PostNuke Module Developers Guide

Marco Canini

Jim McDonald

Gregor J. Rothfuss

PostNuke Module Developers Guide

by Marco Canini, Jim McDonald, and Gregor J. Rothfuss

V2.4 Edition
Published 12nd May 2002

Revision History

Revision 2.4 11rd May 2002 Revised by: mc
Described error and exception handling
Revision 2.3 3rd May 2002 Revised by: mc
Extended Variable validation
Revision 2.2 28th April 2002 Revised by: mc
Added section variable validation
Revision 2.1 27th April 2002 Revised by: mcgjr
Added section on user variables

Table of Contents

Chapter 1. Introduction

1.1. What is a Module?

The PostNuke system allows for expansion of its functionality through the use of modules. A module is set of files containing functions with pre-defined names and roles that integrate very easily with a standard deployment of PostNuke. A module can also include blocks, images, plain HTML files, *etc*.

1.2. Why Write a Module?

There are a number of reasons to write a module. The main reason is because PostNuke does not provide a specific function that you would like it to. Examples of modules that have been developed to date for PostNuke include bulletin boards, galleries, calendars, address books, and MP3 search utilities.

1.3. Status of the Module API

The PostNuke module API is currently in beta. This means that the module functionality may well be increased prior to its first official release. However, creating a module as outlined in this document will work, and continue to work as described for the foreseeable future. Any future versions of the MDG will note where areas and functions have been superceded or deprecated, and developers will have at least 6 months between any major changes in the module design being implemented and backwards compatibility being removed from the core, allowing suitable time for migration.

1.4. On-Going Work

PostNuke is an alpha product and very much a work in progress. There are a number of areas that are currently still under redesign, and when the redesign and resultant new code is in place module developers will need to change their code to be able to support the latest functions. Any change of these areas will have at least one full release cycle where both old and new style code is supported so the transition period will always be a matter of months. All efforts will be made to keep the changes as simple as possible.

Areas that are still defined as to be upgraded before the 1.0 release are as follows:

- The multilingual system. The current multilingual system uses defines, which does not lend itself to high levels of flexibility. This will be solved through some sort of functional interface into the multilingual system, although the details are yet to be defined. The change should involve developers replacing their current _LANGUAGE defines with function calls but should be a relatively simple search and replace operation without changing any logic within the modules.

  The theming/templating system. The theming and templating systems within PostNuke are currently very basic and nonexistent, respectively. A number of PostNuke developers are looking at a much more flexible solution to display content. The change will probably involve developers reworking the display part of their modules, although it is hoped that one of the advantages of the new system is that it will make the entire work of displaying content to users less of a developer task and hence the re-worked code will be a lot simpler than the current display functions.

The hooks system. The hooks system is very new to PostNuke, and it is already recognised that it will require some extra work. The most obvious addition is to allow hooks access to module content, but others might be required as well. This should require minimal changes to development modules, but developers should be aware of this up-coming change both when using and coding hooks.

1.5. This Document

This document gives a step-by-step guide to creating a module for the PostNuke system. It covers all stages from the initial design and registering yourself with the PostNuke community to deploying, certifying, and upgrading your module.

1.6. Related Documents

Other documents that might be of use in conjunction with this guide are the API Reference Command Reference, the Theme Development guide, and the Output Functions Guide. Note that the Theme and Output guide remain unwritten at this point.

1.7. Suggestions and Updates

The PostNuke module system is a work-in-progress. There are no doubt many good ideas out there that have not been incorporated into the PostNuke module system, and if a developer has a request for a particular set of functionality then they can submit it to the PostNuke features request list on SourceForge at the PostNuke Homepage (http://sourceforge.net/projects/post-nuke). If you have found a bug within the current module system then you can submit it to the bug list at the same address.

Please note that the main requirement for the PostNuke module design is stability. Due to this it is possible that your request for new or updated functionality will get refused on the grounds that it is too specific, can easily be built from core API functions, or carries out work that should rightly be done by a module. In such situations the PostNuke team will always try to provide a simple alternative, but please remember that submission of a new or updated addition to the module design does not guarantee inclusion.

Chapter 2. PostNuke Architecture

This chapter describes the basic architecture of PostNuke, explains the major parts, and contains information on the design choices made for the system.

2.1. Variable handling

TBD: Accepting arguments [$args Vs. pnVarCleanFromInput()] - priority etc. Why you HAVE to pass all variables through pnVarCleanFromInput etc Variable scope, what to store in session vars

2.2. User variables

An user variable is an entity identified by a name that stores a value owned by exactly one user. PostNuke offers two API functions to manipulate user variables, they are `pnUserGetVar()` and `pnUserSetVar()`. The purpose of `pnUserGetVar()` is to allow read access to one user variable. In contrast to that, `pnUserSetVar()` allows write access to one user variable. The *`$name`* parameter is checked against metadata to make sure the variable is registered. PostNuke keeps some metadata about every user variable, so you can't access the *`$name`* user variable if its metadata is not registered. A module can register a new user variable by providing its metadata only if it has the right permissions (permissions are checked by the registration function). Usually the registration process should take place at initialization time for a module that wants to use the *`$name`* user variable during its life cycle.

PostNuke doesn't impose any restriction on the value of *`$name`* except for duplicate and reserved names. As of this writing, the list of reserved names consists of

uid

: The user id.

name

: The user name (full spelling).

uname

: The user name (short form, 'nick').

email

: The email address of the user.

url

: The user url.

status

: The status of the user (active, inactive, deleted etc).

auth_module

: The autentication module that was last used for this user.

You are advised (even for performance reasons) to use the following naming convention: *$name* := $module_name . '_' . $real_name

To register the *$name* user variable you have to use the module API function `register_user_var()` exported by the Modules module. Here is an example:

```
$module_name = 'MyModule';
$variable_name = 'MaxLinesPerPage';
$metadata['label'] = $module_name . '_' . $variable_name;
$metadata['dtype'] = _UDCONST_INTEGER; //one of the values defined for dynamic user data variable ty
$metadata['default'] = 20;
$metadata['validation'] = 'num:>=:10&num:<=:100';

pnModAPILoad('Modules', 'admin');

$result = pnModAPIFunc('Modules', 'admin', 'register_user_var', $metadata);
if (!isset($result)) {
  // pnModAPIFunc() failed
} elseif ($result == false)
  // registration failed
} else {
  // registration succeeded
}
```

As you can see in this example, a descriptive array for the new user variable is created first, and later register_user_var is called with that array as parameter. Meaningful keys for the array are: label, dtype, default and validation. The *label* field is mandatory; it specifies the user variable name as you'll refer later in `pnUserGetVar()` and `pnUserSetVar()` *$name* parameter. The *dtype* field is mandatory; it can take one of the following values: _UDCONST_STRING, _UDCONST_TEXT, _UDCONST_FLOAT, _UDCONST_INTEGER. You should obviously choose the right value for the data type that the new user variable will contain. The *default* field is optional; it's used when the user has not yet set a value for the new user variable. The *validation* field is optional; refer to the next section to get an overview of variable validation. To unregister an user variable you have to call the `unregister_user_var()`, which is located in the users module admin API. You should call that API only at uninstallation time for your modules. Keep in mind that by calling `unregister_user_var()` all the existing values for that user variable will be deleted from user data.

As described in this document, PostNuke offers support for module variables too. If you get confused from that, and can't see the distinction between these different things, here is a little explanation to cover that issue. Module variables are system-wide variables, shared between each module user, like configuration variables. They are not owned by any particular user, and even if they are often protected by permissions for write access, they are typically administrative-side variables. You are encouraged to use them when you have a need to give administrators the possibility to choose some behaviours of your module. But when those behaviours are more related to user preferences you should avoid using module variables and register a new user variable to be used in your code. As example you can consider the above code listing, where a new user variable is registered to allow every single module user to choose his own MaxLinePerPage setting. Now it's reasonable to have done this choice, but here we could have choosen a unique shared module var as well. On the other hand there are some cases in wich you don't have this kind of freedom, for example consider the authldap module. It needs to access a LDAP server, so it needs a variable that contains the LDAP server hostname. Obviously this variable should be a module variable, and access to it should be granted only to administrators with the right permissions. We invite you to ponder this issue for a while before you settle on module vars or user vars.

2.3. Variable validation

PostNuke includes a transparent mechanism for variable validation. It's currently used by two API functions: `pnUserValidateVar()` and `pnUserSetVar()`. The validation works thanks to the Dynamic User Data architecture. As you saw in the above section, with Dynamic User Data you can register new user variables simply by providing its metadata. That metadata can also contain a special field denoted by the validation key. A powerful syntax has been invented for this field. All what you need to do is to follow the right syntax and write your own validator(s), later you register it with user variable metadata and now you can get rid of validation in your module functions. Simply when you call `pnUserSetVar()`, PostNuke will automatically apply your validator(s) and if the check fails you will be notified of that by the return value. To compensate for all this loss of control on validation, a new API function has been created. You can validate an user variable value with the `pnUserValidateVar()` function. That gives you the possibility to first validate all variables from user input and second update them if all validation checks have succeeded.

Here is the grammar for the validation string:

```
validation_string := validator_list
validator_list := validator [ + '&' + validator_list ]
validator := ['!' +] type + ':' + operator + ':' + param
```

Reserved characters to be escaped with a preceeding '\' are: ':' and '&'

`type` can be one of these values: 'num', 'string', 'stringlen', 'func'

`operator` is type-sensitive:

valid operators for *`num`* type are: ==, !=, <, >, <=, >=

valid operators for *`string`* type are: is, contains, starts, ends, regex

valid operators for *`stringlen`* are the same as *`num`* type.

there's only one valid operator for *`func`* type: it's a string composed from ModName + ',' + FuncName. FuncName MUST be exported as an user API function from ModName module.

`param` is the second parameter to be used with operator, except for the *`func`* type: here param is the second parameter that will be passed to FuncName function.

You can create complex validators simply by concatenating them with the logic & (AND) operator.

Here are some examples:

```
// validation string = "string:starts:foo bar"
// validation will succeed
pnUserSetVar("myVar", "foo bar is better than bar foo");
// validation will fail
pnUserSetVar("myVar", "bar foo is ugly");

// validation string = "string:starts:foo\\: bar&stringlen:<=:16"
// NOTE: if you need to use the ':' character you have to
//       escape it with a preceding '\'
// validation will succeed
pnUserSetVar("myVar", "foo: bar is good");
// validation will fail, the string is too long
pnUserSetVar("myVar", "foo: bar is better");
```

```
// validation string = "!string:regex:/(censored1|censored2)/"
// NOTE: the negation operator before the string type
// validation will succeed
pnUserSetVar("myVar", "i’m a good boy, i’m not posting something bad");
// validation will fail
pnUserSetVar("myVar", "i’m a bad boy, you are a censored1");

// validation string = "num:>=:1&num:<=:10"
// validation will succeed
pnUserSetVar("myVar", "5");
// validation will fail
pnUserSetVar("myVar", "12");

// validation string = "func:MyModule,MyFunc:none"
// IMPORTANT: if your validation function works only with the
//            variable value you must specify that param has not
//            to be passed to function.
//            You achieve that by simply setting it to ’none’
// validation will succeed
pnUserSetVar("myVar", "Homer Simpson");
// validation will fail
pnUserSetVar("myVar", "Marco Canini");

// MyModule user API

function MyModule_userapi_MyFunc($args)
{
    extract($args);  // $value
    $ssconn = StarShip::openConnection();
    return !$ssconn->isAlienLifeForm($value);
}
```

2.4. Error handling

PostNuke is capable of error handling through a powerful exception handling system. Since the PHP language doesn’t support language-level exceptions, PostNuke provides an artificial mechanism to deal with exceptions. PostNuke divides exceptions into two types: system exceptions and user exceptions. System exceptions are used by PostNuke API functions, but you can use them if it’s meaningful in that such situation; for example consider the DATABASE_ERROR exception, you are strongly encouraged to use this exception when a database error occurs and not to use your own exception. As another example consider the BAD_PARAM exception, you should choose to use that exception in your module functions and API functions when passed parameters are considered wrong. Finally system exceptions are well known exceptions for which PostNuke can undertake particular actions like logging or emailing, on the other and user exceptions are not known by PostNuke, and since they are indistinguishable, PostNuke will treat them as they were all the same thing. Another good point in distinction between system and user exceptions is the fact that you should not leave uncaught user exceptions as you can do for system exceptions. Hence you should catch all user exceptions instead of throwing back them to PostNuke, this because user exceptions can be seen as soft exceptions, so you could be in the position of doing other actions and/or returning a properly formatted

error message that will look better than the default PostNuke exception caught error message. However it's not illegal to throw back user exceptions to PostNuke, so fell free to do that if it's the case. On the other hand you should avoid to catch system exceptions, except particular cases. A system exception is an hard exception, this means that something very wrong happened and PostNuke should be noticed of that. You achieve this simply by throwing back system exceptions. Also here there are particular circumstances in which you could and perhaps should catch system exceptions. For example consider the `pnUserGetVar()` API function: it raises a NO_PERMISSION system exception in the case you don't have right permission, however you weren't in the position to get access level for user variables, so it's perfectly acceptable here to catch this exception and go ahead when it's meaningful to go ahead.

Now it's the moment to explore how PostNuke permits to deal with exceptions. Here we begin by exposing how to catch exceptions. When a function, that pontetially can raise exceptions, outcomes with a void value you MUST check if some exception was raised. You can do that by calling the `pnExceptionMajor()` function and comparing its return value with the PN_NO_EXCEPTION constant. If they are different you know that an exception was raised. The `pnExceptioMajor()` return value can assume one of these values: PN_NO_EXCEPTION, PN_USER_EXCEPTION, PN_SYSTEM_EXCEPTION. Obviously the value PN_NO_EXCEPTION indicates that no exception was raised, and PN_USER_EXCEPTION stays for user exception was raised and PN_SYSTEM_EXCEPTION stays for system exception was raised. When you see that an exception was raised you have two possibilities: throw it back or handle it. To throw back an exception you have only to return with a void value. To handle an exception you have to check for the exception type, id and value if one.

Consider the following example:

```
$res = pnModFunc('MyModule', 'user', 'MyFunc');
if (!isset($res) && pnExceptionMajor() != PN_NO_EXCEPTION) {
    // Got an exception
    if (pnExcepionMajor() == PN_SYSTEM_EXCEPTION) {
        return; // throw back
    }
    // Got a user exception
    if (pnExceptionId() == 'MyException1') {
        $value = pnExceptionValue();
        $output->Text("Syntax error at line: ".$value->lineNumber);
    } elseif (pnExceptionId() == 'MyException2') {
        /* Do something useful */
    } else { // MyException3
        /* Do something useful */
    }
    // reset exception status
    // NOTE: it's of vital importance to call this function
    //       before returning
    pnExceptionFree();
    return $output->GetOutput();
}
```

To throw exception you use a unique function: `pnExceptionSet()`. You simply call it by passing the exception major, id and value if one; and after this call you return void.

Consider the following example:

```
class MyException1
{
    var $lineNumber;
```

```
}

/* ... */

MyModule_user_MyFunc()
{
    /* ... */
    if ($syntax == false) {
        // Syntax error
        $exc = new MyException1;
        $exc->lineNumber = $line;
        pnExceptionSet(PN_USER_EXCEPTION, 'MyException1', $exc);return;
    }
    /* ... */
    pnExceptionSet(PN_USER_EXCEPTION, 'MyException2');
    /* ... */
    pnExceptionSet(PN_USER_EXCEPTION, 'MyException3');
    /* ... */
    return true;
}
```

Note that no value is associated to MyException2 and MyException3, so there is no need to create a class for exception value. As you can see exception handling is very powerful but also boring and tedious. However you can always choose to not use user exceptions and always throw back system exceptions. But keep in mind that good error handling is not something that should be left for last. It should be part of the development process. Note that is wrong to not check exception status after a call to a function that can pontentially raise something. And note also that if you choose to handle one or more exceptions you MUST call `pnExceptionFree()` before exiting, otherwise the trust relationship on which the exception handling mechanism is based won't work and you will produce very bad things. An ulterior thing for who of you aims to code an official PostNuke module: you MUST always check for possibly raised exceptions and not code with the thought that something will never happen; you MUST also raise DATABASE_ERROR in every function that do queries. To get a better understanding of exception handling functions you should now look at PostNuke API Command Reference.

Chapter 3. PostNuke Module Design

The PostNuke module system design has been carried out by the PostNuke development team to allow for the maximum flexibility to developers whilst ensuring that the module can be accessed in a generic fashion by the PostNuke core, other modules, and remote systems given access through other interfaces such as XML-RPC. The main design characteristics of the module system are listed below.

3.1. Separation of User and Administrator Functions

Separation of user and administrator functions allows for a much cleaner module. It speeds up the responsiveness of the module in the most-often used cases (*i.e.* user actions) as the module only needs to load the code that is required of it. It allows for work one one area of the code (*e.g.* an admin GUI redesign) to take place without affecting the other areas. And it also gives an extra layer of security to help ensure that privileged functions cannot be executed inadvertently from user areas.

3.2. Separation of Display and Operational Functions

Separation of display and operational functions allows for areas within and without PostNuke to use the functionality supplied with a module. This is most obvious in the case of modules with blocks. where the block might display its own information but use the module functions to gather that information. Other modules where this is hugely important are the utility modules; things like comments and rating systems, that have no real use on their own but can be coupled with other modules to provide generic and site-wide functionality at very little cost to the module developer.

3.3. Single Directory Installation

Having a single install directory allows for much easier maintenance of large PostNuke systems, and far easier install and removal of modules both for the module developer and for the site administrator. Dependencies of the layout on the filesystem are no longer required, and as such the module designer does not need to worry about on which systems his module might be deployed, and how it needs to interact with the underlying operating system to function correctly.

3.4. External Access to Module Functions

Allowing access to module functions from external (*i.e.* non-PostNuke) systems is a very desirable thing to do. By allowing this, the PostNuke system becomes a content repository, where information can be accessed in ways other than through the standard web interface. An example of this power can be seen through use of the XML-RPC interface that is provided with PostNuke and which allows other systems to obtain information directly from modules without going through the web interface, thus allowing things like easy syndication of a site’s content.

Chapter 4. PostNuke Module Operations

This chapter covers how modules interact with PostNuke. The information in this chapter is correct for the 0.71 release of PostNuke, for other releases please get the most recent copy of the Module Developers Guide.

4.1. Locating Modules

All PostNuke modules must be placed within their own subdirectory of the 'modules' directory to be recognised. Modules placed anywhere else within the filesystem will not be located correctly.

4.2. Working out Module Functionality

A module might have administration or user functionality, or both. PostNuke works out which functionality each module has by looking for the files 'pnadmin.php' or 'pnadminapi.php' to confirm administration functionality, and 'pnuser.php' or 'pnuserapi.php' to confirm user functionality. Lack of these files results in PostNuke assuming that this specific module functionality does not exist.

4.3. Initialising Modules

Initialisation of modules is through the *modname*`_init()` in the 'pninit.php' file within the module's directory. function. No other function is called when the module is initialised.

4.4. Activating/Deactivating Modules

Activation and deactivation of modules is done through field settings within the appropriate database table. Unlike earlier versions of PostNuke, no physical changes to the module directories is made to infer the activation status of the module.

4.5. Calling Module Functions

Module functions are called through the `pnModFunc()` and `pnModAPIFunc()` functions. No direct calling of module functions is allowed, even from within the same module.

4.6. Creating Module URLs

4.7. Direct URLs to functions

URLs for new-style modules go through the 'index.php' entry point, and are defined by a number of parameters. The parameters that currently decide which particular module function to call are as follows:

module

The name of the module. This corresponds to the well-known name of the module, which can be found through the modules administration interface

type

The type of the module function. This is currently either 'user' for user functions or 'admin' for administrative functions.

func

The name of the function itself. This is module-dependent.

If any of these parameters are undefined within a URL PostNuke will apply defaults to them. Note that bot the names of the parameters and their default values might change, and as such it is not recommended to create direct URLs for anything but to either go through the PostNuke main page or to use the `pnModURL()` function to generate URLs that will always be internally consistent for any given version of PostNuke.

Chapter 5. Before Starting Your Module

There are a number of steps that need to be taken before you can start building your module.

5.1. Choose a Name for Your Module

Choosing a name for your module is important, as this is the main way that your module will be known throughout the PostNuke community. The name should be related to the functionality that the module provides, but also be specific enough to be able to discern it from separate modules that might offer similar functionality.

Module names are case-sensitive. For this reason, it is highly recommended that all modules names are lower-case only.

5.2. Decide on the Type of Your Module

There are two broad types of module available in PostNuke. *Item modules* are modules which contain their own content and operate on that content, whereas *utility modules* are modules which contain additional information or functionality for the content of other modules. Examples of item modules are news, FAQ, and downloads. Examples of utility modules are comments, ratings, and global index. Utility modules can either work in the same way as item modules, or they can operate through the use of *hooks*, which allow module functions to be acted upon without being explicitly called by other modules. Hooks are normally used for items that are not part of a piece of content but directly related to it

5.3. Register Your Module Name

Registering your module is not compulsory, but it is a very good idea. By registering your module you can ensure that no other official PostNuke module will take the name that you have chosen for your module. Two modules with the same name will not operate correctly on a single PostNuke site, so it is beneficial to both yourself and the PostNuke community in general to have a unique name.

> **Note:** Need information on how to register module names and get a module ID

5.4. Obtain a Copy of The PostNuke API Reference Guide

A copy of the PostNuke API reference guide is essential when developing a module. This guide covers all of the core functionality that the PostNuke system provides and provides example code for every API function available.

> **Note:** Add link to pnAPI download

5.5. Read the 'Notes on Developing Modules' Section

The section entitled 'Notes on Developing Modules' includes a lot of miscellaneous information that does not fit in other sections of this document. It should be read fully before any attempt to design or develop a module is started.

5.6. Understand the Following Areas

5.6.1. Difference Between GUI and Operational Functions

Understanding the difference between GUI and operational functions is critical when building a good module. Proper separation of these functions will allow other modules to be able to access the functionality of your module and incorporate it into their modules. It will also allow methods of access apart from those that the standard web-based PostNuke system.

5.6.2. Difference Between User and Administrative Functions

Understanding the difference between user and administrative functions is very important when building a good module. The separation of these types of actions allows for

5.6.3. The PostNuke Security Model

The PostNuke security model is a very important area to understand before coding a module. Developers should understand which parts of their module need to be protected, and exactly how this is accomplished.

> **Note:** Add description of the security model.

5.6.4. Function Return Codes

Every well-defined module function must return the appropriate return codes. Return codes are the main way in which a module communicates with the PostNuke core, and as such it is vital that the correct return codes are used.

The following return codes should be used when returning control to the PostNuke core from any module function:

text string

Returning a text string implies that the module function has finished its work and has output to be displayed in the appropriate place on the PostNuke web page. PostNuke will take the returned output and display it as appropriate. Note that all output from modules is displayed verbatim, with no escaping of HTML characters. This is to allow for formatted output from the module functions.

`true`

Returning boolean `true` implies that the module function has finished its work and set up an appropriate redirect to send the user to a page that will have display output. The PostNuke core will take no further action as far as this module is concerned.

`false`

Returning boolean `false` implies that the module function has finished its work but not set up an appropriate redirect to send the user to a page that will have display output. The PostNuke core will set an appropriate redirect for this module.

Note that none of these functions carry any information about the success or failure of the attempted operation that the module function was undertaking.

5.6.5. Where Modules Fit in PostNuke

Modules cover two separate areas of PostNuke. The first is administration of core functions, (*e.g.* users, permissions), and the second is extension of system functionality (*e.g.* downloads, web links). As each of these areas are not core this implies two things. First is that no module is actually required - the PostNuke system would work without anything in its modules directory, although its functionality would be severely limited and there would be no configuration options available. Second is that modules should not remove any core functionality when they are installed, operated, or removed.

5.7. Design Your Module!

An often overlooked point is that the module should be designed before being coded. This will allow for far easier coding later on, and an understanding of how the module fits into the generic PostNuke module structure. Some of the points that should be considered are:

- What data does the module store? How should the module data best be stored? Is the data hierarchical or flat?
- What does the module do with the stored data? How is the data displayed, how much data is displayed at any one time? What options should the user have to view the data in different ways?
- How does the module interact with other modules? Does it compete directly with other modules? If so, does it make sense to follow their module API to allow for greater interoperability between similar modules? Can it use other modules for part of its functionality? Is it better written as an extension to a current module rather than starting again from scratch?

5.8. Consider Including the Standard Module Functions

There are a number of standard module functions that allow a newly written module to interface with other parts of the PostNuke system. These functions have predefined inputs and outputs, allowing external modules and core functions to use them effectively without needing to tailor their operation to each separate module. The best example of these functions is the 'search' function, which passes in a simple text string and requires that an array is passed back about all items within the module that match the string.

Note: Clarify the search function

If your module does not have these functions then it will not integrate fully with the other parts of the PostNuke system. It is recommended that these functions are supplied if they make any sense in the context of your module.

5.9. Use Standard Function Names

There are a number of function names that are considered standard *i.e.* they have well-known meanings and are used in a number of modules. Using the standard function names makes it easier for other module developers to use your module. Some of the standard functions are shown below.

> **Note:** The below list is subject to addition as more standard functions are introduced - the template module supplied with your copy of PostNuke should have the most up-to-date set of standard functions available.

5.9.1. User Display Functions

- `main()` - the default function to call, normally just presents the user menu
- `view()` - display an overview of all items, normally paged output
- `display()` - display a single item in detail, given an identifier for that item

5.9.2. User API Functions

- `getall()` - get basic information on all items, can take optional parameters to obtain a subset of all items
- `get()` - get detailed information on a specific item

5.9.3. Administration Display Functions

- `main()` - the default function to call, normally just presents the user menu
- `view()` - display an overview of all items, normally paged output, with relevant administrative options. Note that it is possible to combine this function with the user `view()` function
- `new()` - display a form to obtain enough information from the user to create a new item
- `create()` - take the information from the form displayed by the administration `new()` function and pass it on to the administration API for creating the item
- `modify()` - display the details of a current item given the item description, and present the relevant fields for modification
- `update()` - take the information from the form displayed by the administration `modify()` function and pass it on to the administration API for modifying the item
- `delete()` - display confirmation for deletion of an item, and if confirmed pass the relevant information on to the administration API for deleting the item

- `modifyconfig()` - display the details of the module's current configuration, and present the relevant fields for modification
- `updateconfig()` - take the information from the form displayed by the administration `modifyconfig()` function and update the relevant module configuration variables

5.9.4. Administration API Functions

- `create()` - create a new item
- `delete()` - delete a current item
- `update()` - update the information about a current item

5.10. Find Out What Utility Modules Are Available

There are a number of utility modules available to carry out features that are required by many item modules within PostNuke. Examples of available utility modules are comments, ratings, and categorisation. Take a look at mods.postnuke.come to find out what other utility modules are available and if they can be used in lieu of parts of the code that you would otherwise be writing for your own module.

> **Note:** Point to a repository of utility modules and functions.

Chapter 6. Module Directory Structure

PostNuke modules have a very specific directory structure. This allows the PostNuke system to use a generic system to access all modules without needing to know specific information about each separate module that is built. Following the directory structure as laid out below is an absolute requirement of any PostNuke-compliant module.

> **Note:** Extra files and directories in addition to those shown below are allowed. Also, if any of the files below are not required (*e.g.* the module does not have database tables of its own so it does not require the pntables.php file) then they do not need to exist. However, files that perform the functions outlined below must comply with the file naming convention to allow the PostNuke system to load the suitable files at the appropriate times to ensure correct operation of the module.

> **Note:** This shows the example layout that a gallery module might have. Other modules will have different names for their top-level directory and blocks as appropriate for their specific functionality.

```
modules/                              ❶
        gallery/                      ❷
                pnadmin.php           ❸
                pnadminapi.php        ❹
                pnblocks/             ❺
                          snapshot.php ❻
                pnimages/             ❼
                          admin.png   ❽
                pninit.php            ❾
                pnlang/               (10)
                       deu/           (11)
                           admin.php  (12)
                           init.php   (13)
                           manual.html (14)
                           snapshot.php (15)
                           user.php   (16)
                       eng/           (17)
                           admin.php  (18)
                           init.php   (19)
                           manual.html (20)
                           snapshot.php (21)
                           user.php   (22)
                       ...
                pntables.php          (23)
                pnuser.php            (24)
                pnuserapi.php         (25)
                pnversion.php         (26)
```

❶ The top-level directory in PostNuke for modules

❷ The directory that contains all of the module code (in this case the module is named ’gallery’)

❸ The file that contains all administrative GUI functions for the module

❹ The file that contains all administrative operational functions for the module

❺ The directory that contains all blocks associated with the module

❻ A file that contains a block associated with this module; in this case it displays a random snapshot from the gallery

❼ The directory that contains all images for the module

❽ The image for the administration icon of the module

❾ The file that contains initialisation functions for the module

(10) The directory that contains all language translation files for the module

(11) The directory that contains all German language translation files for the module

(12) The file that contains German language translations for the administrative GUI functions for the module (*i.e.* pnadmin.php)

(13) The file that contains German language translations for the initialisation functions for the module (*i.e.* pninit.php)

(14) The file that contains the German language translation of the manual for the module

(15) The file that contains German language translations for the snapshot block

(16) The file that contains German language translations for the user GUI functions for the module (*i.e.* pnuser.php)

(17) The directory that contains all English language translation files for the module

(18) The file that contains English language translations for the administrative GUI functions for the module (*i.e.* pnadmin.php)

(19) The file that contains English language translations for the initialisation functions for the module (*i.e.* pninit.php)

(20) The file that contains the English language translation of the manual for the module

(21) The file that contains English language translations for the snapshot block

(22) The file that contains English language translations for the user GUI functions for the module (*i.e.* pnuser.php)

(23) The file that contains all information on database tables for the module.

(24) The file that contains all user GUI functions for the module

(25) The file that contains all user operational functions for the module

(26) The file that contains all version and credit information for the module

Chapter 7. Building Your Module

7.1. Make Your Initial Directory

Create the directory to hold the module files. This directory must be created under the 'modules' directory in the PostNuke install, and must be created with the name of your module as registered at the PostNuke modules site.

7.2. Copy the Module Template

Copy over all of the files from the template directory into you newly created module directory. These files set up the basic structure for your module and allow you to get to work creating your module very quickly.

7.3. Code your Database Tables

Coding your database tables requires you to edit the pntables.php file in your module directory. This file gives information on the structure of the tables used by this module, although it does not carry out any actions itself. The structure information is wrapped in a function (*modname*_pntables()) for easy access by the PostNuke system. An annotated copy of the template pntables.php file is available in the standard PostNuke distribution as part of the Template module.

If your module uses tables specified by another module then you can either remove the pntables.php file completely from your module directory, or have a suitably named function that just returns an empty array.

> **Caution**
>
> If you attempt to use the same table name as another module or the PostNuke core then your module will fail in unexpected ways. Try to give your tables unique names, preferably based on your module name.

7.4. Write your Initialisation Functions

Module initialisation functions are required for three separate actions. These actions are initialisation of the module's tables and configuration, upgrade of the module's tables and configuration, and deletion of the module's tables and configuration. Each of these items are generally only ever called once, although if a site administrator desires they should be able to initialise and delete a module as many times as they wish. It should be assumed that whenever these functions are called the PostNuke system has already loaded the relevant information from `pntables.php` and it is available in the information returned by `pnDBGetTables()`.

An annotated copy of the template `pninit.php` file is available in the standard PostNuke distribution as part of the Template solution.

7.5. Test Your Initialisation Routines

Once the database structure and initialisation files are in place they should be tested by using the modules administration area of your PostNuke system to test initialising and deleting your module. You should manually check that the database table created is correct, and that deleting a module removes all of the relevant configuration variables and database tables. Once you are happy that the module initialisation functions are working correctly you should carry out an initialisation so that work on the administration and user functions can proceed with suitable database tables in place.

7.6. Write your Administration Functions

With your database tables in place the next step is to write some administration functions. The administration functions that you will write depend on the nature of your module, however most modules have at least the following items:

- add a new item
- modify an existing item
- delete an existing item

Each of these items is normally broken down into three separate pieces. The first piece is part of the GUI and displays a form with suitable fields for user input. The second piece is part of the API and carries out the requested operation. The third piece is another part of the GUI and gathers information from the form displayed by the first piece and passes it as arguments to the second piece. The interaction between the three pieces is shown in the diagram below.

As mentioned earlier in the document, it is vital that the separation between the GUI and API functions is clear. If you are unsure about whether part of a function should be in the GUI or the API, take a look at what it does. If it is directly involved with user interaction (gathering information from the user or displaying information to the user) then it is a GUI function. If it is involved with obtaining or updating information in the PostNuke system itself (normally in a database table) then it is an API function.

Annotated copies of the template `pnadmin.php` and `pnadminapi.php` files are available in the standard PostNuke distribution in the Template module.

7.7. Test Your Administration Routines

Once the administration functions are in place they should be tested by using the administration area of your module to carry out the basic functionality that you have created. The operation of the module functions should be checked against the information in the database to ensure that they are storing and displaying the data correctly.

7.8. Write your User Functions

Once the administration functions are in place to manipulate your module's data then you can write the user functions to display the data. As with the administration functions the user functions that you will write depend on the nature of your module, however most modules have at least the following items:

- overview of a number of items
- detailed view of a single item

Each of these items is normally broken down into two separate pieces. The first piece is part of the GUI and gathers information from the user as to which item they wish to view, passes it on to the API piece, and displays the resultant information. The second piece is part of the API and obtains the required information for the display piece. The interaction between the three pieces is shown in the diagram below.

Annotated copies of the template `pnuser.php` and `pnuserapi.php` files are available in the standard PostNuke distribution as part of the Template module.

7.9. Test Your User Routines

Once the user functions are in place they should be tested by operating the module in the same way that a normal user would. The operation of the module functions should be checked against the information in the database to ensure that they are displaying the data correctly.

7.10. Write your Blocks

You might want your module to include *blocks*. Blocks are smaller functional units of a module that display specific information, and generally show up down the left and right hand sides of a page. Blocks are relatively simplistic items, and can either use their module's API functions to obtain information or use their own direct SQL query. Although they are packaged as part of the module they are not directly related to it except that they use the same database tables, and as such they might have to load the module's database table information directly through the use of the `pnModDBInfoLoad()` function if they intend to access the module's tables directly.

An annotated copy of the template `first.php` block file is available in the standard PostNuke distribution as part of the Template module.

7.11. Test Your Blocks

Once the blocks are in place they should be tested by displaying them through the Blocks administration system. The blocks should be checked against the database and the user functions to ensure that they are displaying the data correctly.

7.12. Document Your Module

Documenting your module is a vital step. There are two areas in which your module will need documentation: user information and API information. The first area is covered by producing a manual and placing it in the appropriate place in the directory hierarchy. The second area is covered by writing a short description of each API function, noting the parameters and return values that it has, and placing that at the head of the function. Coding the documentation in the style of PHPDoc (http://www.phpdoc.de/) will allow for automatic parsing of the documentation by other developers who wish to use your module.

7.13. Package Your Module

At this stage the module should be ready for packaging. The two most widely used packaging formats are WinZip (.zip extension) and compressed TAR (.tar.gz extension). If possible, package the module with both formats. If not then just package it with the format that you have and ask someone on the PostNuke modules site if they can package it in the other format.

Chapter 8. Interacting With Other Modules

8.1. Overview

When designing your module you may well find that there is some functionality that you require in the module that is already available to you in other modules. Utility modules have been designed specifically to provide additional often-used functionality for modules in a standard way, and sometimes the functionality of an entire module might be used as part of your module. Functionality can be obtained either from the display part of the module or the API itself, depending on the specific requirements in the new module. Interaction with other modules is carried out in different ways depending on the type of module being written and the level of specific control the module requires over the function being called.

8.2. Hooks

Hooks are a way of adding functionality to modules without the modules themselves knowing what the functions might be. The operation of hooks are controlled by the site administrator, so the decision as to which pieces of extra functionality to use and which not is in their hands rather than the module developer.

Hooks are called for specific actions that take place in a module. At current, the actions that hooks are enabled for are as follows:

- Addition of a category
- Deletion of a category
- Transformation of category data into a standard PostNuke format
- Display of a category
- Addition of an item
- Deletion of an item
- Transformation of item data into a standard PostNuke format
- Display of an item

> **Note:** The terms *category* and *item* are quite broad. Category is used to define any database entity that contains other categories or items, whilst item is used to define any database entity that holds content. Due to this definition it is possible for an item to be a category as well, although this is an unlikely state of affairs and it should be obvious to a module developer which parts of the system deal with categories and which with items.

Hooks are the recommended way of extending the functionality of your module, and use of the appropriate pnAPI hook functions as described below is considered mandatory for a compliant module.

8.2.1. Calling Hooks

If you are developing an item module then you should allow utility modules to add functionality to the item module. This is carried out through use of the `pnModCallHooks()` function. This function should be placed wherever a

specific action is carried out by the item module, where the current specific actions that the hooks system is able to operate on are:

The hook calls should be made at the appropriate level depending on the action that is being taken. With the current hooks, addition and deletion hooks should be called at the API level, and display hooks should be called at the GUI level.

The `pnModCallHooks()` function takes a number of parameters, which are explained below:

hookobject

The object for which the hooks are to be called - currently either *`category`*, or *`item`*, as described above

hookaction

The action for which the hooks are to be called - currently one of *`create`*, *`delete`*, *`transform`*, or *`display`*

obid

An ID that, within the scope of the module and object, uniquely defines the entity for which the hook is being called

extrainfo

This is extra information that is required by the hook function, and is dependent on the hook action being called. Information on the information required by each hook is covered below.

For `create` hooks a string that can be used in conjunction with the obid as part of a URL to access the object. For example, if your `gallery_user_display()` function uses a variable *`picid`* to define the particular picture that a user wishes to look at then the URL would be something like 'index.php?module=gallery&func=display&picid=4' and the identification part of the URL would be something like 'picid=4' so you would pass 'picid' to this hook.

For `display()` hooks a URL that can be used by the hooks to return to a suitable page once they have finished any work that they might have to do. This is normally just the standard display URL for this function.

For `transform()` hooks an array of items that contain text-based content that can be transformed. This is normally all text-based items.

The `pnModCallHooks()` function returns different information depending on value of *`hookaction`*. If *`hookaction`* is *`display`* then the hook will return extra output that should be displayed directly following the display for the item itself. If *`hookaction`* is *`create`* or *`delete`* then the function will return either *`true`* or *`false`* depending on the success or failure of the hooks. If *`hookaction`* is *`transform`* then the hook will return an equivalent array to that which was passed in, with the items suitable transformed.

As an example of calling hooks, if you were developing the 'Gallery' module and were displaying a picture, after the display of the picture you would want to call the hooks to add any other functionality available and required by the site administrator. To do this you would use the following lines:

```
$output->SetInputMode(_PNH_VERBATIMINPUT);
$output->Text(pnModCallHooks('item',
                             'display',
                             $pictureid,
```

```
                                pnModURL('gallery',
                                         'user',
                                         'display',
                                         array('pictureid' => $pictureid))));
$output->SetInputMode(_PNH_PARSEINPUT);
```

which would add the verbatim output of the hooks to the current output. It is worth noting again here the from this code it can be seen that the module itself needs no information on what hooks, if any, exist, it just calls the function and lets the PostNuke core deal with what extra output should be added to this item.

One important area to understand is where exactly in your code to call hooks. For example, if you were displaying a thumbnail view of 100 pictures from your Gallery module, should you call an item display hook for each picture? The answer to this is somewhat dependent on the nature of your module, but in general you should only call display hooks when you are displaying the details of a single item rather than an overview of a large number of items (of course, if all of those items are in a single category then you should call a display hook for that category). However, the transform hook should be called whenever you are displaying content regardless of it if is just an overview, as the overview information could require transformation before display.

The annotated Template module in the standard PostNuke distribution contains notes on calling hooks within an item module.

8.2.2. Writing Hooks

If you are developing a utility module then you probably want to allow your module to be called as a hook. This requires the module functions to be able to be called as hooks, and the module to register and unregister its hooks as required.

8.2.2.1. Writing Hook Functions

Hook functions are very similar to standard module functions, but they have a number of extra restrictions placed on them to be able to work as hooks:

- Hook functions must be able to operate correctly given only two arguments in their arguments array - *`obid`* and *`returnurl`*, as these are the only parameters that are passed to the function if it is called as a hook. Other parameters can be allowed by the function but they must be optional, and default to suitable values if not present such that the function will work appropriately.
- Hook functions must not rely on other hooks to exist, or to have been called already or in future. The order of calling a list of hooks is undefined, and depending on the site administrator's preferences particular hooks might never be called.
- Hook functions must not call the `pnModCallHooks()` function, or functions that might themselves call `pnModCallHooks()`. If a hook does this it risks getting the code into an infinite loop.

The Ratings module that comes with the core PostNuke distribution has an example hook function that shows how to fit within these guidelines whilst still producing a general-purpose function.

8.2.2.2. Registering Hooks

Once your module has hook-capable functions in place they need to be registered on initialisation of the module so that the administrator can configure their applicability, and other modules can access them through the `pnModCallHooks()` function. This is carried out through use of the `pnModRegisterHook()` function. This function should be placed within the *`modname`*`_init()` function of your module and given appropriate parameters to register the relevant hook-capable module functions within your module as hooks.

The `pnModRegisterHook()` function takes a number of parameters, which are explained below:

hookobject

: The object for which the hook is to be registered - currently either *`category`*, or *`item`*, as described above

hookaction

: The action for which the hook is to be registered - currently one of *`create`*, *`delete`*, *`transform`*, or *`display`*

hookarea

: The area that the hook function covers - currently either *`GUI`* (for functions that are in `pnuser.php` and `pnadmin.php`) or *`API`* (for functions that are in `pnuserapi.php` and `pnadminapi.php`)

hookmodule

: The name of the module in which the hook function exists - normally the name of the module calling this function

hooktype

: The type of the hook function - currently either `user` or `admin`

hookfunc

: The name of the hook function

The `pnModRegisterHook()` function returns *`true`* if the registration was successful, and *`false`* if the registration is unsuccessful.

As an example of registering hooks, if you were developing the 'globalid' utility module (which gives every piece of content in PostNuke a separate ID) and had a `globalid_admin_create()` function which created an entry in the global ID table for this particular piece of content then you would register this as a creation hook. To do this you would use the following lines within `globalid_init()`:

```
if (!pnModRegisterHook('item',
                       'create',
                       'API',
                       'globalid',
                       'admin',
                       'create')) {
    return false;
}
```

which would register this hook to be called every time a hook-enabled module someone creates an item (a similar but separate call would be needed to register this hook for the creation of categories as well).

The Ratings module that comes with the core PostNuke distribution has detailed comments on registering hooks within a utility module.

8.2.2.3. Unregistering Hooks

If your module has hook-capable functions that are registered when the module is initialised they need to be unregistered when the module is deleted. This is carried out through use of the `pnModUnregisterHook()` function. This function should be placed within the *`modname`*`_delete()` function of your module and given appropriate parameters to unregister the functions that were previously registered hooks when the module was initialised.

The `pnModUnregisterHook()` function takes the same parameters as the `pnModUnregisterHook()` function.

The Ratings module that comes with the core PostNuke distribution has detailed comments on unregistering hooks within a utility module.

8.3. Function Calls

Another way of accessing the functionality of other modules is by calling their functions directly with the `pnModFunc()` function. Doing this allows a number of advantages over hooks, but also a number of disadvantages. In general, calling functions directly is more flexible as the module developer understands exactly which functions they are calling and can also pass additional arguments to the function to customise its abilities. The disadvantages are that the module named in the function call needs to be installed and active on the system for the calls to work, and if this is replace by a different module providing similar functionality it will not work correctly.

Using direct function calls to other modules is fine within a module, but the developer should consider the implications of this on systems that might not have the modules that they are using installed. Also, even if direct function calls are used then the module developer should still call hooks at the appropriate places in the code to allow for other extended functionality to be added to the module.

An example of where direct function calls might be used within the Gallery module would be if the module developer wanted users to be able to rate various aspects of the picture displayed such as ’use of colour’ and ’originality’. In this case a simple hook would not be able to accommodate this requirement, so the developer would instead make explicit calls to the ’Ratings’ utility module to display a number of separate ratings, each with its own identifier. The hook call would still be made, which might also add a rating to the picture, but in this case the value could be considered as the overall rating for the picture rather than that just for a specific part.

Chapter 9. Upgrading Your Module

Note: Add information how module upgrades interact with the installer

Chapter 10. Notes on Developing Modules

10.1. Use pnAPI

pnAPI is the PostNuke Application Programming Interface, a way for modules to interact with the PostNuke core without needing to access tables and internal structures directly. The API also allows for the underlying implementation details of PostNuke to be hidden from developer so that they can write modules in a standard fashion and not worry about what might change under the hood. This is very important for a system such as PostNuke which has undergone, and continues to undergo, radical changes in the core design to allow it to be faster, more secure, and more flexible.

pnAPI is the only supported way of accessing core information. Module developers must use these methods of obtaining information from the PostNuke core system; failure to do so will very likely result in their module not working when the next version of PostNuke is released.

10.2. Security

Security is a very important part of PostNuke. All modules should subscribe to the PostNuke Security model to ensure that they operate correctly within all environments. For full information on security refer to the PostNuke Security Model documentation, however the main points as regards modules are covered briefly below.

> **Note:** Add information on the PostNuke security model.

10.2.1. Variable Handling

All variables that come in to or go out of PostNuke should be handled by the relevant `pnVar*()` functions to ensure that they are safe. Failure to do this could result in opening security wholes at either the web, filesystem, display, or database layers. Full information on these functions is given in the PostNuke API Guide, and examples of their use are shown throughout the template module.

It can be assumed that any variables passed to functions in the PostNuke API will be handled correctly, and as such these variables do not need to be prepared with the `pnVar*()` functions.

10.2.2. Authorisation

All items displayed for users and actions carried out by administrators must be authorised through use of the `pnSecAuthAction()` function. This function underlies the entire PostNuke permissions system and as such must be used wherever an access check is required.

10.2.3. Reserved Variable Names

PostNuke has a number of variables which are reserved. These variables should not be used within modules as they can conflict with the PostNuke core and cause unpredictable results.

The current list of variables which are reserved are as follows:

file
func
loadedmod
module
name
op
pagerstart
pagertotal
type

In addition, all one-letter variables are reserved.

10.2.4. Page Path

All input from webpages goes through a two-stage process. The first part is displaying the information to be entered in a form, and the second is obtaining that information and passing it on to the module API. In addition to the visible information, there is often a number of hidden items of information in the first page that is used in the second page. To ensure that any attempt to add, delete, or change information in the PostNuke system goes through the full two-stage method of gathering and processing the information the two functions `pnSecGenAuthKey()` and `pnSecConfirmAuthKey()` must be used in the appropriate places. The Template module in the standard PostNuke distribution contains a number of functions that use these API calls, and care should be taken to note where they are used so that developed modules will have the same level of protection against fraudulent administrator requests.

10.3. Output

All output generated by module functions must be returned to the PostNuke core. No output of any type should be made directly from the module; doing this is not supported and will break in future versions of PostNuke.

> **Note:** Review for templating requirements

10.4. Using OO Code

Modules can be written as classes if desired, however the API as described in the rest of this document must still be adhered to. The simplest way of doing this is to use compatibility functions, for example:

```
function mymod_user_main()
{
    // Instantiate
    $obj = new myClass();

    // Call relevant method and return output
    return $obj->usermain();
}
```

Chapter 11. Module Developer's Checklist

The following checklist presents a number of items that need to be checked throughout the process of designing, building, and releasing a module.

Initial

- Decide on module type
- Choose name for module
- Register name for module
- Obtain and read MDG documentation
- Obtain and read API documentation

Module Design

- Separate User and Administration Functions
- Separate GUI and API Functions
- Design data tables
- Note which utility modules are of use
- Note which standard module functions apply
- Create module security schema

Module Build

- Copy template module directory
- Create database tables
- Create database initialisation routines
- Test database initialisation routines
- Write administration functions
- Test administration functions
- Test administration functions
- Write user functions
- Test user functions
- Write blocks
- Test blocks
- Document module API
- Package module

Module Checks

- No global variables used
- No PostNuke reserved variable names used
- No `echo()` or `print()` statements used
- All operations protected by `pnSecAuthAction()`
- All forms results protected by `pnSecConfirmAuthKey()`
- All form variables obtained by `pnVarCleanFromInput()`
- All output parsed through `pnVarPrepForDisplay()` or `pnVarPrepHTMLDisplay()`
- All suitable output censored by `pnVarCensor()`

- All variables in SQL queries protected by `pnVarPrepForStore()`
- All variables in filesystem access protected by `pnVarPrepForOS()`
- Calls to `pnModCallHooks()` in all appropriate locations