SWSOFT™

PLESK™

SWsoft, Inc.

Plesk Modules

Programming Guide

(Revision 1.2)

SWSOFT™

ISBN: N/A
SWsoft, Inc.
13755 Sunrise Valley Drive
Suite 325
Herndon
VA 20171 USA
Phone: +1 (703) 815 5670
Fax: +1 (703) 815 5675

Contents

CHAPTER 1

Preface

In This Chapter

Documentation Conventions

Before you start using this guide, it is important to understand the documentation conventions used in it.

Typographical Conventions

The following kinds of formatting in the text identify special information.

Formatting convention	Type of Information	Example
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.	Go to the QoS tab.
	Titles of chapters, sections, and subsections.	Read the Basic Administration chapter.
Italics	Used to emphasize the importance of a point, to introduce a term or to designate a command line placeholder, which is to be replaced with a real name or value.	The system supports the so called *wildcard character* search.
`Monospace`	The names of commands, files, and directories.	The license file is located in the `httpdocs/common/licenses` directory.
Preformatted	On-screen computer output in your command-line sessions; source code in XML, C++, or other programming languages.	`# ls –al /files` `total 14470`
Preformatted Bold	What you type, contrasted with on-screen computer output.	`# cd /root/rpms/php`
CAPITALS	Names of keys on the keyboard.	SHIFT, CTRL, ALT

KEY+KEY	Key combinations for which the user must press and hold down one key and then press another.	CTRL+P, ALT+F4

Feedback

If you spot a typo in this guide, or if you have thought of a way to make this guide better, we would love to hear from you!

If you have a suggestion for improving the documentation (or any other relevant comments), try to be as specific as possible when formulating it. If you have found an error, please include the chapter/section/subsection name and some of the surrounding text so that we could find it easily.

Please submit a report by e-mail to userdocs@swsoft.com.

CHAPTER 2

Plesk Modules

This guide is written for the programmers who would like to implement a kind of a plug-in for Plesk for UNIX/Linux.

- Section *Basics* presents the plug-in technology of Plesk. This part of documentation shows the difference between various Plesk-specific extension techniques, explains the tenets of Plesk modules architecture, and touches upon the facilities provided by Plesk for better integration with Plesk modules.
- Section *Creating Modules* can serve as a guide on creating Plesk modules from scratch. This section sets the programming task, gives step-by-step instructions for implementing all parts of the module's code, and explains how to create a distribution package ready for commercial use.
- Section *API Reference* presents the Modules API that can be used for creating a module fully compliant with Plesk.

In This Chapter

Basics

This chapter states the concepts of Plesk plug-in technology known as *Plesk Modules*:

- *Overview* shows the purpose of modules in Plesk architecture, explains the difference between modules and other ways of extending the logic of Plesk, and shows why modules are more useful for the customer.
- Section *How a Module Builds into Plesk Architecture* gives a common idea of what a module presents, how it is structured, and how this structure mounts in the file and folder structure of Plesk.
- The *Plesk Modules Support Architecture* section explains how Plesk supports modules and what internal mechanisms of Plesk are involved. This section specifies the boundary between the area of Plesk support and the area of the module’s activity (with regard to each operation that can be performed against the module).

Overview

Plesk provides the mechanism of extending its base functionality with additional modules. A module is a kind of plug-in built into Plesk and providing access to its own functionality or to the functionality of an external service from Plesk Control Panel (PCP).

Purpose

The main advantage of Plesk is the opportunity to manage a wide range of functionality from a single control panel. In this sense, integrating additional applications and services with Plesk via the mechanism of modules gains benefit as follows:

- PCP provides means for installing/uninstalling modules easily,
- modules integrate with Plesk Language system,
- modules integrate with Plesk Help system,
- modules use skins provided by Plesk,
- Plesk provides Modules API specially designed to create Plesk-specific modules,
- modules are registered in Plesk, which prevents from casual deletion of 'module' applications,
- a module installed in Plesk cannot be used from outside – it can be used from PCP only.

The extension technology used in Plesk creates an illusion of seamless interaction between a module and Plesk: once embedded, the module will be displayed on PCP along with native modules of Plesk. Nevertheless, a module is an autonomous software unit. It can be installed independent of Plesk, it does not depend on Plesk modifications, and Plesk does not depend on the module's modifications either.

Where Applicable

Plesk provides more than one technique to access third party applications and services from PCP.

- The *Application Vault technology* allows the user to upload a web application to Plesk for storage and to deploy it on customers' domains on demand.
- The *Custom Button technology* allows quick access to any functionality from PCP, both server-side and located in the Internet, by placing a link of the required resource on Control Panel.
- The *Plesk Modules technology* extends Plesk with *new administering facilities* and provides control of them from PCP. You need to create the code of the new functionality, but it is not necessary to issue a new version of Plesk – the new functionality will be just embedded into Plesk in the form of a module. Also, this technology helps extend the logic of Plesk by integrating it with *ready-made applications* or *services*. Integrated units become accessible from PCP.

Custom Buttons and Plesk Modules are very close in their result: both provide access to third-party functionality via links. Custom Buttons are very easy to create, while Plesk Modules require implementing the *management logic.* Use Modules if you need Plesk-specific support (see above) and protection from casual actions.

There are several modules shipped with Plesk:

- Acronis TrueImage Server management module;

- Battlefield 1942 Server Manager (a game server);
- Counter-Strike Game Server;
- Samba Fileserver Configuration module;
- Firewall;
- Remote Admin for SiteBuilder2;
- Virtual Private Networking module (establishes secure connections on base of standard channels);
- Watchdog (monitoring of Plesk Control Panel services).

Target Audience

Plesk Modules support was designed with Plesk Administrator in mind. It provides the following enhancements:

- Plesk Modules make it possible to extend the administering functionality of Plesk 'on the spot', without issuing a new version of Plesk;
- Plesk Modules allow the management of additional applications/services from PCP and provide Plesk Administrator with the main advantage of Plug-in technology - simple install and registration procedure.

How a Module Builds into Plesk Architecture

Plesk is made up of the following blocks:

- Plesk Control Panel (presentation tier);
- Plesk database and all clients' databases hosted on a Plesk server and managed via Plesk (data storage tier);
- Native modules of Plesk (AppVault, Event Manager, etc.) that implement the logic of Plesk (middle tier);
- Custom modules that serve as Plesk extensions and provide their own GUI and logic. In this sense, custom modules contribute both to the presentation tier and the middle tier of Plesk.

Custom modules are optional in the above structure of Plesk. They can be added to this structure and removed from it when necessary, thus presenting *independent extension units* of Plesk.

Plesk Module Structure

A typical module is a package containing four kinds of code:

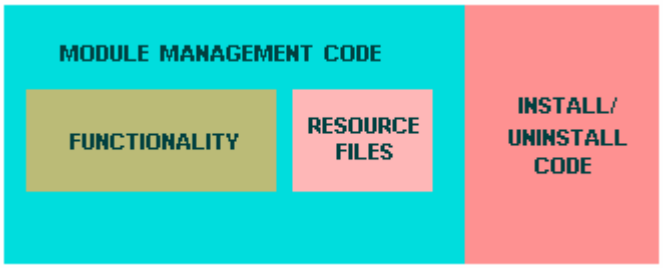

Figure 1: The logical structure of a Plesk Module

Module's functionality. This is the source and/or executable code of a module that implements the logic extending Plesk. This is the code that works when Plesk Administrator triggers any function of the module from GUI. This part of a custom module is optional as the logic of the module can be located beyond the module itself, e.g. a module can provide access to a remote or server-side service.

File types. PHP script files, C utilities.

Location. PHP files are located in the `<plesk_root_dir>/admin/htdocs/modules/<module_name>/` folder, C utilities can be located either in the `<plesk_root_dir>/admin/bin/modules/<module_name>/` folder, or in the `<plesk_root_dir>/admin/sbin/modules/<module_name>/` one.

Module Management code. This code should implement the GUI and backend. This part of the module is always present. Also, this part of code can include a configuration file used to change system settings for smooth execution of the module. This file is optional.

File types. PHP files, C utilities, INCLUDE configuration file.

Location. PHP files are located in the `<plesk_root_dir>/admin/htdocs/modules/<module_name>/` folder, C utilities can be located either in the `<plesk_root_dir>/admin/bin/modules/<module_name>/` folder, or in `<plesk_root_dir>/admin/sbin/modules/<module_name>/`. The configuration file can be located in the module's package at `<plesk_root_dir>/admin/conf/modules/<module_name>/.`

Resource files. This part of a module includes icons (GUI images), help files, localization files, templates. All these files of a custom module are optional.

File types. Icons are presented by GIF files, help files are typical HTML, localization files are PHP files, and templates are files in TPL format.

Location. Resource files are deployed to folders `<plesk_root_dir>/admin/htdocs` and `<plesk_root_dir>/admin /plib`:

- icons are located at `/htdocs/modules/<module_name>/images/`;
- help files are stored in `/htdocs/modules/<module_name>/locales/<locale_name>/help/`;
- localization files are located at `plib/modules/<module_name>/locales/<locale_name>.`
- template files are stored in `plib/templates/modules/<module_name>/`.

Install/uninstall code. These are the module's install/uninstall scripts. Most often, this part of code is located in the module's installation package (RPM/DEB/SH) rather then in the module itself. If the SH installation package is used, the uninstall script should be located within the module's body at `<plesk_root_dir>/var/modules/<module_name>/.`

Recognizing a Module in Plesk Folder System

Plesk allocates resources of installed modules in its file and folder hierarchy as shown below.

	`<plesk_root_dir>/`			This is the root directory of Plesk. Here `<plesk_root_dir>` stands for the fully qualified path of a directory where Plesk Server Administrator is installed. In Unix, this path is normally `/usr/local/psa`. In Debian, it is `/opt/psa`.
		`admin/`		This folder is meant to store everything required for module administering. Namely, this folder contains PHP, binary and image files that implement the *GUI and logic* required for this task.
			`bin/`	This folder stores binary utilities meant for low-level (system) operations (normally, compiled C programs). These can be the files of the module's functionality and the module management files as well. Besides binary files, this folder stores symlinks of binary utilities located in the `sbin/` folder and executed with setuid root privileges. To see the internal structure of this folder, proceed to page 12.
			`htdocs/`	This folder stores files used to display the module management page(s) in a browser. These can be PHP form files referring to the module management part as well as resource files, e.g. HTML help files, icons displayed on navigation buttons To see the internal structure of this folder, go to page 12.
			`plib/`	This directory stores PHP files of the module's functionality and the module management files as well. Also, the directory stores resource files of the module, e.g. PHP localization files and templates. To see the folders nested within this one, go to page 13.
			`sbin/`	This folder contains binary utilities meant for low-level operations on the system (normally, compiled C programs) that should be executed with setuid root privileges. These can be the files of the module's functionality and the module management files as well. Also, this folder contains a setuid root wrapper utility meant for this task. To see the internal structure of this folder, go to page 14.
			`conf/`	This folder contains the configuration file of Apache server that runs Plesk. Also, this folder can contain special files (one for an individual module) that modify this configuration for a certain module. To see the internal structure of this folder, proceed to page 14.

		var/	This folder is meant to store auxiliary files and components of Plesk – log files, SSL certificates, etc. In case of modules, this folder can contain the uninstall script for a module wrapped into a SH distribution package.

admin/bin Folder

This Plesk folder is meant to store binary files (executables, libraries, compiled C programs) and symlinks of binaries located in folder `<plesk_root_dir>/admin/sbin`.

	modules/		This Plesk folder contains a <module_name> folder for each installed module.
		<module_name>/	This folder stores binary utilities and symlinks of a particular module.

Here a module can locate binaries referring to its logic and the management part. Also, here the module can store symlinks of the binaries that require setuid root privileges. These symlinks reference the `mod_wrapper` utility of Plesk located in the `<plesk_root_dir>/admin/sbin` folder beside the binaries that match the module's symlinks.

admin/htdocs Folder

This Plesk folder stores files somehow related with GUI, i.e. meant for the display in the browser. A module locates its source files in this folder as follows.

	modules/					This Plesk folder contains a <module_name> folder for each installed module.
		<module_name>/				The folder stores PHP files that define GUI of a certain module, including the index.php file that serves as an entry point to GUI of a module. Also, this folder contains the following nested folders:
			locales/			This is the 'root' folder for one or several branches, each meant for a certain locale.
				<locale_name 1>/		This folder isolates help files referring to the first supported locale.
					help/	The folder contains HTML help files for this locale.

				<locale_name 2>/	This folder isolates help files referring to the second supported locale.
				help/	The folder contains HTML help files for this locale.
			…		
				<locale_name N>/	This folder isolates help files referring to the N supported locale.
				help/	The folder contains HTML help files for this locale.
			images/		This folder stores GIF image files (icons) of the module's GUI.

admin/plib Folder

This Plesk folder stores PHP files implementing the logic of Plesk units. In case of Plesk modules, this folder is used to store PHP files referring to the logic and management of the module.

	modules/			This Plesk folder contains a <module_name> folder for each installed module.
		<module_name>/		This Plesk folder isolates a branch that stores localization files for all installed modules in the locales folder. Also, it stores the locale.php file that ships with the module and resolves the current locale of Plesk.
			locales/	This folder can store localization files of two types: messages_<locale_name>.php stores error/warning messages and GUI text, and conhelp_<locale_name>.php stores context help messages. Each locale supported by the module can have one or both these files in this folder.
	templates/			This Plesk folder isolates a branch storing template files used by FastTemplate engine to generate GUI pages.

		`modules/`		This Plesk folder contains a `<module_name>` folder for each installed module.
			`<module_name>/`	This folder contains TPL files (HTML templates that define the layout of module pages)

admin/sbin Folder

This Plesk folder is meant to store binary files (executables, libraries, compiled C programs) having symlinks of the same name in folder `<plesk_root_dir>/admin/bin`.

	`modules/`		This Plesk folder contains a <module_name> folder for each installed module.
		`<module_name>/`	This folder stores binary utilities of a particular module.

Here a module can have binaries of its logic and management that need to be executed with setuid root privileges. These binaries should be located in this folder as Plesk stores its `mod_wrapper` utility here that seves as a wrapper for module utilities. Symlinks of setuid binaries reference this mod_wrapper utility, so when called via a symlink, mod_wrapper finds a binary of the same name in the `<plesk_root_dir>/admin/sbin` folder and executes it as a utility with root access permissions.

admin/conf Folder

Plesk uses this folder to store the `httpsd.conf` configuration file. The settings set in this file can be modified for a certain module if place the `httpsd.<module_name>.include` configuration file for this module beside the main file.

Plesk Modules Support Architecture

Being part of Plesk functionality, a module presents a standalone unit able to take care of itself during the whole lifetime. If written and composed correctly, a module is capable of the following:

- it can deploy itself to the specified server directory on demand;
- it can provide GUI for the management of its functionality, activate its functionality once the user has performed certain actions via GUI;
- it can implement its own context help and provide it on demand;
- it can include localization support and provide it on demand;
- finally, it can uninstall itself on demand.

Thus, Plesk modules support technology should undertake just a small part of cares, in particular, providing means to access the module via Plesk Control Panel, triggering the install/uninstall procedure, and providing interactive collaboration between a module and Plesk Help System/Plesk Language System. Let us look at the mechanisms provided by Plesk to implement custom modules support.

Plesk Module Manager

Module Manager is a part of Plesk that provides GUI to operate modules, namely, to install the specified module to Plesk, to display the list of all modules registered in Plesk and accessible for use, to display GUI of the selected module, and to uninstall the module.

To get a better vision of where Plesk support ends and the module's work begins, there is a table presented below:

Task	What Module Manager does		What Module does
Module's install procedure	Allows the user to select an RPM/DEB/SH package via GUI.		
	- triggers unpacking of the package to the server.		- having got the command to unpack itself, the RPM/DEB/SH package deploys its contents (file and folder structure) to the server directories specified in the package.
			- the RPM/DEB/SH package registers itself in the RPM/DEB/SH system managed by the operating system.
	- after the module is registered, the user can apply the Update operation on the Modules page to see the button of the new module on it.		- the module is registered in the specified tables of Plesk database.

Viewing all modules available	Uses GUI to display the buttons of all available modules registered in Plesk database.		
Opening module's GUI and performing operations on module	Once the user has pressed the module's button, Module Manager displays the default page of the module (an entry point of the module's GUI).		- the default page should provide graphics means to run the module and/or navigation means to move to other pages of the module.
			- at least one (main) page of the module should provide an opportunity to return to Module Manager (Plesk GUI concept implies that any page contains a link of a higher-level page).
Module's uninstall procedure	Allows the user to select the candidates for deletion in the list of available modules via GUI, and to trigger the uninstall procedure as well.		- the RPM/DEB gets the command to delete the contents of its package and performs the task. In case of a shell package (SH) Module Manager deletes the package on its own by triggering the uninstall script stored in the `<plesk_root_dir>/var/modules/ <module_name>` folder.
	- after the module is uninstalled, the user can apply the Update command on the Modules page and see that the module's button is disappeared.		- the module unregisters itself in Plesk database.

Localization Support

Plesk Language Support presents a part of Plesk functionality that works on the level of Plesk rather than on the modules support level. In other terms, the use of this mechanism is standard for *any* component of Plesk that needs localization for its GUI or messages.

When displaying messages, context help, and HTML Help files, Plesk Language Support should know what language to use to display the information. The rule is that the language matches the locale settings defined for the whole Plesk. Once installed, Plesk uses the default locale, but the locale can be changed at any moment of Plesk lifetime. Thus, *every time* the module starts running, it relies on the value of Plesk global variable that holds the current locale of Plesk.

Locale Resolution Mechanism

The locale resolution mechanism should be implemented in the module itself, in its `locale.php` file stored in the module's folder `%plesk_dir%/admin/plib/modules/<module_name>`. This mechanism is triggered each time a module is started, and it works as follows. First comes the checkup of whether the module contains message and context help localization files that support the current locale of Plesk.

To localize messages, a module should contain the following resources in the `%plesk_dir%/admin/plib/modules/<module_name>/locales/<locale_name>` folder for every supported locale:

- one `messages_<locale>.php` file for every supported locale. These files are meant to output messages and GUI labels in a proper language;
- one `conhelp_<locale>.php` file for every supported locale. These files are meant to output context help messages in a proper language.

Localization files `messages_<locale>.php` and `conhelp_<locale>.php` have a very simple structure and consist of records like `'<identifier>' => '<message>'` making up an associative array.

If found, the required localization files are selected, otherwise selected are the files matching the default locale of the module (normally en-US). Later on, *Plesk global array of messages* is extended with records from the `messages_<locale>.php` file, and *Plesk global array of context help messages* is added with context help records from the `conhelp_<locale>.php` file and the path of this file. These settings are stored in the global scope until the current Plesk session is terminated.

Thus, when a module is called for the first time within the current Plesk session, the module's localization resources are put into Plesk Language System and later on localization of the module's *messages* and *GUI* runs in a standard way. Once the module needs to output a message to the user in response to any event, or when it is necessary to localize text on GUI elements before display, Plesk Language System reads the required *message* from the global array of Plesk by its *identifier*. The found message is returned to the calling page of the module.

The context help localization system functions similar, but having received the context help message from the global array, the localization system passes it to Plesk Help System for display.

As for localization of the module's HTML Help, it is tied to the name of the `/<locale_name>` folder and is not determined by the standard mechanism of Plesk Language Support. To make localization of HTML Help files possible, a module should have at its path `%plesk_dir%/admin/htdocs/modules/<module_name>/locales` as many `<locale_name>` folders, as the number of supported locales. Every such folder should contain a set of HTML files in a proper language.

Help Support

Plesk Help Support proposes support for two kind of help – context help and HTML help. Getting a context help message in a proper language is performed by means of Plesk Language System. Having received a message, Plesk Help System displays it to the left, in a special Help section of the navigation pane of Plesk Control Panel.

To support HTML Help, a module should contain HTML files in its `%plesk_dir%/admin/htdocs/modules/<module_name>/locales/<locale_name>/help` folder. This can be a single file presenting a brief information about the module's purpose, or a multi-page manual arranged into a hierarchy of folders if necessary. HTML Help files are displayed in a separate browser window above other opened documents.

A certain page of HTML Help can be bound with a definite form of the module. This means that triggering help from a certain form of the module will lead to the display of the related HTML file.

Binding HTML Help files with the module's forms is an optional feature, but the presence of a help file called by default is a strong recommendation.

Creating Modules

What Code Makes Up a Module

This chapter begins with an overview of what code should be created within a module, why this code is necessary, and how this code will function. This knowledge will help with a *preliminary* estimate of the amount of work on your module. The chapter continues with the *Programming Guide* that can be used as a tutorial. Each step explains a specific aspect of the workflow.

Main code

Creating a module is a way out when it is necessary to extend Plesk with some extra functionality. Actually, creating a module begins with the code that implements this *functionality*. There are three ways how it can be obtained:

- the code can be written from scratch;
- a third-party code can be used;
- you can use a remote service that implements the required functionality.

Plesk does not impose any restrictions on the location of the functional code – it can reside on Plesk server or in the Internet. In the last case, the functional code is not included into the distribution package of the module.

Neither does Plesk impose constraints on the format of files implementing the module's functionality. These can be executables, dynamic libraries, scripts, or a combination of them. They can be arranged into a convenient folder system. The only restriction refers to the use of script files – developers should choose between the scripting languages supported in Plesk (PHP, Perl).

The functional code makes up the core of the module. To integrate this code with Plesk and make it function, some *management code* is needed. This is the point where the real work for the programmer begins – it is necessary to wrap the selected code into the module ready for integration with Plesk.

The management code always includes two parts - GUI and backend. GUI is implemented by PHP forms and provides the user with module management facilities (start/stop buttons, navigation through forms, etc.).

The backend part serves as a glue between GUI of the module and its functionality. Backend files solve the task of handling events occurring on module forms, plus they perform various operations invisible to the user (interaction with the module's functionality, data processing, global array operations, database transactions, etc.). The module's backend can have a two-level structure: reading data from forms, data validation, and other tasks are implemented in PHP files, whereas low-level (system) operations are entrusted to C binaries which are more effective for that. The core functionality of the module can be accessed from any of these levels.

The below figure demonstrates the principles of interaction between main parts of a module.

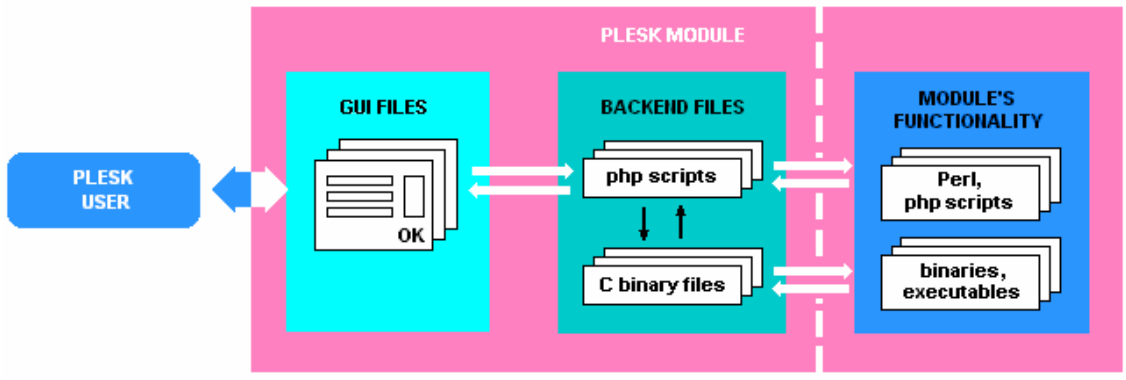

Figure 2: The interaction between main parts of a module

Optional capabilities

For deeper integration with Plesk, modules are allowed to utilize the following Plesk features:

- FastTemplate is a third-party technology that implements generating HTML pages out of PHP forms using templates. This technology is supported in API Reference that contains a special 'form' class for that. To utilize the FastTemplate technology, you need to create forms based on this class, plus simple template files should be included into the module package.
- Plesk Language Support is a feature that extends Plesk-specific localization techniques to integrated programming units. A module can utilize this feature for its GUI text, messages, and context help. This is worth doing as the localization of module messages can be managed on the level of Plesk, along with Plesk messages.

  Note: Plesk localization support is only provided for the *management code* of the module. The localization of the *functional core* is the sole responsibility of its vendors. If this code implements its own localization feature, then it will work autonomously and Plesk will not be able to interfere in any way. Plesk just reads data from stdout of the functional code and passes strings to GUI of the module in the form they have been read.

- Plesk Help Support can be useful if it is necessary to bind GUI forms of a module with certain help files. In this case, calling help from a certain page of a module will load an HTML help file associated with this page. And again, the functional core could have its own help files managed by the logic of this core. Plesk only manages the use of help files related to the *management code* of the module.

Customizing system settings

Changing system settings (e.g. access permissions, user groups, etc.) may be necessary for smooth running of a module. When trying to execute a module, Plesk always checks whether this module requires any changes in system settings. To inform Plesk that such changes are necessary, you can add a special script file to the module package.

Distribution package

When all parts of the module are ready, they need to be packed into a distribution package. Plesk Module Manager recognizes three types of distribution packages, that is: RPM, DEB, SH. These packages have much in common when it comes to creating them. All of them require a specification file and source files (the module package), and all of them can contain install/uninstall scripts (if necessary).

Programming Guide

This tutorial will guide you through the process of creating a Plesk Module, which includes 10 steps as follows:

1. making up the hierarchy of folders of the module package;
2. adding the core functionality;

   Next steps form the management code of the module. They include: creating GUI and the backend code, adding localization and help support, and customizing the module's environment.
3. designing GUI of the module;
4. designing icons;
5. implementing Help system of the module;
6. creating backend of the module (creating PHP files and C utilities of the module's backend);
7. localizing the module;
8. customizing system settings;

   Once the module package is ready, it remains to wrap it into the distribution package.
9. creating install/uninstall scripts;
10. packing the module into an RPM/SH/DEB package.

Step 1. Making up the hierarchy of folders

The initial step is creating the folder structure of the module package. In many respects it repeats the folder structure of Plesk. The reason for this is the use of RPM/SH/DEB technology for building a distribution package of the module. When installing a package, this technology just *reflects* its files and folders to the folder structure of the destination system. Thus, a module package should present a copy of Plesk folder system. Its folder structure should specify the destination paths of the module's files *physically* by locating them in proper folders of the package.

The following table explains what folders can be added to the module package. The `$module` folder stands for the name of the module, `$locale` means the name of a particular locale, e.g. `en-US`.

	`<plesk_root_dir>/`			Mandatory. This folder matches the root directory of Plesk. In Unix, `<plesk_root_dir>` is normally `/usr/local/psa`. In Debian, it is `/opt/psa`.
		`admin/`		Mandatory.
			`bin/`	Optional. This branch

			modules/					should be added if the module has binary files either in its functional core, or in the management code.
				$module/				
		htdocs/						Mandatory.
			modules/					Mandatory.
				$module/				Mandatory. Stores GUI files of the module named `$module`.
					locales/			Optional. Is created if the module implements its own Plesk-specific help.
						$locale/		Optional. This branch is necessary if the module contains HTML help files. If it does, there must be as many such branches as the number of languages in which help is written.
							help/	
					images/			Optional. Contains an icon displayed on the module's button and image files of the module's GUI.
		plib/						Mandatory.
			modules/					Mandatory. This branch stores backend PHP files of the module. PHP and Perl files of the functional core (if any) are located here too.
				$module/				
					locales/			Optional. If the module uses Plesk localization support, this folder is necessary to store localization files for all supported locales.
			templates/					Optional. If the module uses FastTemplate technology for generating its forms, this branch stores template files in the `$module` folder.
				modules/				
					$module/			

			sbin/			Optional. This branch is necessary if the module package has binaries (in its core or management code) that should be executed with setuid root privileges.
				modules/		
					$module/	
			conf/			Optional. This folder is necessary if the module requires some system settings to be modified in order to run properly.
		var/				Optional. This branch is necessary if the module will be distributed in a SH package. The $module folder will store the uninstall script of the module.
			modules/			
				$module/		

Thus, the first step should result in the folder structure of the module which contains all mandatory folders and some (or all of) optional, depending on what code makes up a module and what Plesk specific features are implemented in it.

Step 2. Adding the core functionality

The step is optional and can be skipped if the core functionality of the module is a remote service or application. If the functionality is created from scratch or a third-party application is used, and if this code is executed on Plesk server, then this code should be included into the module structure.

When allocating the files of the functional code within the module's folder structure, the following recommendations could help:

- Binary files (compiled C utilities, dynamic libraries) and symlinks are put to `<plesk_root_dir>/admin/bin/modules/$module/`.
- Binaries referenced by symlinks and executed with setuid root privileges are located at `<plesk_root_dir>/admin/sbin/modules/$module/`.
- PHP and Perl script files are put to `<plesk_root_dir>/admin/plib/modules/$module/`.
- Icons are put to `<plesk_root_dir>/admin/htdocs/modules/<module_name>/images.`
- HTML help files are located at `<plesk_root_dir>/admin/modules/<module_name>/locales/<locale_name>/help` according to the supported locale.

Thus, step 2 should result in the files of the functional code allocated within the module folder structure according to the file type.

Step 3. Designing GUI of the module

This step is very important and cannot be skipped as here begins the work on the module's management code which will serve as a bridge between Plesk and the module's core functionality.

The creation of the management code starts with designing its visible part – the module's GUI. A GUI consists of one or several forms that allow the user to start/stop the module, to define settings, and so on. Each form is presented in the module package by a separate PHP file. This file should solve the task of setting necessary parameters required by the form, plus it should contain the code that creates HTML of all GUI elements of the form.

Plesk Modules API

GUI elements used to compose a form can be created using functions and classes of Plesk Modules API. The API functions are as follows:

Forming GUI Elements
pm_comm_button
pm_link_button
pm_pathbarMaker
pm_pathbarDestructor
Navigation
pm_go_to
pm_go_to_uplevel

Plesk Modules API classes implementing GUI elements are as follows:

- The pm_Form class is designed as the HTML form generator. A special feature of this class is FastTemplate support, though using this technology is not a requirement. Also, this class provides parameters that help bind a form with the relevant help file.
- The pm_cList class generates a list element displayed on the browser's page as a multi-column table. Supported features are sorting, filtering, paging.
- The pm_Pathbar class presents a GUI element always displayed on top of Plesk pages and used to show the path by which the user has got to the current page. This control is a convenient navigation means that allows the user to get to any previous page displayed in the pathbar GUI element as a link.

Entry Point to GUI

When the user clicks on the module's button on the Modules page, Plesk looks through the related module package for the `index.php` file which serves as an entry point to GUI of the module.

This file is a suitable place to create the main form of the module as well as to set the table of commands recognized by the module. For instance, this table can be implemented as a *switch* construct:

```
$cmd = get_gpc('cmd');
```

```
switch ($cmd) {
case 'beginEdit':
      … // handling
      break;
case 'reset':
      … // handling
      break;
case 'remove':
      … // handling
      break;
default:
      … // handling
}
```

The code written within each *case* element is the point where GUI links the required backend functionality (creates objects, calls functions, handles errors, etc.).

Please note that the above example is not a recommendation on how to write the `index.php` file. The only requirement is that this file is *present* in the module package at `<plesk_root_dir>/admin/htdocs/modules/<module_name>` and provides access to the rest of the module's functionality directly or indirectly (referencing other PHP files).

Creating PHP Forms (Requirements and Recommendations)

Simple modules can have all necessary GUI elements located on a single form, while complex modules require a multi-page GUI. All forms of a complex GUI make up a hierarchy that can be arranged in a convenient folder structure. Besides, Plesk style requires that every form has an Up Level button.

It is proposed to locate PHP 'form' files in folder `<plesk_root_dir>/admin/htdocs/modules/<module_name>` of the module package beside with the `index.php` file. If these files are arranged into a folder structure, the entire structure can be located within this folder.

Finally, a couple of words regarding the use of Plesk Modules API. Though Plesk provides a special API for creating the module's GUI, using it is not a requirement. There are some advantages of using this API, that is:

- GUI elements created on basis of Modules API classes have a Plesk styled look.
- Such elements provide additional convenient features.

In respect of GUI forms, such convenient features are: *Plesk Help Support* provided for help topics associated with forms, and the *FastTemplate* technology. And again, the developer is free to select *any convenient way* to create module's forms. The purpose of Modules API is just to propose one particular way.

Utilizing FastTemplate Technology

If you choose in favor of `pm_Form` class of Modules API, this is the point to decide whether FastTemplate is a good choice for generating forms of the module (as using the `pm_Form` class does not make it *necessary* to use templates). This sub-topic demonstrates the power of FastTemplate and proves that using it is really easy.

Using FastTemplate implies creating *template files* that define the static part of the HTML page. These files should be created as standard HTML files, except they should contain *templates* rather than static HTML text of GUI elements. These files should be saved with the TPL extension (`<file_name>.tpl`) in `<plesk_root_dir>/admin/plib/templates/modules/<module_name>`.

The following example demonstrates the simplicity of an HTML template file:

```
<div style='margin: 20px 0 10px;'>
      {INFO}
</div>
<div class="toolsArea">
       <fieldset>
             <legend>{TOOLS_LEGEND}</legend>
             <table width="100%" cellspacing="0" cellpadding="0"
border="0"><tr><td>
             {START_BUTTON}{STOP_BUTTON}{REFRESH_BUTTON}
             </td></tr></table>
       </fieldset>
</div>
<div class="listArea">
       <fieldset>
             <legend>{DISKS_LIST_LEGEND}</legend>
             <table width="100%" cellspacing="0" cellpadding="0"
border="0"><tr><td>
                    {DISKS_LIST}
            </td></tr></table>
       </fieldset>
</div>
```

Here is how this template page looks like in the browser:

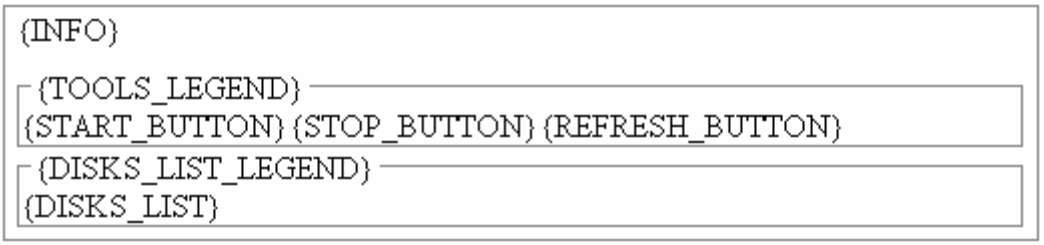

Figure 3: The Disks.tpl template file

Templates are labels enclosed in curly brackets. When the FastTemplate engine (integrated with Plesk) parses this page, templates serve as tokens to be replaced with HTML code of certain GUI elements. The advantage is the flexibility of making pages - each time this HTML is generated dynamically. The below figure illustrates the result of work of the FastTemplate engine:

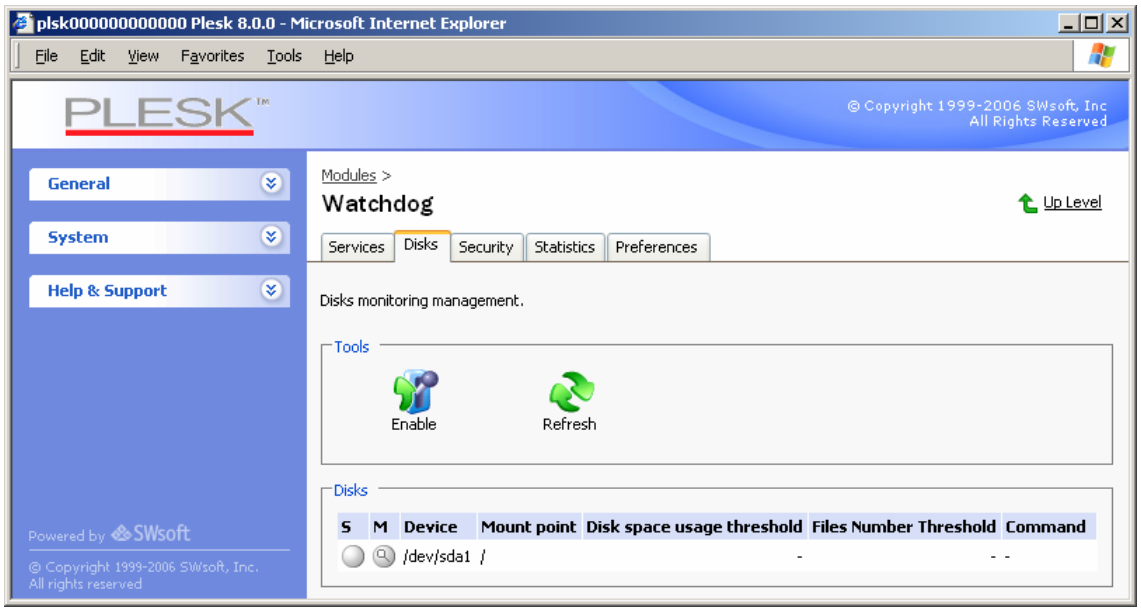

Figure 4: The page generated using the Disks.tpl template file

From the programming standpoint, the Disks form shown on the above figure is a tab of the Watchdog form. So, there must be a class inherited from the abstract `pm_Form` class and implementing the Watchdog form and one more 'tab' class that inherits from 'Watchdog'. To utilize FastTemplate, the 'Watchdog' (parent) class should have its `define()` method overridden as follows:

```
// function define ()
$templates = array(
      'tab' => 'disks.tpl',
      …
);
```

This code defines short names of template files (`disks.tpl`) and associates them with real templates ({TAB}). (Actually, the `'tab'` item of the `$templates` array stores the code that detects the template file of the currently active tab – Services, Disks, Security, Statistics, or Preferences. The `disks.tpl` file is hard-coded in this example for better understanding.)

The 'tab' (child) class should contain the following code in its overridden `assign()` method:

```
// function assign ()
$info = wd__lmsg("Disks monitoring management.");
$vars = array(
      'PAGE_TITLE' => wd__lmsg("Watchdog"),
      'CONTEXT' => safetyjs($context),
      'INFO' => $info,
      'TOOLS_LEGEND' => wd__lmsg("Tools"),
      'START_BUTTON' => '',
      'STOP_BUTTON' => '',
      'REFRESH_BUTTON' => '',
      'DISKS_LIST_LEGEND' => wd__lmsg("Disks"),
      'DISKS_LIST' => $DisksList->get(),
      );
```

This code declares templates and associates them with localized strings (via localization keys) or functions generating HTML of relevant GUI elements.

All methods of class pm_Form and their calling order are stated in the class pm_Form topic of API Reference.

Thus, the step devoted to GUI of the module should result in a set of PHP files generating GUI, including the index.php one, and optionally in a set of template files, all of them located in proper folders of the module's package.

Step 4. Designing icons

This step is optional and consists in designing at least one icon in the GIF format that will be displayed above the module's link button on the Modules page. This step makes the Modules page look stylish, but if you skip it, the module's link button will be displayed with the default icon anyway. Beside this icon, the module's GUI can use icons on its forms, all in the GIF format, 32x32 pixels large.

Once the icons are ready, put them to folder <plesk_root_dir>/admin/htdocs/modules/<module_name>/images of the distribution package.

Thus, this step should result in at least one GIF icon located at a definite path in the module's distribution.

Step 5. Implementing Help System of the module

This step is optional. However it is recommended that GUI of the module does provide help topics for module forms.

A standard help topic is an HTML file. You can create a single help file that would describe the module in common words, and this file would be invoked from the main module page and displayed in a separate browser window above other windows. Or this file can be a multi-page manual with an index page. Besides, you can provide each page of a multi-page module with its own help file. When the user asks for help on a certain page of such a module, the associated help topic is displayed in a separate browser window.

Creating a single-language help

Determine GUI pages that require help, and create help files for them. HTML help files can have any names, but these names should be *unique* within the module's package. By default, help files of a module are written in English (the default language of Plesk). Once the help files are ready, add them to the module's package. E.g. if written in English, they are put to folder <plesk_root_dir>/admin/modules/<module_name>/locales/en-US/help.

Creating a multi-language help

If the multi-language help is required and English (or other) help files are already available, they should be localized manually, their names *kept unchanged*.

Then all localized help files need to take a certain place within the module's distribution package. The <plesk_root_dir>/admin/modules/<module_name>/locales folder can contain multiple <locale_name> subfolders, one for each locale. All localized help files should be distributed between package folders <plesk_root_dir>/admin/modules/<module_name>/locales/<locale_name>/help according to the locales they refer to.

Note: `<locale_name>` is formatted as specified in IETF RFC1766, ISO 639, and ISO 3166 (it should look as `<language code>-<country code>`, e.g. en-US for USA English, en-GB for British English, de-DE for German in Germany, de-AT for German in Austria, etc.).

Binding a help file with a form

To invoke a certain help file, a GUI form should contain the code as described below. The code example is extracted from the main page of the Firewall module (ships with Plesk).

```
// set help support for firewall module
 $vars['BODY_ONLOAD'] = "SetHelpModule('firewall');";
// associate this form with help file firewall_main.html
$vars['CONTEXT'] = 'firewall_main';
```

The above example demonstrates how to bind the `firewall_main.html` help file with an instance of the class derived from `pm_Form`. This instance has a `$var` member variable that stores a set of parameters, including BODY_ONLOAD (sets a function to execute when the body of the form is loaded) and CONTEXT (is used to store the name of the HTML help file).

This binding should be set in every PHP file implementing the form with a help file. When help is invoked for such a page, Plesk will search for help files at `<plesk_root_dir>/admin/modules/<module_name>/locales/<locale_name>/help` where `<locale_name>` matches the locale currently set in Plesk.

If you prefer to use a custom class (not derived from `pm_Form`) for your GUI forms, this class can support Plesk Help System if you undertake two steps:

- HTML of the form generated by some member function of the custom class should contain the handler of the `body.onload` event which in turn should contain the following lines:
  ```
  SetContext('<help_file>', '');
  SetHelpModule('<module_name>');
  ```
- The body of the generated HTML page should contain the declaration of the following Plesk script files:
  ```
  <script language="javascript" type="text/javascript"
  src="/javascript/common.js"></script>
  <script language="javascript" type="text/javascript"
  src="/javascript/chk.js.php"></script>
  ```

Thus, this step should result in a set of HTML help files put to the module's distribution package according to the supported language, and (optionally) in binding forms with certain help files.

Step 6. Creating backend of the module

This step is mandatory as it refers to creating the *management code* of the module. If the module's GUI is ready, it is necessary to bind the visible part of the module with its functional part. This binding includes:

- Reading the data entered to fields of the module forms,
- Validating this data,
- Passing this data to the functional code of the module for processing,
- Reading the result returned by the functional code,
- Passing the result to GUI,

- Handling errors on all steps of data processing.

Using Modules API in the backend code

To perform these tasks, the functions of Plesk Modules API come to help. They can be grouped as follows:

Parameter Operations
pm_get_gpc
pm_get_locale
pm_isset_gpc
pm_set_gpc
Data Formatting and Conversion
pm_ldate
pm_ldatetime
pm_ltime
pm_plesk_mail
pm_safetyhtml
pm_size_b_printing
pm_size_kb_printing
pm_size_mb_printing
pm_size_pretty_printing
pm_time_pretty_printing
Error Handling/Messaging
pm_alert
pm_lmsg
pm_psaerror
pm_topnote
pm_warning
System
pm_util_exec

pm_util_io_exec

Besides, Plesk Modules API exposes the pm_Checker class that can be used when writing the backend code. This class provides a variety of member functions for data format checkup.

The above API resources are proposed for use in PHP code.

The structure of the backend code

In fact, the backend code is not homogeneous in the sense of its file types. The whole backend code can be conditionally considered as a two-level structure.

- The *upper level* of the backend code is written in PHP, it can use Modules API. This code implements operations listed at the beginning of this topic. In common words, this part of backend interacts with GUI, handles data, and performs operations on the core functionality of the module.
- The *lower level* of the backend code is binary. Its files (normally compiled C utilities) implement *system operations*, for instance, references to databases residing on Plesk server, file and folder operations, etc.

Allocating backend files

The upper-level code should be located at `<plesk_root_dir>/admin/plib/modules/<module_name>`.

To allocate the low-level code, keep in mind the following:

- if the utility can be executed with any access permissions, then it is put to folder `<plesk_root_dir>/admin/bin/modules/<module_name>`.
- if the utility should be executed with setuid privileges, then it is put to folder `<plesk_root_dir>/admin/sbin/modules/<module_name>` and a symlink of the same name referencing the `mod_wrapper` module is put to `<plesk_root_dir>/admin/bin/modules/<module_name>`.

The `admin/sbin` folder of Plesk stores a special `mod_wrapper` module. This module can provide its own context to execute a utility with root user permissions using the setuid technique. When activated, a symlink of this module triggers `mod_wrapper` to start, and this module looks through the `/sbin` folder and all its subfolders for the utility with the name matching the symlink name.

So, this step can result in some PHP files implementing the interaction between GUI and core functionality of the module, and in a set of utilities for low-level operations, all of them allocated within the module's package as described above.

Step 7. Localizing the module

This step can be skipped if the module will always show its messages and GUI text in English. In this case all strings with messages can be simply hardcoded in the module's code. However, if the module is a commercial product, using it becomes more comfortable if it provides a GUI/message localization feature. So, if you decide to provide such a feature, Plesk in turn can provide you with a Plesk-enabled language support technique which is described below.

The concept of Plesk Language support has already been considered in brief in the Basics section. Here we describe the steps that will provide a module with Plesk-enabled localization support.

Creating localization files

All messages/GUI text of a module are accumulated into two localization files with fixed names:

- `messages_<locale_name>.php` (for warnings, error messages, and GUI text),
- `conhelp_<locale_name>.php` (for context help messages),

where `<locale_name>` is formatted as `<language code>-<country code>` according to IETF RFC1766, ISO 639, and ISO 3166 , e.g. en-US for USA English, en-GB for British English, and so on.

A localization file contains an array of pairs like `<key>=><localized_string>` where `<localized_string>` is a string being localized and `<key>` is a unique localization key.

```
global $my_module_conhelp_arr;
$my_module_conhelp_arr = array(
      ‘module_activate’ => ‘Start the module.’,
      ‘module_deactivate’ => ‘Close the module.’,
      …
      );
```

Each supported locale should have one or both localization files. If a module supports several locales and contains several localization files of the same type (e.g. `messages`_en-US.php, `messages`_de-DE.php, etc.), the collection of localization keys should be similar in all these files (while the translations vary for different languages).

When ready, *all* localization files are added to the same folder of the module package (`<plesk_root_dir>/admin/plib/modules/<module_name>/locales`).

Using localization keys in the code

Now let us look how these localization pairs are used in the code. The principle is that PHP files of the module’s management code *contain references by localization keys rather than hardcoded message strings*. Here is the code snippet from the Firewall module (ships with Plesk) that demonstrates how `<key>=><localized_string>` pairs are defined in the arrays of messages and then used to describe the link button and to show the page title.

```
/*********************/
/* conhelp_en-US.php */
/*********************/
<?php
global $firewall_conhelp_arr;
$firewall_conhelp_arr = array(
      ‘firewall_activate’ => ‘Configure the system.’,
      …
      );
?>

/*********************/
/* messages_en-US.php */
```

```
/**********************/
<?php
global $lmsg_arr;
$lmsg_arr = array(
      'activateform__activating_page_title' => 'Activating
configuration',
      …
      );
?>

/************************/
/* FirewallMainForm.php  */
/************************/
<?php
…
$vars = array();
$vars[‘ACTIVATE_BUTTON’] = link_button(‘firewall_activate’, ...);
…
$vars['PAGE_TITLE'] =
safetyhtml(lmsg('activateform__activating_page_title'));
…
?>
```

How localization works

Here is the way how Plesk localization technique works. When the localized module is started for the first time within the current Plesk session, all `<key>=><localized_string>` pairs stored in localization files of the current locale (or of the default one if the current locale is not supported in the module) are added to the global array of Plesk messages. When parsing the management code of a module, PHP engine comes across references by a localization key and retrieves the value of this key from the global scope of Plesk. Here you should note that Plesk does not undertake any measures to add the contents of localization files to the global scope. *This task should be implemented by the programmer in the code of the module.* See the example further in the text.

Locale resolution mechanism

And the last unclear aspect is: when started, how does the module detect the current locale of Plesk? Plesk does not serve activated modules, so it will not inform the module about the current locale. The module should get this information itself. Here a specially created `locale.php` file comes to help. This file should be created by the developer and added to the module’s package at `<plesk_root_dir>/plib/modules/<module_name>`.

The name of this file is not strictly fixed, and the file can even be missing. In fact, it is created with the only purpose in mind – to isolate the code that detects the current locale in Plesk. There are no limitations on how this can be done and in which file this code is located, provided this other file is included in a proper place.

The following example is the `locale.php` file written for the Firewall module (ships with Plesk). The task of this code is: to detect the current locale of Plesk, to check whether the current locale is supported by the module and to handle the situation if it does not, and to load localized messages to the global scope.

```
/**************/
/* locale.php */
```

```
/**************/
<?php
/**
 * This function loads the module's locale files. This file
 * automatically calls this function
 */
function load_module_locale()
{
      // Get the current locale
      $locale = get_locale();
      $defaultLocale = 'en-US';
      if (!is_dir(PRODUCT_ROOT_D .
"/admin/plib/modules/firewall/locales/$locale"))
            $locale = $defaultLocale;
      // Load messages to the global scope      global $lmsg_arr;
$lmsg_main = $lmsg_arr;
      if ($locale != $defaultLocale)
      include_once
("modules/firewall/locales/$locale/messages_$locale.php");
      $lmsg_custom = $lmsg_arr;
      require_once
("modules/firewall/locales/$defaultLocale/messages_$defaultLocale.php"
);
      $lmsg_arr = array_merge($lmsg_main, $lmsg_arr, $lmsg_custom);
      unset($lmsg_custom);
      unset($lmsg_main);
      global $session;
      $reload = false;
      $specific_conhelp = $session->getParam('specific_conhelp');
      if (!isset($specific_conhelp["firewall__$defaultLocale"])) {
            $specific_conhelp["firewall__$defaultLocale"]
                  = array('arr_name' => 'firewall_conhelp_arr',
                    conhelp_file' =>
"modules/firewall/locales/$defaultLocale/conhelp_$defaultLocale.php");
            $reload = true;
      }
      if (!isset($specific_conhelp["firewall__$locale"])) {
            $specific_conhelp["firewall__$locale"]
                  = array('arr_name' => 'firewall_conhelp_arr',
                  'conhelp_file' =>
"modules/firewall/locales/$locale/conhelp_$locale.php");
            $reload = true;
      }
      if ($reload) {
            $session->putParam('specific_conhelp',
$specific_conhelp);
            go_to($_SERVER['REQUEST_URI'], 'self',
'refresh_leftframe();');
      }
}
load_module_locale();
?>
```

So, this step should result in the localization files created and allocated properly, and in the locale resolution file as well.

Step 8. Customizing system settings for the module

This step is optional, and it is proposed that the developer takes a decision regarding it based on the following. Sometimes smooth execution of a module requires such actions as changing owners (and access permissions) of certain files, making up groups of special users, customizing network settings, etc. These changes are made on the system level, and the module can declare them in a special file. If these modifications of system settings are not necessary, you can skip this step.

If the module requires some modifications in the current system settings, it should contain the file named `httpsd.<module_name>.include` in folder `<plesk_root_dir>/admin/conf` of the module's package. This file contains a shell script that runs system/custom utilities to perform the required modifications.

When Plesk tries to execute a module, it always checks whether the module contains this file at the specified path, and executes the script if the file is found.

Here is the example of this script:

```
#!/bin/sh
echo 0 > /proc/sys/net/ipv4/ip_forward
@@IPTABLES@@ -F
@@IPTABLES@@ -X
@@IPTABLES@@ -Z
@@IPTABLES@@ -P INPUT ACCEPT
@@IPTABLES@@ -P OUTPUT ACCEPT
@@IPTABLES@@ -P FORWARD DROP

if [ -f '@@PRODUCT_ROOT_D@@/var/modules/@@MODULE@@/openvpn.pid' ];
then
    kill `cat '@@PRODUCT_ROOT_D@@/var/modules/@@MODULE@@/openvpn.pid'`
 rm -f '@@PRODUCT_ROOT_D@@/var/modules/@@MODULE@@/openvpn.pid'
fi
```

This script executes a set of commands of the `iptables` utility, after which the `openvpn.pid` module is used to modify network settings.

Thus, if not skipped, this step should result in the file with a shell script created and put to the proper folder of the module package.

Step 9. Creating install/uninstall scripts

At this step all module files are ready for the distribution, and it is necessary to make a decision regarding scripts that serve two points of the module's lifetime.

Scripts That Work Before and After Installing/Uninstalling a Module

These scripts are included in the RPM/SH/DEB distribution package. When the package is being built, these scripts are copied to its folders (in RPM, these are folders PREIN, POSTIN, PREUN, POSTUN). These scripts are optional. They are necessary for operations like creating/destroying database objects, registering/unregistering the module package in a special table of Plesk database, generating/destroying system objects (files, folders), and so on.

The full set of such scripts is as follows (the example is given for an RPM package, the naming format is not fixed):

RPM Package Folder	Script	Expected Tasks
PREIN	`rpm_<module_name>_install_pre.sh`	Can set system and DB parameters, configuration files, etc.
POSTIN	`rpm_<module_name>_install_post.sh`	Can register the module in Plesk DB tables, create DB and system objects, set various module parameters, access permissions to files, etc.
PREUN	`rpm_<module_name>_uninstall_pre.sh`	Can remove the module and all related options from the DB, etc.
POSTUN	`rpm_<module_name>_uninstall_post.sh`	Can clean out all marks of the module's presence from Plesk DB and system.

One of the most important tasks of installing/uninstalling modules is interacting with Plesk database. First of all, any module must register itself in the Modules table of Plesk database, which makes sense to be done in the postinstall script. This table has the following fields:

Field	Type	Restrictions	Description
`id`	`INT`	`UNSIGNED AUTO_INCREMENT PRIMARY KEY`	module's id
`name`	`VARCHAR(255)`	`BINARY NOT NULL, UNIQUE`	module's name
`packname`	`VARCHAR(255)`	`BINARY NOT NULL`	package name (can match the module's name)
`display_name`	`VARCHAR(255)`	`BINARY NOT NULL`	displayed module's name
`version`	`VARCHAR(30)`	`BINARY NOT NULL`	module version
`release`	`INT`	`UNSIGNED NOT NULL`	package release
`description`	`VARCHAR(255)`		description of the module
`icon`	`VARCHAR(255)`	`BINARY NOT NULL`	icon shown for the module on Control Panel

A record written to this database should be unique (duplicate module installations are inadmissible), so it is a good practice to check this table for a similar record already available before the insert.

When removing a module, the related record should be cleaned out from the `Modules` table, which is normally done in the preuninstall script.

Besides registering itself, a module may need to create its own internal tables in Plesk database in order to register custom settings in them. This can be done in the postinstall script. The naming format for such tables is `module_<module_name>_*` (e.g. `module_fileserver_users`). SQL commands creating tables can be included to the postinstall script directly. Also, it's a good practice to isolate SQL code in separate files (formatted as standard shell scripts but not declared as such) and include them to the postinstall script. The following code can serve as an example of an internal MySQL table:

```
CREATE TABLE module_fileserver_users (
      id    int(10) unsigned NOT NULL auto_increment,
      name  VARCHAR(255) CHARACTER SET ascii COLLATE ascii_general_ci
NOT NULL default '',
      sys_name    VARCHAR(255) CHARACTER SET ascii COLLATE
ascii_general_ci NOT NULL default '',
      password    VARCHAR(255) CHARACTER SET ascii COLLATE ascii_bin
NOT NULL default '',
      password_type enum('plain', 'crypt') NOT NULL default 'plain',
      PRIMARY KEY (id),
      UNIQUE KEY name (name)
) TYPE=MyISAM;
```

When the module is uninstalled, all its internal tables need to be deleted from Plesk database too. This task is performed by the preuninstall script.

When ready, these 'pre' and 'post' scripts are added to the RPM/SH/DEB package during its build (see Step 10 of this tutorial).

Uninstall Script for SH Distribution Packages

One more situation that should be handled 'in advance' is the use of a SH package when installing a module. A SH module package is a set of shell commands that install a module. In contrast to RPM and DEB, such a package is not registered in the system, so the system does not remember the installation history of the module and will not be able to uninstall it. This should be done by a special `uninstall` script located in folder `<plesk_root_dir>/var/modules/<module_name>` of the module package. This shell script deletes all files and folders of a module and triggers pre- and postuninstall scripts if necessary.

The `uninstall` script is activated in Plesk as follows: when the administrator selects the delete operation for a module via Plesk GUI, Plesk passes control to ModuleManager that in turn looks for the `uninstall` script in `<plesk_root_dir>/var/modules/<module_name>/` and triggers it to execute.

Thus, this step should result in a set scripts meant to pre- and postprocess the module's install/uninstall, and (optionally) in one more uninstall script necessary if one chooses in favor of the SH package to distribute the module.

Step 10. Creating the Distribution Package

At this step, all components of the module's package are ready and the package itself is fully formed. Now it's time to decide what distribution technology to choose. Plesk recognizes three types of distribution packages - RPM technology which is 'native' for Unix operating system, SH technology, and DEB technology that targets Debian Linux.

Creating an RPM package

Since the code of the module is ready and packed into a folder structure, it remains to wrap it into an RPM package, which implies two steps:

- creating the SPEC file for the RPM package,
- assembling the RPM package.

A standard SPEC file should have a name formatted as follows:

```
<package name>-<application version>-<release number>.spec
```

A SPEC file contains the information necessary to build an RPM package. The header of this file contains a standard set of information that includes the application's description, its name, version number, release number, etc. Also, this file contains instructions on the building process, lists of application files and lists of third-party applications necessary for the install procedure.

The following SPEC file created for the `Counter-Strike2 Server version 3.1.0.6` illustrates the above:

```
%define  name    cs-gs2
%define  display_name  Counter-Strike2 Server
%define  cs_gs_version  3.1.0.6
%define  mod_version   10000
%define  display_mod_version %{cs_gs_vers}

Summary: Counter-Strike and Counter Strike:Source game server module
for Plesk
Name: %{name}
Version: %{display_mod_version}
Release: %{rels}
Copyright: GPL
Vendor: SWsoft
Group: Amusements/Games
Packager: SWSoft Inc <info@swsoft.com>
BuildRoot: %{cs_gs_buildroot}
Prefix: %{product_root_d}
Prereq: psa >= 7.5.0
Requires: psa-security = 7.5
Provides: plesk-module

%description
%{name} is Counter-Strike game server module for Plesk.
%prep
[ -f %{cs_gs_buildroot}/../rpm_cs-gs2_install_pre.sh ]
[ -f %{cs_gs_buildroot}/../rpm_cs-gs2_install_post.sh ]
[ -f %{cs_gs_buildroot}/../rpm_cs-gs2_uninstall_pre.sh ]
%pre
 %include %{cs_gs_buildroot}/../rpm_cs-gs2_install_pre.sh
%post
 %include %{cs_gs_buildroot}/../rpm_cs-gs2_install_post.sh
%preun
 %include %{cs_gs_buildroot}/../rpm_cs-gs2_uninstall_pre.sh
%files
```

```
 %defattr(-, root, psaadm)
%{product_root_d}/admin/bin/modules/%{name}
%{product_root_d}/admin/sbin/modules/%{name}

%defattr(-, root, root)
%{product_root_d}/etc/modules/%{name}/
%{product_root_d}/admin/htdocs/modules/%{name}/
%{product_root_d}/admin/htdocs/images/modules/%{name}/
%{product_root_d}/admin/plib/modules/%{name}/
%{product_root_d}/admin/plib/templates/modules/%{name}/
%{product_root_d}/admin/htdocs/images/custom_buttons/cs-small.gif
%{product_root_d}/var/modules/%{name}/
%changelog
* Tue Nov 08 2005 Plesk Inc <info@swsoft.com>
- First build
```

Once the SPEC file is ready, it's time to build the RPM package. This can be done by the following command executed under RedHat Linux:

```
# rpmbuild --bb --target=noarch /usr/src/redhat/SPECS/<spec_file_name
>.spec
```

Once this command is executed, a special message informs the user about the folder where the resulting RPM package is located. E.g. the above example will put the RPM package to the `/usr/src/redhat/RPMS/noarch` folder.

Visit http://www.rpm.org/RPM-HOWTO/build.html to learn more about the process of building RPM packages.

Creating a SH package

First comes the preparatory step at which the module package gets compressed. Pack the module into a TAR archive to merge its hierarchical folder structure into a single file, and then compress the resulting TAR file into a ZIP archive file. E.g. module `psa-sbm2 v1.1-0021` module can be packed into the `psa-sbm2.tar` file and then compressed into the `psa-sbm2.tar.gz` archive file.

To wrap the compressed module into a SH package, proceed through the following steps:

- create an empty SH file;
- create the contents of this file;
- encode the module package using UUENCODE;
- append the encoded module package to the SH package.

A SH file should have a name as follows:

```
<application name>-<version number>-<release number>.sh
```

First it is necessary to create an empty SH file, e.g. `psa-sbm2-1.1-0021.sh`, and make it executable using the following commands:

```
> psa-sbm2-1.1-0021.sh
chmod 777 psa-sbm2-1.1-0021.sh
```

The contents of this file should contain a header where it is specified that the package refers to Plesk modules:

```
#!/bin/sh
#Provides: plesk-module
```

Next comes the section that describes how the module package will be decoded from UUENCODE as well as enumerates the archive files, includes the code that will unpack this archive, copies the module's files to a specified location, and cleans out temporary installation files and folders.

The following example is an extract of the SH file of the SiteBuilder module (ships with Plesk):

```
#!/bin/sh
#Provides: plesk-module

initial_conf()
{
      PRODNAME = "psa"
      PRODUCT_NAME = "Plesk"
      product_full = "Plesk"
      PRODUCT_FULL_NAME = "Plesk"
      product_etc = "/etc/${PRODNAME}"
}

set_module_params()
{
      module = "sbm2"
      module_full = "Remote Admin for SiteBuilder2"
      packname = "psa-sbm2"
      module_current_version = "1.1 21"
      module_ver =
"$PRODUCT_ROOT_D/admin/plib/modules/${module}/version"
      module_conf =
"$PRODUCT_ROOT_D/admin/plib/modules/${module}/config.php"
```

```
      module_version = "1.1"
      module_release = "0021"
      module_sql_file =
"$PRODUCT_ROOT_D/etc/modules/${module}/sbm2_db.sql"
}

mysql_query()
{
      result = `echo "$query" | $mysql 2>/dev/null`
      return $result
}

set_common_params()
{
      initial_conf
      set_module_params

      # set mysql params
      mysql_client = "$MYSQL_BIN_D/mysql"

      mysql_passwd_file = "$product_etc/.${PRODNAME}.shadow"
      admin_passwd = `cat "$mysql_passwd_file"'
      mysql = "$mysql_client -N -uadmin -p$admin_passwd -D${PRODNAME}"
}

sbm2_install_db()
{
      echo "===> Installing database"
      query = `cat $module_sql_file`
      mysql_query
}

sbm2_install_registration()
{
      echo "===> Registering module..."
      query = "
      REPLACE INTO \`module_2_sbm_config\` VALUES ('3', 'version',
'$module_version-__release__');
      REPLACE INTO Modules (\`name\`, \`packname\`, \`version\`,
\`release\`, \`description\`, \`icon\`, \`display_name\`) VALUES
('$module', '$packname', '$module_version',  '$module_release', 'The
SBM2 module provides functionality for remote administration of
SiteBuilder2.', '/modules/$module/images/icon.gif', '$module_full');"
      mysql_query
      echo "===> Module has been registered successfully."
      echo $module_current_version >$module_ver
}

sub_which()
{
      local prog = "$1"
      if [ "X$prog" = "X" ]; then
            return 1
      fi;
      IFS =:
      for E in $PATH; do
            if [ -f "$E/$prog" -a -x "$E/$prog" ]; then
                  echo "$E/$prog"
            return 0
            fi
```

```
      done
      return 1
}

sbm2_unpack_tar()
{
      uudecode $0
      GTAR = `sub_which gtar`
      echo "===> Installing module ${packname}"
      $GTAR --owner=root --group=psaadm -C $PRODUCT_ROOT_D/admin -xzf
${packname}.tar.gz
      mv -f $PRODUCT_ROOT_D/admin/app-key-handler.module-sitebuilder-2
$PRODUCT_ROOT_D/bin
      mkdir -p $PRODUCT_ROOT_D/etc/modules/$module
      mv -f $PRODUCT_ROOT_D/admin/sbm2_db.sql
$PRODUCT_ROOT_D/etc/modules/$module
      if [ ! -d $PRODUCT_ROOT_D/var/modules/$module ]; then
             mkdir $PRODUCT_ROOT_D/var/modules/$module
      fi
      mv -f $PRODUCT_ROOT_D/admin/uninstall
$PRODUCT_ROOT_D/var/modules/$module/
      chmod 550 $PRODUCT_ROOT_D/var/modules/$module/uninstall
$PRODUCT_ROOT_D/bin/app-key-handler.module-sitebuilder-2
      rm -f ${packname}.tar.gz
      echo "===> OK"

}

oper = "install"
set_common_params

sbm2_unpack_tar
sbm2_install_db $oper
sbm2_install_registration

ln -s ${PRODUCT_ROOT_D}/lib / 2>/dev/null
exit 0
begin 644 psa-sbm2.tar.gz
```

Then the module package is encoded using the UUENCODE utility and added to the the SH file using the following command:

```
uuencode psa-sbm2.tar.gz psa-sbm2.tar.gz >> psa-sbm2-1.1-0021.sh
```

Creating a DEB package

The procedure of packing a module into the DEB package resembles that one with RPM packages. But there are two differences, that is:

- a CONTROL file is used instead of SPEC;
- assembling the package is done using the dpkg utility.

The CONTROL file contains various values which the package management tool will use to manage the package. Though there are no format restrictions on the CONTROL file name, it is recommended that the developers follow a standard naming convention for such files:

```
<package name>-<module version>-<release number>.control
```

A CONTROL file should contain the following control information necessary for the source package:

- Source: the name of the source package. Optional.
- Section: the section of the distribution the source package goes into (in Debian: main, non-free, or contrib., plus logical subsections, e.g. ‘admins’, ‘doc’, ‘libs’).
- Priority: describes how important it is that the user installs this package. This item can be left as ‘optional’.
- Maintainer: the name and email address of the maintainer.
- Build-Depends: the list of packages required to build your package. This item is optional.
- Standards-Version: the version of the Debian Policy standards this package follows, the versions of the Policy manual you read while making your package. This item is optional.

The information that describes the source package is as follows:

- Package: the name of the binary package. This is usually the same as the name of the source package.
- Version: the version information related to the application itself and to Plesk.
- Installed-size: the amount of disk space (in bytes) required for a given application.
- Architecture: the CPU architecture the binary package can be compiled for.
- Depends: the packages on which the given one depends on. The package will not be installed unless these packages are installed. This item should be used only if the application absolutely will not run (or will cause severe breakage) unless a particular package is present.
- Provides: this field specifies that the package targets a Plesk moodule.
- Description: a short description of the package.

<Here is the place where the long description goes> The Package field is specified using the following format:

```
Package: psa-module-<MODULE_NAME_IN_LOW_CASE>
```

The Version field is formatted as follows:

```
Version: <MODULE_VER>-<PLESK_VERSION><MODULE_RELEASE>
```

Here <MODULE_VER> stands for the version of the packed module, <PLESK_VERSION> means the version of Plesk (e.g. 80 for Plesk v8.0), and <MODULE_RELEASE> means the ordinal number of the module’s build.

The Depends field has the following format:

```
Depends: psa (>= 8.0)<MODULE_REQUIRES>
```

I.e. the required packages should follow one another without spaces, their versions specified in brackets as shown in the format string.

The field coming after the Description field should meet the requirements to follow: this should be a paragraph which gives more details about the package. There must be no blank lines, but one can put a single . (dot) in a column to simulate that. Also, there must be no more than one blank line after the long description.

Here is a sample CONTROL file:

```
Source: @@PRODNAME@@-cs-gs2
Section: non-free/net
Priority: extra
Maintainer:  <info@swsoft.com>
Standards-Version: 3.5.8
Build-Depends: debmake

Package: @@PRODNAME@@-cs-gs
Architecture: any
Provides: plesk-module
Pre-Depends: @@PRODNAME@@ (>= @@PRODVERSION@@)
Depends: @@PRODNAME@@-security-@@RELEASE_VERSION@@, @@PRODNAME@@ (>=
@@PRODVERSION@@)
Description: Counter Strike: Source game server module for Plesk.
```

To build a DEB package, one can use the dpkg utility and issue the following command:

```
dpkg -b ${build_dir} ${package_file}
```

For example,

```
dpkg -b /usr/local/modulebuilder/build/ cs-gs2-3.5.8-4_deb31_80
/usr/local/modulebuilder/build/psa-module- cs-gs2-3.5.8-8004_all.deb
```

Visit http://www.debian.org/doc/maint-guide/index.en.html to learn more about the process of building DEB packages.

API Reference

Plesk Modules API is designed to provide a third party developer with means of creating custom Plesk-specific modules. This API is a collection of classes and independent functions.

Modules API Functions

This section of Plesk Modules API Reference contains syntax and semantic information for the Plesk Modules API functions.

FUNCTION	DESCRIPTION
pm_alert	Shows the specified error message in case an error occurs.
pm_comm_button	Returns the HTML formatted code used to represent a button HTML element on the HTML page.
pm_get_gpc	Returns the value of the specified variable from the array of parameters passed in to a script via GET, POST, or COOKIE HTML method.
pm_get_locale	Gets the currently set locale.
pm_go_to	Opens a certain URL in the specified window frame in the browser, plus allows to specify some actions to perform before URL is opened.
pm_go_to_uplevel	Loads the upper-level URL to the currently active window frame in the browser. Also, allows to specify some actions to perform before URL is opened.
pm_isset_gpc	Checks whether the specified variable is present in the array of parameters passed in to a script via GET, POST, or COOKIE HTML method.
pm_ldate	Returns the specified date in the format defined in the `$def_date_format` global variable.
pm_ldatetime	Returns the specified date in the format defined in the `$def_datetime_format` global variable.
pm_link_button	Returns the HTML formatted code used to represent a link button HTML element on the HTML page.
pm_lmsg	Translates the specified index into the message text taking into account the current locale settings.

pm_ltime	Returns the specified date in the format defined in the `$def_time_format` global variable.
pm_plesk_mail	Forms an email envelope for Plesk email user on basis of the standard set of information (to, subject, message, etc).
pm_psaerror	This function throws Plesk fatal exception.
pm_safetyhtml	Modifies the specified HTML string (replaces HTML escape sequences with codes, etc.)
pm_set_gpc	Sets the specified variable to the global array of parameters passed in to a script via GET, POST, or COOKIE HTML method.
pm_size_b_printing	Accepts a string with the number of bytes and returns the formatted value in bytes for printing.
pm_size_kb_printing	Accepts a string with the number of bytes and returns the formatted value in kilobytes for printing.
pm_size_mb_printing	Accepts a string with the number of bytes and returns the formatted value in megabytes for printing.
pm_size_pretty_printing	A universal function that converts the passed in number of bytes to the best format (bytes/ KB/MB/GB/TB) so that the resulting value has a non-zero integer part as small as possible.
pm_time_pretty_printing	Converts the number of seconds to a pretty format (e.g. exp: 2day(s) 12:45)
pm_topnote	The function adds a warning message with the specified text to the global array of messages of type MSG_WARNING according to its ID.
pm_util_exec	Safely runs the specified utility within Plesk exec wrapper.
pm_util_io_exec	Safely executes the specified utility, passing in the data to the utility's input and reading the utility's output.
pm_warning	Adds a warning message with the specified text and without ID to the global array of messages of type MSG_WARNING.

In addition to these API functions, there are two more GUI functions that serve the pathbar GUI control located on the form. The `pm_pathbarMaker()` function should be called to guarantee the creation of the only instance of the `pm_Pathbar` class on the form. The `pm_pathbarDestructor()` function is designed to guarantee the destruction of the pathbar with its parameters being saved to the global scope first.

FUNCTION	DESCRIPTION
pm_pathbarMaker	Creates the only instance of the pm_Pathbar class on the HTML form.
pm_pathbarDestructor	Saves parameters of the pathbar GUI control to the global scope before this control is destroyed.

pm_alert

Shows the specified error message in case an error occurs.

Syntax

```
pm_alert($msg)
```

Parameters

msg

A *string* value with the message to display.

Returns

Nothing.

Include: pm.php.

pm_comm_button

Returns the HTML formatted code used to represent a button HTML element on the HTML page.

Syntax

```
pm_comm_button ($name, $conhelp, $handler, $enabled,
$tabindex)
```

Parameters

name

A *string* value with the button name.

conhelp

A *string* value with the context help index for this button. By default, this parameter holds an empty space.

handler

A *string* value with the code of the onClick handler written in Javascript. By default, this parameter holds an empty space.

enabled

A *boolean* value with the button status. Is set to *true* if the button is enabled, *false* otherwise. The default value is *true*.

tabindex

A *boolean* value that is *true* if the button should have a tab index on the form, *false* otherwise. The default value is *false*.

Returns

A *string* value that holds the HTML formatted code of the button's representation on the HTML page.

Remarks

If a button name consists of more than one word, the relevant *name* parameter should look as a set of words glued with the '_' (underscore) character.

A valid conhelp parameter should go without the 'b_' prefix.

Include: pm.php.

pm_get_gpc

Returns the value of the specified variable from the array of parameters passed in to a script via GET, POST, or COOKIE HTML method.

Syntax

```
pm_get_gpc ($name, $default_value)
```

Parameters

name

A *string* value that specifies the name of a variable to look in the array of parameters set via GET, POST, or COOKIE.

default_value

A *boolean* value that is *true* if it is necessary to return the default value of this variable from the common array of parameters set via GET, POST, and COOKIE in case the specified variable has not been set in it. By default, it is set to *false*.

Returns

A *string* value of the specified variable.

Remarks

In case the parameter name is specified incorrectly and Plesk fails to find it in the global array of parameters, it throws Plesk fatal exception.

Include: pm.php.

pm_get_locale

Gets the locale currently set for this session in the global set of parameters.

Syntax

```
pm_get_locale()
```

Returns

A *string* value with the current locale.

Include: pm.php.

pm_go_to

Opens a certain URL in the specified window frame in the browser, plus allows to specify some actions to perform before URL is opened.

Syntax

```
pm_go_to ($url, $target, $onLoad)
```

Parameters

url

A *string* value with the URL to load.

target

A *string* value that specifies the target window frame. By default, the value is ‘self’, which means that URL will be loaded to the currently active window frame.

onLoad

A *string* value with the code in Javascript that will be executed before the URL is loaded. By default, this value is set to *false.*

Returns

Nothing.

Remarks

A well-formed Javascript text passed in via the *onLoad* parameter should not contain double quotes.

Include: pm.php.

pm_go_to_uplevel

Loads the upper-level URL to the currently active window frame in the browser. Also, allows to specify some actions to perform before URL is opened.

Syntax

```
pm_go_to_uplevel ($page, $onLoad)
```

Parameters

page

A *string* value that specifies the page from which to go up. By default, the value is NULL (which means the current page).

onLoad

A *string* value with the code in Javascript that will be executed before the URL is loaded. By default, the value is *false*.

Returns

Nothing.

Remarks

The *page* parameter can hold any HTML page identifier specified by the developer for use in a module.

A well-formed Javascript text held in the *onLoad* parameter should not contain double quotes.

Include: pm.php.

pm_isset_gpc

Checks whether the specified variable is present in the array of parameters passed in to a script via GET, POST, or COOKIE HTML method.

Syntax

```
pm_isset_gpc ($name)
```

Parameters

name

A *string* value that specifies the name of a variable to look in the array of parameters set via GET, POST, or COOKIE.

Returns

A *boolean* value. Is *true* if the specified variable is set in the array of parameters passed in to a script via GET, POST, or COOKIE. *false* otherwise.

Include: pm.php.

pm_ldate

Returns the specified date in the format defined in the `$def_date_format` global variable.

Syntax

```
pm_ldate ($timestamp, $format, $gmdate)
```

Parameters

timestamp

A *string* value with the specified time. Optional. Is equal to the current time by default.

format

A *string* value with the format of the time passed in. By default, the string is empty.

gmdate

A *boolean* value that is *true* if the resulting time should be GMT (Greenwich Mean Time), *false* otherwise. Is *false* by default.

Returns

A *string* value with the date formatted according to the mask stored in the `$def_date_format` global variable.

Remarks

If the *format* parameter is empty, the format of the `$def_date_format` global variable will be applied.

Include: pm.php.

pm_ldatetime

Returns the specified date in the format defined in the `$def_datetime_format` global variable.

Syntax

```
pm_ldatetime ($timestamp, $format, $gmdate)
```

Parameters

timestamp

A *string* value with the specified time. Optional. Is equal to the current time by default.

format

A *string* value with the format of the time passed in. By default, the string is empty.

gmdate

A *boolean* value that is *true* if the resulting time should be GMT (Greenwich Mean Time), *false* otherwise. Is *false* by default.

Returns

A *string* value with the timestamp formatted according to the mask stored in the `$def_datetime_format` global variable.

Remarks

If the *format* parameter is empty, the format of the `$def_datetime_format` global variable will be applied.

Include: pm.php.

pm_link_button

Returns the HTML formatted code used to represent a link button HTML element on the HTML page.

Syntax

```
pm_link_button ($name, $conhelp, $href, $enabled,
$tabindex, $new_window, $lock)
```

Parameters

name

A *string* value with the link button name.

conhelp

A *string* value with the context help index for this button. By default, this parameter holds an empty space.

href

A *string* value with a reference to the specified URL. By default, this parameter holds '#', which means that a mouseclick on this link button will open the same page on which this link is clicked.

enabled

A *boolean* value with the button status. Is set to *true* if the button is enabled, *false* otherwise. The default value is *true*.

tabindex

A *boolean* value that is *true* if the button should have a tab index on the form, *false* otherwise. The default value is *false.*

new_window

A *boolean* value that is *true* if a mouseclick on this button should open URL in a new window of a browser, *false* otherwise. The default value is *false.*

lock

A *boolean* value that is *true* if the button is locked, *false* otherwise. The default value is *true.*

Returns

A *string* value that holds the HTML formatted code of the link button's representation on the HTML page.

Remarks

If a button name consists of more than one word, the relevant *name* parameter should look as a set of words glued with the '_' (underscore) character.

A valid *conhelp* parameter should go without the 'b_' prefix.

Include: pm.php.

pm_lmsg

Translates the specified index into the message text taking into account the current locale settings.

Syntax

```
pm_lmsg ($index, $ar1, $ar2, $ar3, $ar4, $ar5, $ar6, $ar7,
$ar8)
```

Parameters

index

A *string* value with index of the message to extract.

ar1

A *string* value with a reserved string parameter. By default, it holds an empty string (' ').

ar2

A *string* value with a reserved string parameter. By default, it holds an empty string (' ').

ar3

A *string* value with a reserved string parameter. By default, it holds an empty string (' ').

ar4

A *string* value with a reserved string parameter. By default, it holds an empty string (‘ ’).

ar5

A *string* value with a reserved string parameter. By default, it holds an empty string (‘ ’).

ar6

A *string* value with a reserved string parameter. By default, it holds an empty string (‘ ’).

ar7

A *string* value with a reserved string parameter. By default, it holds an empty string (‘ ’).

ar8

A *string* value with a reserved string parameter. By default, it holds an empty string (‘ ’).

Returns

A *string* value that holds a localized message extracted by the specified index from a localization file referring to a current locale.

Remarks

It is recommended that parameters *ar1* to *ar8* are always left as they are. This function refers to the Zend engine.

Include: pm.php.

pm_ltime

Returns the specified date in the format defined in the `$def_time_format` global variable.

Syntax

```
pm_ltime ($timestamp, $format, $gmdate)
```

Parameters

timestamp

A *string* value with the specified time. Optional. Is equal to the current time by default.

format

A *string* value with the format of the time passed in. By default, the string is empty (‘ ’).

gmdate

A *boolean* value that is *true* if the resulting time should be GMT (Greenwich Mean Time), *false* otherwise. Is *false* by default.

Returns

A *string* value with the time formatted according to the mask stored in the `$def_time_format` global variable.

Remarks

If the *format* parameter is empty, the format of the `$def_time_format` global variable will be applied.

Include: pm.php.

pm_plesk_mail

Forms an email envelope for Plesk email user on basis of the standard set of information (to, subject, message, etc).

Syntax

```
pm_plesk_mail ($to, $subject, $message,
$additional_smtp_headers, $from_header)
```

Parameters

to

A *string* value that enumerates email addresses of the recipients.

subject

A *string* value with the message subject.

message

A *string* value with the text of the message.

additional_smtp_headers

A *boolean* value which is *true* if some extra smtp headers need to be formed, *false* otherwise. Is *false* by default.

from_header

A *boolean* value that is *true* if the FROM header needs to be added to the letter's envelope, *false* otherwise. Is *false* be default.

Returns

A *string* value that contains the resulting text of the email envelope.

Remarks

A valid *to* parameter accepts a string with one to many addressees formatted as follows: `"Person1 <email@domain.com>, assa@bessa.com, ...".` Also, it can accept an array of addressees: `array($email1 => Person1, ...)`

Include: pm.php.

pm_psaerror

This function throws Plesk fatal exception.

Syntax

```
pm_psaerror ($message)
```

Parameters

message

A *string* value with a system message to display if Plesk fatal exception occurs.

Returns

Nothing.

Include: pm.php.

pm_safetyhtml

Modifies the specified HTML string (replaces HTML escape sequences like '}', '{', quotes with codes, etc.).

Syntax

```
pm_safetyhtml ($string)
```

Parameters

string

A *string* value with the HTML text to format.

Returns

A *string* value with the modified HTML text.

Include: pm.php.

pm_set_gpc

Sets the specified variable to the global array of parameters passed in to a script via GET, POST, or COOKIE HTML method.

Syntax

```
pm_set_gpc ($name, $value)
```

Parameters

name

A *string* value that specifies the name of a variable to set to the array of parameters received via GET, POST, or COOKIE.

value

A *string* value to set to the specified variable in the array of parameters obtained via GET, POST, or COOKIE.

Returns

Nothing.

Include: pm.php.

pm_size_b_printing

Accepts the number of bytes and returns the formatted string with the value in bytes for printing.

Syntax

```
pm_size_b_printing ($bytes, $addB, $precision)
```

Parameters

bytes

An *integer* value that specifies the number of bytes.

addB

A *boolean* value that is *true* if the ‘bytes’ word should be added to the resulting string, *false* otherwise. The default setting is *false.*

precision

A *string* value that specifies precision of the value to show. It is set to -1 by default (only the integer part of the value should be shown).

Returns

A *string* value that holds the passed in number of bytes in a special format.

Code Example

```
$num_bytes = 1024;

$num_bytes_to_print = pm_size_b_printing($num_bytes, true, 2);
$num_bytes_to_print1 = pm_size_b_printing($num_bytes, true, 0);
$num_bytes_to_print2 = pm_size_b_printing($num_bytes,true);
$num_bytes_to_print3 = pm_size_b_printing($num_bytes);

echo $num_bytes_to_print;
echo $num_bytes_to_print1;
echo $num_bytes_to_print2;
echo $num_bytes_to_print3;

// the result will look as follows:
// 1024.00 bytes
// 1024. bytes
// 1024. bytes
// 1024.
```

Remarks

The above code snippet shows the results of three invocations of this function, each time with different parameters.

Include: pm.php.

pm_size_kb_printing

Accepts the number of bytes and returns the formatted string with the value in kilobytes for printing.

Syntax

```
pm_size_kb_printing ($bytes, $addB, $precision)
```

Parameters

bytes

An *integer* value that specifies the number of bytes.

addB

A *boolean* value that is *true* if the ‘kilobytes’ word should be added to the resulting string, *false* otherwise. The default setting is *false.*

precision

A *string* value that specifies precision of the value to show (if the value in KB is float type formatted). It is set to -1 by default (only the integer part of the value in KB should be shown).

Returns

A *string* value that holds the passed in number of bytes transferred to KB and formatted as specified.

Code Example

```
$num_bytes = 5000;

$num_bytes_to_print = pm_size_kb_printing ($num_bytes, true, 2);
$num_bytes_to_print1 = pm_size_kb_printing ($num_bytes, true, 0);
$num_bytes_to_print2 = pm_size_kb_printing ($num_bytes,true);
$num_bytes_to_print3 = pm_size_kb_printing ($num_bytes);

echo $num_bytes_to_print;
echo $num_bytes_to_print1;
echo $num_bytes_to_print2;
echo $num_bytes_to_print3;

// the result will look as follows:
// 4.88 kilobytes
// 4. kilobytes
// 5. kilobytes
// 5.
```

Remarks

The above code snippet shows the results of four invocations of this function, each time with different parameters.

In case the *precision* parameter is a *zero* or a *positive value*, the function calculates the float number of kilobytes in the passed in number of bytes. Then the function trims the resulting value from the right so that only the required number of fractional digits is left.

In case the *precision* parameter is set to a negative value or not specified, the function transfers the passed in number of bytes to kilobytes and then rounds up the result to its integer part. The resulting string contains an integer part of the value and a dot delimiter, the fractional part of the value is missing.

Include: pm.php.

pm_size_mb_printing

Accepts the number of bytes and returns the formatted string with the value in megabytes for printing.

Syntax

```
pm_size_mb_printing ($bytes, $addB, $precision)
```

Parameters

bytes

An *integer* value that specifies the number of bytes.

addB

A *boolean* value that is *true* if the ‘megabytes’ word should be added to the resulting string, *false* otherwise. The default setting is *false.*

precision

A *string* value that specifies precision of the value to show (if the value in MB is float type formatted). It is set to -1 by default (only the integer part of the value in MB should be shown).

Returns

A *string* value that holds the passed in number transferred to MB and formatted as specified.

Code Example

```
$num_bytes = 5 000 000;

$num_bytes_to_print = pm_size_mb_printing ($num_bytes, true, 2);
$num_bytes_to_print1 = pm_size_mb_printing ($num_bytes, true, 0);
$num_bytes_to_print2 = pm_size_mb_printing ($num_bytes,true);
$num_bytes_to_print3 = pm_size_mb_printing ($num_bytes);

echo $num_bytes_to_print;
echo $num_bytes_to_print1;
echo $num_bytes_to_print2;
echo $num_bytes_to_print3;

// the result will look as follows:
// 4.76 megabytes
// 4. megabytes
// 5. megabytes
// 5.
```

Remarks

The above code snippet shows the results of four invocations of this function, each time with different parameters.

In case the *precision* parameter is a *zero* or a *positive value*, the function calculates the float number of megabytes in the passed in number of bytes. Then the function trims the resulting value from the right so that only the required number of fractional digits is left.

In case the *precision* parameter is set to a negative value or not specified, the function transfers the passed in number of bytes to megabytes and then rounds up the result to its integer part. The resulting string contains an integer part of the value and a dot delimiter, the fractional part of the value is missing.

Include: pm.php.

pm_size_pretty_printing

A universal function that converts the passed in number of bytes to the best format (bytes/ KB/MB/GB/TB) so that the resulting value has a non-zero integer part as small as possible.

Syntax

```
pm_size_pretty_printing ($bytes, $addB, $precision)
```

Parameters

bytes

An *integer* value that specifies the number of bytes.

addB

A *boolean* value that is *true* if the ‘bytes’ suffix should be added to the resulting string in addition to the ‘kilo’/’mega’/’giga’/’tera’ word, *false* otherwise. The default setting is *false.*

precision

A *string* value that specifies precision of the resulting value to show. It is set to -1 by default (only the integer part of the value should be shown).

Returns

A *string* value that holds the passed in number of bytes transferred to the best format and output as specified.

Code Example

```
$num_bytes_to_print = pm_size_pretty_printing (1023, true, 2);
$num_bytes_to_print1 = pm_size_mb_printing (5000, true, 2);
$num_bytes_to_print2 = pm_size_mb_printing (5000000,true);
$num_bytes_to_print3 = pm_size_mb_printing (5000000000, false, 2);
$num_bytes_to_print4 = pm_size_mb_printing (5000000000000);

echo $num_bytes_to_print;
echo $num_bytes_to_print1;
echo $num_bytes_to_print2;
echo $num_bytes_to_print3;

echo $num_bytes_to_print4;

// the result will look as follows:
// 1023 bytes
// 4.88 kilobytes
// 5. megabytes
// 4.65 giga
// 5. tera
```

Remarks

The above code snippet shows the results of four invocations of this function, each time with different parameters.

If the *bytes* parameter is less than 1024 bytes (KB), the function returns a string with an integer formatted number of bytes. The *precision* parameter is not considered.

In case the *precision* parameter is a *zero* or a *positive value*, the function calculates the float number in KB/MB/GB/TB (the best format is selected). Then the function trims the resulting value from the right so that only the required number of fractional digits is left.

In case the *precision* parameter is set to a negative value or not specified, the function transfers the passed in number of bytes to KB/MB/GB/TB (the best format is selected) and then rounds up the result to its integer part. The resulting string contains an integer part of the value and a dot delimiter, the fractional part of the value is missing.

If the *addB* parameter is set to *true*, then the string contains the value and the relevant description (kilobytes, megabytes, gigabytes, terabytes). In case *addB* is set to *false*, the 'bytes' suffix is missing, and the string contains the short description (kilo, mega, giga, tera).

Include: pm.php.

pm_time_pretty_printing

A universal function that converts the number of seconds to a pretty format. If the number of seconds is less than a day, the time format is selected, otherwise the day and time format is applied.

Syntax

```
pm_time_pretty_printing ($tval, $format)
```

Parameters

tval

A *string* value with the number of seconds to print.

format

A *string* value with the time format to apply. By default, the string is empty (' ').

Returns

A *string* value formatted as specified.

Remarks

If the format parameter is not specified, the function applies default formats `$def_date_format` and `$def_time_format`.

Include: pm.php.

pm_topnote

The function adds a warning message with the specified text to the global array of messages of type MSG_WARNING according to its ID.

Syntax

```
pm_topnote ($msg, $id)
```

Parameters

msg

A *string* value with a message test.

id

A *string* value with the identifier of the warning message.

Returns

Nothing.

Remarks

The message will remains in the array until it is removed from it explicitly. I.e. it doesn't disappear once displayed.

Include: pm.php.

pm_util_exec

Checks whether the utility exists and can be executed with the specified parameters. If the checkup is successful, safely runs the utility within Plesk exec wrapper.

Syntax

```
pm_util_exec ($command, $args, $which, $mod_name)
```

Parameters

command

A *string* value with the name of the utility to run.

args

A *string* value with the array of arguments to use when triggering the utility. By default, an empty array is set.

which

A *string* value that specifies the approach to execution error reporting. If set to 'msg', tells the function to display a standard error message and to return *false.* If set to 'lst', tells the function to display a standard error message and to return *false* if the utility fails, otherwise to return an array of strings from the utility's output. By default, the 'msg' value it specified.

mode_name

A *stirng* value with the module name the utility refers to. Is NULL by default.

Returns

In case the *which* parameter is ‘msg’, a *boolean* value is returned - *true* if the utility has been executed OK, *false* otherwise. In case the *which* parameter is ‘lst’, a *mixed* value is returned - an array of strings from the utility’s output if the utility has been executed OK, *false* otherwise.

Include: pm.php.

pm_util_io_exec

Safely runs the specified utility within Plesk exec wrapper. Passes in the data to the utility’s stdin and reads the result from its stdout.

Syntax

```
pm_util_io_exec ($cmd, $args, $in_str, &$out, $mod_name)
```

Parameters

command

A *string* value with the name of the utility to run.

args

A *string* value with the array of arguments to use when triggering the utility. By default, an empty array is set.

in_str

A *string* value with the data to pass in to the utility’s input.

out

A reference of a *string* value with the data read from the utility’s output.

mode_name

A *stirng* value with the module name the utility refers to. Is NULL by default.

Returns

A *boolean* value which is *true* if the utility has been executed OK, *false* otherwise.

Include: pm.php.

pm_warning

Adds a warning message with the specified text and without ID to the global array of messages of type MSG_WARNING.

Syntax

```
pm_warning ($msg)
```

Parameters

cmd

A *string* value with the message text to display.

Returns

Nothing.

Remarks

The message is removed from the array once displayed.

Include: pm.php.

pm_pathbarMaker

Creates an instance of the `pm_Pathbar` class that presents a pathbar GUI control on the HTML page.

Syntax

```
pm_pathbarMaker()
```

Parameters

No

Returns

An *instance* of the *pm_Pathbar class*.

Remarks

The function is designed to control the creation of the only instance of the bathbar GUI control on the HTML page. If the creation of the instance fails, Plesk throws an error message.

Include: pm.php.

pm_pathbarDestructor

Saves parameters of the pathbar GUI control to the gobal scope before this control is destroyed.

Syntax

```
pm_pathbarDestructor()
```

Parameters

No

Returns

A *boolean* value which is always *true*, both in case parameters have been flushed successfully or in case the update is not required.

Remarks

If called for the form not having an instance of the pathbar GUI control yet, creates this instance. Next step is saving pathbar parameters to the global scope.

Include: pm.php.

Modules API Classes

This section of Plesk Modules API Reference contains syntax and semantic information for the Plesk Modules API classes that can be used for programming purposes when creating a module.

CLASS	DESCRIPTION
pm_Checker	Provides a variety of member functions for data format checkup.
pm_cList	Presents a GUI element of the same name. Is abstract, needs to be extended.
pm_Form	Implements the creation of the HTML form based on the specified set of templates. Is abstract, needs to be extended.
pm_Pathbar	Implements the path bar GUI element. Is abstract, needs to be extended.

class pm_Checker

The pm_Checker class provides a variety of member functions for data format checkup.

Methods	Description
login ($login)	Checks whether the Plesk Server Administrator login is valid.
sys_login ($login)	Checks whether the system login is valid.
sys_passwd ($login, $password)	Checks whether the system password is valid.
ftp_login ($login)	Checks whether the FTP login is valid.

ftp_passwd ($password)	Checks whether the FTP password is valid.
pg_login ($login)	Checks whether the PostgreSQL database login is valid.
pg_passwd ($login, $password)	Checks whether the PostgreSQL database password is valid.
mailname ($mail_name)	Checks whether the specified email name is valid.
mail_passwd ($login, $password)	Checks whether the password to the specified email account is valid.
resp_name ($resp_name)	Checks whether the name of an autoresponder is valid.
domain ($dom_name)	Checks whether the specified domain name is valid.
idn_rfc_domain ($idn_dom_name)	Checks whether the specified domain name in IDN format is valid..
rfc_domain ($dom_name)	Checks whether the specified domain name is valid as per RFC 1035.
cert_domain ($dom_name)	Checks whether the domain name specified in the SSL certificate is valid.
idn_domain ($idn_dom_name)	Checks whether the IDN formatted domain name is valid.
atdomain ($name)	Validates both parts of an email name (standing before and after the at-sign).
subdomain ($name)	Checks whether the subdomain name is valid.
idn_subdomain ($idn_name)	Checks whether the subdomain name formatted as IDN is valid.
dns_dom($dom_name, $allow_mask)	Checks whether the specified domain name is well-formatted for use in the DNS system.
dns_dom_t($dom_name, $allow_mask)	Checks whether the specified DNS domain name can be used as a DNS template.

chk_ip_t($ip_address)	Checks whether the specified IP address can be used in the DNS template.
url($url, $proto_required)	Checks whether the specified URL is well-formatted.
localUrl($url)	Checks the local part of URL (following the domain part) for disallowed characters.
mailto($href)	Checks whether a ‘mailto’ string is well-formatted.
shortUrl($url)	Checks whether the specified short URL (like http://domain.com) is well-formatted.
protectedDirName($dir)	Checks whether the specified protected directory name is well-formatted.
siteAppInstallPrefix($install_prefix)	Checks whether the specified module application’s installation directory name is well-formatted.
realm($realm)	Checks whether the specified caption displayed in the authorization window for a protected web folder is well-formatted.
dbName($db)	Checks whether the specified database name is formatted properly.
dbUserName($usr)	Checks whether the specified DB user name is formatted properly.
personalName($name)	Checks whether the specified personal name is formatted properly.
companyName($company)	Checks whether the specified company name is formatted properly.
phone($phone)	Checks whether the specified phone number is well-formatted.
fax($fax)	Checks whether the specified fax number is well-formatted.

email($email)	Checks whether the specified email address is well-formatted.
address($address)	Checks whether the specified address is well-formatted.
city($city)	Checks whether the specified city name is well-formatted.
state($state, $country)	Checks whether the specified state name is well-formatted.
zip($zip, $country)	Checks whether the specified zip code is well-formatted.
country($country)	Checks whether the specified country code is well-formatted and exists in the list of country codes of Plesk.
us_states()	Returns an array of US states and places of US military presence.
ca_states()	Returns an array of Canadian states.
ip($ip)	Checks whether the specified IP address is well-formatted.
ip_address_and_mask($ip_address, $ip_mask, $valid)	Checks whether the specified IP address presents the expected type of IP address (non-masked IP address, masked IP address, masked IP range, etc.).
ip_interface($ip_address, $ip_mask)	Checks whether the specified IP address and IP mask are valid settings for a network interface.
ip_address($str, $valid, $valid_formats)	Checks whether the specified IP address satisfies the selected validation rule and IP format.
cidr_addr($cidr)	Checks whether the IP address specified in the CIDR notation is well-formatted.
mask($mask)	Checks whether the specified subnet IP mask is valid.
netaddr($netaddr, $netmask)	Checks whether the IP address presents a network address.

filename($filename)	Checks whether the specified file name is formatted properly.
filepath($filepath, $base, $user)	Checks whether the path exists and the specified user has necessary access permissions for this folder.
int($num)	Checks whether the specified integer value is well-formatted.
spamassassinPattern($pattern)	Validates the SpamAssassin pattern that specifies groups of addresses for white and black lists.
FTPMessage($msg)	Checks whether the specified FTP message is well-formatted.

Include: pm.php.

pm_Checker:: login Method

Checks whether Plesk Server Administrator's login is valid.

Syntax

```
public static function login ($login)
```

Parameters

login

A *string* value that specifies Plesk Server Administrator's login.

Returns

An *integer* which is 1 if the login is considered valid, *false* (0) otherwise.

Remarks

A valid Plesk Server Administrator login should be 1 to 20 characters long, its first character being a Latin letter or a digit, and its other characters being Latin letters, digits, and characters '_' (underscore), '.' (dot), and '-' (hyphen).

pm_Checker:: sys_login Method

Checks whether the system login is valid.

Syntax

```
public static function sys_login ($login)
```

Parameters

login

A *string* value that specifies the system login.

Returns

An *integer* which is 1 if the login is valid, *false* (0) otherwise.

Remarks

A system login can be an empty string or a string up to 17 characters long. A valid system login should contain a lowercase Latin letter at its first position, and the remaining characters can be lowercase Latin letters, digits, and characters '_' (underscore) and '-' (hyphen).

pm_Checker:: sys_passwd Method

Checks whether the system password is valid.

Syntax

```
public static function sys_passwd ($login, $password)
```

Parameters

login

A *string* value that specifies the system login.

password

A *string* value that specifies the system password.

Returns

A *boolean* set to *true* if the system password is valid, *false* otherwise.

Remarks

A valid system password allows ASCII characters (with codes 0 to 127) only. Also, it shouldn't contain the system login, and it should be 5 to 14 characters long.

pm_Checker:: ftp_login Method

Checks whether the FTP login is valid.

Syntax

```
public static function ftp_login ($login)
```

Parameters

login

A *string* value that specifies FTP login.

Returns

An *integer* which is 1 if the login is valid, *false* (0) otherwise.

Remarks

A valid login consists of 0 to 16 symbols and can contain Latin letters, digits, and characters ‘_’ (underscore) and ‘-’ (hyphen).

pm_Checker:: ftp_passwd Method

Checks whether the FTP password is valid.

Syntax

```
public static function ftp_passwd ($password)
```

Parameters

password

A *string* value that specifies FTP password.

Returns

A *boolean* - *true* if the FTP password is valid, *false* otherwise.

Remarks

A valid FTP password can be an empty string or a string of ASCII symbols (with codes 0 to 127) only. Escape characters ‘\n’ (LF, ASCII code 10) and ‘\r’ (CR, ASCII code 13) are not allowed.

pm_Checker:: pg_login Method

Checks whether the PostgreSQL database login is valid.

Syntax

```
public static function pg_login ($login)
```

Parameters

login

A *string* value that specifies PostgreSQL database login.

Returns

An *integer* which is 1 if the PostgreSQL database login is valid, *false* (0) otherwise.

Remarks

A PostgreSQL database login can be an empty string or a string up to 17 characters long. A valid login should contain a lowercase Latin letter at its first position, and the remaining characters can be lowercase Latin letters, digits, and characters ‘_’ (underscore) and ‘-’ (hyphen).

pm_Checker:: pg_passwd Method

Checks whether the PostgreSQL database password is valid.

Syntax

```
public static function pg_passwd ($login, $password)
```

Parameters

login

A *string* value that specifies the PostgreSQL database login.

password

A *string* value that specifies the PostgreSQL database password.

Returns

A *boolean* set to *true* if the PostgreSQL database password is valid, *false* otherwise.

Remarks

A valid PosrgreSQL database password allows ASCII characters (with codes 0 to 127) only. Also, it shouldn't contain the system login, and it should be 5 to 14 characters long.

pm_Checker:: mailname Method

Checks whether the specified email name is valid.

Syntax

```
public static function mailname ($mail_name)
```

Parameters

mail_name

A *string* value that specifies the email name to check.

Returns

An *integer* which is 1 if the email name is considered valid, 0 otherwise.

Remarks

A valid name of the email specified in the mail_name parameter can contain letters, digits, dots, '_' (underscore), '+' and '-' (hyphen) characters. It should begin with a letter, digit, underscore, or a hyphen. E.g. ann.miller-77.

pm_Checker:: mail_passwd Method

Checks whether the password to the specified email account is valid.

Syntax

```
public static function mail_passwd ($login, $password)
```

Parameters

login

A *string* value that specifies the email account name.

password

A *string* value that specifies the password to the email account.

Returns

A *boolean* set to *true* if the email password is valid, *false* otherwise.

Remarks

A valid email password should be 5 to 14 characters long, it should contain ASCII characters (with codes 0 to 127) only, and a password shouldn't contain the name of the related email account.

pm_Checker:: resp_name Method

Checks whether the specified string can be used as a valid name of an autoresponder.

Syntax

```
public static function resp_name ($resp_name)
```

Parameters

resp_name

A *string* value that specifies the name of an autoresponder to check.

Returns

A *boolean* set to *true* if the name of the autoresponder is valid, *false* otherwise.

Remarks

The autoresponder name shouldn't be empty or longer than 245 characters.

pm_Checker:: domain Method

Checks whether the specified domain name is valid.

Syntax

```
public static function domain ($dom_name)
```

Parameters

dom_name

A *string* value that specifies the domain name to check.

Returns

A *boolean* set to *true* if the domain name is well-formatted, *false* otherwise.

Remarks

This function is a particular case of pm_Checker::rfc_domain, except it imposes tighter restrictions on the length of the passed in domain name.

A valid domain name shouldn't be longer than 245 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa' (reverse resolution from an IP address to the fully qualified domain address). IP addresses are also disallowed.

A valid domain name specified in the dom_name parameter should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters. E.g. `Plesk-8.main-support.support.com.`

pm_Checker:: idn_rfc_domain Method

Checks whether the specified domain name in IDN format is valid.

Syntax

```
public static function idn_rfc_domain ($idn_dom_name)
```

Parameters

idn_dom_name

A *string* value that specifies the domain name in IDN format.

Returns

A *boolean* set to *true* if the domain name is well-formatted, *false* otherwise.

Remarks

This function is a particular case of pm_Checker::rfc_domain, except it converts the passed in domain name to the ASCII format first.

A valid domain name shouldn't be longer than 255 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa' (reverse resolution from an IP address to the fully qualified domain address). IP addresses are also disallowed.

A valid domain name specified in the idn_dom_name parameter should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters. E.g. `Plesk-8.main-support.support.com.`

pm_Checker:: rfc_domain Method

Checks whether the specified domain name is valid according to RFC 1035.

Syntax

```
public static function rfc_domain ($dom_name)
```

Parameters

dom_name

A *string* value that specifies the domain name to check.

Returns

A *boolean* set to *true* if the domain name is well-formatted, *false* otherwise.

Remarks

A valid domain name shouldn't be longer than 255 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa' (reverse resolution from an IP address to the fully qualified domain address). IP addresses are also disallowed.

A valid domain name specified in the dom_name parameter should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters. E.g. `Plesk-8.main-support.support.com.`

pm_Checker:: cert_domain Method

Checks whether the domain name specified in the SSL certificate is valid.

Syntax

```
public static function cert_domain ($dom_name)
```

Parameters

dom_name

A *string* value with the domain name specified in the SSL certificate.

Returns

An *integer* which is 1 if the login is valid, *false* (0) otherwise.

Remarks

The *dom_name* parameter can specify the entire domain name, e.g. `plesk.com`, or a range of subdomains, which can be done using a wildcard, e.g. `*.plesk.com`. The allowed format is `<label>[.<label>[…]]`, where a label is a wildcard or a block of 63 characters or less, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters. The specified domain name shouldn't be longer than 255 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa' (reverse resolution from an IP address to the fully qualified domain address). IP addresses are also disallowed.

pm_Checker:: idn_domain Method

Checks whether the IDN formatted domain name is valid.

Syntax

```
public static function idn_domain ($idn_dom_name)
```

Parameters

idn_dom_name

A *string* value that specifies the domain name in IDN format.

Returns

A *boolean* set to *true* if the domain name is well-formatted, *false* otherwise.

Remarks

This function is a particular case of pm_Checker::domain, except it converts the passed in domain name to the ASCII format first.

A valid domain name shouldn't be longer than 245 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa' (reverse resolution from an IP address to the fully qualified domain address). IP addresses are also disallowed.

A valid domain name specified in the idn_dom_name parameter should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters. E.g. `Plesk-8.main-support.support.com.`

pm_Checker:: atdomain Method

Validates both parts of an email name (standing before and after the at-sign).

Syntax

```
public static function atdomain ($name)
```

Parameters

name

A *string* value that specifies the email name to check.

Returns

A *boolean* set to *true* if both parts of the specified email name are well-formatted, *false* otherwise.

Remarks

The first part (preceding the at-sign) can be an empty string or a string up to 17 characters long. It should begin with a lowercase Latin letter and can contain lowercase Latin letters, digits, and characters '_' (underscore) and '-' (hyphen).

The second part (following the at-sign) shouldn't be longer than 245 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa'. It cannot be an IP address either. It should present a valid domain name formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters.

pm_Checker:: subdomain Method

Checks whether the subdomain name is valid.

Syntax

```
public static function subdomain ($name)
```

Parameters

name

A *string* value that specifies subdomain name to check.

Returns

A *boolean* set to *true* if the subdomain name is well-formatted, *false* otherwise.

Remarks

A valid subdomain name shouldn't be longer than 253 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa'. IP addresses are also disallowed. It should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters.

pm_Checker:: idn_subdomain Method

Checks whether the subdomain name in IDN format is valid.

Syntax

```
public static function idn_subdomain ($idn_name)
```

Parameters

name

A *string* value that specifies the IDN formatted subdomain name.

Returns

A *boolean* set to *true* if the subdomain name is well-formatted, *false* otherwise.

Remarks

This function is a particular case of pm_Checker::subdomain, except it converts the passed in subdomain name to the ASCII format first.

A valid subdomain name shouldn't be longer than 253 characters, shouldn't be 'localhost.rev' or ending with '.in-addr.arpa'. IP addresses are also disallowed. It should be formatted as a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]]`, each label up to 63 characters long, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters.

pm_Checker:: dns_dom Method

Checks whether the specified domain name is well-formatted for use in the DNS system.

Syntax

```
public static function dns_dom ($dom_name, $allow_mask)
```

Parameters

dom_name

A *string* value that specifies the domain name.

allow_mask

A *boolean* value that indicates whether the domain's DNS name can be masked: ALLOW_MASK if it can, NOT_ALLOW_MASK otherwise. By default, it is set to ALLOW_MASK.

Returns

An *integer* which is 1 if the domain name format fits well to be used as a DNS name, 0 otherwise.

Remarks

The *dom_name* parameter can specify the entire domain name, e.g. `plesk.com`, or a range of subdomains, which can be done using a wildcard, e.g. `*.plesk.com.`

The *allow_mask* parameter serves to indicate which format is used currently – the full one, or the masked one.

Thus, if *allow_mask* is set to NOT_ALLOW_MASK, the full format of the DNS domain name is expected and a standard format checkup is applied. A valid DNS domain name shouldn't be longer than 254 character and should have a format as follows: `<label>.[<label>.[…]]`, where a label can be up to 63 characters long and composed of Latin letters, digits and '-' (hyphen) characters.

If *allow_mask* is ALLOW_MASK, a valid DNS domain name can be formatted similar to a non-masked DNS domain name, but the first label can be substituted by '*' (a wildcard).

pm_Checker:: dns_dom_t Method

Checks whether the specified DNS domain name can be used as a DNS template.

Syntax

```
public static function dns_dom_t ($dom_name, $allow_mask)
```

Parameters

dom_name

A *string* value that specifies the domain name.

allow_mask

A *boolean* value that indicates whether the DNS name can be masked: ALLOW_MASK if it can, NOT_ALLOW_MASK otherwise. By default, it is set to ALLOW_MASK.

Returns

An *integer* which is 1 if the specified domain name can be used as a DNS template, 0 otherwise.

Remarks

The *dom_name* parameter can specify the entire domain name, e.g. plesk.com, or a range of subdomains, which can be done using a wildcard, e.g. *.plesk.com, or a DNS template, e.g. <template>.plesk.com.

The *allow_mask* parameter serves to indicate which format is used currently – the full one, or the masked one. If *allow_mask* is set to NOT_ALLOW_MASK, the full format of the domain name is expected and a standard format checkup is applied. If *allow_mask* is ALLOW_MASK, a wildcard is searched within the domain name and a proper format checkup is used.

A valid DNS template can have a format as follows:
`T.|(<label>.[<label>.[…]][T.])`, where `T` stands for the `<domain>` template, and `<label>` can be up to 63 characters long and composed of Latin letters, digits and '-' (hyphen) characters. E.g. `somedomain.<domain>.com.` If the masked form of the DNS template is allowed, the first label can be substituted by a wildcard ('*').

pm_Checker:: chk_ip_t Method

Checks whether the specified IP address can be used in the DNS template.

Syntax

```
public static function chk_ip_t($ip_address)
```

Parameters

ip_address

A *string* value that specifies the IP address to check.

Returns

A *boolean* set to *true* if the IP address is not NULL or 0.0.0.0, or if it is equal to `<ip>`, *false* otherwise.

pm_Checker:: url Method

Checks whether the specified URL is well-formatted.

Syntax

```
public static function url ($url, $proto_required)
```

Parameters

url

A *string* value that specifies the URL to check.

proto_required

A *boolean* value which is *true* if the URL requires a protocol (ftp://, http://, or https://), *false* otherwise. Is set to *false* by default.

Returns

A *boolean* which is *true* if the URL format is considered valid, *false* otherwise.

Remarks

A valid URL can have a format as follows:
`http(s)://<domain><path/file><parameters>` or
ftp://<user:password><domain:port><path/file><parameters>.

If the *proto_required* parameter is set to *false*, the URL is missing its protocol part (ftp://, http://, or https://).

The function checks the main part of URL that follows the protocol part (if any). The part in focus shouldn't contain space characters, single and double quotes, and apostrophes. Allowed are the valid formats of a domain name (see the Remarks section for domain()) and of an IP address (see the Remarks section for ip()).

pm_Checker:: localUrl Method

Checks the local part of URL (following the domain part) for disallowed characters.

Syntax

```
public static function localUrl ($url)
```

Parameters

url

A *string* value that specifies the URL to check.

Returns

An *integer* which is 1 if the local URL is well-formatted, 0 otherwise.

Remarks

A valid local URL should have a leading '/' (slash) character and shouldn't contain space characters, single and double quotes, and apostrophes.

pm_Checker:: mailto Method

Checks whether a ‘mailto’ string is well-formatted.

Syntax

```
public static function mailto ($href)
```

Parameters

href

A *string* value that specifies an email address referenced from an HTML page using the ‘mailto’ protocol.

Returns

A *boolean* which is *true* if the ‘mailto’ string is well-formatted, *false* otherwise.

Remarks

A valid ‘mailto’ string should have the following format: mailto:<email_name>@<domain>.

The `<email_name>` part should be at least 1 character long (it shouldn’t be missing), and it can contain Latin letters, digits, dots, ‘_’ (underscore), ‘+’ and ‘-’ (hyphen) characters. It should begin with a Latin letter, a digit, an underscore, or a hyphen.

The `<domain>` part should be formatted as `<label>[.<label>[…]]`, where a label is a block of 63 characters or less, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and ‘-’ (hyphen) characters.

pm_Checker:: shortUrl Method

Checks whether the specified short URL (e.g. http://domain.com) is well-formatted.

Syntax

```
public static function shortUrl ($url)
```

Parameters

url

A *string* value that specifies the short URL to check.

Returns

A *boolean* which is *true* if the short URL is well-formatted, *false* otherwise.

Remarks

A valid short URL should have a format as follows: `[<protocol>]<host_name>`. The protocol part (ftp://, http://, https://) can be missing. The host part can be a valid domain name formatted as `<label>[.<label>[…]]`, where a label is a block of 63 characters or less, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and ‘-’ (hyphen) characters. Or the host part can be a valid IP address.

pm_Checker:: protectedDirName Method

Checks whether the specified protected directory name is well-formatted.

Syntax

```
public static function protectedDirName ($dir)
```

Parameters

dir

A *string* value that specifies the directory name to check.

Returns

A *boolean* which is *true* if directory name is well-formatted, *false* otherwise.

Remarks

A well-formatted protected directory name can contain Latin letters, digits, dots (‘.’), slash characters (‘/’), and hyphens (‘-’). The following chains are not allowed: ‘/../’, ‘//’, ‘./’ and ‘../’ at the beginning of the directory name, ‘/./’, ‘/..’, ‘/.’, and ‘.’ at the end of the directory name. The directory name shouldn’t be longer than 247 characters after the leading and trailing spaces and ‘/’ characters are removed.

pm_Checker:: siteAppInstallPrefix Method

Checks whether the name of the installation directory for a site application is well-formatted.

Syntax

```
public static function siteAppInstallPrefix
($install_prefix)
```

Parameters

install_prefix

A *string* value that specifies the name of the folder where the site application will be unpacked.

Returns

A *boolean* which is *true* if installation folder name is well-formatted, *false* otherwise.

Remarks

A valid name of a site application’s installation folder can contain Latin letters, digits, underscores (‘_’), dots (‘.’), slash characters (‘/’), and hyphens (‘-’). The chains that are not allowed are: a leading or trailing dot (‘.’) or a slash (‘/’), combinations like ‘./’ and ‘../’ at the beginning of the install_prefix string, combinations like ‘/..’, ‘/.’ At the end of the install_prefix string, and combinations like ‘/../’, ‘//’, ‘/./’ within this string.

The function returns *true* if the specified folder name satisfies the above listed conditions, or if it is equal to SITEAPP_PREFIX_ROOT (the current folder).

pm_Checker:: realm Method

Checks whether the specified caption displayed in the authorization window for a protected web folder is well-formatted.

Syntax

```
public static function realm ($realm)
```

Parameters

realm

A *string* value that specifies the text (the caption of a protected folder) shown in the authorization window when a user tries to access a protected resource.

Returns

A *boolean* which is *true* if the caption of a protected folder is well-formatted, *false* otherwise.

Remarks

A valid protected folder caption shouldn't be double quoted.

pm_Checker:: dbName Method

Checks whether the specified database name is formatted properly.

Syntax

```
public static function dbName ($db)
```

Parameters

db

A *string* value that specifies the database name to check.

Returns

A *boolean* which is *true* if the DB name is well-formatted, *false* otherwise.

Remarks

A well-formatted database name should not be longer than 63 characters and should only consist of Latin letters, digits, hyphens ('-') and underscores ('_').

pm_Checker:: dbUserName Method

Checks whether the specified DB user name is formatted properly.

Syntax

```
public static function dbUserName ($usr)
```

Parameters

usr

A *string* value that specifies the DB user name to check.

Returns

A *boolean* which is *true* if the DB user name is well-formatted, *false* otherwise.

Remarks

A well-formatted DB user name should be 1 to 16 characters long, should begin with a Latin letter, and shouldn't contain anything except Latin letters, digits, and underscores ('_').

pm_Checker:: personalName Method

Checks whether the specified personal name is formatted properly.

Syntax

```
public static function personalName ($name)
```

Parameters

name

A *string* value that specifies the name to check.

Returns

A *boolean* which is *true* if the personal name is well-formatted, *false* otherwise.

Remarks

A well-formatted personal name can consist of any characters, should not be longer than 255 characters, and shouldn't be enclosed in space or tab characters.

pm_Checker:: companyName Method

Checks whether the specified company name is formatted properly.

Syntax

```
public static function companyName ($company)
```

Parameters

company

A *string* value that specifies the company name to check.

Returns

A *boolean* which is *true* if the company name is well-formatted, *false* otherwise.

Remarks

A well-formatted company name should be 1 to 100 characters long.

pm_Checker:: phone Method

Checks whether the specified phone number is well-formatted.

Syntax

```
public static function phone ($phone)
```

Parameters

phone

A *string* value that specifies the phone number to check.

Returns

A *boolean* which is *true* if the phone number is well-formatted, *false* otherwise.

Remarks

A well-formatted phone number shouldn't be longer than 30 characters, should start with a digit, and can contain digits and delimiters like parentheses '(' and ')', hyphens ('-') and '+' characters.

pm_Checker:: fax Method

Checks whether the specified fax number is well-formatted.

Syntax

```
public static function fax ($fax)
```

Parameters

fax

A *string* value that specifies the fax number to check.

Returns

A *boolean* which is *true* if the fax number is well-formatted, *false* otherwise.

Remarks

A well-formatted fax number shouldn't be longer than 30 characters, should start with a digit, and can contain digits and delimiters like parentheses '(' and ')', hyphens ('-') and '+' characters.

pm_Checker:: email Method

Checks whether the specified email address is well-formatted.

Syntax

```
public static function email ($email)
```

Parameters

email

A *string* value that specifies the email name to check.

Returns

A *boolean* which is *true* if the email name is well-formatted, *false* otherwise.

Remarks

A valid email name should have the following format: mailto:<email_name>@<domain>.

The <email_name> part should be at least 1 character long (it shouldn't be missing), and it can contain Latin letters, digits, dots, '_' (underscore), '+' and '-' (hyphen) characters. It should begin with a Latin letter, a digit, an underscore, or a hyphen.

The <domain> part should be formatted as <label>[.<label>[…]], where a label is a block of 63 characters or less, beginning with a Latin letter and composed of Latin letters (A-Z, a-z), digits (0-9) and '-' (hyphen) characters.

pm_Checker:: address Method

Checks whether the specified address is well-formatted.

Syntax

```
public static function address ($address)
```

Parameters

address

A *string* value that specifies the address to check.

Returns

A *boolean* which is *true* if the address is well-formatted, *false* otherwise.

Remarks

A well-formatted address should be a string 1 to 255 characters long. Any characters are allowed.

pm_Checker:: city Method

Checks whether the specified city name is well-formatted.

Syntax

```
public static function city ($city)
```

Parameters

city

A *string* value that specifies the city name to check.

Returns

A *boolean* which is *true* if the city name is well-formatted, *false* otherwise.

Remarks

A well-formatted city name should be a string 2 to 50 characters long.

pm_Checker:: state Method

Checks whether the specified state name is well-formatted.

Syntax

```
public static function state ($state, $country)
```

Parameters

state

A *string* value that specifies the state name to check.

country

A *string* value that specifies what country the state in focus belongs to. Holds an empty string by default.

Returns

A *mixed* value. If the state name is well-formatted and the specified country is US or Canada, returns the array of strings containing states and their 2-character codes in the format `'<state_name>=><state_code>'`. If the state name is well-formatted and the country is unknown or not specified, returns the string with the passed in state, otherwise returns *false.*

Remarks

A well-formatted state name for a country different from US or Canada should be a string 0 to 50 characters long.

pm_Checker:: zip Method

Checks whether the specified zip code is well-formatted.

Syntax

```
public static function zip ($zip, $country)
```

Parameters

zip

A *string* value that specifies the zip code to check.

country

A *string* value that specifies what country the zip code belongs to. Holds an empty string by default.

Returns

A *boolean* which is *true* if the zip code is well-formatted, *false* otherwise.

Remarks

A well-formatted zip code name should be a string 0 to 10 characters long. The US zip code should be a string of digits in one of the following formats: xxxxx, xxxxx-xxxx.

pm_Checker:: country Method

Checks whether the specified country code is well-formatted and exists in the list of country codes of Plesk.

Syntax

```
public static function country ($country)
```

Parameters

country

A *string* value that specifies the country code to check.

Returns

A *boolean* which is *true* if the country code is well-formatted and exists in the list of country codes of Plesk, *false* otherwise.

Remarks

A well-formatted country code should be a 2-character string. E.g., CA, US, etc. The list of country codes accepted by Plesk is available in the `countries.php3` file. It is defined as follows:

```
$countries = array (
'AF' => 'Afghanistan',
'AX' => 'Aland Islands',
'AL' => 'Albania',
'DZ' => 'Algeria',
'AS' => 'American Samoa',
'AD' => 'Andorra',
'AO' => 'Angola',
'AI' => 'Anguilla',
'AQ' => 'Antarctica',
'AG' => 'Antigua and Barbuda',
'AR' => 'Argentina',
'AM' => 'Armenia',
'AW' => 'Aruba',
'AU' => 'Australia',
'AT' => 'Austria',
'AZ' => 'Azerbaijan',
```

```
'BS' => 'Bahamas',
'BH' => 'Bahrain',
'BD' => 'Bangladesh',
'BB' => 'Barbados',
'BY' => 'Belarus',
'BE' => 'Belgium',
'BZ' => 'Belize',
'BJ' => 'Benin',
'BM' => 'Bermuda',
'BT' => 'Bhutan',
'BO' => 'Bolivia',
'BA' => 'Bosnia and Herzegovina',
'BW' => 'Botswana',
'BV' => 'Bouvet Island',
'BR' => 'Brazil',
'IO' => 'British Indian Ocean Territory',
'BN' => 'Brunei Darussalam',
'BG' => 'Bulgaria',
'BF' => 'Burkina Faso',
'BI' => 'Burundi',
'KH' => 'Cambodia',
'CM' => 'Cameroon',
'CA' => 'Canada',
'CV' => 'Cape Verde',
'KY' => 'Cayman Islands',
'CF' => 'Central African Republic',
'TD' => 'Chad',
'CL' => 'Chile',
'CN' => 'China',
'CX' => 'Christmas Island',
'CC' => 'Cocos (Keeling) Islands',
'CO' => 'Colombia',
'KM' => 'Comoros',
'CG' => 'Congo',
'CD' => 'Congo, the Democratic Republic of the',
'CK' => 'Cook Islands',
'CR' => 'Costa Rica',
'CI' => 'Cote D'Ivoire',
'HR' => 'Croatia',
'CU' => 'Cuba',
'CY' => 'Cyprus',
'CZ' => 'Czech Republic',
'DK' => 'Denmark',
'DJ' => 'Djibouti',
'DM' => 'Dominica',
'DO' => 'Dominican Republic',
'EC' => 'Ecuador',
'EG' => 'Egypt',
'SV' => 'El Salvador',
'GQ' => 'Equatorial Guinea',
'ER' => 'Eritrea',
'EE' => 'Estonia',
'ET' => 'Ethiopia',
'FK' => 'Falkland Islands (Malvinas)',
'FO' => 'Faroe Islands',
'FJ' => 'Fiji',
'FI' => 'Finland',
'FR' => 'France',
'GF' => 'French Guiana',
'PF' => 'French Polynesia',
```

```
'TF' => 'French Southern Territories',
'GA' => 'Gabon',
'GM' => 'Gambia',
'GE' => 'Georgia',
'DE' => 'Germany',
'GH' => 'Ghana',
'GI' => 'Gibraltar',
'GR' => 'Greece',
'GL' => 'Greenland',
'GD' => 'Grenada',
'GP' => 'Guadeloupe',
'GU' => 'Guam',
'GT' => 'Guatemala',
'GN' => 'Guinea',
'GW' => 'Guinea-Bissau',
'GY' => 'Guyana',
'HT' => 'Haiti',
'HM' => 'Heard Island and McDonald Islands',
'VA' => 'Holy See (Vatican City State)',
'HN' => 'Honduras',
'HK' => 'Hong Kong',
'HU' => 'Hungary',
'IS' => 'Iceland',
'IN' => 'India',
'ID' => 'Indonesia',
'IR' => 'Iran, Islamic Republic of',
'IQ' => 'Iraq',
'IE' => 'Ireland',
'IL' => 'Israel',
'IT' => 'Italy',
'JM' => 'Jamaica',
'JP' => 'Japan',
'JO' => 'Jordan',
'KZ' => 'Kazakhstan',
'KE' => 'Kenya',
'KI' => 'Kiribati',
'KP' => 'Korea, Democratic People's Republic of',
'KR' => 'Korea, Republic of',
'KW' => 'Kuwait',
'KG' => 'Kyrgyzstan',
'LA' => 'Lao People's Democratic Republic',
'LV' => 'Latvia',
'LB' => 'Lebanon',
'LS' => 'Lesotho',
'LR' => 'Liberia',
'LY' => 'Libyan Arab Jamahiriya',
'LI' => 'Liechtenstein',
'LT' => 'Lithuania',
'LU' => 'Luxembourg',
'MO' => 'Macao',
'MK' => 'Macedonia, the Former Yugoslav Republic of',
'MG' => 'Madagascar',
'MW' => 'Malawi',
'MY' => 'Malaysia',
'MV' => 'Maldives',
'ML' => 'Mali',
'MT' => 'Malta',
'MH' => 'Marshall Islands',
'MQ' => 'Martinique',
'MR' => 'Mauritania',
```

```
'MU' => 'Mauritius',
'YT' => 'Mayotte',
'MX' => 'Mexico',
'FM' => 'Micronesia, Federated States of',
'MD' => 'Moldova, Republic of',
'MC' => 'Monaco',
'MN' => 'Mongolia',
'MS' => 'Montserrat',
'MA' => 'Morocco',
'MZ' => 'Mozambique',
'MM' => 'Myanmar',
'NA' => 'Namibia',
'NR' => 'Nauru',
'NP' => 'Nepal',
'NL' => 'Netherlands',
'AN' => 'Netherlands Antilles',
'NC' => 'New Caledonia',
'NZ' => 'New Zealand',
'NI' => 'Nicaragua',
'NE' => 'Niger',
'NG' => 'Nigeria',
'NU' => 'Niue',
'NF' => 'Norfolk Island',
'MP' => 'Northern Mariana Islands',
'NO' => 'Norway',
'OM' => 'Oman',
'PK' => 'Pakistan',
'PW' => 'Palau',
'PS' => 'Palestinian Territory, Occupied',
'PA' => 'Panama',
'PG' => 'Papua New Guinea',
'PY' => 'Paraguay',
'PE' => 'Peru',
'PH' => 'Philippines',
'PN' => 'Pitcairn',
'PL' => 'Poland',
'PT' => 'Portugal',
'PR' => 'Puerto Rico',
'QA' => 'Qatar',
'RE' => 'Reunion',
'RO' => 'Romania',
'RU' => 'Russian Federation',
'RW' => 'Rwanda',
'SH' => 'Saint Helena',
'KN' => 'Saint Kitts and Nevis',
'LC' => 'Saint Lucia',
'PM' => 'Saint Pierre and Miquelon',
'VC' => 'Saint Vincent and the Grenadines',
'WS' => 'Samoa',
'SM' => 'San Marino',
'ST' => 'Sao Tome and Principe',
'SA' => 'Saudi Arabia',
'SN' => 'Senegal',
'CS' => 'Serbia and Montenegro',
'SC' => 'Seychelles',
'SL' => 'Sierra Leone',
'SG' => 'Singapore',
'SK' => 'Slovakia',
'SI' => 'Slovenia',
'SB' => 'Solomon Islands',
```

```
'SO' => 'Somalia',
'ZA' => 'South Africa',
'GS' => 'South Georgia and the South Sandwich Islands',
'ES' => 'Spain',
'LK' => 'Sri Lanka',
'SD' => 'Sudan',
'SR' => 'Suriname',
'SJ' => 'Svalbard and Jan Mayen',
'SZ' => 'Swaziland',
'SE' => 'Sweden',
'CH' => 'Switzerland',
'SY' => 'Syrian Arab Republic',
'TW' => 'Taiwan',
'TJ' => 'Tajikistan',
'TZ' => 'Tanzania, United Republic of',
'TH' => 'Thailand',
'TL' => 'Timor-Leste',
'TG' => 'Togo',
'TK' => 'Tokelau',
'TO' => 'Tonga',
'TT' => 'Trinidad and Tobago',
'TN' => 'Tunisia',
'TR' => 'Turkey',
'TM' => 'Turkmenistan',
'TC' => 'Turks and Caicos Islands',
'TV' => 'Tuvalu',
'UG' => 'Uganda',
'UA' => 'Ukraine',
'AE' => 'United Arab Emirates',
'GB' => 'United Kingdom',
'US' => 'United States',
'UM' => 'United States Minor Outlying Islands',
'UY' => 'Uruguay',
'UZ' => 'Uzbekistan',
'VU' => 'Vanuatu',
'VE' => 'Venezuela',
'VN' => 'Vietnam',
'VG' => 'Virgin Islands, British',
'VI' => 'Virgin Islands, U.S.',
'WF' => 'Wallis and Futuna',
'EH' => 'Western Sahara',
'YE' => 'Yemen',
'ZM' => 'Zambia',
'ZW' => 'Zimbabwe',
);
```

pm_Checker:: us_states Method

Returns an array of US states and places of US military presence.

Syntax

```
public static function us_states ()
```

Parameters

No

Returns

An *array* of strings containing US states and their 2-character codes in the format `'<state_name>=><state code>'`.

pm_Checker:: ca_states Method

Returns an array of Canadian states.

Syntax

```
public static function ca_states ()
```

Parameters

No

Returns

An *array* of strings containing Canada states and their 2-character codes in the format `'<state_name>=><state code>'`.

pm_Checker:: ip Method

Checks whether the specified IP address is well-formatted.

Syntax

```
public static function ip ($ip)
```

Parameters

ip

A *string* value that specifies the IP address to check.

Returns

A *boolean* which is *true* if the IP address is well-formatted, *false* otherwise.

Remarks

A well-formatted IP address is set using the dotted-decimal notation, e.g. 192.123.12.1. This notation implies that an IP address consists of four blocks, each holding a value 0 to 255, delimited by dots. A valid address shouldn't be equal to 0.0.0.0.

pm_Checker:: ip_address_and_mask Method

Checks whether the specified IP address presents the expected type of IP address (non-masked IP address, masked IP address, masked IP range, etc.).

Syntax

```
public static function ip_address_and_mask ($ip_address,
$ip_mask, $valid)
```

Parameters

ip_address

A *string* value that specifies the IP address used to specify a valid subnet.

ip_mask

A *string* value that specifies the IP mask used to specify a valid subnet.

valid

A *string* value that specifies the validation rule to apply. Is set to IP_ADDRESS_ANY by default. The following validation rules are available:

- IP_ADDRESS_ANY allows the use of all rules available.
- IP_ADDRESS_BLOCK checks whether the IP address presents an address block (a group of network addresses) rather than a certain address.
- IP_ADDRESS_NETMASK checks whether the given IP address is an individual IP address with a mask.
- IP_ADDRESS selects the rule that checks whether the given IP mask specifies an individual IP address.

Returns

A *boolean* which is *true* if the specified IP address and IP mask are not 0.0.0.0 and the rule applied to them succeeds. Otherwise returns *false.*

Remarks

This function applies the selected validation rule(s) to the combination of the IP address and IP mask to check whether the IP address really belongs to the expected type (a non-masked IP address, a masked IP address, an IP range, etc.).

- The validation rules work as follows:
- IP_ADDRESS_BLOCK checks whether the IP address specifies a group of IP addresses (i.e. whether the IP address presents an address block). This rule requires the use of a masked IP address format, and the requirement is that the host bits are set to ‘0’. E.g. `123.123.0.0/16` is a range of addresses, but `123.123.0.1/16` is a particular IP address in a class B network.
- IP_ADDRESS_NETMASK checks whether the IP address is an individual masked IP address. The rule succeeds if the IP mask specifies the entire IP address, e.g. `123.123.123.1/32` or `123.123.123.1/255.255.255.255`. The rule fails if the IP address has ‘0’ bits in its host part, e.g. `123.123.0.0/16`, or if it has all ‘1’ bits in the host part (a broadcast address), e.g. `123.123.123.255/255.255.255.0`.
- IP_ADDRESS checks whether the IP address in focus is a non-masked individual IP address. E.g. `123.123.123.123`.
- IP_ADDRESS_ANY indicates to the function that all the rules should be applied to the IP address one after another until any of them succeeds. If none of the above rules succeeds, this rule also fails.

pm_Checker:: ip_interface Method

Checks whether the specified IP address and IP mask are valid settings for a network interface.

Syntax

```
public static function ip_interface ($ip_address, $ip_mask)
```

Parameters

ip_address

A *string* value that specifies the IP address to check.

ip_mask

A *string* value that specifies the IP mask.

Returns

A *boolean* which is *true* if the specified IP address and IP mask are considered valid for a network interface, *false* otherwise.

Remarks

The requirements to the format of an IP address are as follows: it shouldn't be equal to 0.0.0.0, neither should it belong to broadcast addresses (formatted as x.x.x.255 or 255.x.x.x) and loopback addresses (127.x.x.x). A valid IP address that can be used for a network interface should belong to class A, B, or C.

The specified IP mask shouldn't be equal to 0.0.0.0 either.

Also, the function checks whether the specified IP address and IP mask do not specified a subnet of the minimal size.

pm_Checker:: ip_address Method

Checks whether the specified IP address satisfies the selected validation rule and IP format.

Syntax

```
public static function ip_address ($str, $valid,
$valid_formats)
```

Parameters

str

A *string* value that specifies the IP address to check.

valid

A *string* value that specifies the validation rule to apply. Is set to IP_ADDRESS_ANY by default. The following validation rules are available:

- IP_ADDRESS_ANY allows the use of all rules available.

- IP_ADDRESS_BLOCK checks whether the IP address presents an address block (a group of network addresses) rather than a certain address.
- IP_ADDRESS_NETMASK checks whether the given IP address is an individual IP address with a mask.
- IP_ADDRESS selects the rule that checks whether the given IP mask specifies the whole IP address.

valid_formats

A *string* value that specifies the allowed IP address format. Is set to IP_ADDRESS_ANY_FORMAT by default. The following formats are available:

- IP_ADDRESS_MASK_FORMAT allows the IP address with a wildcard ('*').
- IP_ADDRESS_BLOCK_FORMAT allows the IP address in the CIDR notation (`xxx.xxx.xxx.xxx/n` where `n` is a number of '1' network bits in the mask).
- IP_ADDRESS_NETMASK_FORMAT allows the IP address with a network mask in the format as follows: `xxx.xxx.xxx.xxx/xxx.xxx.xxx.xxx (<IP_address>/<IP_mask>).`
- IP_ADDRESS _FORMAT allows a typical IP address in the dotted-decimal notation, i.e. `xxx.xxx.xxx.xxx.`
- IP_ADDRESS_ANY_FORMAT allows the use of any above format.

Returns

A *boolean* which is *true* if the checked IP address satisfies the validation rule(s) applied and is formatted as specified by the valid_formats constant. Otherwise returns *false*.

Remarks

This function validates the passed in IP address based on two constants that specify the format of the IP address and the validation rule(s) that should be applied to discover whether the IP address in focus really matches this format.

Here is the table of the admissible combinations of the validation rules and formats:

Format / Validation rule(s)	IP_ADDRESS_MASK_FORMAT	IP_ADDRESS_BLOCK_FORMAT	IP_ADDRESS_NETMASK_FORMAT	IP_ADDRESS_FORMAT
IP_ADDRESS_BLOCK	yes	yes	yes	no
IP_ADDRESS_NETMASK	no	yes	yes	no
IP_ADDRESS	yes	yes	yes	yes
IP_ADDRESS_BLOCK| IP_ADDRESS_NETMASK | IP_ADDRESS	yes	yes	yes	no
IP_ADDRESS_BLOCK| IP_ADDRESS_NETMASK	yes	yes	yes	no

IP_ADDRESS_NETMASK | IP_ADDRESS	yes	yes	yes	no
IP_ADDRESS_BLOCK| IP_ADDRESS	yes	yes	yes	no

The validation rule(s) to apply is (are) specified in the valid parameter. They work as follows:

- IP_ADDRESS_BLOCK checks whether the IP address specifies a group of IP addresses (i.e. whether the IP address presents an address block). This rule requires the use of a masked IP address format, and the requirement is that the host bits are set to '0'. E.g. `123.123.0.0/16` is a range of addresses, but `123.123.0.1/16` is a particular IP address in a class B network.
- IP_ADDRESS_NETMASK checks whether the IP address is an individual masked IP address. The rule succeeds if the IP mask specifies the entire IP address, e.g. `123.123.123.1/32` or `123.123.123.1/255.255.255.255`. The rule fails if the IP address has '0' bits in its host part, e.g. `123.123.0.0/16`, or if it has all '1' bits in the host part (a broadcast address), e.g. `123.123.123.255/255.255.255.0`.
- IP_ADDRESS checks whether the IP address in focus is a non-masked individual IP address. E.g. `123.123.123.123`.
- IP_ADDRESS_ANY indicates to the function that all the rules should be applied to the IP address by after another until any of them succeeds. If none of the above rules succeeds, this rule fails.

The valid_formats parameter informs the function what IP address format is validated.

- IP_ADDRESS_MASK_FORMAT means that an IP address with a wildcard (*) is expected. A wildcard is allowed in place of any block, except the first one. E.g. `123.123.123.*.`
- IP_ADDRESS_BLOCK_FORMAT means that the IP address presents a group of addresses and the CIDR notation is used to indicate the network size, e.g. `123.123.0.0/16` (16 leftmost bits are the network ones). Or this format can be applied to specify a particular IP address and to provide it with the information about the network size, e.g. `123.123.123.123/24` (the host part is the rightmost block of 8 bits). If used to specify a particular non-masked IP address, this format should look as follows: `123.123.123.123/32.`
- IP_ADDRESS_NETMASK_FORMAT is a standard dotted-decimal notation with a mask (`xxx.xxx.xxx.xxx/xxx.xxx.xxx.xxx`). The first part specifies the IP address, the second part presents a mask applied to this address. This format can be used to specify a group of IP addresses (address block), e.g. `123.123.123.0/255.255.255.0`, a masked IP address, e.g. `123.123.123.123/255.255.255.0`, or a non-masked IP address, e.g. `123.123.123.123/255.255.255.255`.
- IP_ADDRESS _FORMAT is a non-masked dotted decimal format (`xxx.xxx.xxx.xxx`). It is used to specify an individual IP address, no information about the network size provided. E.g. `123.123.123.1.`
- IP_ADDRESS_ANY_FORMAT means that the format check will be successful if the IP address matches any of the above formats.

pm_Checker:: cidr_addr Method

Checks whether the IP address specified in the CIDR notation is well-formatted.

Syntax

```
public static function cidr_addr ($cidr)
```

Parameters

cidr

A *string* value that specifies the CIDR address to check.

Returns

A *boolean* which is *true* if the CIDR address is well-formatted, *false* otherwise.

Remarks

CIDR specifies an IP address range by the combination of an IP address and its associated network mask. CIDR notation uses the following format: `xxx.xxx.xxx.xxx/n` where `n` is the number of (leftmost) '1' bits in the mask. E.g., 192.168.12.0/23.

pm_Checker:: mask Method

Checks whether the specified subnet IP mask is valid.

Syntax

```
public static function mask ($mask)
```

Parameters

mask

A *string* value that specifies the IP mask to check.

Returns

A *boolean* which is *true* if the IP mask is valid, *false* otherwise.

Remarks

A well-formatted IP mask should be formed according to the dotted-decimal notation. This notation implies that an IP mask consists of four blocks, each holding a value 0 to 255, delimited by dots. Besides, an IP mask shouldn’t have its leftmost bits set to ‘0’ and its rightmost bits set to ‘1’. The rightmost block(s) should be equal to 255, the intermediate blocks are expected to be within the following range: 128, 192, 224, 240, 248, 252, 254, 255. The leftmost block(s) should be set to 0.

pm_Checker:: netaddr Method

Checks whether the IP address presents a network address.

Syntax

```
public static function netaddr ($netaddr, $netmask)
```

Parameters

netaddr

A *string* value that specifies the IP address to check.

netmask

A *string* value with a subnet mask to be used for a checkup.

Returns

A *boolean* which is *true* if the IP address presents a network address, *false* otherwise.

Remarks

A valid format of an IP address or an IP mask is a dotted-decimal notation (`xxx.xxx.xxx.xxx`).

pm_Checker:: filename Method

Checks whether the specified file name is formatted properly.

Syntax

```
public static function filename ($filename)
```

Parameters

filename

A *string* value that specifies the file name to check.

Returns

A *boolean* which is *true* if the file name is well-formatted, *false* otherwise.

Remarks

A well-formatted file name should not contain single quotes.

pm_Checker:: filepath Method

Checks whether the path exists and the specified user has necessary access permissions for this folder.

Syntax

```
public static function filepath ($filepath, $base, $user)
```

Parameters

filepath

A *string* value that specifies the file path to check.

base

A *string* value that specifies the folder within which to search the specified folder. Is set to '/' by default.

user

A *string* value that specifies the user who should have access to the folder being checked. Is set to 'psaadm' by default.

Returns

A *boolean* which is *true* if the file path really exists in the specified base folder and the user has access permissions for this folder, *false* otherwise.

pm_Checker:: int Method

Checks whether the specified integer value is well-formatted.

Syntax

```
public static function int ($num)
```

Parameters

num

A *string* value that specifies the integer value to check.

Returns

A *boolean* which is *true* if the passed in value is formatted as integer, *false* otherwise.

Remarks

An integer value format allows any number of decimal digits (0-9), possibly preceded by – or +.

pm_Checker:: spamassassinPattern Method

Validates the SpamAssassin pattern that specifies groups of addresses for white and black lists.

Syntax

```
public static function spamassassinPattern ($pattern)
```

Parameters

pattern

A *string* value that specifies the SpamAssassin pattern.

Returns

A *boolean* which is *true* if the pattern being checked is valid, *false* otherwise.

Remarks

In SpamAssassin, one can create patterns to specify groups of addresses to be filtered off as spam, or always trusted (black and white lists). This can be done using two wildcards: '*' (an asterisk) to match any characters in an email address to check, and '?' (a question mark) to match only one character. E.g. if one specifies a pattern like myclub?@somedomain.com, this pattern will succeed for email addresses like myclub1@somedomain.com, myclub2@somedomain.com, myclubX@somedomain.com , etc. If one specifies a pattern like `*.hotmail.com`, the pattern succeeds for any email address with the `hotmail.com` domain part.

Thus, a valid pattern can be formatted as `<domain>` or `<email>@<domain>`, where `<domain>` is a series of labels glued with '.' (dot) characters, i.e. `<label>[.<label>[…]].` Each label can be up to 63 characters long, beginning with a letter and composed of letters, digits (0-9) and '-' (hyphen) characters. The allowed formats for the domain part are ASCII and IDN.

Both email and domain parts of the email address can have a wildcard or more at any position, or a wildcard can substitute `<label>`, `<email>`, or the entire `<domain>`.

pm_Checker:: FTPMessage Method

Checks whether the specified FTP message is well-formatted.

Syntax

```
public static function FTPMessage ($msg)
```

Parameters

msg

A *string* value that specifies the message to check.

Returns

A *boolean* which is *true* if the passed in message is well-formatted, *false* otherwise.

Remarks

A well-formatted FTP message allows printable characters only.

class pm_cList

The pm_cList class presents a GUI element of the same name. This class is designed as the *base* one - it shouldn't be used as it is, but can serve as a parent from which user defined classes can inherit.

The pm_cList class generates a 'list' GUI element displayed on the browser's page as a table that has a caption and several columns (according to the fields got by a query from the database), each having its name.

The data currently displayed in the table is stored in a special multidimensional array (the `list_` member variable) whose format can be defined by the user as desired.

The table columns are rather complex entities described by a number of parameters. These parameters also require a multidimensional array (the `columns_` member variable) and, among others, provide the following capabilities:

- If a certain table column has a 'filter' parameter set for it, the list can be filtered by this column. To set the filtering parameter, you should pass in some value to the instance of the `pm_cList` class. As a result, the table will display only the rows that store this specified value in the 'filter' column.
- If a certain table column has a 'sort' parameter set for it, the list can be sorted by this column.

The class provides a variety of opportunities to customize the look of the HTML table (list element). E.g. the table is normally set to output one-level data, though it can be customized to output an item in a table row and a set of sub-items in sub-rows following the "parent" row.

Also, the pm_cList class provides an internal bitmask that can be used to store any state of the HTML table or the state of any parameter.

To make the navigation through the multi-page list easier, every page displays the paging control that looks nearly as follows:

```
First 1 2 3 4 5 Last
```

The look of this control is customizable: a special member variable (`pagingSize_`) holds the number of pages to show in this control to the left and to the right of the current page.

The creation of an HTML page with a 'list' GUI control implies a definite set of invocations, that is:

```
// instantiate class my_cList
$myList = new my_cList;

// get filtering, sorting, and paging parameters from the global array
// of settings and set the matching class properties
$myList->setFSP();
// initialize the instance – get columns info, data to display, etc.
$myList->init();
// form HTML code of the page with the list (table)
```

```
$myListObj = $myList->get();
```

Initialization of an instance of class derived from `pm_cList` is not performed in `init()` in full. The matter is that the structure of multi-dimensional arrays `list_` and `columns_` is not fixed (the developers should resolve this structure in a derived class themselves). Since the `pm_cList` class knows nothing about the structure of these arrays, it does not contain any code that would initialize them. To fill the gap, you need to create a special member function in your derived class that would form a query to the database and initialize member variables `list_` and `columns_` with data. This special function should have a name as follows:

```
fetch<derived_class_name> ()
```

(the name of the derived class should be used in the lower case, no parameters are required).

In case an error occurs when reading these parameters, the `fetch<derived_class_name>` function should set its `error_` flag to *true* and assign a message text to its `errorMsg_` variable using the `setError()` member function. Finally, this function can apply custom sorting, filtering, and paging. If custom sorting has been applied, the `sortApplayed_` member variable should be set to *true* in order to prevent from original re-sorting.

Here is the list of constant values defined in class `pm_cList`:

Constant	Value	Description
`DEFAULT_FLAGS`	`0`	*Integer*. A bitmask used to specify user defined flags.
`DEFAULT_SORT`	`'name'`	*String*. The field by which the list is sorted.
`DEFAULT_FILTER`	`' '`	*String*. The value by which the list is filtered.
`DEFAULT_FILTER_BY`	`'name'`	*String*. The field by which the list is filtered.
`DEFAULT_FILTER_SHOW_COOKIE`	`'cListShowSearch'`	*String*. The name of the cookie file that stores the state of the search elements (a special input form and buttons Search and Show All). They can be displayed (then the cookie file stores '1') or hidden ('0').
`DEFAULT_PAGE`	`0`	*Integer*. The ordinal number of the current list page set by default.
`DEFAULT_PAGE_SIZE`	`25`	*Integer*. The number of lines displayed on a single page.

Properties	Access	Description

`list_`	protected	*Array*. Is designed to hold the data displayed in the list. Presents an array that can be formatted as follows:
		
		`list_ = array(`
		
		`<id>=>array(<data_name>=><data_value>,…),`
		
		…
		
		`)`
		
		The meaning of collections associated with `<id>` is user defined. E.g. `<id>` can stand for the identifier of the list item (a row in the table), `<data_value>` can mean the column name, and `<data_value>` can be the value displayed in the table at the intersection of `<id>` and `<data_name>`.
		
		The `list_` array is created as an empty array by default.

`listHeader_`	protected	*String*. Specifies the object of search that will be displayed on the page as follows:
		
		`17 of 30 <objects> found`
		
		or
		
		`<objects> not found.`

`columns_`	protected	*Array*. Holds a collection of sub-arrays, each describing a column displayed in the list. The format of the overloaded array can be defined as desired. The current implementation looks as follows:
		
		`columns_ = array(`
		
		`<id> = array('name'=><value>, 'sort'=><value>, 'reverse'=><value>, …),`
		
		`…)`

`parentList_id_`	protected	*Integer*. The identifier of the owner of the listed items (e.g. *domain id* for the list of email names, etc.).
`flags_default_`	protected	*Integer*. The default value stored in the internal bitmask. Is set to DEFAULT_FLAGS by default.
`sort_default_`	protected	*String*. Specifies the default field by which sorting should by applied. Is set to DEFAULT_SORT by default.
`sortApplayed_`	protected	*Boolean*. If set to *true*, indicates that sorting has already been applied. Is used in the overloaded `fetch<derived_class_name>` method to prevent from 'native' sorting in case custom one has been done. Is *false* by default.
`filter_default_`	protected	*String*. Specifies the default filtering parameter (value). Is set to DEFAULT_FILTER by default.
`filterBy_`	protected	*String*. Specifies the field by which filtering performs. Is set to DEFAULT_FILTER_BY by default.
`filterShowCookie`	protected	*String*. Specifies the name of the cookie file that stores the state as whether the search elements (a special input field, buttons Search and Show All) are displayed or hidden on the page. The cookie stores '1' if the search elements are displayed, '0' otherwise. The property is set to DEFAULT_FILTER_SHOW_COOKIE by default.
`pagingSize_`	protected	*Integer*. Specifies the number of pages to be shown in the paging control to the left and to the right of the current page .E.g. If the list is 10 pages long, the currently active page is 2, and pagingSize = 2, then the paging control will look as follows: `First 1 2 3 4 Last` Is set to 2 by default.
`error_`	public	*Boolean*. If this flag is set to *true*, an error has occurred and can be reported. Is set to *false* by default.
`errorMsg_`	public	*String*. Holds an error message. Holds an empty string by default.

`removeTarget_`	protected	*String.* Specifies the URI of the page (normally reduced to the path in the context of the same site) displayed after the Remove selected button is pressed. Is set to the $PHP_SELF global variable by default (this variable stores the script being currently executed).

`_locales`	protected	*Array.* Is designed to hold the list of keys and associated GUI messages. In the `pm_cList` class, the default `_locales` array is defined as follows:
		
		`array (`
		
		`'search_result'=> 'class_cList__search_result',`
		
		`'not_found' => 'class_cList__not_found',`
		
		`'header' => 'class_cList__header',`
		
		`'empty'=> 'class_cList__empty'`
		
		`);`

Methods	Access	Description
pm_cList()	private	Constructor. Initializes member variables.
init()	public	Initializes the instance of the `pm_cList` class .
setFSP($cmd)	public	Retrieves the specified setting (filtering, sorting, paging) from the global array of parameters to the instance of class `pm_cList`.
isCMDSupported($cmd)	public	Checks whether the specified command is registered in the class.
setError_($msg)	public	Resets the error reporting object by setting the `error_` flag to *false* and `errorMsg_` to an empty string.
setFilter($filter)	public	Sets the specified filtering parameter to the instance of class `pm_cList.`
getFilter()	public	Returns the filtering parameter currently stored in the instance of class `pm_cList.`

clearFilter()	public	Resets the filtering parameter stored in the instance of class `pm_cList`.
getFlag($bit_no)	public	Checks whether the specified bit in the internal bitmask of `pm_cList` is set to '1'.
setFlag($bit_no, $value)	public	Sets the bit in the internal `pm_cList` bitmask to the specified value ('1' or '0').
getFlags()	public	Returns the whole bitmask.
setFlags($value)	public	Sets the whole bitmask to the specified value.
isCustomFilterSet()	protected, overridable	Checks whether the filtering parameter is set in the instance of class `pm_cList`.
isFilterSet()	protected, overridable	Checks whether the filtering parameter is set in the instance of the class the `isFilterSet` method belongs to.
setSort($sort)	public	Sets the specified sorting parameter to the instance of class `pm_cList`.
getSort()		Returns the sorting parameter set in the instance of class `pm_cList`.
setPageSize($page_size)	public	Recalculates the current page using the new page size.
setPage($page)	public	Sets the specified page as current.
getPage()	public	Returns the current number of the page that displays a portion of the whole list.
getPageSize()	public	Returns the current page size (in lines).
length()	public	Returns the current number of items in the list.
lengthTotal()	public	Returns the number of items on all pages of the non-filtered list.
lengthFilter()	public	Returns the number of items on all pages of the filtered list.
lengthActive()	public	Returns the actual number of items on all pages of the list.
lengthPage()	public	Returns the number of items on the current page of the filtered list.

listName()	public	Returns the name of the list (which normally matches the name of the class in the lower case).
listBegin()	public	Returns the key of the first item in the list.
listNext()	public	Returns the key of the next item in the list.
id($id)	public	Returns the key of the specified item in the list.
name($id)	public	Returns the value stored in the ‘name’ parameter of the specified list item.
get()	public	Forms HTML code of the page that will be displayed in the browser.
fetchRemoveFunction()	protected, overridable	Forms the javascript code of the function which will be called to handle the Remove Selected button pressure on the module’s page.
fetchSearchFunction()	protected, overridable	Forms the javascript code of several functions which will be called to handle the pressure of buttons Search, Show All, Hide Search, Show Search located on the module’s page.
fetchSelectFunction($ch_name, $misc_code)	protected, overridable	Forms the javascript code of the function that will be called to handle the selection of all checkboxes in the list.
fetchSelectOnChangeFunction()	protected, overridable	Forms the javascript function which will be called to handle change of state of the checkbox referring to a certain list item.
fetchSortFunction()	protected, overridable	Forms the javascript code of the function sorting the list.
fetchPagingFunction()	protected, overridable	Forms the javascript code of two functions that handle transfer to a different page of the list (a click on the paging control) and recalculate the page size.
getButtonsLayout()	protected, overidable	Forms HTML code that defines the layout and the collection of buttons that will be displayed on the module’s page above the list.
getSearch_()	protected, overridable	Forms HTML code of buttons Search and Show All displayed on the page right above the list element.
getRemoveSelectedButton($enabled)	protected, overridable	Forms HTML code that specifies the Remove Selected button.

notShowDel()	public	Disables the display of the Remove Selected button on the page containing the list.
getTable_()	protected, overridable	Forms HTML code of the table that presents a list on the browser page.
getTableHeader_()	protected, overridable	Forms HTML code of the table header.
getLastColumnTitle($title, $checkbox)	protected, overridable	Forms HTML code of the 'select all' checkbox element displayed in the table header.
getTableBody_()	protected, overridable	Forms HTML code of the table body.
postTr($id)	protected, overridable	Specifies HTML code forming a sub-row below the specified row in the table.
getDel_($id, $disabled, $checked, $readonly)	protected, overridable	Forms HTML code of a 'checkbox' GUI element anchored to the table row which is specified by ID.
getPaging_()	protected, overridable	Forms HTML code of 'paging' GUI elements.
noPageApplayed()	protected	Sets the instance of class `pm_cList` to state "paging applied".
noFilterApplayed()	protected	Sets the instance of class `pm_cList` to state "filtering applied".

Include: pm.php.

pm_cList:: pm_cList Method

Constructor. Initializes member variables.

Syntax

```
function pm_cList ()
```

Parameters

No

Returns

Nothing.

Remarks

The `removeTarget_` member variable is set to the value stored in the global `$PHP_SELF` variable (the script being executed).

pm_cList:: init Method

Initializes the instance of the `pm_cList` class .

Syntax

```
function init ()
```

Parameters

No

Returns

A *boolean* which is *true* if all initialization stages have been passed successfully, *false* otherwise.

pm_cList:: setFSP Method

Gets the specified setting (filtering, sorting, paging) from the global array of parameters.

Syntax

```
function setFSP ($cmd)
```

Parameters

cmd

A *string* value with a command that specifies the setting to be applied. Supported are the following commands:

- ‘setFilter’ - tells the function to apply filtering settings;
- ‘clearFilter’ - tells the function to clear filtering settings;
- ‘setSort’ - tells the function to apply sorting settings;
- 'setPage' - tells the function to set the current page;
- 'setPageSize' - tells the function to set the page size.

Returns

A *boolean* which is *true* if the specified setting has been applied successfully, *false* otherwise. If one passes in *false* as a command, the function returns *true.*

Remarks

This function looks for the specified setting in the global array of parameters passed to a script via GET, POST, or COOKIE HTML method:

- The ‘filter’ parameter is checked if the ‘setFilter’ command is specified;
- The ‘sort’ parameter is checked if the ‘setSort’ command is specified;
- The ‘page’ parameter is checked if the ‘setPage’ command is specified;

- The ‘page_size’ parameter is checked if the ‘setPageSize’ command is specified.

In case the passed in command is not recognized, Plesk throws an error message.

One may need to define a custom command if the derived class is extended with new properties that should be read from the global array of parameters. It is important that all new commands are registered in the isCMDSupported method. Also, every command triggers a special member function that reads the global parameter and puts the obtained value to the matching class member variable. The code of the overloaded method can look as follows:

```
static function setFSP($cmd)
{
      switch ($cmd){
      case false:
                  return true;
      case ‘<new_command>’:
                  return $this-
>set<new_param>(get_gpc(‘<new_param>’));
      …
      psaerror(‘Command not found:’.$cmd);
      }
}
```

pm_cList:: isCMDSupported Method

Checks whether the specified command is registered in the class.

Syntax

```
function isCMDSupported ($cmd)
```

Parameters

cmd

A *string* value that specifies the command to be checked. Supported are the following commands:

- ‘setFilter’ - means the setting of filtering settings;
- 'clearFilter' - means the removal of filtering settings;
- 'setSort' - means the settings of sorting settings;
- 'setPage' - means the setting of the current page;
- 'setPageSize'- means the setting of the page size.

Returns

A *boolean* which is *true* if the specified command is present in the list of known commands, *false* otherwise.

Remarks

To extend the list of known commands, one can overload this method for the derived class as follows:

```
function isCMDSupported($cmd)
{
      arr$ = array(false, ‘setFilter’, ‘clearFliter’, ‘setSort’,
‘setPage’, ‘setPagingSize’, <new_command>);
```

```
      return in_array($cmd, $arr);
}
```

pm_cList:: setError_ Method

Sets an error message to the instance of class `pm_cList`.

Syntax

```
public function setError_($msg)
```

Parameters

msg

A *string* that contains the text of the error message.

Returns

Nothing.

pm_cList:: setFilter Method

Sets the specified filtering parameter to the 'list' GUI element and flushes the new setting to the database.

Syntax

```
function setFilter ($filter)
```

Parameters

filter

A string that specifies the filtering parameter (value) to be set. Is NULL by default.

Returns

A boolean value that is true if the new filter has been set successfully, *false* otherwise.

Remarks

If the new filter is NULL (the *filter* parameter is not specified), then the default filtering parameter (`filter_default_` variable) is set for in 'list' GUI element.

pm_cList:: getFilter Method

Returns the filtering parameter currently set in the 'list' GUI element.

Syntax

```
function getFilter ()
```

Parameters

No

Returns

A *string* value with the filtering parameter currently stored in the instance of class `pm_cList`.

pm_cList:: clearFilter Method

Resets the filtering parameter (value) in the 'list' GUI element and writes the new setting to the database.

Syntax

```
function clearFilter ()
```

Parameters

No

Returns

A *boolean* value. Always returns *true* after the filtering parameter is reset.

pm_cList:: getFlag Method

Checks whether the specified bit in the internal bitmask of the instance of class `pm_cList` is set to ‘1’.

Syntax

```
function getFlag ($bit_no)
```

Parameters

bit_no

An *integer* that specifies the ordinal number of the bit to check in the bitmask.

Returns

A *boolean* value that is *true* if the specified bit in the bitmask is set to ‘1’, *false* otherwise.

pm_cList:: setFlag Method

Sets the bit in the internal bitmask of the instance of class `pm_cList` to the specified value (‘1’ or ‘0’).

Syntax

```
function setFlag ($bit_no, $value)
```

Parameters

bit_no

An *integer* that specifies the ordinal number of the bit to set in the bitmask. Only positive values are allowed.

value

An *integer* that should be set to the specified bit of the bitmask. Allowed values are ‘0’ and ‘1’.

Returns

A *boolean* value that is *true* if the *bit_no* parameter is not negative and the specified bit is set in the bitmask, *false* otherwise.

pm_cList:: getFlags Method

Returns the whole bitmask of the instance of class `pm_cList`.

Syntax

```
function getFlags ()
```

Parameters

No

Returns

An *integer* value that presents a bitmask of the instance of class `pm_cList`.

pm_cList:: setFlags Method

Sets the whole bitmask of the instance of class `pm_cList` to the specified value.

Syntax

```
function setFlag ($value)
```

Parameters

value

An *integer* that should be written to the bitmask. Is NULL by default.

Returns

A *boolean* value that is *true* if the *value* parameter is an integer and the specified value is set to the bitmask successfully, *false* otherwise.

Remarks

If the *value* parameter is NULL, the bitmask is set to the value currently stored in the `flags_default_` variable of the instance of class `pm_cList`.

pm_cList:: isCustomFilterSet Method

Checks whether the filtering parameter (value) of the calling instance of class `pm_cList` is set.

Syntax

```
function isCustomFilterSet ()
```

Parameters

No

Returns

A *boolean* value that is *true* if the filtering parameter of the instance of class `pm_cList` holds a non-empty string, *false* otherwise.

Remarks

If overloaded in the derived class with its own 'filter' variable, this function is meant to read this *current* 'filter' variable (referenced by `$this`). If one needs to read the *parent* filtering parameter, this can be done by calling to the pm_cList::isFilterSet() function that reads the 'filter' variable belonging to the same class the `isSetFilter` method belongs to.

pm_cList:: isFilterSet Method

Checks whether the filtering value is set in the instance of class `pm_cList` to which the `isFilterSet` method belongs to.

Syntax

```
function isFilterSet ()
```

Parameters

No

Returns

A *boolean* value that is *true* if the filtering value of the instance of class pm_cList is not an empty string, *false* otherwise.

Remarks

This member function calls the pm_cList::isCustomFilterSet() method.

If overloaded in the derived class that has its own 'filter' variable, this function is meant to check both the *parent* and *current* 'filter' variables, e.g. this can be done as follows:

```
return (self::isCustomSet() || parent::isFilterSet());
```

Thus, this function will return *true* if any of these 'filter' variables is set.

pm_cList:: setSort Method

Sets the specified sorting parameter to the instance of class `pm_cList.`

Syntax

```
function setSort ($sort)
```

Parameters

sort

A *string* that specifies the sorting parameter (the name of any column stored in the `columns_` array). The value passed in to the function via the *sort* parameter can also specify the *sorting order* – if it has the *'_reverse'* suffix appended to the name of the column, this means the reverse sorting order. E.g. '*name*' means direct sorting order, '*name_reverse'* means the reverse order (the column stored in the `columns_` array will have the *'name'* parameter, the suffix is never stored). Is NULL by default.

Returns

A *boolean* value. Always returns *true* after the sorting parameter is put to the instance of class `pm_cList` and the entire list of parameters is flushed to the database.

Remarks

If the *sort* parameter is not NULL, the function looks through the `columns_` array, picks out the columns for which sorting is allowed, and searches the specified sorting parameter among the parameters of such columns. Once this parameter is found, it is put to the instance of class `pm_cList` the `pm_cList::sortApplayed_` flag is disabled, after which the entire list of parameters of class `pm_cList` is flushed to the database.

In case the sorting parameter is not found in the `columns_` array, or the *sort* input parameter is NULL, the value currently stored in the `pm_cList::sort_default_` variable becomes a sorting parameter. Then the `pm_cList::sortApplayed_` flag is disabled and all parameters are written to the database.

pm_cList:: getSort Method

Returns the sorting parameter set in the instance of class `pm_cList`.

Syntax

```
function getSort ()
```

Parameters

No

Returns

A *string* value that holds the current sorting parameter stored in the instance of class `pm_cList.`

pm_cList:: setPageSize Method

Recalculates the current page using the new page size, sets the new page size to the instance of class `pm_cList`, and writes the updated setting to the database.

Syntax

```
function setPageSize ($page_size)
```

Parameters

page_size

An *integer* that specifies the new page size.

Returns

A *boolean* value. Is always set to *true* after the new page size is set and the updated settings are flushed to the database.

Remarks

If the new page size is a negative value, the page size is set to DEFAULT_PAGE_SIZE.

pm_cList:: setPage Method

Sets the specified page as current and writes the updated setting to the database.

Syntax

```
function setPage ($page)
```

Parameters

page

An *integer* that specifies the number of the page to display currently. Is set to DEFAULT_PAGE by default.

Returns

A *boolean* value. Is always set to *true* after the new page is set as current and the updated settings are flushed to the database.

Remarks

If the new page number is a negative value, page number 0 is set as the current one.

pm_cList:: getPage Method

Returns the current number of the page used to display a portion of the whole list.

Syntax

```
function getPage ()
```

Parameters

No

Returns

A *string* value that holds the number of the page that displays a portion of the whole list.

pm_cList:: getPageSize Method

Returns the page size (in lines) currently set in teh instance of class `pm_cList.`

Syntax

```
function getPageSize ()
```

Parameters

No

Returns

An *integer* value that holds the current number of lines on the page that displays the portion of the list.

pm_cList:: length Method

Returns the current number of items in the list.

Syntax

```
function length ()
```

Parameters

No

Returns

An *integer* value that holds the number of items currently present in the `pm_cList::list_` associative array.

pm_cList:: lengthTotal Method

Returns the number of items on all pages of the non-filtered list.

Syntax

```
function lengthTotal ()
```

Parameters

No

Returns

An *integer* value that holds the number of items on all pages of the non-filtered list.

pm_cList:: lengthFilter Method

Returns the number of items on all pages of the filtered list.

Syntax

```
function lengthFilter ()
```

Parameters

No

Returns

An *integer* value that holds the number of items on all pages of the filtered list.

pm_cList:: lengthActive Method

Returns the actual number of items on all pages of the list.

Syntax

```
function lengthActive ()
```

Parameters

No

Returns

An *integer* value that holds the actual number of items displayed on all pages of the list.

pm_cList:: lengthPage Method

Returns the number of items displayed on the current page of the filtered list.

Syntax

```
function lengthPage ()
```

Parameters

No

Returns

An *integer* value that holds the number of items displayed on the current page of the filtered list.

pm_cList:: listName Method

Returns the name assigned to the list programmatically.

Syntax

```
function listName ()
```

Parameters

No

Returns

A *string* value with the list name read from the instance of class `pm_cList.` Normally, the list name matches the name of the relevant class in the lower case.

pm_cList:: listBegin Method

Returns the key of the first item in the list.

Syntax

```
function listBegin ()
```

Parameters

No

Returns

A *mixed* value. If the `pm_cList::list_` array is empty, returns *false*. Otherwise returns the key of the first element in the array.

Remarks

The function positions the internal pointer of the `pm_cList::list_` associative array to its first element and returns its key.

pm_cList:: listNext Method

Returns the key of the next item in the list.

Syntax

```
function listNext ()
```

Parameters

No

Returns

A *mixed* value. If the `pm_cList::list_` array is empty or the element in focus is the last one in the array, returns *false*. Otherwise returns the key of the next element in the array.

pm_cList:: id Method

Returns the key of the specified item in the list.

Syntax

```
function id ($id)
```

Parameters

id

An *mixed* value that specifies the item in the `pm_cList::list_` array by its <id> value (`list_ = array(<id> = array(<name>=><value>, …), …)`. Is set to 0 by default.

Returns

A *mixed* value that holds the key of the specified item in the `pm_cList::list_` associative array.

Remarks

If the item is not specified (the *id* input parameter is set to 0), the function returns the key of the current item in the list.

If the specified key is not found in the array, the function returns 0.

pm_cList:: name Method

Returns the value stored in the ‘name’ parameter of the specified list item.

Syntax

```
function name ($id)
```

Parameters

id

A *mixed* value that specifies the item in the `pm_cList::list_` array by its <id> value `(list_ = array(<id> = array(<name>=><value>, …), …)`. Is set to 0 by default.

Returns

A *mixed* value that can hold either the key of the current item in the list (if the item is not specified, i.e. the *id* input parameter is set to 0), or the value stored in the ‘name’ parameter of the specified item in the `pm_cList::list_` associative array `(list_ = array(<id>=array(‘name’=><value>…), … ))`.

pm_cList:: get Method

Forms HTML code of the page to be displayed in the browser.

Syntax

```
function get ()
```

Parameters

No

Returns

A *string* value that holds HTML code of the page with all required GUI elements (buttons, remove elements, etc.) to be displayed in the browser. If the HTML code failed to be formed OK, an empty string is returned.

Remarks

If the `columns_` array is not defined or empty, the function cannot be executed and Plesk throws an error message.

If it is necessary to form extended HTML code covering the creation of *custom* GUI elements, the `get()` member function handles this situation by calling to a `javascript()` member function that is supposed to be in a class derived from `pm_cList`. After calling to the `javascript()` function, `get()` forms HTML code of a standard Plesk styled page containing a list GUI element and a set of buttons.

To generate HTML of GUI elements normally present on the page, `get()` invokes the following member functions:

- getButtonsLayout()
- fetchPagingFunction()
- fetchSelectFunction()
- fetchSelectOnChangeFunction()
- fetchSortFunction()
- fetchSearchFunction()
- fetchRemoveFunction()

pm_cList:: fetchRemoveFunction Method

Forms the javascript code of the function called to handle the Remove Selected button pressure on the module's page.

Syntax

```
function fetchRemoveFunction ()
```

Parameters

No

Returns

A *string* value that presents a code snippet in javascript. The snippet contains the text of the function that loads the `removeTarget_` page after the Remove Selected button is pressed.

Remarks

This function is called from within the pm_cList::get() member function that forms HTML of the module's page. The javascript code obtained after calling to `fetchRemoveFunction()` is inserted into the resulting HTML. If overloaded, `fetchRemoveFunction()` allows the user to implement a different or extended delete operation for selected list items.

pm_cList:: fetchSearchFunction Method

Forms the javascript code of several functions called to handle the pressure of buttons Search, Show All, Hide Search, Show Search located on the module's page.

Syntax

```
function fetchSearchFunction ()
```

Parameters

No

Returns

A *string* value that presents a portion of code in javascript containing several functions that handle the pressure of the following buttons: Search, Show All, Hide Search, Show Search.

Remarks

This function is called from within the pm_cList::get() member function that forms HTML of the module's page. The javascript code obtained after calling to `fetchSearchFunction()` is inserted into the resulting HTML. If overloaded, `fetchSearchFunction()` allows the user to implement different or extended handlers of pressing buttons Search, Show All, Hide Search, Show Search, or to add handlers for custom buttons related with search operations.

pm_cList:: fetchSelectFunction Method

Forms the javascript code of the function called to handle the selection of all checkboxes in the list.

Syntax

```
function fetchSelectFunction ($ch_name, $misc_code)
```

Parameters

ch_name

A *string* value that specifies the name of the checkbox element located in the table header. Is set to '`del[ ]`' by default.

misc_code

A *string* value that can hold user defined code that will be appended to the code of the generated function right after the selection code. Holds an empty string by default.

Returns

A *string* value that contains the code of the function called to handle the change of state of the checkbox located in the table header.

Remarks

This function is called from within the pm_cList::get() member function that forms HTML code of the module's page. The javascript code obtained after calling to `fetchSelectFunction()` is inserted into the resulting HTML. If overloaded, `fetchSelectFunction()` allows the user to implement a different or extended handler of selecting all items in the list.

pm_cList:: fetchSelectOnChangeFunction Method

Forms the javascript function to handle change of state of a checkbox referring to a certain list item.

Syntax

```
function fetchSelectOnChangeFunction ()
```

Parameters

No

Returns

A *string* value that holds the code of the handler function to be called after a certain checkbox has changed its state (the matching list item has been selected/released).

Remarks

This function is called from within the pm_cList::fetchSelectFunction() member function right after the code that changes the state of the specified checkbox and prior to the user defined code set by the *misc_code* parameter of `pm_cList::fetchSelectFunction`.

The `pm_cList::fetchSelectOnChangeFunction` method is the base one – the function specified within this method contains the only line of code that just returns *true*. To overload this member function without changing the calling one (`fetchSelectFunction()`), do it as follows:

```
// the body of the overloaded fetchSelectOnChangeFunction()
return '
function ' . get_class($this) . 'SelectOnChange(o)
{
      // define custom operations here
      return true;
}
```

pm_cList:: fetchSortFunction Method

Forms the javascript code of the function sorting the list by the column.

Syntax

```
function fetchSortFunction ()
```

Parameters

No

Returns

A *string* value containing the code of the function that determines the list column by which to sort and applies sorting to it.

Remarks

This function is called from within the pm_cList::get() member function that forms HTML code of the module's page. The javascript code obtained after calling to `fetchSortFunction()` is inserted into the resulting HTML.

pm_cList:: fetchPagingFunction Method

Forms the javascript code of two functions that handle transfer to a different page of the list (a click on the paging control) and recalculate the page size.

Syntax

```
function fetchPagingFunction ()
```

Parameters

No

Returns

A *string* value that contains two functions, one used to handle transfer to a different page of the list, and another meant to recalculate the page size (the number of lines to display on the page at a time).

pm_cList:: getButtonsLayout Method

Forms HTML code that defines the layout and set of buttons that will be displayed on the page of the module above the list.

Syntax

```
function getButtonsLayout ()
```

Parameters

No

Returns

A *string* value that contains HTML code that defines the layout and set of buttons that will be displayed on the module's page above the list.

pm_cList:: getSearch_ Method

Forms the HTML code of buttons Search and Show All displayed on the page right above the list element.

Syntax

```
function getSearch_ ()
```

Parameters

No

Returns

A *string* value that contains the HTML code snippet defining buttons Search and Show All.

pm_cList:: getRemoveSelectedButton Method

Forms HTML code that specifies the Remove Selected button.

Syntax

```
function getRemoveSelectedButton ($enabled)
```

Parameters

enabled

A *boolean* value that should be set to *true* to allow the display of the Remove Selected button on the page, otherwise it should be *false*. Is *true* by default.

Returns

A *string* value that holds HTML code of the Remove Selected button.

pm_cList:: notShowDel Method

Disables the display of the Remove Selected button on the page containing the list.

Syntax

```
function notShowDel ()
```

Parameters

No

Returns

Nothing.

pm_cList:: getTable_ Method

Forms HTML code of the table that presents a list on the browser page.

Syntax

```
function getTable_ ()
```

Parameters

No

Returns

A *string* value that holds HTML code of the table that presents the list GUI element.

Remarks

The function defines HTML tags `<table>` and `</table>` and the code lying within. This code includes table parameters, the code of the table header obtained by calling to the getTableHeader_() member function, and the code of the table body generated by the getTableBody_() member function.

pm_cList:: getTableHeader_ Method

Forms HTML code of the table header.

Syntax

```
function getTableHeader_ ()
```

Parameters

No

Returns

A *string* value that holds HTML code of the table header.

Remarks

To form the table header, the function defines HTML tags `<tr>` and `</tr>` and the code lying within. This code can include adding the 'select all' checkbox element, after which every column is assigned a set of parameters (size, column-generating function name, sorting, sorting icons, context help).

This member function is invoked in the pm_cList::getTable_() method.

pm_cList:: getLastColumnTitle Method

Forms HTML code of the 'select all' checkbox element displayed in the table header.

Syntax

```
function getLastColumnTitle ($title, $checkbox)
```

Parameters

title

A *string* value that can specify the HTML attribute describing the column title. Holds an empty string by default.

checkbox

A *boolean* value. If set to *true*, indicates that HTML code of the 'select all' checkbox element should be generated. If set to *false*, a cell with a zero width is generated. Is set to *true* by default.

Returns

A *string* value that holds HTML code of the 'select all' checkbox element.

Remarks

The function is invoked from within the pm_cList::getTableHeader_() method.

pm_cList:: getTableBody_ Method

Forms HTML code of the table body.

Syntax

```
function getTableBody_ ()
```

Parameters

No

Returns

A *string* value that holds HTML code of the table body.

Remarks

The function generates HTML code of the table body, specifying parameters (a checkbox element inserted, sorting) for each table row. After a row is defined, the function invokes the postTr() member function to define a sub-row or several (e.g. to display a list of subdomains below the domain).

The function is invoked from within pm_cList::getTable_().

pm_cList:: postTr Method

Specifies HTML code forming a sub-row below the specified row in the table.

Syntax

```
function postTr ($id)
```

Parameters

id

An *integer* value. Specifies the table row that requires a sub-row to be displayed lower. Holds an empty string by default.

Returns

A *string* value that holds the HTML code of a sub-row. Returns an empty string in the current implementation.

Remarks

This function is called from within the pm_cList::getTableBody_() member function after the HTML code of a next table row is generated.

This function is reserved for overriding in derived classes. Currently, it returns an empty string. Potentially, it can generate HTML code of a sub-row that will be inserted into HTML of the table after every `<tr>...</tr>` block.

pm_cList:: getDel_ Method

Forms HTML code of the 'checkbox' GUI element anchored to the specified table row.

Syntax

```
function getDel_ ($id, $disabled, $checked, $readonly)
```

Parameters

id

An *integer* value that specifies the table row.

disabled

A *boolean* value. If set to *true*, specifies the disabled checkbox. Is set to *false* by default.

checked

A *boolean* value that specifies the state of the checkbox. If set to *true*, means the checked checkbox. Is *false* by default.

readonly

A *boolean* value. If set to *true*, the state of the checkbox cannot be changed. Is set to *false* by default.

Returns

A *string* value that holds HTML code of the 'checkbox' GUI element displayed in a table row specified by its identifier.

pm_cList:: getPaging_ Method

Forms HTML code of GUI elements referring to paging.

Syntax

```
function getPaging_ ()
```

Parameters

No

Returns

A *string* value that holds HTML code of 'paging' GUI elements.

Remarks

The function defines the output of the list info (how many items are found, etc.), of line “Number of entries per page: 10 25 100” , and of the paging control (First 2 3 4 5 Next) if paging is allowed.

pm_cList:: noPageApplayed Method

Sets the instance of class `pm_cList` to state “paging applied”.

Syntax

```
function noPageApplayed ()
```

Parameters

No

Returns

Nothing.

pm_cList:: noFilterApplayed Method

Sets the instance of class `pm_cList` to state “filtering applied”.

Syntax

```
function noFilterApplayed ()
```

Parameters

No

Returns

Nothing.

class pm_Form

The pm_Form class is designed as a creator of the HTML form. This is an abstract 'parent' class from which user defined classes can inherit.

The creation of the HTML form implemented by this class is based on the FastTemplate technology - the pm_Form class supports the rFastTemplate PHP extension (implemented `in class.rFastTemplate.`php) that serves for creating HTML pages using templates, including *dynamic* ones. The pm_Form class aggregates an instance of the rFastTemplate class, and some functions of pm_Form serve to pass in the data to this instance.

In brief, the FastTemplate technology operates the following elements. An instance of the rFastTemplate class has three main arrays:

- TEMPLATE – stores *arrays*, one for each template file containing variables (like {TITLE}, {NAME}, etc.). Each array has a *name* by which the template file will be referenced, and *a set of parameters*, among which the base one is the *name of the template file*.
- VAR – stores *pairs (variable, value)*. Each variable present in the template file(s) should be resolved in this array.
- HANDLE – stores a *handle* or several, each referencing the text of a certain *template file* with its variables substituted by text.

The main steps of the FastTemplate technology are:

- define – the TEMPLATE array is filled with filenames mapped to names by which templates will be referred.
- assign - the VAR array is filled with pairs (variable, value).
- parse – the specified template file is parsed, its variables are interpolated to text taken from the VAR array, and the HANDLE array is extended with a handle internally referencing the resulting text.
- FastPrint: prints out the contents of the specified handle from the HANDLE array.

The rFastTemplate technology ‘recognizes’ *dynamic* templates which are not separate files but special *dynamic blocks* nested within ‘parent’ template files. These dynamic template blocks are added to the TEMPLATE array by define_dynamic operation and are stored in it with a special ‘dynamic’ mark.

The details on the implementation and use of the FastTemplate technology can be found at www.thewebmasters.net or in other Internet resources. Also, section *Creating Modules* of this documentation gives an example on how to create forms using the FastTemplate technology (see topic Step 3. Designing GUI of the module in the *Programming Guide*).

Constant	Value	Description
`TEMPLATES_DIR`	`PRODUCT_ROOT. ‘/admin/plib/templates/’`	*String*. Specifies the directory from which template files are searched and loaded by rFastTemplate.
`TEXT_SIZE`	`25`	*Integer*. Specifies the default size of the input field.
`REQ`	`get_asterisk()`	*String*. The string returned by the `get_asterisk()` function specifies the ‘*’ character as the ‘*required*’ indication for the field.
`ERROR_FIELD`	`‘class=”error” ’`	*String*. Specifies the class that should be used to draw an error message field in the form.

Properties	Access	Description
`action`	protected	*Boolean*. Specifies what will be set to the ACTION variable of the form template. Is *false* by default.
`_show_warnings`	protected	*Boolean*. If set to *true*, allows the display of warnings and error messages in a special area of the form. Is *true* by default.
`_show_events`	protected	*Boolean*. If set to *true*, indicates that the current form can handle a javascript event, *false* otherwise. Is *true* by default.

Methods	Access	Description
pm_Form()	public	Constructor. Initializes member variables.

addTemplates ($templates)	protected	Adds an array of templates (pairs like {name, filename}) to the instance of class `pm_Form`.
define()	public, overridable	Pushes all templates stored in the instance of class `pm_Form` to the TEMPLATE array of the internal rFastTemplate instance.
define_dynamic($Macro, $Parent)	protected	Adds the name of the dynamic template block (nested within the parent template file) to the TEMPLATE array of the internal rFastTemplate instance.
addVars($vars)	protected	Adds an array of variables (pairs like {variable, value}) to the instance of class `pm_Form`.
addHideableControls($control)	protected	Generates HTML of all GUI controls of the form and adds this HTML code to the instance of class `pm_Form.`
addHideableControl($control, $var, $category, $available)	protected	Forms HTML of the area containing the specified control and adds this HTML to the instance of class `pm_Form.`
getRefreshButton($conhelp, $href)	protected, overridable	Forms HTML code of the Refresh link button.
getPreviousButton()	protected, overridable	Forms HTML of the link button referencing the previous page of the wizard.
getUpdateButton($conhelp, $onclick)	protected, overridable	Forms HTML code of the OK or Finish button.
getCancelButton($conhelp, $href)	protected, overridable	Forms HTML code of the Cancel button.
assign()	public, overridable	Fills the VAR array of the internal rFastTemplate instance with data.
parse($name, $tmpl)	public, overridable	Triggers the parse operation in the internal rFastTemplate instance.
clear_dynamic($tmpl)	protected	Removes the specified dynamic block definition from the parent template.
clear($handle)	protected	Clears all or specified internal references (handles) stored in the HANDLE array of the rFastTemplate instance.

fetch()	public	Returns the text referenced by the 'DOCUMENT' handle of the HANDLE array of the rFastTemplate instance.
FastPrint()	public	Prints out the text referenced by the 'DOCUMENT' handle of the HANDLE array of the internal rFastTemplate instance.

Include: pm.php.

pm_Form:: pm_Form Method

Constructor. Initializes member variables.

Syntax

```
public function pm_Form ()
```

Parameters

No

Returns

Nothing.

Remarks

The function sets the action member variable to the value of the $PHP_SELF global variable (the current script), instantiates the internal object of type rFastTemplate and sets its template directory to TEMPLATES_DIR.

pm_Form:: addTemplates Method

Adds an array of templates (pairs like {name, filename}) to the instance of class `pm_Form.`

Syntax

```
protected function addTemplates ($templates)
```

Parameters

templates

An *array* of templates that should be added to the instance of class `pm_Form` and then pushed to the internal rFastTemplate instance using the define() method.

Returns

A *boolean* which is *true* if the passed in array of templates has been added successfully, *false* otherwise.

Remarks

The function fails if the passed in parameter is not an array. In this case the function throws a special error and returns *false*.

The array of templates passed in via the *templates* parameter should be formatted as `array(<template_name> => <template_file_name>, …)`, e.g. `array('tickets' => 'tickets.tpl', 'customers' => 'all_customers.tpl')`. The template file should be located in the directory declared in the TEMPLATES_DIR constant.

pm_Form:: define Method

Pushes all templates stored in the instance of class `pm_Form` to the TEMPLATE array of the internal rFastTemplate instance.

Syntax

```
public function define ()
```

Parameters

No

Returns

A *boolean* which is *true* if the `pm_Form` instance stores at least one template.

Remarks

This function needs to be overridden.

pm_Form:: define_dynamic Method

Adds the name of the dynamic template block nested within the parent template file to the TEMPLATE array of the internal rFastTemplate instance.

Syntax

```
protected function define_dynamic ($Macro, $Parent)
```

Parameters

Macro

A *string* that specifies the name of the dynamic template block nested within the template file.

Parent

A string that specifies the 'parent' – the *template name* stored in the TEMPLATE array of the internal rFastTemplate instance and referencing the name of the *template file* within which the dynamic block is nested. Holds an empty string by default.

Returns

A *boolean* which is always *true*.

pm_Form:: addVars Method

Adds an array of variables (pairs like {variable, value}) to the instance of class `pm_Form.`

Syntax

```
protected function addVars ($vars)
```

Parameters

vars

An *array* of variables that should be added to the instance of class `pm_Form.`

Returns

A *boolean* which is *true* if the passed in array of variables has been merged with `pm_Form::vars` successfully, *false* otherwise.

Remarks

The function fails if the passed in parameter is not an array. In this case the function throws a special error and returns *false.*

The array of variables passed in via the *vars* parameter should be formatted as `array(<variable> => <value>, …)`, e.g. `array('TITLE' => 'Administering FTP', 'USER' => 'Admin')`. The array of variables stored in the instance of class `pm_Form` should resolve all variables present in the template files added to array of templates of this `pm_Form` instance.

pm_Form:: addHideableControls Method

Generates HTML of all GUI controls of the form and adds this HTML code to the instance of class `pm_Form.`

Syntax

```
protected function addHideableControls ($control)
```

Parameters

control

A *mixed* value that can hold an *array* of control names or a *string* with a single control name.

Returns

A *boolean* which is *true* if all resulting arrays of variables have been added successfully, *false* otherwise.

Remarks

The function invokes the pm_Form::addHideableControl() member function for each item of the passed in collection of controls (or for a single one).

pm_Form:: addHideableControl Method

Forms HTML of the area containing the specified control and adds this HTML code to the instance of class `pm_Form.`

Syntax

```
protected function addHideableControl ($control, $var, $category, $available)
```

Parameters

control

A *string* that specifies the name of the control.

var

A *mixed* value that is expected to contain the name of the control (*string*) in the upper case. Is *false* by default. If not defined (equal to *false*), this parameter will be assigned the name of the control automatically.

category

A *string* that specifies the type of the control. Is set to '*formControl*' by default.

available

A *boolean* value. If set to *true*, indicates that the specified control is enabled. Is *true* by default.

Returns

A *boolean* which is *true* if the generated HTML code has been added to the instance of class `pm_Form` successfully, *false* otherwise.

Remarks

The function reads the visibility status of the specified control from the global scope. Depending on the results, the function adds two pairs (variable, text) to the instance of class `pm_Form`. The text contains HTML code of the visible/invisible area in which the control will be placed:

- pairs `'BEGIN_VIS_$var' => '<tboby style = "display::none">'` and `'END_VIS_$var' => '</tbody>'` will be added in case the visibility status of the specified control is 'hide';
- `'BEGIN_VIS_$var'` and `'END_VIS_$var'` associated with HTML of the area (including the specified control) will be added in case the visibility status of the specified control is 'show'.

pm_Form:: getRefreshButton Method

Forms HTML code of the Refresh link button.

Syntax

```
protected function getRefreshButton ($conhelp, $href)
```

Parameters

conhelp

A *string* value that holds the context help that will be displayed when one points at the Refresh button with a mouse. Holds an empty string by default.

href

A *string* value that holds the destination URI that will be loaded once the Refresh button is clicked. Holds an empty string by default.

Returns

A *string* value that holds HTML of the Refresh link button.

Remarks

This function can be overridden.

pm_Form:: getPreviousButton Method

Forms HTML of the link button referencing the previous page of the wizard.

Syntax

```
protected function getPreviousButton ()
```

Parameters

No

Returns

A *string* that holds HTML of the link button referencing the previous page of the wizard or the current page in case it is the last one (or the only one).

Remarks

This function can be overridden.

pm_Form:: getUpdateButton Method

Forms HTML code of the OK or Finish button.

Syntax

```
protected function getUpdateButton ($conhelp, $onclick)
```

Parameters

conhelp

A *string* value that holds the context help to be displayed when one points at the OK/Finish button with a mouse. Holds an empty string by default.

onclick

A *string* value that holds the javascript code to execute when one clicks on the OK/Finish button. Is set to `'return update_oC(document.forms[0])'` by default.

Returns

A *string* that holds HTML code of the OK/Finish button.

Remarks

The button name is Finish if it is the wizard page, else the button name is OK.

This function can be overridden.

pm_Form:: getCancelButton Method

Forms HTML code of the Cancel button.

Syntax

```
protected function getUpdateButton ($conhelp, $href)
```

Parameters

conhelp

A *string* value that holds the context help to be displayed when one points at the Cancel button with a mouse. Holds an empty string by default.

href

A *mixed* value that holds the destination URI that will be loaded once the Cancel button is clicked. Is NULL default.

Returns

A *string* that holds HTML code of the Cancel link button.

Remarks

This function can be overridden.

pm_Form:: assign Method

Fills the VAR array of the internal rFastTemplate instance with data.

Syntax

```
public function assign ()
```

Parameters

No

Returns

A *boolean* which is *true* if the array of variables stored in the instance of class `pm_Form` is not empty and has been passed to the internal rFastTemplate instance successfully, *false* otherwise.

Remarks

This function needs to be overridden.

pm_Form:: parse Method

Triggers the parse operation in the internal rFastTemplate instance.

Syntax

```
public function parse ($name, $tmpl)
```

Parameters

name

A *string* value that holds the name of the *handle* (an item of the HANDLE array of the internal rFastTemplate instance). The handle will reference the text of the specified template file with all its variables substituted by values during parsing.

tmpl

A *mixed* value that can hold either a *string*, or an *array* of strings, each holding a template name to search in the TEMPLATE array of the the internal rFastTemplate instance.

Returns

A *boolean* which is always *true.*

Remarks

The function invokes the `parse()` member function of the the internal rFastTemplate instance, passing in two parameters: the name of the destination *handle* (the *name* input parameter) and the name of the template file (or a collection) to parse (the *tmpl* input parameter).

Once this function returns, the HANDLE array of the the internal rFastTemplate instance gets a single handle in case the *tmpl* input parameter is a string, or as many handles as the number of items in the array of templates in case the *tmpl* input parameter is an array. Each handle references the text of one template file.

This function can be overridden.

pm_Form:: clear_dynamic Method

Removes the specified dynamic block definition from the parent template.

Syntax

```
protected function clear_dynamic ($tmpl)
```

Parameters

tmpl

A *mixed value* that can be a *string* holding the name of the dynamic template block, or an *array* of strings of this kind.

Returns

Nothing.

Remarks

The function strips all of the *unparsed* dynamic blocks named as specified in the *tmpl* input parameter from the parent template. On the level of code, this means that all such blocks stored in the TEMPLATE array of the rFastTemplate instance will have their ‘result’ parameters set to empty strings.

This member function will take effect only if the parse operation has not been applied to the parent template yet, as all BEGIN and END lines of dynamic blocks are removed from the template during parsing, after which the dynamic block cannot be recognized.

pm_Form:: clear Method

Clears all or specified internal references (handles) stored in the HANDLE array of the rFastTemplate instance.

Syntax

```
protected function clear ($handle)
```

Parameters

handle

A *mixed* value that can hold a *string* with the *handle* (an item of the HANDLE array of the internal rFastTemplate instance) or an *array* of strings of this kind. A handle references the text of a template file with all its variables substituted by matching values.

Returns

Nothing.

Remarks

If the *handle* input parameter is set to NULL, the whole HANDLE array of the internal rFastTemplate instance is released.

pm_Form:: fetch Method

Returns the text referenced by the 'DOCUMENT' handle of the HANDLE array of the rFastTemplate instance.

Syntax

```
public function fetch ()
```

Parameters

No

Returns

A *string* value that holds the text referenced by the 'DOCUMENT' handle of the HANDLE array of the rFastTemplate instance.

pm_Form:: FastPrint Method

Prints out the text referenced by the 'DOCUMENT' handle of the HANDLE array of the internal rFastTemplate instance.

Syntax

```
public function FastPrint ()
```

Parameters

No

Returns

A *boolean* which is always *true.*

Remarks

Before displaying the text of the page, the function clears all event and warning messages if their display is not suppressed in the `_show_warnings` and `_show_events` flags.

class pm_Pathbar

The pm_Pathbar class presents a GUI element of the same name. A pathbar is a GUI element displayed on top of Plesk pages and looking like a path string. For instance, the pathbar for any service on `example.com` can look as follows:

Domains > example.com >

The pathbar GUI element is designed to display the navigation history of the current page. The current page itself is not displayed in the pathbar. The names of previous pages serve as bookmarks, i.e. they can behave as links if one wants to proceed to the relevant page.

To describe the pathbar GUI element, the `pm_Pathbar` class has a special multi-dimensional array. Its first element stores the *session context* – a *string* that specifies the section of the navigation pane of Plesk Control panel from which the navigation is started (e.g. ‘clients’, ‘domains’, ‘server’, etc.). The first element of the array (the context) is always present.

Elements of the 'pathbar' array indexed 1 and higher specify previous pages (the current page is not stored in the 'pathbar' array). Each element of the 'pathbar' array is an *array* of parameters making up the description of the relevant page. The page names are displayed in the order they arrive in the array. The 'pathbar' array gets a new item to its tail at drill-down to some new page and gets truncated at drill-up to the previous page.

A page is specified in the 'pathbar' array as follows:

```
$pathbar = array(
      [0] => ‘<session_context>’,
      [1] => array(
             ‘url’ => ‘<page_url>’,
             // the key associated with the page caption to be
displayed:
             ‘name’ => ‘<localization_key>’,
             // 'action' parameter of the form:
             ‘page’ => ‘<PHP_script_file_name>’,
             ‘previous_page’ => ‘<prev_page_url>’,
             ‘return_conhelp’ => ‘<conhelp_localization_key>’
             ),
      [2] => array(…),
      [3] => array(…),
      …
);
```

Thus, a page is described by five parameters: its URL, URL of the previous page, its title, its 'action' parameter, and its context help. A page is identified by two parameters - by its URL and by a PHP script file set in its ACTION parameter. The remaining parameters serve as an additional description.

Methods	Access	Description
pm_Pathbar()	public	Constructor. Initializes member variables.
Destructor()	public	Saves the current session parameters to the global scope.
set($url, $name, $in_context, $return_conhelp, $page, $top_page)	public	Rewrites the 'pathbar' array at drill-down or drill-up in the hierarchy of pages.
del($url, $page)	public	Truncates the 'pathbar' array up to the specified page at drill-up.
reset($context)	public	Clears the 'pathbar' array and sets the specified context.
getLatestURL()	public	Returns the identifier of URL of the page displayed previously.

setPage($page)	public	Sets the name of the PHP script file for the page being displayed.
getStruct($page)	public	Returns the 'pathbar' array truncated right before the specified page.
getHistory($page)	public	Forms HTML code of the pathbar GUI element.
getButton($page)	public	Forms HTML code of the Up Level button.
getSelfPath($page)	public	Returns a 'pathbar' sub-array describing the specified page.
getSelfURL($page)	public	Returns URL of the specified page.
getUpLevelPath($page)	public	Returns a 'pathbar' sub-array describing the page located one level higher in the hierarchy than the specified page.
getUpLevelURL($page)	public	Returns URL of the page located one level higher in the hierarchy than the specified page.
getUpLevelPage($page)	public	Returns the name of the PHP script file associated with the page located one level higher in the hierarchy than the specified page.

Include: pm.php.

pm_Pathbar::pm_Pathbar Method

Constructor. Initializes member variables.

Syntax

```
function pm_Pathbar ()
```

Parameters

No

Returns

Nothing.

Remarks

The function reads the current session parameters from the global scope.

pm_Pathbar::Destructor Method

Saves the current session parameters to the global scope.

Syntax

```
function Destructor()
```

Parameters

No

Returns

A *boolean* value that is always *true* after the session parameters are written to the global scope.

Remarks

The function should be called before destroying the instance of the `pm_Pathbar` class.

pm_Pathbar::set Method

Rewrites parameters of the pathbar GUI element at drill-down or drill-up in the hierarchy of pages.

Syntax

```
function set($url, $name, $in_context, $return_conhelp, $page,
$top_page)
```

Parameters

url

A *string* value that holds URL of the page being displayed. Is set to NULL by default.

name

A *string* value that holds the localization key associated with the page title to be displayed. The format of the key is ‘pb_<php_file_name_no_extension>’. Is set to NULL by default.

in_context

A *string* value that holds the passed in context of the page. Is set to NULL by default (will be read from the global scope).

return_conhelp

A *string* value that holds the context help localization key for the current page. The format of the key is ‘to_<php_file_name_no_extension>’. Is set to NULL by default.

page

A *string* value that holds the name of the PHP script file (without extension) set in the *action* parameter of the page being displayed. Is set to NULL by default.

top_page

A *string* value that holds the top-level page to be displayed in the pathbar GUI element. Should be specified at drill-down. Is set to NULL by default.

Returns

A *boolean* value which is always *true* after the pathbar parameters are set anew.

Remarks

The *in_context* parameter can be used if it is necessary to pass the required *session context* (a string that specifies the section of the navigation pane of Plesk Control panel from which the navigation is started, e.g. 'clients', 'domains', etc.). If left equal to NULL, the *in_context* parameter indicates than the session context should be read from the global scope.

The the collection of bookmarks stored in the 'pathbar' array is either added with a new item or truncated, depending on the direction one moves through the hierarchy of pages.

pm_Pathbar::del Method

Truncates the 'pathbar' array up to the specified page at drill-up.

Syntax

```
function del($url, $page)
```

Parameters

url

A *string* value that specifies URL of the page that will be the top-level one in the pathbar. Is NULL by default.

page

A *string* value that holds the PHP script (the file name) of the page to be the top-level one in the pathbar. Is NULL by default.

Returns

A *boolean* value which is always *true* after the 'pathbar' array is rewritten.

Remarks

The function rewrites its 'pathbar' array so that it becomes truncated up to the element specified by the *url* and *page* input parameters. If these parameters are set to NULL, the function uses parameters set for the currently active page.

pm_Pathbar::reset Method

Removes all elements from the 'pathbar' array and sets the specified context.

Syntax

```
function set ($context)
```

Parameters

context

A *string* value that holds the new *session context* to be put to 'pathbar' array after it is cleaned out. Is *false* by default.

Returns

Nothing.

Remarks

The session context is a *string* that specifies the section of the navigation pane of Plesk Control Panel from which the navigation is started (e.g. ‘clients’, ‘domains’, ‘server’, etc.).

If the *context* input parameter is *false*, the old context is set anew.

pm_Pathbar::getLatestURL Method

Returns the identifier of the URL of the page displayed previously.

Syntax

```
function getLatestURL()
```

Parameters

No

Returns

A *mixed* value that can hold *false* if 'pathbar' array is empty, or a *string* that identifies URL of the previous page.

Remarks

The page displayed previously is the last item of the 'pathbar' array.

The returned string is formatted as '`?previous_page=<encoded_url>`' or '`&previous_page=<encoded_url>`'. Here `<encoded_url>` stands for URL of the previous page encoded the same way as data posted from a WWW form: all non-alphanumeric characters except hyphens (-), underscores (_), and dots (.) are replaced with a percent (%) sign followed by two hex digits, spaces are encoded as plus (+) signs.

pm_Pathbar::setPage Method

Sets the name of the PHP script file for the current page.

Syntax

```
function setPage ($page)
```

Parameters

page

A *string* value that holds the name of the PHP script file specifying the page.

Returns

Nothing.

pm_Pathbar::getStruct Method

Returns the copy of the 'pathbar' array truncated right before the specified page.

Syntax

```
function getStruct ($page)
```

Parameters

page

A *string* value that holds the name of the PHP script file associated with the specified page. Is set to NULL by default.

Returns

An *array* that holds the copy of the 'pathbar' array truncated right before the specified page (the specified page is included).

Remarks

If the *page* input parameter is NULL, the returned array contains the structure of the pathbar GUI element ('pathbar' array) ending with the specified (or current) page.

pm_Pathbar::getHistory Method

Forms HTML code of the pathbar GUI element.

Syntax

```
function getHistory ($page)
```

Parameters

page

A *string* value that specifies the name of the PHP script file associated with the page to be displayed. Is NULL by default.

Returns

A *string* value that holds HTML code of the pathbar GUI element.

Remarks

The *page* input parameter specifies the page that will be displayed in the browser, so it will not be a part of the pathbar. If set to NULL, the *page* parameter specifies the currently active page.

pm_Pathbar::getButton Method

Forms HTML code of the Up Level button.

Syntax

```
function getButton ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (PHP script file) of the page containing the Up Level button. Is NULL by default.

Returns

A *string* value that holds HTML code of the Up Level button.

Remarks

If the *page* input parameter is set to NULL, the *action* parameter of the current page is used.

pm_Pathbar::getSelfPath Method

Returns a 'pathbar' sub-array describing the specified page.

Syntax

```
function getSelfPath ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (the PHP script file) of the specified page. Is set to NULL by default.

Returns

A *mixed* value that can hold the 'pathbar' *sub-array* being searched, or *false* if the specified page is not found in the parent array or refers to the ‘root’ level of hierarchy (displayed in the navigation pane of Plesk Control Panel).

pm_Pathbar::getSelfURL Method

Returns URL of the specified page.

Syntax

```
function getSelfURL ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (the PHP script file) of the specified page. Is set to NULL by default.

Returns

A *string* value that holds URL of the specified page.

Remarks

If the specified page refers to the ‘root’ level of hierarchy (displayed in the navigation pane of Plesk Control Panel), the function returns URL from the global scope, or the ‘/’ string if failed to find such.

pm_Pathbar::getUpLevelPath Method

Returns a 'pathbar' sub-array describing the page located one level higher in the hierarchy than the specified page.

Syntax

```
function getUpLevelPath ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (the PHP script file) of the specified page. Is set to NULL by default.

Returns

A *mixed* value that can hold the 'pathbar' *sub-array* of the upper-level page in the hierarchy (as compared to the specified one), or *false* if the searched page refers to the ‘root’ level of hierarchy (displayed in the navigation pane of Plesk Control Panel).

pm_Pathbar::getUpLevelURL Method

Returns URL of the page located one level higher in the hierarchy than the specified page.

Syntax

```
function getUpLevelURL ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (the PHP script file) of the specified page. Is set to NULL by default.

Returns

A *string* value that holds URL of the page located one level higher in the hierarchy than the specified page.

Remarks

If the target page refers to the ‘root’ level of hierarchy (displayed in the navigation pane of Plesk Control Panel), the function returns URL from the global scope, or the ‘/’ string if failed to find such.

pm_Pathbar::getUpLevelPage Method

Returns the name of the PHP script file associated with the page located one level higher in the hierarchy than the specified page.

Syntax

```
function getUpLevelPage ($page)
```

Parameters

page

A *string* value that holds the *action* parameter (the PHP script file) of the specified page. Is set to NULL by default.

Returns

A *string* value that holds the PHP script file name associated with the page located one level higher than the specified one.

Remarks

If the target page refers to the ‘root’ level of hierarchy (displayed in the navigation pane of Plesk Control Panel), the function returns an empty string.