel Link' => $this->url()-
>makeUrl('phpfoxsample.add')
                ),
            )
        );

        return 'block';
    }
}

?>
```
Now we have the PHP block file. Now we need to create the HTML block file:
`/module/phpfoxsample/template/default/block/panel.html.php`
In that file add:
HTML:
```
I am a panel block...
```
We are going to shortly go over what we just added in these files, however first let us focus on registering our new block. Log into your AdminCP and go to:
`Extensions >> Module >> Add Component`
Make sure when you create the block that the form looks like this:



You will notice the key difference from creating a block from a controller is we identify the **Type** to be set to **Block**.

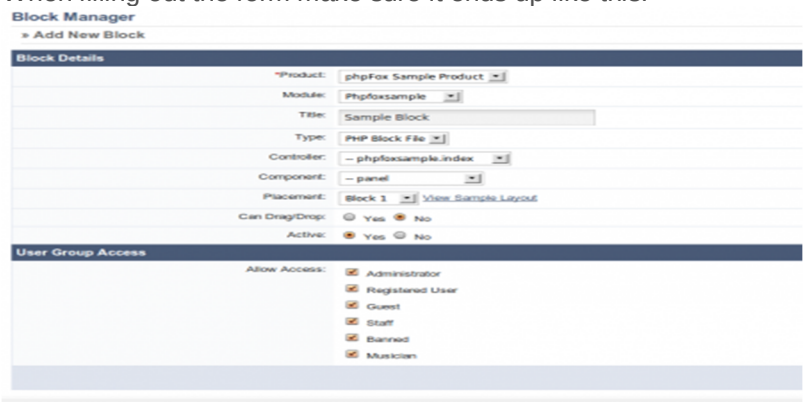
Submit the form and now you have registered your block.

Connecting a Block to a Controller

Now that we have registered our controller and block we can now connect them by adding a CMS (Content Management System) block. Log into your AdminCP and go to:

```
CMS >> Blocks >> Add New Block
```

When filling out the form make sure it ends up like this:



Here is a summary of what each field means:

- **Product** - *Name of your product.*
- **Module** - *Name of the module this controller is a part of.*
- **Title** - *Title of this block. It can be anything that will identify what this block does.*

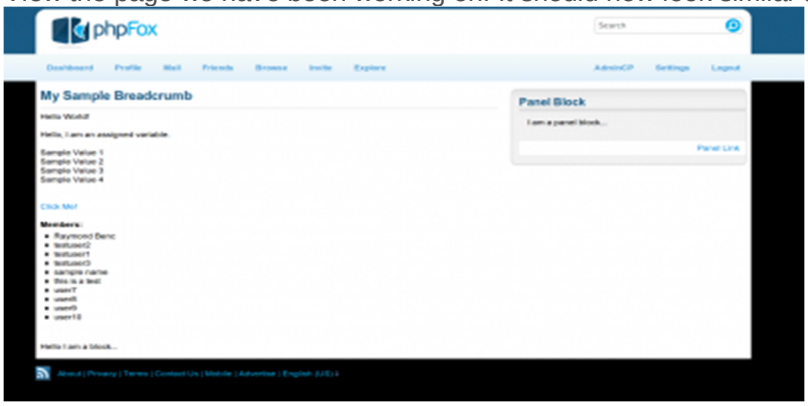
- **Type** - *Identifies what type of block this is. In this case we went with **PHP Block File**.*
- **Controller** - *The controller we are going to connect our block to.*
- **Component** - *The block we are going to connect.*

- **Placement** - *Where we plan to position the block. If you click on **View Sample Layout** you will see there are 10 spots to choose from.*
- **Can Drag/Drop** - *Controls if the block can be dragged and dropped on a page that has support for such a feature.*
- **Active** - *Controls if the block is live or not.*

- **Allow Access** - *Controls what user groups can view the block.*

Once you submit the form you will have successfully connected your first block with a controller.

View the page we have been working on. It should now look similar to this:



Understanding the Block File

Let us look over the block file we created earlier that is now in charge of displaying the block on our index controller. Open the file:

```
/module/phpfoxsample/include/component/block/panel.class.php
```

In the file you will notice we used the assign() method to assign certain variables for the HTML template. The first variable we assigned is:

PHP:

```
'sHeader' => 'Panel Block'
```

The variable **sHeader** controls the title of the block. If you take a quick look on the page we are working on at the top of your newly created block you will find the phrase **Panel Block**.

Next, we have the variable **aFooter** and this controls the menu found on the bottom of your block.

Open the HTML file for this block:

```
/module/phpfoxsample/template/default/block/panel.html.php
```

You will notice in that HTML file all we have is the following code:

HTML:

```
I am a panel block...
```

The reason there is HTML that comes before and after what is currently in your block file is because of the **return** we have in the PHP block file:

PHP:

```
return 'block';
```

If in a PHP block file you return a string it must be the name of the template that will act as the outer shell of the block. In this case we returned the string **block**. That means the HTML template that will act as a shell of this block is the template file:

```
/theme/frontend/default/template/block.html.php
```

In that file you will notice all the other HTML that is used to build up the entire block. The part where your newly created block fits in is where we have:

HTML:

```
{layout_content}
```

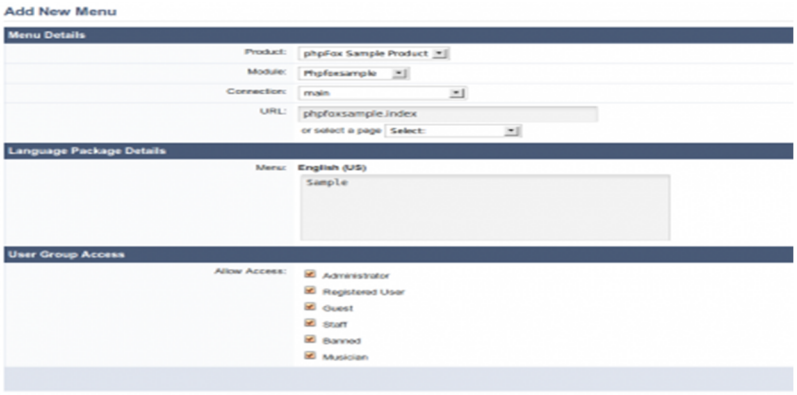
The purpose of the shell template file is so if we want to make a change in the future with how a block looks or if a designer wants the block to look different they do not have to worry about developers and their block files as the shell controls how the block will look and not how the block itself will behave. That and we won't have to update all the blocks just to match our new design or your new design.

Adding a Menu to a Controller

Now that we have created our controller let's make things easier by creating a menu so we can easily reach the page in the future. To do that log into your AdminCP and go to:

```
CMS >> Menus >> Add New Menu
```

When you fill out the form make sure it looks like this:

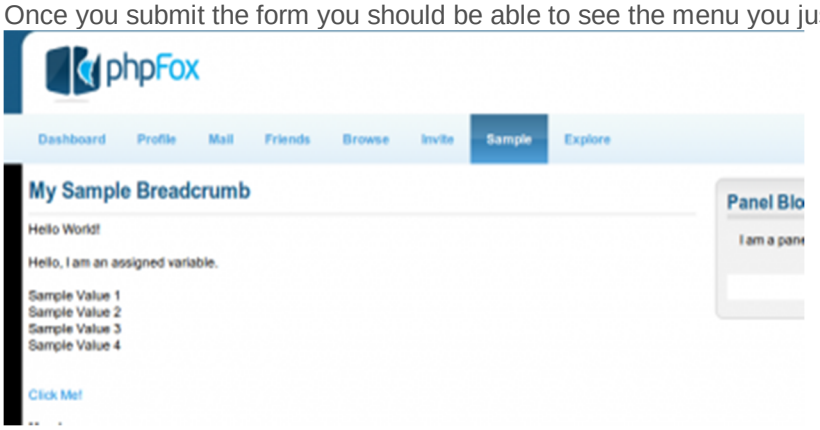


Here is a summary of what each field means:

- **Product** - *Name of your product.*
- **Module** - *Name of the module this controller is a part of.*
- **Connection** - *Where we are going to place the menu. In this case we are placing it in the **main** menu bar.*

- **URL** - *Link to the menu. The period in the link represents a forward slash "/".*
- **Menu** - *Name of the menu.*
- **Allow Access** - *Defines which user group can see the menu.*

Once you submit the form you should be able to see the menu you just added in the main menu bar.



Plugins

Plugins are used to interact with the core product, modules or other 3rd party plugins without the need to modify the source code as it simply plugs into 3rd party code. When building an application it is important to keep as much as you can within your own products modules, however in order for an application to fully extend the product the usage of plugin hooks are vital. Before we go into how to create a plugin let us look into how a plugin hook looks in the code. Hooks can be found in PHP and HTML files. First, open the file:

```
/module/blog/include/service/process.class.php
```

Look for:

PHP:

```php
(($sPlugin = Phpfox_Plugin::get('blog.service_process__start')) ? eval($sPlugin) : false);
```

This is how a hook call looks in PHP code. The name of this hook would be:

```
blog.service_process__start
```

Next, let's open the HTML file:

```
/module/blog/template/default/controller/add.html.php
```

Look for:

HTML:

```
{plugin call='blog.template_controller_add_hidden_form'}
```

The name of this hook would be:

```
blog.template_controller_add_hidden_form
```

Let's now look into how to create plugins. We are going to go into how to create the same plugin in 2 different ways. Creating the plugin either way is fine and is really up to you as a developer. Open the file:

```
/module/profile/template/default/block/info.html.php
```

Look for:

HTML:

```
{plugin call='profile.template_block_info'}
```

The name of this hook is:

```
profile.template_block_info
```

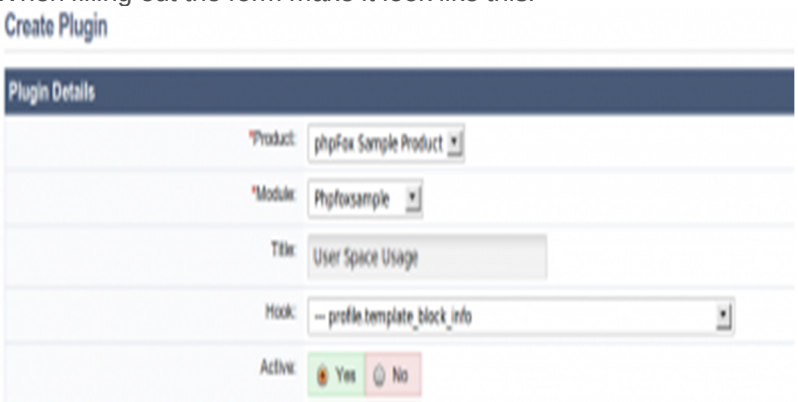
Continue below to complete the plugin.

Creating a Plugin from the AdminCP

To create a plugin from the AdminCP log into your AdminCP and go to:

```
Extensions >> Plugin >> Create New Plugin
```

When filling out the form make it look like this:



For the field **PHP Code** add the following:

PHP:

```php
$aUser = $this->getVar('aUser');
echo '<div class="info">';
```

```
echo '<div class="info_left">Space Usage:</div><div class="info_right">' .
Phpfox::getLib('phpfox.file')->filesize($aUser['space_total']) . '</div>';
echo '</div>';
```

Save the form and visit your profile. In the **Basic Info** block you should find the **Space Usage** information. Here is an example of how it could look:

Creating a Plugin from a PHP File

We are now going to create a plugin very similar to the one we just did when creating it from the AdminCP earlier. Create the file:

```
/module/phpfoxsample/include/plugin/profile.template_block_info.php
```

The reason the name of the PHP file **profile.template_block_info.php** is because the name of the hook we are working with is **profile.template_block_info**. So all we do is add the PHP file extension after the name of the hook. In that file add:

PHP:

```
<?php
$aUser = $this->getVar('aUser');
echo '<div class="info">';
echo '<div class="info_left">Space Usage PHP:</div><div class="info_right">
' . Phpfox::getLib('phpfox.file')-
>filesize($aUser['space_total']) . '</div>';
echo '</div>';
?>
```

No log into your AdminCP and go to:

```
Tools >> Maintenance >> Cache Manager
```

Click on the **Clear All**. Note that plugins are cached and when working with plugins from the source code we need to clear the sites cache. Alternatively in the future you can enable a developers setting by going to:

```
Settings >> System Settings >> Manage Settings >> Development
```

Look for the setting **Cache Plugins** and disable it. This way you will not have to clear your sites cache every time you make an update to a plugin. Just make sure to enable this on your live site and that this does not need to be done if you are working with plugins from your AdminCP.

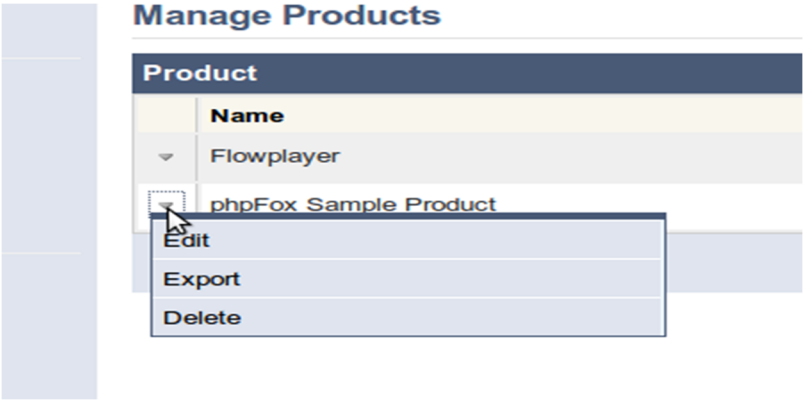
Now that the sites cache has been cleared visit your profile and you should see the phrase **Space Usage PHP** within the**Basic Info** block.

Product Complete - Ready for Export

Congratulations! You have completed your first add-on that is complete with modules, controllers, blocks, menus, plugins and services. If you plan on releasing your product to others you can easily export it from your AdminCP. To do this go to:

```
Extension >> Product >> Import/Export Products
```

Look for the product **phpFox Sample Product** and from the drop down click on **Export**.



It will pack everything into a ZIP archive so you or your clients can download and install your product easily from the AdminCP.

ZIP Archive

Now that your product is completed and if you downloaded the ZIP archive you will notice the name of the file is something like**phpfox-product-phpfox_sample-1.0.zip**. It is important not to change this name as our product is designed to install a product by verifying the name of the ZIP archive. Unzip the archive and you will find that we packed in anything related to your product including the PHP source files. When you or your clients install your product we use the XML information found in the file:

```
/upload/include/xml/phpfox_sample.xml
```

What's Next?

With this article we have gone over the basics on how to create and work with phpFox. There is a lot more that can be done with phpFox and to learn more tips and tricks be sure to keep track of articles in our Developers Documentation section.