Search ↵

Introduction
▶ Modules
▶ Third Party Modules
▼ Development
Creating Modules
Creating Themes
Creating Overlay Themes
Creating Scheduled Cron Jobs
Translations
XML-RPC Calls
▶ API
▶ Module

Creating Modules

October 1, 2023 · 40 min · Jamie Cameron | Suggest Changes

▶ Menu

▶ Table of Contents

This page should be read if you are planning to write your own Webmin module, as it explains all the requirements for creating a usable module.

It assumes that you have a working knowledge of Perl, HTML, and web application concepts. It also focuses towards the new module API in Webmin 1.460 and later.

On this page

Introduction

Webmin is designed to allow the easy addition of new modules without changing any of the existing code. A module can be thought of as something like a Photoshop plugin or iPhone application - it can be written by someone other than the developers of Webmin and distributed under a license the developer chooses.

A module should be written to administer one service or server, such as the Unix password file or the Apache web server. Some complex system functions may even be split over several modules - for example, disk partitioning, mounting disks, and disk quota management are 3 separate modules in

the standard Webmin distribution.

Modules can theoretically be written in any language. However, to make use of the Webmin API Perl version 5.8 or above should be used. A module should be written entirely in Perl, with no C functions or external binary programs. The aim is for modules to be as portable as possible across different Unix systems and CPU types.

Modules written in other languages will not be displayed using the standard Webmin UI and will not be able to call its API. For these reasons, using Perl is strongly recommended.

At their simplest, modules are really just directories of CGI

programs that Webmin's web server runs. However, there are certain rules that should be followed to make sure that they work with the Webmin API, main menu, and access control system. Even though you can just stick any existing CGI script into a module directory, this is not a good idea.

Required files

Every module has its own directory under the Webmin base directory, in which all the module's CGI programs and configuration files must be stored. For example, if the Webmin base was `/usr/libexec/webmin` , a module called foobar would be created or installed in `/usr/libexec/webmin/foobar` .

You can find this base directory by looking at the `root` entry in your `/etc/webmin/miniserv.conf` file. It will differ depending on which operating system Webmin is installed.

For a module to be displayed on the main Webmin menu, it should contain at least the following files. Only `module.info` is mandatory though.

- `module.info`
  This file contains information about the module and the operating systems it runs under. See below for details on its format.
- `images/icon.gif`
  The icon displayed on the main menu for this module. The icon should be 48x48 pixels and should use the same colour scheme as

the other icons on the main menu.

- `lang/en`
  The text strings used by this module, as explained in the **Internationalization** section of this documentation.
- `install_check.pl`
  Program that checks to see if the service or program is installed and usable, returning a non-zero value if so.

Each module name on Webmin's left menu is a link to the module directory. Thus you must have an `index.cgi` file to be displayed when the user clicks on the link. A typical module contains many `.cgi` programs that are linked to from `index.cgi`, each of which performs some function such as displaying a form or

saving inputs from a form.

When you first create a new module, it will not be in the allowed list of any Webmin user and so you will not be able to see it in the main menu. To fix this, you must first delete the file `/etc/webmin/module.infos.cache` to clear the cache of known modules. Then to make your module visible, either edit the file `/etc/webmin/webmin.acl` or use the Webmin Users module to grant yourself access.

The `module.info` file

This file contains meta-information about your module, such as its title, supported operating systems, and category. It is a text file with each line containing a name and value separated by `=`,

a format widely used by Webmin. An example `module.info` file might look like:

```
desc=Foo Web Server
os_support=*-linux
category=servers
```

Required entries are:

- `desc`
  A description for the module, such as **Foo Web Server**. This is the text that will appear on Webmin's left menu.
- `os_support`
  A space-separated list of operating systems that this module supports. The module will only be displayed on the main menu if the OS Webmin is running on is in the list or if there is no os_support line at all. Unless your module configures some service that only exists on a few

operating systems (such as X.Org), this line should be omitted instead of trying to list all of those supported by Webmin. The actual operating system codes used in this line can be seen in the third column of the `os_list.txt` file in the Webmin root directory and are the same as those that can be appended to the names of `config-` files, as explained in the Module Configuration section. To specify only a certain version of some OS, add it to the OS name after a slash. For example, a `module.info` file might contain: `os_support=redhat-linux suse-linux/15.5` . If your module supports all Linux distributions

both no other operating systems, you can use the OS code `*-linux` in this line.

- `category`
  The code for the Webmin menu category to display the module under. This will typically be one of `servers`, `system`, `net` or `hardware`.

Module library

The Webmin web server treats files with the extension `.cgi` as CGI programs, just like most other web servers. All the forms, menus, and other pages in your module will be generated by CGI programs, so knowledge of the basic concepts of CGI programming and HTML is necessary for writing a module.

All CGI programs are run with *root* privileges, which is generally necessary for them to be able to edit configuration files. In some cases your code may drop those privileges by switching to another user, for example if the module's access control settings for some Webmin user specify it.

When writing a new module, you should create a file with the same name as the module's directory, but with `-lib.pl` appended. So if your module directory was `foobar`, you should create `foobar-lib.pl`. This file will contain common functions that your module's CGI programs will call and will in turn call Webmin's initialization functions.

An example library file could look like:

```
=head1 foobar-lib.pl

Functions for managin

  foreign_require("fo
  my @sites = foobar:

=cut

BEGIN { push(@INC, ".
use WebminCore;
init_config();

=head2 get_foobar_con

Returns the Foobar We

=cut
sub get_foobar_config
{
my $lref = &read_file
my @rv;
my $lnum = 0;
foreach my $line (@$l
    my ($n, $v) = spl
    if ($n) {
      push(@rv, { 'na
      }
    $lnum++;
    }
return @rv;
}
```

The first two lines being in the core Webmin API, which exports numerous functions for

parameter parsing, HTML generation, user management, reading and writing config files, and much more. These are fully documented below.

The `init_config();` line calls a Webmin API function to initialize the module's environment. This sets several variables in your module's package, such as the `%config` hash containing the module's current configuration. It also checks if the current user is allowed to access this module, blocks links from un-trusted referers, and much more. See the documentation for `init_config` for a full list of the variables it exports.

Finally, the `get_foobar_config` sub is just an example of a function your

module's CGI scripts might call to read the config file for the server it manages. In a good module design, all access to configuration files is done via functions like this, rather than directly in CGI scripts. This way your functions can be called from other modules and code duplication is reduced.

Note how the file begins with a POD format documentation comment explaining what it does and giving a short snippet of code showing how another module could call this one. Also, individual functions should have POD format comments, as you can see on `get_foobar_config` . This allows other developers to use a command like `perldoc foobar-lib.pl` to see all the documentation.

Module CGI scripts

CGIs are responsible for generating the HTML for pages and forms that the user interacts with. Wherever possible they should use the Webmin UI functions to generate headers, forms, inputs, tables, and so on. This way the UI is consistent and can be overridden by custom themes.

The module's `index.cgi` file might contain code like:

```
#!/usr/bin/perl

require 'foobar-lib.p
ui_print_header(undef

$conf = get_foobar_cc
$dir = find($conf, "r
print &text('index_rc

ui_print_footer("/",
```

The first line is standard for all Perl scripts and must

match the path to Perl on your system. This can be found in the `/etc/webmin/perl-path` file.

The line `require 'foobar-lib.pl';` brings in the module's function library described above and calls Webmin's `init_config` initialization function.

The page's HTML header is generated by the call to `ui_print_header`. The most important parameter is `$text{'index_title'}`, which refers to the `%text` hash that is loaded from the module's `lang/en` file, described below.

The next two lines are calls to functions from the example module's library. The `print` statement output's some HTML, using the

Webmin API function `text` to substitute a programmatically-generated string into a message.

Finally, the call to `ui_print_footer` generates a link back to Webmin's main menu, if needed.

Language files

Webmin has an internationalization system based on the contents of files in each module's `lang` sub-directory. The global default language is English, so each module must have a `lang/en` file containing US English messages used by its CGI scripts. It can also have files for other languages, like `de` form German or `fr` for French. Each file contains lines of text, one per message, formatted like:

```
index_title=Foobar We
index_root=The root d
```

When your code calls `init_config`, this file is read into the module-level hash `%text`. In addition, any strings defined in the appropriate files under Webmin's top-level `lang` directory are also read. These contain useful messages codes like `save`, `delete`, and `index`.

The example `index_root` line contains a placeholder `$1`, which will be replaced by the `text` function with its second parameter. Strings can contain multiple placeholders like this, using the codes `$2`, `$3`, and so on.

Module configuration

Almost all modules have a set of user-editable configuration parameters, available in the `%config` hash which is set by the `init_config function` . When Webmin or a module is installed, a configuration file appropriate for the chosen operating system is copied from the module directory to the Webmin configuration directory for that module, typically something like `/etc/webmin/foobar/config` . It is this file that is read by `init_config` .

In general, module configuration settings are for things that the user may want to edit. These include paths to other config files that the module manages, display preferences, and options that control behavior.

Making the locations of programs and other files editable makes your module more flexible and able to support systems on which config files are in different locations.

In most cases, your module only needs to include a single file named `config` in its base directory, which is copied to `/etc/webmin` at install time. If you are writing a module yourself from scratch, you will need to do this manually with commands like:

```
cd /usr/libexec/webmi
mkdir /etc/webmin/foo
cp config /etc/webmin
```

An example `config` file for your module might contain:

```
foobar_conf=/etc/foob
sort_mode=0
```

In other cases, you might want the default configuration to differ depending on the operating system. For example, Apache is installed in a different place in almost every operating system, but its config always files have the same format. Webmin's core Apache module contains files named like `config-redhat-linux` and `config-solaris`, which define the locations for `httpd.conf` and `apachectl`. At install time the appropriate file is copied to `/etc/webmin/apache/config` and values from it are then used by the Apache Webmin module to find other config files.

User configuration editing

Every module with a `config` file should also have a meta-config file named `config.info` that tells the core Webmin API what values and options are allowed. When a user clicks on a module's **Module Config** link, the page that appears is driven by the contents of the module's `config.info` file. A sample file looks like:

```
foobar_conf=Path to F
sort_mode=Sort users
```

Like most Webmin files, `config.info` is a text file with lines in **name=value** format. Each **name** must match an entry in the `config` file.

The right-hand side is a comma-separated list, with the following elements:

- A human-readable description of this configurable setting.
- A numeric type code that determines how the value can be edited.
- An option comma-separated list of type parameters. Their number and format depends on the type code.

Type code zero is most common and is used for free-text fields. The other possible type codes are:

1. **One of many**. The user can choose one of several options. For this type, the rest of the line is a comma-separated list of **value/display** pairs. The **value** part of each pair is what gets stored in the config file, while the **display** part is what is shown to the user.

2. **Many of many**. The user can choose zero of more of several options. Available options are specified in the same way as type 2.
3. **Optional free text**. The user can either select the default option or enter some value. The rest of the line is the description of the default option (typically something like **None** or **Default mode**)
4. **One of many**. The same as type 1, but uses a menu instead of a row of radio buttons
5. **Unix user**. Displays a selector for a user from the host Webmin is running on.
6. **Unix group**. Displays a group selector from the host Webmin is running on.

7. **Directory**. Like the free text input, but with a directory chooser next to it.
8. **File**. Like the free text input, but with a file chooser next to it.
9. **Multiline free text**. The first **value** after the type is the width of the input and the second the height.
10. **Like type 1**, but with an additional option for entering free text of the user's choice.
11. A parameter of this type does not allow the user to enter anything, but instead puts a section header row containing the description into the configuration form at this point.
12. A field for entering a password, without actually displaying the current value.

Not every configurable parameter needs an entry in `config.info` - only those that the user may want to edit.

Global configuration

The hash `%gconfig` contains global configuration options, typically from the file `/etc/webmin/config`. Some useful entries are:

- `os_type`
  A code for the operating system type detected at install time, such as `debian-linux` or `redhat-linux`.
- `os_version`
  Webmin's internal code for the OS version, such as `5.9`.
- `path`
  The Unix path for this operating system, as a colon separated list of

directories. This is also available in `$ENV{'PATH'}`, as thus to any programs that you module runs.

User interface

Webmin's API contains a large number of functions for generating forms, tables, inputs, and tabs. While a module can create its own HTML with simple Perl print statements, using the API is both easier and produces a more consistent look.

Some example code for creating a form might look like:

```
print ui_form_start("

print ui_table_row($t
    ui_textbox("usern

print ui_table_row($t
    ui_password("pass

print ui_form_end([ [
```

To create a table, you can use code like:

```
print ui_columns_star
foreach my $u (@users
    print ui_columns_
        ui_link("edit
        $u->{'real'}
    }
print ui_columns_end(
```

Some other good guidelines for module user interfaces are:

- Try to follow the layout of core modules. For example, your module's main page `index.cgi` might display a table of objects, each of which contains a link to `edit.cgi`. This page in turn shows a form for editing or creating a user and submits to a script called `save.cgi` to update the underlying config files.
- Don't use Flash or Java unless there is

no other alternative. Most dynamic UIs can be created using JavaScript in modern browsers.

Design goals

A typical Webmin module is written to configure some Unix service, such as Apache, Squid or NFS exports. Most Unix servers are normally configured by editing some text file, which may have a complex format. Any Webmin module that modifies some configuration file must be able to parse all possible options in such a configuration file - even if not all options are presented to the user.

No module should ever corrupt a service configuration file or remove options that it does not understand. Modules should be

able to parse any valid configuration without requiring special comments or a special format. If your module cannot deal with some option in a file, it should be left alone.

Webmin modules should be designed to be easy for novices to use, but still allow the user to do almost everything that could be done by editing the configuration file directly. However, in some cases configurations options will exist that very few users will need to edit or that do not lend themselves to be edited through a GUI. These kind of settings should be left out of your Webmin module if they would clutter up the user interface with' their presence.

Online help

Webmin has support for context-sensitive help, both for an entire page or for individual elements. The `hlink` function outputs HTML for a link that displays a given help page. Help pages are stored in the `help` subdirectory under the module directory and are named simply `page.html` for those in English. So a call to `hlink` like:

```
print ui_table_row(hl
    ui_textbox("usern
```

... would output a link to display the help page in the file `help/username.html` under the module's base directory. This could contain:

```
<header>Foobar Userna

Enter the name of a l

<footer>
```

This file is basically regular HTML, except for the special `<header>` tag which must contain the help page's title.

If the **help** parameter to the `ui_print_header` function is set, a link labeled **Help** to the specified help page is included in the heading. This can be useful if you have created some documentation that explains what the entire page does in general, instead of or as well as documenting fields individually. The same rules about help HTML file selection apply.

Even though online help is not mandatory (or even common) in Webmin modules, it can be useful to provide additional information to users about what a field really means or what

the purpose of a page is. In many cases inputs are not self-explanatory and need additional documentation, so why not make it available from the page itself?

Webmin modules can support multiple languages through the use of alternative translation files in the lang subdirectory. Help pages can exist if more than one language as well, by creating files named like `page.language.html` in the help subdirectory. If such a file exists, it will be used in preference to `page.html` , which is assumed to be in English. For example, to add a Greek version of an existing `name.html` page you would need to create `name.el.html` .

Module packaging

The Webmin Configuration module allows the user to add a new module to their existing setup. Modules must be packaged as a compressed Unix TAR file containing one or more modules. Each module in the TAR file must have all its files in one subdirectory.

To create such a package, you could use commands like:

```
cd /usr/libexec/webmi
tar cvzf /tmp/foomod.
```

The standard extension for Webmin modules is `.wbm.gz` or just `.wbm` if the tar file is not compressed. For themes the extension is usually `.wbt.gz` and for Usermin modules it is `.ubm.gz` .

Webmin modules can also be packaged as RPMs, which are suitable for installing on servers on which the RPM version of Webmin itself is already installed. You can download a script called `makemodulerpm.pl` that can package up a module directory into an RPM by creating the spec file automatically. It will place the resulting RPM file into the `/usr/src/redhat/RPMS/noarch` directory. The RPM name is always `wbm-` followed by the module's directory name or `wbt-` for themes.

Similarly, you can create a Debian package of a module using `makemoduledeb.pl` file. The resulting `.deb` file is placed in the `/tmp` directory.

The package name is always `webmin-` followed by the directory name, for both modules and themes.

Example module

The best way to show what a Webmin module should look like is via an example. You can install a demo module for the imaginary Foobar Webserver by following these steps:

- Login to Webmin as *root* and go to **Webmin → Webmin Configuration → Webmin Modules**
- Select the **From HTTP or FTP URL** option and enter the URL `http://download.webmin.com/download/modules/foobar.wbm.gz` into the adjacent text box

- Click the **Install Module** button

You should now be able to find the **Foobar Webserver** module under the **Servers** category. Its source code is in the `foobar` directory under the Webmin root.

The main page of this module shows a table of websites, with a link to add a new one. Adding or editing a site brings up a separate form for entering its details. This kind of layout is typical in Webmin and should be copied (where appropriate) in your own modules.

The Webmin API

The full API available to modules is documented on the Webmin API page. This covers both the core API and that exported by other modules. You

can call functions from other modules with code like:

```
foreign_require("user
@users = useradmin::l
foreach my $u (@users
  print $u->{'user'},
  }
```

Advanced concepts

Module Access Control

Webmin supports a standard method for restricting which features of a module a user can access. For example, the Apache module allows a Webmin user to be restricted to managing selected virtual servers, and the BIND module allows user to be limited to editing records only in certain domains.

This kind of detailed access control is separate from the first

level ACLs that control which users have access to which modules. As long as your module calls `init_config` , the Webmin API will automatically block users who do not have access to the entire module.

Module access control options are set in the Webmin Users module by clicking on a username and then on the name of a module. The options available are generated by code from the module itself (except for the **Can edit module configuration?** option, which is always present). When the user clicks on **Save** the form parameters are also parsed by code from the module being configured, before being saved in the Webmin configuration directory.

A module wanting to use access control must contain a file called `acl_security.pl` in its directory. This file must contain two Perl functions:

- `acl_security_form(acl)` This function takes a reference to a hash containing the current ACL options for this user, and must output HTML for form inputs to edit those ACL options. You must use the `ui_table_row` function to format your output.
- `acl_security_save(acl, inputs)` . This function must fill in the given hash reference with values from the form created by `acl_security_form` . Form inputs are available in the second parameter to

the function, which is in the same format as the `%in` hash created by the `ReadParse` function.

An example `acl_security.pl` file looks like:

```
require "foomod-lib.p

sub acl_security_form
{
my ($access) = @_;
print ui_table_row("A
  ui_yesno_radio("cre
}

sub acl_security_save
{
my ($access, $in) = @
$access->{'create'} =
}
```

Because these functions are called in the context of your module, the `acl_security.pl` file can require the common functions file used by other CGI programs in the module. This gives you access to all the

standard Webmin functions, and allows you to provide more meaningful inputs. For example, when setting ACL options for the Apache module a list of virtual servers from the Apache configuration is displayed for the user to select from.

If a user has not yet had any ACL options set for a module, a default set of options will be used. These are read from the file `defaultacl` in the module directory, which must contain **name=value** pairs one per line. These options should allow the user to do anything, so that the admin or master Webmin user is not restricted by default.

To actually enforced the chosen ACL options for each user, your module programs must use the

`get_module_acl` function to get the ACL for the current user, and then verify that each action is allowed. When called with no parameters this function will return a hash containing the options set for the current user in the current module, which is almost always what you want. For example:

```
#!/usr/bin/perl
require 'foobar-lib.p
%access = &get_module
$access{'create'} ||
```

When designing a module that some users will have limited access to, remember the user can enter **any** URL, not just those that you link to. For example, just doing ACL checking in the program that displays a form is not enough - the program that processing the form should do all the same

checks as well. Similarly, CGI parameters should never be trusted, even hidden parameters that cannot normally be input by the user.

User and Group Update Notification

Webmin has a feature that allows the Users and Groups module to notify other modules when a Unix user or group is added, updated or deleted. This can be useful if your module deals with additional information that is associated with users. For example, the Disk Quotas module sets default quotas when new users are created, and the Samba Windows File Sharing module keeps the Samba password file in sync with the Unix user list.

To have your module notified when a user is

added, updated or deleted you must create a Perl script called `useradmin_update.pl` in your module directory. This file must contain three functions:

- `useradmin_create_user(user)` This function is called when a new Unix user is created. The `user` parameter is a hash containing the details of the new user, described in more detail below.
- `useradmin_modify_user(user, olduser)` This function is called when an existing Unix user is modified in any way. The `user` parameter is a hash containing the new details of the user, and `olduser` the details of the user before he was modified.

- `useradmin_delete_user(user)` This function is called when a Unix user is deleted. Like the other functions, the `user` hash contains the user's details.

The hash reference passed to each of the three functions has the following keys:

- `user` - The Unix username
- `pass` - Encrypted password, perhaps using MD5 or DES
- `uid` - User's ID
- `gid` - User's primary group's ID
- `real` - Real name for the user. May also contain office phone, home phone and office location, comma-separated
- `home` - User's home directory
- `shell` - Shell command to run

when the user logs in

- `passmode` - Set to 0 if the user has no password, 1 for a lock password, 2 for a pre-encrypted password, 3 if a new password was entered, or 4 if the password was not changed
- `plainpass` - The user's plain-text password, if available

In addition, if the system supports shadow passwords it may also have the keys:

- `change` - Days since 1970 the password was last changed
- `min` - Days before password may be changed
- `max` - Days after which password must be changed

- `warn` - Days before password is to expire that user is warned
- `inactive` - Days after password expires that account is disabled
- `expire` - Days since Jan 1, 1970 that account is disabled

When your functions are called, they will be in the context of your module. This means that your `useradmin_update.pl` script can require the file of common functions used by other CGI programs. The functions can perform any action you like in order to update other configuration files or whatever, but should not generate any output on `STDOUT`, or take too long to execute. An example `useradmin_update.pl` might look like:

```
do 'foobar-lib.pl';

sub useradmin_create_
{
  my ($user) = @_;
  my $lref = &read_fi
  push(@$lref, "$user
  &flush_file_lines($
}
```

Groups update information can also be passed to your module if the `useradmin_update.pl` script contains the functions `useradmin_create_group`, `useradmin_modify_group` and `useradmin_delete_group`. These take group hash references as parameters, which contain the keys:

- `group` - The group name
- `pass` - Rarely-used encrypted password, in DES or MD5 format
- `gid` - Unix ID for the group

- `members` - A comma-separated list of secondary group members

Internationalisation

Webmin provides module writers with functions for generating different text and messages depending on the language selected by the user. Each module that wishes to use this feature should have a subdirectory called `lang` which contains a translation file for each language supported. Each line of a translation file defines a message in that language in the format **messagecode=Message in this language**.

The default language for Webmin is English (code `en`), so every module should have at least a file called lang/en. If any other

language is missing a message, the English one will be used instead. Check the file `lang_list.txt` for all the languages currently supported and their codes. To change the current language, go into the Webmin Configuration module and click on the **Language** icon.

When your module calls the `init_config` function, all the messages from the appropriate translation file will be read into the hash `%text`. Thus instead of generating hard-coded text like this:

```
print "Click here to
```

Your module should use the `%text` hash like so:

```
print $text{'index_st
```

The `lang/en` file would then have a line like:

```
index_startmsg=Click
```

Messages from the appropriate file in the top-level `lang` directory are also included in `%text` . Several useful messages such as `save` , `delete` and `create` are thus available to every module.

In some cases, you may want to include some variable text in a message. Because the position of the variable may differ depending on the language used, message strings can include place-markers like `$1` , `$2` or `$3` . The function text should be used to replace these place-markers with actual values like so:

```
print &text('servercc
```

Your module's module.info file can also support multiple languages by adding a line with the key =desc=*code* for each language, where **code** is the language code. So the German description for your module would be specified with a link like:

```
desc_de=Verwalten von
```

You can also have a separate `config.info` file for each language, whose filename has the language code appended. So the file for German would be named `config.info.de`, and might contain the contents:

```
users_file=Die Benutz
groups_file=Gruppen-D
```

```
show_groups=Details a
```

Help files can also be translated for each language, by creating separate files with the same prefixes as the English help, but with a language code before the `.html` extension. So the introductory help page for our module in German might be named `intro.de.html` .

In all cases, if there is no translation for the user's chosen language then the default (English) will be used instead.

File Locking

Webmin's API has several simple functions for locking files to prevent multiple programs from writing to them at the same time. Module programmers should make use of these

functions in order to prevent the corruption or overwriting of configuration files in cases where two users are using the same module at the same time.

Locking is done by the function `lock_file`, which takes the name of a file as a parameter and obtains and exclusive lock on that file by creating a file with the same name but with `.lock` appended. Similarly, the function `unlock_file` removes the lock on the file given as a parameter. Because the `.lock` file stores the PID of the process that locked the file, any locks a CGI program holds will be automatically released when it exits. However, it is recommended that locks be properly released by calling `unlock_file` or

`unlock_all_files`

before exiting.

The following code shows how the locking functions might be used:

```
lock_file("/etc/somet
open(CONF, ">>/etc/sc
print CONF "some new
close(CONF);
unlock_file("/etc/som
```

Locking should be done as soon as possible in the CGI program, ideally before reading the file to be changed and definitely before writing to it. Files can and should be locked during creation and deletion as well, as should directories and symbolic links before creation or removal. While this is not really necessary to prevent file corruption, it does make the logging of file changes performed by the program more complete, as explained below.

Many other programs also use `.lock` files for the same purpose, but most do not put their process ID in the file. If the `lock_file` function encounters a lock like this, it will wait until it is completely removed before obtaining its own lock, as there is no way to tell if the original process is still running or not.

If you want to just read from a file while being sure that no other process is corrupting it by writing to it, the `lock_file` function takes an optional second parameter that can be set to 1 to indicate a read-only lock. This will prevent other Webmin processes from writing to the same file, but will not block read locks by other scripts.

Safe File Writes

If your module writes to critical system configuration files, you should use IO functions built into the Webmin API instead of Perl's standard `open` function. These protect files from problems like the failure of a script part way through writing a file, lack of disk space, or un-expected termination.

To open a file for writing safely, use the `open_tempfile` function. This writes to a temporary file in the same directory until it is closed with `close_tempfile` , at which point the target file is over-written. For example:

```
open_tempfile(CONFIG,
print_tempfile(CONFIG
close_tempfile(CONFIG
```

The `print_tempfile` function behaves like Perl's built-in `print` ,

but immediately calls `error` to terminate the script if the write fails due to lack of disk space or some other reason.

Functions in the Webmin API that write to files like `flush_file_lines`, `write_file` and `replace_file_line` already call the safe file IO functions internally.

Action Logging

Webmin has support for detailed logging by CGI programs of the actions performed by users for later viewing in the **Webmin Actions Log** module. Logs are also written to the file `/var/webmin/miniserv.log`, this does not contain the information required to work out exactly what each Webmin user had been doing. To improve on this, Webmin now logs detailed information to

the file `/var/webmin/webmin.log` and optionally to files in the directory `/var/webmin/diffs` . Note that nothing will be recorded in this file if logging is not enabled in the Webmin Configuration module.

The function `webmin_log` should be called by CGI programs after they have successfully completed all processing and file updates. The parameters taken by the function are:

- `action` - A short code for the action being performed, like “create”
- `type` - A code for the type of object the action is performed to, like “user”
- `object` - A short name for the object, like “joe” if the Unix

user “joe” was just created

- `params` - A hash ref of additional information about the action
- `module` - Name of the module in which the action was performed, which defaults to the current module
- `host` - Remote host on which the action was performed. You should never need to set this (or the following two parameters), as they are used only for remote Webmin logging
- `script-on-host` - Script name like `create_user.cgi` on the host the action was performed on
- `client-ip` - IP address of the browser that performed the action

All of these parameters can contain any information you want, as they are merely logged to the actions log file and not interpreted by webmin_log in any way. For example, a module might call the function like this:

```
lock_file("/etc/foo.u
open(USERS, ">>/etc/f
print USERS "$in{'use
close(USERS);
unlock_file("/etc/foo
webmin_log("create",
```

Because the raw log files are not easy to understand, Webmin also provides support for converting detailed action logs into human-readable format. The **Webmin Actions Log** `module` makes use of a Perl function in the file `log_parser.pl` in each module's subdirectory to convert logs records from that module into a readable message.

This file must contain the function `parse_webmin_log` , which is called once for each log record for this module. It will be called with the following parameters:

- `user` - The Webmin user who run the program that generated this log record.
- `script` - The filename of the CGI script that generated this log, without the directory
- `action` - Whatever was passed as the action parameter to `webmin_log` to create this log record
- `type` - Whatever was passed as the type parameter to `webmin_log`
- `object` - Whatever was passed as the object parameter to `webmin_log`

- `parameters` - A reference to a hash the same as the one passed to `webmin_log`
- `long` - If non-zero, this indicates that the function is being called to create the description for the **Action Details** page, and thus can return a longer message than normal. You can ignore this if you like.

The function should return a text string based on the parameters passed to it that converts them into a readable description for the user. For example, your `log_parser.pl` file might look like:

```
require 'foobar-lib.p

sub parse_webmin_log
{
my ($user, $script, $
if ($action eq 'creat
    return &text('log
    }
```

```
elsif ($action eq 'de
    return &text('log
    }
else {
    return undef;
    }
}
```

Because the `log_parser.pl` file is read and executed in a similar way to how the `acl_security.pl` file

Webmin Users module, it can require the module's own library of functions just like any module CGI program would. This means that the text function and `%text` hash are available for accessing the module's translated text strings, as in the example above.

Webmin can also be configured to record exactly what file changes have been made by each CGI program before calling `webmin_log` . Under **Logging** in the Webmin

Configuration module is a checkbox labeled **Log changes made to files by each action** which when enabled will cause the `webmin_log` function to use the `diff` command to find changes made to any file locked by each program.

When logging of file changes is enabled, the Action Details page in the actions log module will show the diffs for all files updates, creations and deletions by the chosen action. If locking of directories and symbolic links is done as well, it will show their creations and modifications too.

As well as having their file changes logged, programs can also use the common functions `system_logged` , `kill_logged` and `rename_logged` which

take the same parameters as the Perl `system` , `kill` and `rename` functions, but also record the event for viewing on the **Action Details** page. There is also a `backquote_logged` function which works similar to the Perl backquote operator (it takes a command and executes it, returning the output), but also logs the command. If these functions are used they must be called before `webmin_log` for the logging to be actually recorded, as in this example:

```
if ($pid) {
    kill_logged('TERM
    }
else {
    system_logged("/e
    }
webmin_log("stop");
```

Pre and Post Install Scripts

Webmin allows modules to define scripts that will be run after a module is installed and before it is un-installed. If your module contains a file called `postinstall.pl`, the Perl function `module_install` in this file will be called after the install of your module is complete. Because it is executed in the module's directory, it can make use of the common functions library, like so:

```
require 'foobar-lib.p

sub module_install
{
if (!-r "$config_dire
    copy_source_dest(
    }
}
```

The function will be called when a module is installed from the Webmin Configuration or Cluster Webmin

Servers modules, when a module RPM or Debian package is installed, or when the `install-module.pl` command is used. It will also be called when your module is upgraded or when Webmin is upgraded, so make sure it doesn't over-write.

Similarly, if your module contains a file called `uninstall.pl`, the Perl function `module_uninstall` in that file will be called just before the module is deleted. This can happen when it is deleted using the Webmin Users or Cluster Webmin Servers modules, or when the entire of Webmin is uninstalled. The uninstall function should clean up any configuration that will no longer work when the module is uninstalled, such as

Cron jobs that reference scripts in the module.

Installed Checks

Webmin module writers can call the API function `foreign_installed` to check if the server or service managed by some other module is installed on the system. If you are writing a module that manages some server, you can add a file to your module's directory that provides this information to callers. In addition, this determines if your module appears under **Un-used Modules** on the left menu.

This is done by creating a script called `install_check.pl` that contains the single Perl function `is_installed` . This function takes a mode parameter with the

same meaning as the parameter passed to `foreign_installed`, and must interpret it in the same way. Because most modules don't require an extra level of configuration before use, your function can just return 0 if the server is not installed, or **mode + 1** if it is.

This example code shows how an `is_installed` function might be written:

```
do 'foobar-lib.pl';

sub is_installed
{
my $mode = $_[0];
if (!-r $config{'foo_
        return 0;
        }
else {
        return $mode
        }
}
```

Functions in Other Modules

The standard Webmin modules contain a vast

number of useful functions for parsing and manipulating the configuration files for Apache Webserver, BIND DNS Server, Users and Groups and so on. If your module needs to configure these servers as well in some way, it makes sense to make use of existing functions in the standard modules.

Because the standard modules have typically already been configured with the correct paths for files like `httpd.conf` and `squid.conf`, their functions will use those paths when you call them to read and write configuration files. The actual `%config` settings for another module can also be accessed, so that your module knows what commands to use to apply changes to or

start some server like Apache or Squid.

When you first load the library for some other module with the `foreign_require` function, it is actually executed in a separate Perl module namespace. All of your module's CGI programs and its library will be in the their own namespace, but other foreign module's functions will be put in a namespace with the same name as the Webmin module. This means that you can call those functions with code like `useradmin::list_users()`, and access global variables like `$useradmin::config{'passwd_file'}`. This Perl namespace separation ensures that functions and globals with the same names can exist in both your and the

foreign module, without any clashes. Some things are shared between all modules though, such as caches used by `get_system_hostname`, `load_language`, `read_file_cached` and `get_all_module_infos`, so that loading the library of a new module with foreign_require is not too slow.

Documentation on functions available in other modules can be found on the Webmin API page.

Remote Procedure Calls

Webmin has several API functions for executing code on remote Webmin servers. They are used by some of the standard modules (such as those in the **Cluster** category) to control multiple servers

from a single interface, and may be useful in your own modules as well. These functions, all of which have names starting with `remote_`, let you call functions, evaluation Perl code, and transfer data to and from other system running Webmin.

Before a “master” server can make RPC calls to a remote host, it must be registered in the **Webmin Servers Index** module on the master system. The **Link type** field must be set to **Login via Webmin** and a username and password entered. The user specified should be *root* or *admin*, as others are not by default allowed to accept RPC calls.

RPC is usually used to call functions in other modules on a remote

system, or common functions. This is done with the `remote_foreign_call` function, but before it can be used `remote_foreign_require` must be called to load the library for the module that you want to call. This is very similar to calling functions in other local modules with the `foreign` functions, explained above.

A piece of code that edits a user on a remote system might look like:

```
$server = "www.exampl
$user = "joe";
remote_foreign_requir
@users = remote_forei
($joe) = grep { $_->{
if ($joe) {
    $joe->{'real'} =
    &remote_foreign_c
    }
```

Of course, you need to be familiar with the available functions in other modules, and

also to be sure that the module that you want to call is actually installed and of the right version.

All parameters passed to remote functions are converted to a serialized text form for transfer to the remote server, and any return value is also sent back in serialized form. The API functions `serialize_variable` and `unserialize_variable` are used, but the process is hidden from both the caller and the remote function - they only see scalars and references in their original format. One thing to look out for is circular references though - trying to send a structure that contains links to itself (such as a doubly-linked list) will fail due to the shortcomings of the

`serialize_variable` function. Also, try to avoid using extremely large parameters, such as strings over 1 MB in size, as serialization may make them massive.

Parameters that are references to hashes, arrays or scalars that would normally be filled in by the function will not be transferred properly. For example, the `read_file` function normally fills in the hash referenced by its second argument with the contents of a file. This will not work when it is called remotely, as all parameters and anything that they refer to are 'copied' to the other system.

The `remote_eval` function can be used to execute an arbitrary block of Perl code on a remote system, which

allows you to do things that calls to remote functions cannot. It is the only way to call native Perl functions such as unlink, to read and write arbitrary format files, set global variables and properly call functions that set their parameters. Whatever the Perl code evaluates to will be sent back returned by this function. This example shows `remote_eval` in use:

```
$data = &remote_eval(
    "rename('/etc/foo
    "local \%data;\n"
    "&read_file('/etc
    "return \\%data;\
&write_file('/etc/foo
```

As you can see, proper quoting is necessary when constructing the Perl code string, so that any variable symbols (such as `$`, `%` and `@`) are escape, as is the \ character. The second **module** parameter to

`remote_eval` can be set to `undef`, which indicates that the code should be executed in the global Webmin context, rather than in any module's.

The functions `remote_read` and `remote_write` can be used to transfer the contents of an entire file between the master and remote systems. They are must faster than reading in the file and encoding it for use in the `remote_foreign_call` or `remote_eval functions`, as the file is transferred un-encoded over a separate TCP connection.

If your module makes RPC calls, you may want the user to select a system to make calls to from a menu. A list of the names of all those available can be

obtained from the Webmin Servers Index module with code like this:

```
foreign_require("serv
@allservers = servers
@rpcservers = map { $
```

In addition, all of the `remote` functions will accept undef for the **server** parameter. This indicates that the local system should be used, which never needs to be defined in the Webmin Servers Index module. This is how all of the Cluster category modules can include the **this server** option in their lists of hosts to manage.

Creating Usermin Modules

Usermin has a very similar architecture to Webmin, and so its modules have an almost identical design to Webmin modules. The main difference is

that Usermin is designed to be used by any Unix user on a server to perform tasks that they could perform from he command line. Any third-party Usermin Modules should be written with this in mind.

By default, module CGI programs are run as *root*, just like in Webmin. This is necessary because some tasks (like changing passwords) can only be done as *root*. However, most Usermin modules do not need super-user privileges and so should call the `switch_to_remote_user` API function just after calling `init_config`, in order to lower privileges to those of the logged-in user.

Usermin module can have global

configuration variables that are initially set from the `config` files in the module directory, and are available in `%config`. However, these variables are never editable by the user - they can only be set in the Usermin Configuration module in Webmin.

Per-user configurable options are supported though, using a different mechanism. When the standard `create_user_config_dirs` function is called, the global hash `%userconfig` will be filled with values from the following sources, with later sources overriding earlier ones:

- The `defaultuconfig` file in the module directory This should contain the default options for this module for all users,

to be used if no other settings are made by the user or system administrator.

- The file `defaultuconfig` in the module's directory under `/etc/usermin`. This contains defaults for the module on this system, as set by the system administrator using the second form in the **Usermin Module Configuration** page feature in the **Usermin Configuration** Webmin module.
- The file `config` in the modules' directory in `.usermin` under the user's home directory. This contains options chosen by users themselves.

The editors for the system-wide and per-user configuration variables are defined by the `uconfig.info` file in the module directory. This file has the exact same format as the `config.info` file used for Webmin and Usermin global configuration, explained elsewhere in this document.

If you create your own Usermin module, it should be packaged in exactly the same way as a Webmin module (as a .tar or .tar.gz file). However, the module.info file must contain the line `usermin=1` so that it cannot be installed into Webmin where it would not work properly.

If your module needs to store additional data in the user's `.usermin` directory, it should call the

`create_user_config_dirs` API function first to ensure that directory exists. This in turn sets the `$user_config_directory` and `$user_module_config_directory` global variables, which contain paths to the `.usermin` directory and its per-module sub-directory.

© 1997 - 2024 Webmin