Xaraya Module Developers Guide

Submitted to
Xaraya Development Community

Prepared by
Scot R. Gardner
Xaraya Development Team
September 21, 2003
revised August 15, 2004

• Xaraya Developers Guide

Table of Contents

Xaraya Developers Guide.....2
Abstract.....3
Introduction.....4
Discussion.....5
What is a module?.....5
Why write a module?.....6
Related Documents.....6
Suggestions and Updates.....6
Xaraya Architecture.....6
Variable handling.....6
User Variables.....6
Variable validation.....8
Error handling.....10
Xaraya Module Design.....12
Separation of User and Administrator Functions.....13
Separation of Display and Operational Functions.....13
Single Directory Installation.....13
External Access to Module Functions.....13
Xaraya Module Operations.....13
Locating Modules.....14
Working out Module Functionality.....14
Initializing Modules.....14
Activating/Deactivating Modules.....14
Calling Module Functions.....14
Creating Module URLs.....14
Direct URLs to functions.....14
Before Starting Your Module.....15
Choosing a Name for Your Module.....15
Decide on the Type of Your Module.....15
Register Your Module Name.....16
Obtain a Copy of the Xaraya API Reference Guide.....16
Read the Notes on Developing Modules Section.....16
Understand the Following Areas.....16
Difference between GUI and Operational Functions.....16
The Xaraya Security Model.....17
Function Return Codes.....17

Where Modules Fit in Xaraya.....17
Design Your Module.....18
Consider Including the Standard Module Functions.....18
Use Standard Function Names.....18
User Display Functions.....19
User API Functions.....19
Administration Display Functions.....19
Administration API Functions.....20
Find out What Utility Modules are Available.....20
Module Directory Structure.....20
Building Your Module.....29
Make Your Initial Directory.....29
Copy the Module Example.....29
Code your Database Tables.....29
Write your Initialization Functions.....30
Test Your Initialization Routines.....30
Write Your Administration Functions.....30
Test Your Administration Functions.....31
Write Your User Functions.....31
Test Your User Functions.....31
Write Your Blocks.....32
Test Your Blocks.....32
Document Your Module.....32
Packaging Your Module.....32
Interacting With Other Modules.....33
Overview.....33
Hooks.....33
Calling Hooks.....34
Writing Hooks.....35
Writing Hook Functions.....36
Registering Hooks.....36
Un-registering Hooks.....37
Function Calls.....38
Upgrading Your Module.....38
Notes in Developing Modules.....39
Use xarAPI.....39
Security.....39
Variable Handling.....39
Authorization.....40
Reserved Variable Names.....40
Page Path.....40
Output.....40
Using Object Oriented Code.....41
Recommendations.....41

Glossary.....42
Appendix A.....43
Module Developers Check List.....43

-

Abstract

Xaraya *is currently in beta production and rapidly approaching a final release. The initial installation package includes* **core** *functionality for a basic content management solution listing of registered modules on Xaraya.com includes sixty-two content-based modules registered for a certified Xaraya identification number. Thirty-nine of those modules are currently completed or under development by members of the Xaraya development team.*

Third party developers have a concept or idea that they want to contribute to the project, but lack the proper tools that will allow them to submit **API**[1] *compliant modules for certification. A series of polls posted on Xaraya websites have revealed the need for this type of training document and have produced a list of suggested modules to choose from for use as examples in the development guide. Current research for this project was conducted by reading through the Xaraya code, developer comments and notes as well as existing documentation located in the beta repositories.*

Introduction

Xaraya is an extensible, open source platform written in **PHP**[2] *and licensed under the* **GNU** *General Public License. Xaraya utilizes permissions, data management, and modular systems to dynamically integrate and manage content. Xaraya's modular, database independent architecture introduces the need for add-on modules that will enhance*

[1] **application program interface (API):** *A formalized set of software calls and routines that can be referenced by an application program in order to access supporting network services*

[2] **PHP** *is a widely used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.*

its functionality and produce a customizable solution that will meet any web masters needs.

Xaraya's modular, database-independent architecture provides tools, which separate form, function, content, and design on the site. Xaraya delivers the requisite infrastructure for a fully dynamic multi-platform Content Management Solution (CMS).

** Database-driven website engine (***PHP**-**ADODB** *compliant)*

** Extensible through a variety of third-party modules*

** Powerful security module for multi-level user/administrator logins*

** Distribute workload using User/Administrator management tools*

** Robust article management system*

** Fully editable & manageable News, Links, Downloads, and FAQ Sections*

** **BlockLayout** Theme Engine: Display your site your way*

** News Feed Manager: Access thousands of **RSS**-compliant news feeds*

** Site Statistics: Tracks browser & operating system, top news & articles, and more*

** Easy install on most Unix/Linux and Windows platforms*

** Import information from **PostNuke**, **PHP-Nuke**, **phpBB2** and **Moveable Type** installations*

Xaraya reduces web site development costs by introducing sophisticated administration tools & services, which separate form, function, content, and design. With Xaraya, you work in a simple, structured environment so you can rapidly develop your website with diverse content, including:

** News Articles*

** Web Links Directory*

** Job Boards*

** Frequently Asked Questions*

** File Downloads*

** Photo Gallery*

** Member profiles*

** Web forums (message boards)*

** Articles Repository (i.e.: Knowledge base, cooking recipes, product reviews, news articles, etc.)*

With more than forty active developers spanning five continents and ten languages, the Xaraya Development Team is a blend of some of the best and brightest developers in the open source community. We expect the best of one another. As a result, you can expect the best from Xaraya.

Discussion

- ### What is a module?

The Xaraya system allows for expansion of its functionality using modules. A module is a set of files containing functions with predefined names and roles that integrate very easily with a standard deployment of Xaraya. A module can also include blocks, images, plain HTML files etc.

- ### Why write a module?

There are a number of reasons to write a module. The main reason is that Xaraya does not provide a specific function that you would like. Examples of modules that are currently available for Xaraya include bulletin boards, photo galleries, address books, and personal information managers.

- ## 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.*

- ## Suggestions and Updates

The Xaraya module system is a work-in-progress. There are no doubt many good ideas out there that have not been incorporated into the Xaraya module system, and if a developer has a request for a particular set of functionality then they can submit it to the Xaraya features request list at the Xaraya Homepage. 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 Xaraya module design is stability. Due to this, it is possible that your request for new or updated functionality will be refused because 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 Xaraya 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.*

- ## Xaraya Architecture

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

- ## Variable handling

Variables are loaded through xarVarFetch calls in order to be processed, checked for validity, as well as proper handling on unset vars, etc. Instead of relying on $_POST or $_GET calls throughout your module and then cleaning the input, Xaraya has a very robust way of handling the variable.

An example call would be:

```
    if (!xarVarFetch('itemsperpage', 'int', $itemsperpage,
10, XARVAR_NOT_REQUIRED)) return;

        Fetches the $name variable from input variables and
        validates it by applying the $validation rules.

1st try to use the variable provided, if this is not set
(Or the XARVAR_DONT_REUSE flag is used) then it try to get
the variable from the input (POST/GET methods for now)
Then tries to validate the variable through xarVarValidate.

After the call the $value parameter passed by reference is
set to the variable value converted to the proper type
according to the validation applied.

The $defaultValue provides a default value that is returned
when the variable is not present or doesn't validate
correctly.

 The $flag parameter is a bitmask between the following
constants: XARVAR_GET_OR_POST, XARVAR_GET_ONLY,
 XARVAR_POST_ONLY, XARVAR_NOT_REQUIRED.

 You can force to get the variable only from GET parameters
or POST parameters by setting the $flag parameter to one of
XARVAR_GET_ONLY or XARVAR_POST_ONLY.

 You can force xarVarFetch not to reuse the variable by
setting the $flag parameter to XARVAR_DON_REUSE.

 By default $flag is XARVAR_GET_OR_POST which means that
xarVarFetch will lookup both GET and POST parameters and
that if the variable is not present or doesn't validate
correctly an exception will be raised.

 The $prep flag will prepare $value by passing it to one of
the following:
  XARVAR_PREP_FOR_NOTHING:    no prep (default)
  XARVAR_PREP_FOR_DISPLAY:    xarVarPrepForDisplay($value)
  XARVAR_PREP_FOR_HTML:       xarVarPrepHTMLDisplay($value)
  XARVAR_PREP_TRIM:           trim($value)
```

- ## User Variables

A user variable is an entity identified by a name that stores a value owned by exactly one user. Xaraya offers two **API** *functions to manipulate user variables, they are* `xarUserGetVar()` *and* `xarUserSetVar()`*. The purpose of* `xarUserGetVar()` *is to allow read access to one user variable. In contrast to that,* `xarUserSetVar()` *allows write access to one user variable. The $name parameter is checked against* **metadata** *to make sure the variable is registered. Xaraya keeps some* **metadata** *about every user variable, so you cannot 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.*

Xaraya does not 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 display name on the system. "Name" should be used in any case that you are displaying the user name, IE author fields, etc.

uname

The user name is the system variable used to differentiate one user from another in string vice integer format. This variable should not be used to display information about the user for consistency and security reasons.

email

The email address of the user.

status

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

auth_module

The authentication module 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 types
$metadata['default'] = 20;
$metadata['validation'] = 'num:>=:10&num:<=:100';

xarModAPILoad('Modules', 'admin');

$result = xarModAPIFunc('Modules', 'admin', 'register_user_var', $metadata);
if (!isset($result)) {
  // xarModAPIFunc() 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 `xarUserGetVar()` *and* `xarUserSetVar()` *$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 would contain. The default field is optional; 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 a 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, Xaraya offers support for module variables too. If you get confused from that, and cannot see the distinction between these different things, here is a little explanation.

Module variables are system-wide variables, shared between each module user, like configuration variables. No particular user owns them, and even if they do they are 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 behaviors of your module. However, when those behaviors are related to user preferences you should avoid using module variables and register a new user variable to be used in your code.

As an 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 is reasonable to have selected this choice, but here we could have chosen a unique shared module var as well. On the other hand, you do not have this kind of freedom, for example consider the authldap module in some cases. 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.*

- ## Setting xarUser*Vars

All user vars should be set from adding Dynamic Data on the roles module. This will give you the greatest benefit and will present the variables to all modules on the system.

- ## Error handling

Xaraya is capable of error handling through a powerful exception handling system. Since the **PHP** *language does not*

support language-level exceptions, Xaraya provides an artificial mechanism to deal with exceptions. Xaraya divides exceptions into two types: system exceptions and user exceptions. System exceptions are used by Xaraya **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 considers the BAD_PARAM exception, you should choose to use that exception in your module functions and **API** *functions when passed parameters are wrong. Finally, system exceptions are well known exceptions for which Xaraya can undertake. Xaraya does not know particular actions like logging, emailing, or user exceptions, and since they are indistinguishable, Xaraya will treat them, as they were all the same thing.*

Another good point of 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 them back to Xaraya, this is because user exceptions are the same 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 Xaraya exception caught error message. However, it is not illegal to throw back user exceptions to Xaraya, so fell free to do that if it is the case.

You should avoid catching system exceptions, except in particular cases. A system exception is a hard exception, this means that something very wrong happened and Xaraya should be notified. You can achieve this simply by throwing back system exceptions. In addition, there are particular circumstances in which you could and perhaps should catch system exceptions.

For example consider the `xarUserGetVar()` **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 do so.

Now is the moment to explore how Xaraya permits you to deal with exceptions. Here we begin by exposing how to catch exceptions. When a function that potentially can raise exceptions, outcomes with a void value you MUST check if some exception was raised. You can do that by calling the `xarExceptionMajor()` *function and comparing its return value with the XAR_NO_EXCEPTION constant. If they are different, you know that an exception has been raised.*

The `xarExceptioMajor()` *return value can assume one of these values: XAR_NO_EXCEPTION, XAR_USER_EXCEPTION, XAR_SYSTEM_EXCEPTION. The value XAR_NO_EXCEPTION indicates that no exception was raised, and XAR_USER_EXCEPTION stands for user exception was raised and XAR_SYSTEM_EXCEPTION stands for system exception was raised.*

When you see that an exception was raised you have two options: 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:

```
$res = xarModFunc('MyModule', 'user', 'MyFunc');
if (!isset($res) && xarExceptionMajor() != XAR_NO_EXCEPTION) {
    // Got an exception
    if (xarExcepionMajor() == XAR_SYSTEM_EXCEPTION) {
        return; // throw back
    }
    // Got a user exception
    if (xarExceptionId() == 'MyException1') {
        $value = xarExceptionValue();
        $output->Text("Syntax error at line: ".$value->lineNumber);
    } elseif (xarExceptionId() == 'MyException2') {
        /* Do something useful */
    } else { // MyException3
        /* Do something useful */
    }
    // reset exception status
    // NOTE: it's of vital importance to call this function
    //       before returning
    xarExceptionFree();
    return $output->GetOutput();
}
```

To throw exception you use a unique function: `xarExceptionSet()`*. 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;
        xarExceptionSet(XAR_USER_EXCEPTION, 'MyException1', $exc);return;
    }
    /* ... */
    xarExceptionSet(XAR_USER_EXCEPTION, 'MyException2');
    /* ... */
    xarExceptionSet(XAR_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 not to use user exceptions and always throw back system exceptions.

Keep in mind that good error handling is not something that to leave for last. It should be part of the development process. Note that it is wrong not to check exception status after a call to a function that can potentially raise something. And note also that if you choose to handle one or more exceptions you MUST call `xarExceptionFree()` *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 those of you wanting to code an official Xaraya 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 does queries. To get

a better understanding of exception handling functions you should refer to the Xaraya **API** *Command Reference.*

- ## Xaraya Module Design

The Xaraya module system design allows for the maximum flexibility to developers whilst ensuring that the module can be accessed in a generic fashion by the Xaraya 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.*

- ## 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 area of the code (e.g. an admin **GUI** *redesign) to take place without affecting the other areas. In addition, it gives an extra layer of security to help ensure that privileged functions cannot be executed inadvertently from user areas.*

- ## Separation of Display and Operational Functions

Separation of display and operational functions allows for areas within and without Xaraya 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.

- ## Single Directory Installation

Having a single install directory allows for much easier maintenance of large Xaraya 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 file system 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.

- ## External Access to Module Functions

Allowing access to module functions from external (i.e. non-Xaraya) systems is a very desirable thing to do. By allowing this, the Xaraya 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 Xaraya and which allow for such tools as the Google API to be used, or other webservice events.*

- ## Xaraya Module Operations

This chapter covers how modules interact with Xaraya. The information in this section is correct for the 0.92 release of Xaraya, for other releases please get the most recent copy of the Xaraya Module Developers Guide.

- ## Locating Modules

All Xaraya modules must be placed within their own subdirectory of the 'modules' directory to be recognized. Modules placed anywhere else within the file-system will not be located correctly.

- ## Working out Module Functionality

A module might have administration or user functionality, or both. Xaraya works out which functionality each module has by looking for the files in 'xaradmin' or 'xaradminapi' directories to confirm administration functionality, and in 'xaruser' or 'xaruserapi' directories to confirm user functionality. Lack of these directories results in Xaraya assuming that this specific module functionality does not exist.

- ## Initializing Modules

Initialization of modules is accomplished through the `modname_init()` *in the 'xarinit.php' file within the module's*

directory function. No other functions are called when the module is initialized.

- ## Activating/Deactivating Modules

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

- ## Calling Module Functions

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

- ## Creating Module URLs

- ## 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**

1. *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**

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

- **func**

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

If any of these parameters are undefined within a URL Xaraya will apply defaults to them. Note that both 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 Xaraya main page or to use the `xarModURL()` *function to generate URLs that will always be internally consistent for any given version of Xaraya.*

You can also extend the xarModUrl call by adding a parameter call to the function in the form of an array. `xarModURL('module', 'type', 'function', array('foo' => $bar))` *will create the input parameter of 'foo' having the value of $bar.*

• Before Starting Your Module

There are a number of steps to follow before you can start building your module.

• Choosing 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 Xaraya community. The name should relate 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.

• Decide on the Type of Your Module

There are two broad types of module available in Xaraya. ***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 download. Examples of utility modules are comments, ratings, and global index. Either utility modules work in the same way as item modules, or they can operate with* ***hooks****, which allow module functions to be acted upon without being explicitly called*

by other modules. Hooks are used for items that are not part of a piece of content but directly related to it

- ## 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 Xaraya 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 Xaraya site, so it is beneficial to both yourself and the Xaraya community in general to have a unique name.

You may register your module via the release module on xaraya.com

- ## Obtain a Copy of the Xaraya API Reference Guide

The Xaraya API Reference Guide has been moved to the PHP-Doc Style of documentation. No separate entry will be created for the API guide. Please reference http://docs.xaraya.com/index.php/documentation/c80/ for more information.

- ## 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.

- ## Understand the Following Areas

  - ### 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 Xaraya system.*

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

- ## The Xaraya Security Model

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

The entire Xaraya security model is beyond the scope of this document. The Security System RFC30 is located at *http://docs.xaraya.com//docs/rfcs/rfc0030.html*

- ## 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 Xaraya 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 Xaraya core from any module function:

text string

Returning a text string implies that the modules function has finished its work and has output to be displayed in the appropriate place on the Xaraya web page. Xaraya 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 Xaraya 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 Xaraya core will set an appropriate redirect for this module.

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

• Where Modules Fit in Xaraya

Modules cover two separate areas of Xaraya. 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 is not core this implies two things. First is that no module is actually required – the Xaraya 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 installed, in operation, or removed.

• 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 Xaraya 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?

- **Consider Including the Standard Module Functions**

There are a number of standard module functions that allow a module to interface with parts of the Xaraya 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.

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

- **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.

The list below is subject to addition as more functions that are standard are introduced - the example module supplied with your copy of Xaraya should have the most up-to-date set of standard functions available.

- **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

- **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*

- **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*

- **Administration API Functions**

`create()` – *create a new item*

`delete()` – *delete a current item*

`update()` – *update the information about a current item*

- **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 Xaraya. Examples of available utility modules are comments, ratings, and categorization. Look at xaraya.com 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.

- **Module Directory Structure**

Xaraya modules have a very specific directory structure. This allows the Xaraya 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 Xaraya-compliant module.

Extra files and directories in addition to those shown below are allowed. In addition, 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 xartables.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 Xaraya system to load the suitable files at the appropriate times to ensure correct operation of the module.

This shows the layout of the example module directory. Other modules will have different names for their top-level

directory and blocks as appropriate for their specific functionality

```
modules/                                          (1)
        example/                                  (2)
                xaradmin/                         (3)
                        create.php                (4)
                        delete.php                (5)
                        main.php                  (6)
                        modify.php                (7)
                        modifyconfig.php          (8)
                        new.php                   (9)
                        update.php                (10)
                        updateconfig.php          (11)
                        view.php                  (12)
                        index.html                (13)
                xaradminapi/                      (14)
                        create.php                (15)
                        delete.php                (16)
                        getmenulinks.php          (17)
                        menu.php                  (18)
                        update.php                (19)
                        index.html                (20)
                xarblocks/                        (21)
                        first.php                 (22)
                        others.php                (23)
                        index.html                (24)
                xarimages/                        (25)
                        admin.gif                 (26)
                        admin_generic.gif         (27)
                        preferences.gif           (28)
                        index.html                (29)
                xartemplates/                     (30)
                        blocks/                   (31)
                           first.xd               (32)
                           firstAdmin.xd          (33)
                           othersAdmin.xd         (34)
                           example-firstblock-modify.xd(35)
                        admin-delete.xd           (36)
                        admin-main.xd             (37)
                        admin-menu.xd             (38)
                        admin-modify.xd           (39)
                        admin-modifyconfig.xd     (40)
                        admin-new.xd              (41)
                        admin-view.xd             (42)
                        index.html                (43)
                        user-display.xd           (44)
                        user-main.xd              (45)
                        user-menu.xd              (46)
                        user-usermenu_icon.xd     (47)
                        user-view.xd              (48)
                        user-usermenu_form.xd     (49)
                xaruser/                          (50)
                        display.php               (51)
                        main.php                  (52)
                        usermenu.php              (53)
                        view.php                  (54)
                        index.html                (55)
```

```
                xaruserapi/                          (56)
                        encode_shorturl.php          (57)
                        countitems.php               (58)
                        decode_shorturl.php          (59)
                        get.php                      (60)
                        getall.php                   (61)
                        getmenulinks.php             (62)
                        menu.php                     (63)
                        validateitem.php             (64)
                        index.html                   (65)
                        getitemlinks.php             (66)
        xareventapi.php                              (67)
        xarinit.php                                  (68)
        xartables.php                                (69)
        example.wsdl                                 (70)
        index.html                                   (71)
        xaradmin.php                                 (72)
        xaradminapi.php                              (73)
        xaruser.php                                  (74)
        xaruserapi.php                               (75)
        xarversion.php                               (76)
```

(1) The top-level directory in Xaraya for modules

(2) The directory that contains all of the module code (in this case the module is named 'example')

(3) The directory that contains all administrative GUI functions for the module

(4) create.php
This a standard function that is called with the results of the form supplied by xarModFunc ('example','admin','new') to create a new item.

Syntax: @param $'name' - The name of the item to be created
@param $'number' - The number of the item to be created

(5) delete.php
This standard function is called whenever an administrator wishes to delete a current module item. Note that this function is the equivalent of both the modify() and update() functions as it operates a form and processes its output. This is fine for simpler functions, but for complex operations such as creation

and modification it is generally easier to separate them into separate functions. There is no requirement in the Xaraya MDG to use one or the other, so either or both can be used as seen appropriate by the module developer.

Syntax: @param $'exid' – The id of the item to be deleted
@param $'confirm' – Confirm that this item can be deleted

(6) main.php
This function is the default function, and is called whenever the module is initiated without defining arguments. As such, it can be used for a number of things, but most commonly, it either shows the module menu and returns or calls whatever the module designer feels should be the default function. (This is often the view function)

(7) modify.php
This is a standard admin function that is called whenever an administrator whishes to modify a current module item.

Syntax: @param $'exid' the id of the item to be modified

(8) modifyconfig.php
This is the standard function to modify the configuration parameters for the module.

(9) new.php
This standard function is called whenever the administrator wishes to create a new module item.

(10) update.php
This is the standard function that is called with the results of the form supplied by xarModFunc('example', 'admin','modify') to update a current item.
Syntax: @param $ 'exid' – The id of the item to be updated
@param $ 'name' – The name of the item to be updated

@param $ 'number' – The number of the item to be updated

(11) updateconfig.php
This standard function updates the configuration parameters of the module given the information passed back by the modification form.

(12) view.php
This standard function is called whenever the administrator wishes to view a module item.

(13) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(14) xaradminapi
The directory that contains all administrative database functions for the module.

(15) create.php
This standard function creates an item when the administrator wants to create an item in the database. Parameters are passed from the xaradmin/create.php. This function returns the example item upon success, or raises an exception error upon failing.

*Syntax: **@**param $args ['exid'] the ID of the item*
@param $args ['name'] name of the item
@param $args['number'] number of the item
Returns the example item ID as integer on success, returns false on failure and raises BAD_PARAM, NO_PERMISSION, DATABASE_ERROR

(16) delete.php
This standard function deletes an item in the database. Parameters are passed from the xaradmin/delete.php. This function returns true upon success, or raises an exception error upon failing.

Syntax: @param $args ['exid'] name of the item

Returns the example item ID as integer on success, returns false on failure and raises BAD_PARAM, NO_PERMISSION, DATABASE_ERROR

(17) getmenulinks.php
This is a utility function that passes individual menu items to the main menu.
Returns array containing the menulinks for the admin menu items.

(18) menu.php
This function generates the common admin menu configuration

(19) update.php
This standard function updates an item when the administrator wants to change an item in the database. Parameters are passed from the xaradmin/update.php. This function returns the example item upon success, or raises an exception error upon failing.

*Syntax: **@**param $args ['exid'] the ID of the item*
@param $args [‘name’] name of the item
@param $args[‘number’] number of the item

Returns the example item ID as integer on success, returns false on failure and raises BAD_PARAM, NO_PERMISSION, DATABASE_ERROR

(20) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(21) xarblocks/
This directory contains the files for generating blocks for the module.

(22) first.php
This is the code for the example block for the example module.

(23) others.php

This is the code for the example block for the example module.

(24) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(25) xarimages/
This directory contains image files that are associated with the example module.

(26) admin.gif
This is the image that is displayed in the example module administration overview.

(27) admin_generic.gif
This is an optional image that can be displayed in the example module administration overview.

(28) preferences.gif
This preferences image is used in the example module.

(29) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(30) xartemplates/
This directory contains the template files used by blocklayout for displaying pages within the example module.

(31) blocks/
This directory contains the template files used by blocklayout to display the blocks for the example module.

(32) first.xd
Templates file for the first block in the example module.

(33) firstAdmin.xd

Templates file for the first block administration page for the example module.

(34) othersAdmin.xd
Templates file for the others block administration page for the example module.

(35) example-firstblock-modify.xd
Templates file for the first block administration modify settings page.

(36) admin-delete.xd
Templates file for administration delete function.

(37) admin-main.xd
Templates file for administration main function.

(38) admin-menu.xd
Templates file for the administration menu for Example module.

(39) admin-modify.xd
Templates file for administration modify item page.

(40) admin-modifyconfig.xd
Templates file for administration modify configuration page.

(41) admin-new.xd
Templates file for administration new item page.

(42) admin-view.xd
Templates file for administration view item page.

(43) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(44) user-display.xd
Templates file for the user display page.

(45) user-main.xd

Templates file for the users main page.

(46) user-menu.xd
Templates file for the users menu.

(47) user-usermenu_icon.xd
Template file for example user menu hook.

(48) user-view.xd
Templates file for the user display items page.

(49) user-usermenu_form.xd
Templates file for example hook for user menu.

(50) xaruser/
The directory that contains all users GUI functions for the module.

(51) display.php
This standard function provides detailed information on a single item available from the module.

Syntax: @param $args an array of arguments (if called by other modules)
@param $args ['objectid'] - A generic objected (if called by other modules)
@param $args ['exid'] - The item id used for this module

(52) main.php
This standard function is the default function, and is called whenever the module is initiated without defining arguments. As such, it can be used for a number of things, most commonly, it either shoes the module menu and returns or calls whatever the module designer feels should be the default function. (This is often the view() function)

(53) user-menu.php
This standard function is used to display the user menu hook.

(54) view.php

This standard function is used to provide an overview of all the items available from the module.

(55) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(56) xaruserapi/
The directory that contains all user database functions for the module.

(57) encode_shorturl.php
The standard functions that encode module parameters into some virtual path that will be added to index.php, and decode a virtual path back to the original module parameters.

(58) countitems.php
This utility function counts the number of items held by the module.
Returns the number of items as an integer type with the number of items held by this module.

(59) decode_shorturl.php
This function extracts arguments from short urls and passes them back to the xarGetRequestInfo() api function.

(60) get.php
This standard function retrieves a specific example item from the database.

Syntax: @param $args ['exid' – Id of example item to get
Returns item array, or false on failure
Raises exceptions BAD_PARAM, NO_PERMISSION, DATABASE_ERROR

(61) getall.php
This standard function retrieves all example items from the database.

Syntax: @param $ numitems – The number of items to retrieve (default –1 = all)
@param $ startnum – Start with this item number (default 1)
Returns an array of items, or false on failure
Raises exceptions BAD_PARAM, NO_PERMISSION, DATABASE_ERROR

(62) getmenulinks.php
This utility function will pass individual menu items to the main menu.
Returns an array containing the menu links for the main menu.

(63) menu.php
This standard function will generate the common menu configuration.

(64) validateitem.php
This standard function validates argument arrays that are passed for writing to the database.

(65) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(66) getitemlinks.php
This utility function passes individual item links to other calling functions

Syntax: @param $args ['itemtype'] – Item type (optional)
@param $args ['itemids'] – array of item ids to get
Returns an array containing the itemlink(s) for the item(s)

(67) xareventapi.php
This standard function is the event handler for the system event ServerRequest
Returns bool

(68) xarinit.php

This file contains the initialization, upgrade, and delete functions for the example module. Functions include

example_init() – example_upgrade() – example_delete()

(69) xartables.php
This file contains the table definitions for the example module/

(70) example.wsdl
This file contains an example soap xml schema that connects and returns the number of items in the example database.

(71) index.html
This is a blank index.html file to prevent someone from viewing the directory contents when access this directory via a url.

(72) xaradmin.php
A depreciated file; all admin functions should be separated into individual files and placed into the xaradmin/ directory.

(73) xaradminapi.php
A depreciated file; all adminapi functions should be separated into individual files and placed into the xaradminapi/ directory.

(74) xaruser.php
A depreciated file; all user functions should be separated into individual files and placed into the xaruser/ directory.

(75) xaruserapi.php
A depreciated file; all userapi functions should be separated into individual files and placed into the xaruserapi/ directory.

(76) xarversion.php

This file contains the module ID number, author, and version information for the example module.

- **Building Your Module**

- **Make Your Initial Directory**

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

- **Copy the Module Example**

Copy over all of the files from the example module 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.

- **Code your Database Tables**

Coding your database tables requires you to edit the xartables.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_xartables()) for easy access by the Xaraya system. An annotated copy of the template xartables.php file is available in the standard Xaraya distribution as part of the Template module.

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

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

- ## Write your Initialization Functions

Module initialization functions are required for three separate actions. These actions are initialization 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 called once, although if a site administrator desires they should be able to initialize and delete a module as many times as they wish. It should be assumed that whenever these functions are called the Xaraya system has already loaded the relevant information from `xartables.php` *and it is available in the information returned by* `xarDBGetTables()`*.*

An annotated copy of the template `xarinit.php` *file is available in the standard Xaraya distribution as part of the Example solution.*

- ## Test Your Initialization Routines

Once the database structure and initialization files are in place they should be tested by using the modules administration area of your Xaraya system to test initializing 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 initialization functions are working correctly you should carry out an initialization so that work on the administration and user functions can proceed with suitable database tables in place.

- ## 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**, *gathers information from the form displayed by the first piece, and passes it as arguments to the second piece.*

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, 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 Xaraya system itself (normally in a database table) then it is an API function.

Annotated copies of the template xaruser and xaruserapi function files are available in the standard Xaraya distribution as part of the Example module.

• Test Your Administration Functions

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

• 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.

Annotated copies of the template `xaruser` *and* `xaruserapi` *function files are available in the standard Xaraya distribution as part of the Example module.*

- **Test Your User Functions**

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

- **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 `xarModDBInfoLoad()` *function if they intend to access the module's tables directly.*

An annotated copy of the template `first.php` *block is available in the standard Xaraya distribution as part of the Example module.*

- **Test Your Blocks**

Once the blocks are in place, they need testing by displaying them through the Blocks administration system. The blocks need checking against the database and the user functions to ensure that they are displaying the data correctly.

- **Document Your Module**

Documenting your module is a vital step. There are two areas your module needs 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 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 will allow for automatic parsing of the documentation by other developers who wish to use your module.

- **Packaging 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 on the Xaraya site if someone can package it in the other format.

- **Interacting With Other Modules**

• Overview

When designing your module you may well find there is functionality that you require in a module already available. Utility modules are 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.

• Hooks

Hooks are a way of adding functionality to modules without the modules themselves knowing what the functions might be. The site administrator controls the operations of hooks, 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 Xaraya format

Display of a category

Addition of an item

Deletion of an item

Transformation of item data into a standard Xaraya format

Display of an item

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 xarAPI hook functions as described below is mandatory for a compliant module.

- **Calling Hooks**

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 `xarModCallHooks()` *function. This function placed wherever a specific action is required by the item module, where the current specific actions that the hooks system is able to operate on are:*

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

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

hookobject

The object for which the hooks are 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 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* `example_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=example&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 `xarModCallHooks()` *function returns different information depending on value of hookaction. If hookaction is display then the hook will return extra output to display 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 'Example' 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->Text(xarModCallHooks('item',
                              'display',
                              $pictureid,
                              xarModURL('example',
                                        'user',
                                        'display',
                                        array('pictureid' => $pictureid))));
```

This 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 Xaraya 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 Example module, should you call an item display hook for each picture? The answer to this is somewhat dependent on the nature of your module. 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 Example module in the standard Xaraya distribution contains notes on calling hooks within an item module.

- **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 unregistered its hooks as required.

- **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 are 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 may never be called.

Hook functions must not call the `xarModCallHooks()` *function, or functions that might themselves call* `xarModCallHooks()`*. If a hook does this, it risks getting the code into an infinite loop.*

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

• Registering Hooks

Once your module has hook-capable functions in place they need to be registered on initialization of the module so that the administrator can configure their applicability, and other modules can access them through the `xarModCallHooks()` *function. This is carried out through use of the* `xarModRegisterHook()` *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 `xarModRegisterHook()` *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* `xaruser.php` *and* `xaradmin.php`*) or* ***API*** *(for functions that are in* `xaruserapi.php` *and* `xaradminapi.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 `xarModRegisterHook()` *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 Xaraya 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 (!xarModRegisterHook('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 Xaraya distribution has detailed comments on registering hooks within a utility module.

- **Un-registering Hooks**

If your module has hook-capable functions that are registered when the module is initialized they need to be unregistered when the module is deleted. This is carried out through use of the `xarModUnregisterHook()` *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 initialized.*

The `xarModUnregisterHook()` *function takes the same parameters as the* `xarModUnregisterHook()` *function.*

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

- **Function Calls**

Another way of accessing the functionality of other modules is by calling their functions directly with the `xarModFunc()` *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 pass additional arguments to the function to customize 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 Example module would be if the module

developer wanted users to be able to rate various aspects of the picture displayed such as 'use of color' 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.

-

- **Upgrading Your Module**

Add information how module upgrades interact with the installer

-

- **Notes in Developing Modules**

xarAPI is the Xaraya Application Programming Interface, a way for modules to interact with the Xaraya core without needing to access tables and internal structures directly. The API also allows for the underlying implementation details of Xaraya 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 Xaraya, which has undergone, and continues to undergo, radical changes in the core design to allow it to be faster, more secure, and more flexible.

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

- **Use xarAPI**

-

- **Security**

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

- **Variable Handling**

All variables that come in to or go out of Xaraya should be handled by the relevant `xarVar*()` *functions to ensure that they are safe. Failure to do this could result in opening security holes at the web, file-system, display, or the database layers. Full information on these functions is in the Xaraya* **API** *Guide, and examples of their use are shown throughout the example module.*

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

- **Authorization**

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

- **Reserved Variable Names**

Xaraya has a number of variables, which are reserved. These variables are not be used within modules as they can

conflict with the Xaraya core and cause unpredictable results.

The current lists 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.

- **Page Path**

All input from web pages goes through a two-stage process. The first part is displaying the information 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 are 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 Xaraya system goes through the full two-stage method of gathering and processing the information the two functions `xarSecGenAuthKey()` *and* `xarSecConfirmAuthKey()` *must be used in the appropriate places. The Example module in the standard Xaraya distribution contains a number of functions that use these* **API** *calls, and note where they are used so that developed modules will have the same level of protection against fraudulent administrator requests*

- **Output**

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

• Using Object Oriented Code

Modules written as classes is allowed, 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();
}
```

• Recommendations

If you are new to Xaraya, it is highly recommended that you familiarize yourself with Xaraya by reading the Xaraya Installation and Getting Started Guide. The guide is currently available via the **BitKeeper** *repository located on the Xaraya.com Web site. As a side note, the guide is currently a work in progress, once completed the guide will be available by means that are more accessible.*

Xaraya News Groups:

The following lists of news groups are available via news.xaraya.com. These news groups are also available via your web browser at the following Web sites.
http://www.xaraya.com

- *Ddf.public – DDF Public List*
- *Xaraya.announce – Xaraya Announcements List*
-
- *Xaraya.devel – Xaraya Member List*
- *Xaraya.bk-notices – Xaraya BitKeeper Notices*
- *Xaraya.documentation – Xaraya Documentation List*
- *Xaraya.bugs – Xaraya Bugs List*
- *Xaraya.ui – Xaraya User Interface Lists*
- *Xaraya.knowledge-base – Xaraya Knowledge Base List*
- *Xaraya.marketing – Xaraya Marketing*
- *Xaraya.patches – Xaraya Patches*
- *Xaraya.qa – Xaraya Quality Assurance*
- *Xaraya.user – Xaraya User Discussion List*

- *Xaraya.user-arabic – Xaraya Arabic User Discussion List*
- *Xaraya.user.chinese – Xaraya Chinese User List*
- *Xaraya.user.danish – Xaraya Danish User Discussion List*
- *Xaraya.user.dutch – Xaraya Dutch User Discussion List*
- *Xaraya.user.french – Xaraya French User Discussion List*
- *Xaraya.user.german – Xaraya German User Discussion List*
- *Xaraya.user.greek – Xaraya Greek User Discussion List*
- *Xaraya.users.hungarian – Xaraya Hungarian User Discussion List*
- *Xaraya.user.italian – Xaraya Italian User Discussion List*
- *Xaraya.user.polish – Xaraya Polish User Discussion List*
- *Xaraya.user.portuguese – Xaraya Portuguese User Discussion List*
- *Xaraya.user.russian – Xaraya Russian User Discussion List*
- *Xaraya.user.spanish – Xaraya Spanish User Discussion List*

Glossary

API
application program interface (API): *A formalized set of software calls and routines that can be referenced by an application program in order to access supporting network services*

BitKeeper
BitKeeper: *is a scalable configuration management system, supporting globally distributed development, disconnected operation, compressed repositories, change sets, and repositories as branches.*

Blocklayout

Blocklayout*: is the theme-rendering engine intended to give theme developers a maximum of control over the appearance and functionality of their Xaraya website. The RFC for Blocklayout is located at http://docs.xaraya.com/docs/rfcs/rfc0010.html.*

GUI

gui: *Acronym for* **graphical user interface**. *A computer environment or program that displays, or facilitates the display of, on-screen options, usually in the form of icons (pictorial symbols) or menus (lists of alphanumeric characters) by means of which users may enter commands. Note 1: Options are selected by using the appropriate hardware (e.g., mouse, designated keyboard keys, or touchpad) to move a display cursor to, or on top of, the icon or menu item of interest. The application or function so represented may then be selected (e.g., by clicking a mouse button, pressing the "enter" key, or by touching the touchpad). Note 2: Pronounced "gooey."*

LDAP

LDAP: *Abbreviation for lightweight directory access protocol. A simplified version of the X.500 standard, which version consists of a set of protocols developed for accessing information directories. [After Bahorsky]*

METADATA

Metadata*: is machine understandable information for the web. The W3C Metadata Activity addressed the combined needs of several groups for a common framework to express assertions about information on the Web, and was super ceded by the W3C Semantic Web Activity.*

MOVABLE TYPE

Movable Type*: is Six Apart's powerful, customizable publishing system, which installs on web servers to enable*

individuals or organizations to manage and update weblogs, journals, and frequently updated website content.

PHP

PHP*: is a widely used general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.*

POSTNUKE

PostNuke: *is a community, content, collaborative management system, a C3MS.*
More information is located at http://www.postnuke.com

RSS

*RDF Site Summary (***RSS***): - also referred to as Rich Site Summary - is a method of describing news or other Web content that is available for "feeding" (distribution or syndication) from an online publisher to Web users. RSS is an application of the Extensible Markup Language (XML) that adheres to the World Wide Web Consortium's Resource Description Framework (RDF). Originally developed by Netscape for its browser's Netcenter channels, the RSS specification is now available for anyone to use.*

XMLRPC

XMLRPC: *is a spec and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.*

- # Appendix A

- # Module Developers Check List

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

- *Initial*

1.Decide on the module type
2.Choose a name for your module

3.Register your module name
4.Obtain and read the Module Developers Guide
5.Obtain and read the API Documentation

- *Module Design*

1.Separate User and Administration Functions
2.Separate GUI and API functions
3.Design data tables
4.Note, which utility modules are of use
5.Note, which standard module functions apply
6.Create module security schema

- *Module Build*

1.Copy the example module directory
2.Create database tables
3.Create database initialization routines
4.Test database initialization routines
5.Write administration functions
6.Test administration functions
7.Write user functions
8.Test user functions
9.Write blocks
10.Test blocks
11.Document module API
12.Package your module

- *Module Checks*

1.No global variables used
2.No Xaraya reserved variable names used
3.No echo() or print() statements used
4.All operations protected by xarSecAuthAction()
5.All form results protected by xarSecConfirmAuthKey()
6.All form variables obtained by xarVarFetch()
7.All output is passed through transform hooks.
8.All output parsed through xarVarPrepForDisplay() or xarVarPrepForHTMLDisplay()
9.All variables in SQL queries protected by using bindvars
10.All variables in filesystem access protected by xarVarPrepForOS()
11.Calls to xarModCallHooks() in appropriate locations

Sources Cited

SearchWebServices.

SearchWebServices. 22 Sept 2003

<http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci813358,00.html>

Telecom Glossary 2k.

Telecom Glossary 2k. 29 Feb. 2001

< http://www.atis.org/tg2k/ irc.html>

W3C Technology and Society Domain.

W3C Technology and Society Domain. 05 April 2001

< http://www.w3.org/Metadata/>

Xaraya Development Team About Xaraya

John Cox. 27 April. 2003

< http://www.xaraya.com/index.php/news/c28/>

Xaraya Documentation.

Xaraya Documentation. 15 August 2004

< http://docs.xaraya.com/index.php/documentation/72>