WebAccess Plugin Paper

The following document describes a new functionality that is currently under development. Not all information is final and can be subject to change.

Table of Contents

1 Goal

The plugin architecture allows the development of custom features, integration of third party applications and community cooperation in the development of new functionalities.

The server administrator is able to add plugins to the webaccess by simply adding the placing the plugin files in the plugins directory of the webaccess. The webaccess automatically detects the installed plugins that are placed in that directory.

2 Plugin anatomy

A plugin can interact with the webaccess in a couple of different ways.

- **Adding new components**
  A plugin can create their own modules, views, widgets and sets of CSS rules and add them to the webaccess through a simple plugin. These new components have the same control and freedom as any other native component in the webaccess. This allows developer to create their own full integration with third party applications and create new interfaces to combine information stored in their account.
- **Extending existing components**
  A plugin can extend existing components (native or implemented by other plugins) and add or change their functionalities. You can, for example, customize the operations of a module to rewrite the request to the server when retrieving the calendar items or you can change the way the drag and drop functions.
- **Hooks**
  A plugin can hook in at certain points in the code to stop the native webaccess. The plugin then has the opportunity to modify objects and data before the native code finishes. The plugin developer can now scan the body text just before it is displayed and replace all the telephone numbers in the body of an email to automatically link to their skype when the user clicks on them. He can also detect addresses in the text and add an overlay contain a Google Maps image displaying the location (See Image 1). Adding their own items to menus and adding extra tabs in dialogs are also in the realm of possibilities.
- **Library functionality**
  A plugin can also be used to add a library to the webaccess. These libraries do not necessarily have to interact with the webaccess directly, but can be used by other plugins.

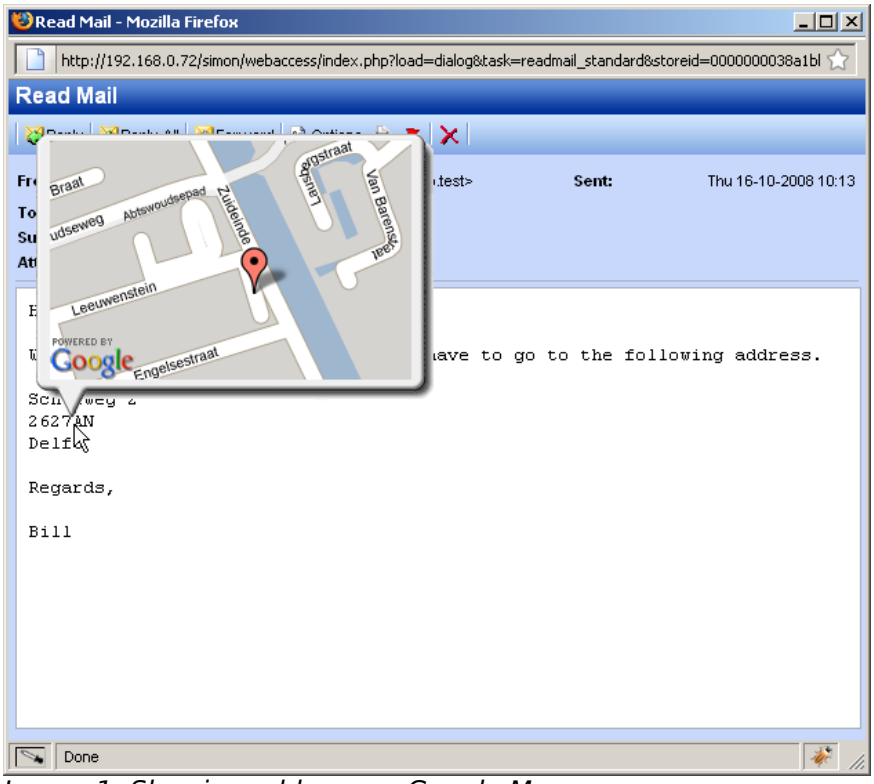

Image 1: Showing address on Google Maps

2.1 Plugin structure

In the root of the webaccess folder there is a plugin directory. Inside that directory each plugin has their own folder.

The plugin is defined in the manifest.xml file that contains the configuration of the plugin in XML-format. This file is stored in the folder of the plugin. This manifest contains information like name of the author, name of the plugin and a description.

It also contains information about what files the webaccess should include. Both on the server side and on the client side. Each component that you want to add is listed here.

In order to place hooks you will also need to define a plugin class. In this class the developer can register to any hook in the system. When this hook is triggered the plugin class is notified and the developer will be able to execute code he wishes to have executed at that point in the webaccess code. For the client side and the server side a separate class will need to be defined. One in PHP and the other one in Javascript. More on this plugin class later on in this document.

To implement new functionalities it is also possible to add complete new components as modules, views, dialogs, CSS files, widgets, etc. These components will have all access and permissions the native webaccess components have.

You can also define your own translations for your plugin. In your XML manifest you simply define the folder that contains all the different translations and the webaccess automatically includes the selected language. More on how to use these translations and the exact folder structure can be found later in this document.

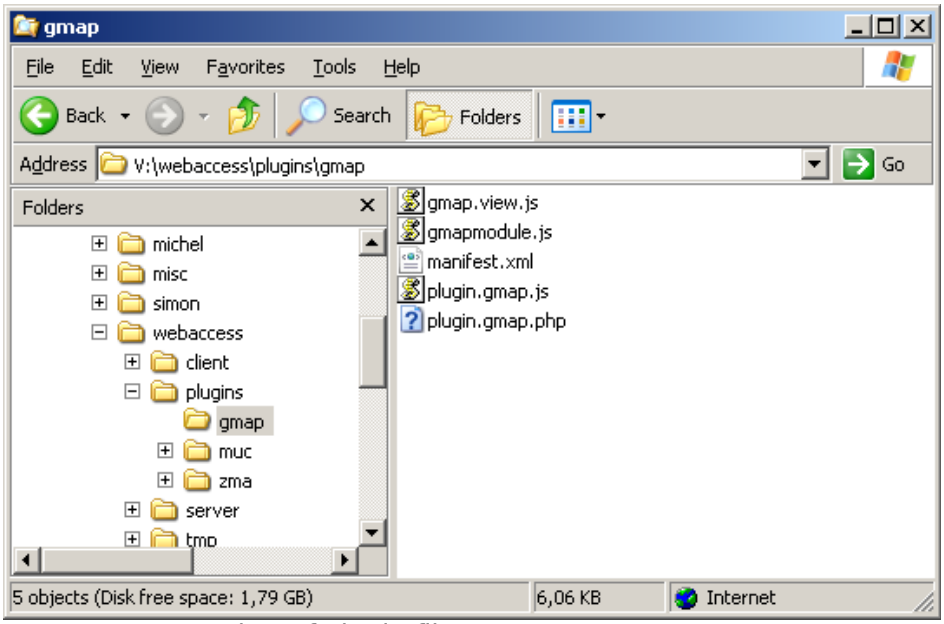

Image 2: Screenshot of plugin file structure

2.2 Configuration

The configuration file config.php can define three options for the Plugin System.

Configuration key	Default value	Description
PATH_PLUGIN_DIR	“plugins”	Defines the path to the directory that contains all the plugins.
ENABLE_PLUGINS	true	Enables or disables all the plugins.
DISABLED_PLUGINS_LIST	(empty string)	A semicolon separated list of the names of the plugins that will be disabled.

An example of the configuration values set in the config.php can be found in the following lines of code.

```
// Define the path to the plugin directory (No slash at the end)
define("PATH_PLUGIN_DIR", "plugins");


// Define the path to the plugin directory
define("ENABLE_PLUGINS", true);


// Define list of disabled plugins separated by semicolon
define("DISABLED_PLUGINS_LIST", 'gmap;spellchecker');
```

2.3 Manifest

The manifest defines the contours of your plugin. It defines, besides the some descriptive information about the plugin, the files and modules that need to be included by the Webaccess.

Below is the structure of the manifest XML of a fictive example plugin.

```
<plugin version="1">
	<info>
		<version>1,0</version>
		<title>Zarafa Example Plugin</title>
		<author>S.A.Koster</author>
		<authorURL>http://www.zarafa.com</authorURL>
		<description>Example of plugin</description>
	</info>

	<resources>

		<client>
			<clientfile>plugin.example.js</clientfile>
			<clientfile type="module">examplepluginmodule.js</clientfile>
			<clientfile type="css">example.css</clientfile>
		</client>

		<server>
			<serverfile>plugin.example.php</serverfile>
			<serverfile type="module" module="examplepluginmodule">
				class.examplepluginmodule.php
			</serverfile>
		</server>

		<dialogs>
			<dialog>
				<name>exampledialog</name>
				<file>dialogs/exampledialog.php</file>
			</dialog>
		</dialogs>

		<language>translations</language>

	</resources>
</plugin>
```

The info part describes some basic information regarding the plugin: Version, title, author information and a description.

The resources part describes what resources the plugin uses.

2.3.1 Client files

```
(...)

<client>
	<clientfile>plugin.example.js</clientfile>                        // Plugin class
	<clientfile type="module">examplepluginmodule.js</clientfile>     // Module
	<clientfile type="css">example.css</clientfile>                   // CSS file
	<clientfile>widget.example.js</clientfile>                        // Widget
	<clientfile>view.example.js</clientfile>                          // View
	<clientfile>example.js</clientfile>                               // Other file

	<clientfile load="main">load.main.js</clientfile>    // Load only in main window
	<clientfile load="dialog">load.dialog.js<clientfile> // Load only in dialog
	<clientfile>load.all.js</clientfile>                 // Load in main and dialog
</client>

(...)
```

The client portion contains CLIENTFILE-nodes. Each node links to a file that must be included on the client-side of the webaccess. There are few file types that can be included in this way. The

code example above indicates what type of files can be included. Every CLIENTFILE-node only contains the filename (and path to that file) except for modules and CSS-files. The two exceptions require a type attribute. The webaccess requires modules to be specifically identified as modules and CSS files need to be included in a different way than Javascript-files.

File type	XML Node	Attribute(s)	Value
JS Plugin class	clientfile	None	Path to file
JS Module	clientfile	type=module	Path to file
JS Widget	clientfile	None	Path to file
JS View	clientfile	None	Path to file
Other JS file	clientfile	None	Path to file
CSS file	clientfile	type=css	Path to file

It is also possible to say that a file must only be loaded in the main window or only in the dialogs. You can do this by specifying the load attribute with either the value “main” (load only in main window) or the value “dialog” (load only in dialogs). If you do not specify the attribute the file will be loaded in the both the main window as in the dialogs.

Load in	XML Node	Attribute(s)	Value
Main window	clientfile	load=main	Path to file
All dialogs	clientfile	load=dialog	Path to file
Main and all dialogs	clientfile	None	Path to file

2.3.2 Server files

```
(...)

<server>
      <serverfile>plugin.example.php</serverfile>                 // Plugin class
      <serverfile type="module" module="examplepluginmodule">     // Module
            class.examplepluginmodule.php
      </serverfile>
      <serverfile>example.php</serverfile>                        // Other file
</server>

(...)
```

The server portion contains SERVERFILE-nodes. Each node links to a file that must be included on the server-side of the webaccess. The code above shows three file types.

The first one is a plugin class file. This file is included without any specific attributes.

The second type is a module file that needs to attributes: type and module. The type attribute identifies the file as a module and the module-attribute defines the class name of the module. As mentioned before the modules need to be identified as such and that is why module-files need these extra attributes.

The third type is a PHP file that you want included in the server-side webaccess. This file can contain another PHP class, extra functions or another PHP library for example.

File type	XML Node	Attribute(s)	Value
PHP Plugin class	serverfile	None	Path to file
PHP Module	serverfile	type=module module=CLASSNAME	Path to file
Other PHP file	serverfile	None	Path to file

2.3.3 Dialogs

```
(...)
<dialogs>
      <dialog>
            <name>exampledialog</name>
            <file>dialogs/exampledialog.php</file>
      </dialog>
</dialogs>
(...)
```

The dialogs portion contains DIALOG-nodes that each define a dialog that is added to the webaccess by this plugin. In these DIALOG-nodes we define the (system)name of the dialog and the path to the PHP-file that defines the dialog.

When the webaccess opens a dialog is supplies a TASK-argument in the URL. When the (system)name is passed as TASK-argument (in combination with the plugin name) the dialog defined by the plugin is opened.

The PHP-file contains the setup of the dialog. In this setup the JS-files, CSS-files, module-files, etc. that have to be included will be defined as well as the rest of the configuration for a dialog. How this works is described in the Webaccess Tutorials Document.

2.3.4 Translations

```
(...)
<language>translations</language>
(...)
```

The translations portion only defines the directory that contains the translations. How to set up these translations will be described later on in this document.

2.4 Plugins on the client-side

The plugin developer can add completely new components (like modules, views, widgets, etc.) to the webaccess. These components will have the same access and permissions as the native webaccess components have. The syntax and structure of these components does not differ from native components. As mentioned before it is important to have the modules identified in the manifest XML.

When extending existing components you can add a normal JS file where you extend the existing JS objects through prototyping. You can also add your own libraries through this way.

The added components are only used when the webaccess directly instantiates the component. A module, view or widget have to specifically be called before it will get executed. The developer can also define a plugin class to let the plugin interact directly with the webaccess. When the plugin class is initialized it has the ability to register to hooks. When the webaccess reaches a point in the code where a hook has been placed all plugins that have registered to that hook will be notified and will have the opportunity to execute their code before the native code will continue.

2.4.1 Plugin class

The plugin class on the client-side is a prototyped Javascript object/class. The class is always instantiated when it exists. It is possible to create a plugin that does not have a plugin class.

```
/**
 * Example plugin class
 */

Pluginexample.prototype = new Plugin;
Pluginexample.prototype.constructor = Pluginexample;
Pluginexample.superclass = Plugin.prototype;

function Pluginexample(){}

Pluginexample.prototype.init = function(){
      this.registerHook("main.hierarchymodule.sharedFoldersPane.buildup");
}

Pluginexample.prototype.execute = function(eventID, data){
      switch(eventID){
            case "main.hierarchymodule.sharedFoldersPane.buildup":
                  this.doExampleStuff(data);
                  break;
      }
}

Pluginexample.prototype.doExampleStuff = function(data){
      // Do stuff
}
```

The structure of such a class looks like the code shown above.

```
Pluginexample.prototype = new Plugin;
Pluginexample.prototype.constructor = Pluginexample;
Pluginexample.superclass = Plugin.prototype;
```

These lines are needed for all class components in the webaccess. The plugin class is called “Pluginexample” and it extends the superclass “Plugin”.

```
function Pluginexample(){}
```

The constructor does not do much in this example, but you can of course set the properties you will need in this class. You cannot not use the constructor to register to hooks.

```
Pluginexample.prototype.init = function(){
	this.registerHook("main.hierarchymodule.sharedFoldersPane.buildup");
}
```

The init-function is called to initialize the plugin class. In this function you can start registering to hooks. This can be done by using the registerHook function that the plugin class inherits. Each hook is identified by a string. The hook that is placed above is identified by the string “main.hierarchymodule.sharedFoldersPane.buildup”. This specific hook allows you to place links under the hierarchy list in the webaccess. If you want to define more hooks you can call this method multiple times.

```
Pluginexample.prototype.execute = function(eventID, data){
	switch(eventID){
		case "main.hierarchymodule.sharedFoldersPane.buildup":
			this.doExampleStuff(data);
			break;
	}
}
```

The execute function is called when the plugin class has been registered to one or more hooks and that hook is triggered by the native webaccess code. The two arguments that are supplied are the identifier of the hook that is triggered and an object that contains the data that you can modify.
The data object contains references to the data that the plugin is allowed to change. In the example of the hook “main.hierarchymodule.sharedFoldersPane.buildup” the data object will contain a reference to an HTML DOM element. You can use this reference to add your own HTML (read: links) to the pane below the hierarchy list. Another example can be that the data object contains a list of items that will be added to a (context)menu.
It is best to create a switch statement in this execute function so you can create separate function for each hook. This is what is done in the code above. Note that in order to do that you have to pass on the data object to that separate function.

```
Pluginexample.prototype.doExampleStuff = function(data){
	// Do stuff
}
```

In the code above you can define all the actions that you want to do when the hook is triggered. This can mean adding items to menus or changing the output that will be displayed (so you can add mouse over hover functionalities like Google Maps or linking telephone numbers to Skype when they are clicked on).

2.4.2 Client-side Plugin Manager API

The client-side Plugin Manager has the following API functions available.

<boolean> pluginExists (<string> pluginname)

This method checks if another plugin is loaded. If a plugin is available it will return TRUE, otherwise it will return FALSE.

```
if(webclient.pluginManager.pluginExists("gmap")){
	this.registerHook("client.module.readmailitemmodule.setbody.predisplay");
	this.registerHook("client.module.readmailitemmodule.setbody.postdisplay");
}
```

2.5 Plugins on the server-side

The plugin developer can add completely new components to the server-side as well. On the server-side this means that new modules can be added. These modules have the same access and permissions as other native webaccess modules. The syntax and structure of these modules does not differ from native modules. As mentioned before it is important to have the modules identified as such in the manifest XML.

When extending existing components you can add a normal PHP file where you extend the existing PHP classes. You can also add your own libraries through this way.

As on the client-side, the server-side modules are only used when the webaccess or other plugins directly instantiate the modules. They has to be specifically called before it will get executed. The developer can also define a plugin class on the server-side to let it interact directly with the webaccess. When the plugin class is initialized it has the ability to register to hooks. When the webaccess reaches a point in the code where a hook has been placed all plugins that have registered to that hook will be notified and will have the opportunity to to execute their code before the native code will continue.

2.5.1 Plugin class

The plugin class is always instantiated when it exists. It is possible to create a plugin that does not have a plugin class.

```
<?php
/**
 * Example Plugin class
 */
class Pluginexample extends Plugin {
	function Pluginexample(){}
	function init(){
		$this->registerHook('dialogs.general.buildMenu');
	}
	function execute($eventID, &$data){
		switch($eventID){
			case 'dialogs.general.buildMenu':
				$this->addDialogMenuItems($data);
				break;
		}
	}
	function addDialogMenuItems(&$data){
		// Do stuff
	}
}
?>
```

The structure of such a class looks like the code shown above. You should be able to note that it looks very similar to the client-side plugin class.

```
class Pluginexample extends Plugin {
```

The plugin class is called “Pluginexample” and it extends the superclass “Plugin”. It inherits some basic functionalities that are needed in this class.

```
function Pluginexample(){}
```

The constructor does not do much in this example, but you can set the properties you will need in this class. You cannot use the constructor to register to hooks.

```
function init(){
      $this->registerHook('dialogs.general.buildMenu');
}
```

The init-function is called to initialize the plugin class. This function is used for the same actions as the client-side Javascript plugin class is used for. In this function you can start registering to hooks. This can be done by using the registerHook function that the plugin class inherits. Each hook is identified by a string. The hook that is placed above is identified by the string “dialogs.general.buildMenu”. This specific hook allows you add menu items to the top menu of all dialogs. If you want to define more hooks you can call this method multiple times.

```
function execute($eventID, &$data){
      switch($eventID){
            case 'dialogs.general.buildMenu':
                  $this->addDialogMenuItems($data);
                  break;
      }
}
```

The execute function is called when the plugin class has been register to one or more hooks and that hook is triggered by the native webaccess code. The two arguments that are supplied are the identifier of the hook that is triggered and an array that contains the data you can modify. This is similar to the execute function of a plugin class on the client-side. In PHP it is important to keep in mind that if you want to modify the data in the $data variable, you have to pass it as reference. You will be able to change the data directly because of this reference. To read more about references you can consult the PHP manual reference on http://www.php.net/references.

The ampersand (&) symbol when defining the arguments of the function is vital to pass the data as a reference. The $data variable (array) will contain the data that you can change. It is advisable to use a switch statement and a separate function for each hook to keep the handling of multiple hooks apart.

```
function addDialogMenuItems(&$data){
      // Do stuff
}
```

In the code above you can define all the actions that you want to do when the hook is triggered. Note that the $data variable is again passed as a reference.

2.5.2 Storing data in session

In the webaccess the contents of the *$_SESSION* superglobal is not stored in the session. The Plugin Class offers an easy mechanism to store data and preserve it over multiple requests in a session on the server. Storing the data in a session should only be done when this is necessary or if it will improve the performance.

Each Plugin Class inherits the *setData* method that allows the plugin to store data in the session. The first argument supplies the key under which the data must be stored. This key will not interfere with the data of other plugins, so it is not a problem to have two plugins storing different data under the same key. The second argument contains the data that will be stored. The following example shows the use of this method.

```
<?php
class Pluginsugarcrm extends Plugin {
```

```
      (...)
      function storeAuthenticationData(){
            $data = Array(
                  'username' => $this->username,
                  'sessionid' => $this->sessionid
            );
            $this->setData("authData", $data);
      }
}
?>
```

As soon as the *setData* method is called the data is stored in the session. It could be that you want to wait before the data is actually written to the session file. For this you can supply a third argument to indicate whether you want to automatically save the data in the session file or whether you want to queue the data until you have added all the data and write everything at once. The latter case is useful when you have a large set of data that you want to add.

The following example shows how you can postpone the writing command until you have finished adding all the data.

```
<?php
class Pluginsugarcrm extends Plugin {
      (...)
      function storeCache(){
            // No writing...
            $this->setData("configuration", $this->configuration, false);
            // No writing just yet...
            $this->setData("lastopenedfile", $this->lastopenedfile, false);
            // Now you can write!
            $this->setData("authData", $this->authData, true);
      }
}
?>
```

In the last example you can also omit the third argument in the last call to *setData*. The default value is set to true.

When you want to retrieve the data from the session you must use the *getData* method. Supply the key under which you filed the saved data as argument and the method will return the data. It will return NULL when there is not data filed under the supplied key.

```
<?php
class Pluginsugarcrm extends Plugin {
      (...)
      function getAuthData(){
            $this->authData = $this->setData("configuration");
      }
}
?>
```

2.5.3 Server-side Plugin Manager API

The server-side Plugin Manager has the following API functions available.

<boolean> pluginExists (<string> pluginname)

This method checks if another plugin is loaded. If a plugin is available it will return TRUE, otherwise it will return FALSE.

```
if($GLOBALS['PluginManager']->pluginExuists("gmap")){
      // Do stuff that only has to happen when the plugin "gmap" is available
}
```

2.6 Dialogs

The way to create a dialog is discussed in the tutorial on how to create a new dialog.

A plugin can define their own dialogs the same way the native webaccess code does. In the manifest you must define the task (identifier of the dialog) and the path to the file that sets up the dialog. The manifest portion below defines two example dialogs.

```
(...)
<dialogs>
      <dialog>
            <name>exampledialog</name>
            <file>dialogs/exampledialog.php</file>
      </dialog>

      <dialog>
            <name>secondialog</name>
            <file>dialogs/secondialog.php</file>
      </dialog>
</dialogs>

(...)
```

To open a dialog you need to pass the task identifier and the plugin name as arguments in the URL. The following example Javascript code shows how this is done. The first call opens a modal dialog and the second opens a normal dialog. Both calls will have a slightly different set of arguments that can be used. However the first four arguments are the same.

```
webclient.openModalDialog(
      moduleObject,             // Module object or module ID. Also -1 for no module.
      'firstexampledialog',                                 // Name
      DIALOG_URL+'task=exampledialog&plugin=example',      // URL and arguments
      600,                                                  // Width
      450,                                                  // Height
      plugin_example_exampledialog_callback                 // Callback function
);
```

```
webclient.openWindow(
      moduleObject,             // Module object or module ID. Also -1 for no module.
      'secondexampledialog',                                // Name
      DIALOG_URL+'task=seconddialog&plugin=example',      // URL and arguments
      600,                                                  // Width
      450,                                                  // Height
      false                                                 // Not resizable
);
```

2.7 Translations

The webaccess uses the GNU gettext for translations. A plugin can define their own translations in their own plugin folder. In the manifest you define the path to the folder from the root of the plugin folder. In the example plugin as shown in Image 3 the definition in the manifest will be the following.

```
(...)
<language>trans</language>
(...)
```

2.7.1 Translations file structure

The defined translations folder needs to have a specific file structure. For each supported language you have to define a subfolder with the language code in the name. The folder name will also contain the extension ".utf-8". The folder names will look like the following.

```
en_US.utf-8        // US English
nl_NL.utf-8        // Dutch
de_DE.utf-8        // German
fr_FR.utf-8        // French
```

In this folder you have to define a subfolder called LC_MESSAGES. In the LC_MESSAGES folder you can define your .PO file. You have to transform the .PO file into the binary .mo file so it can be used by the gettext functionality in the webaccess.

The name of the .MO file is bound to the name of the plugin. If your plugin is called "example" your translations file will have to be called "plugin_example.mo". The string "plugin_example" is also called the domain of the translations for that specific plugin.

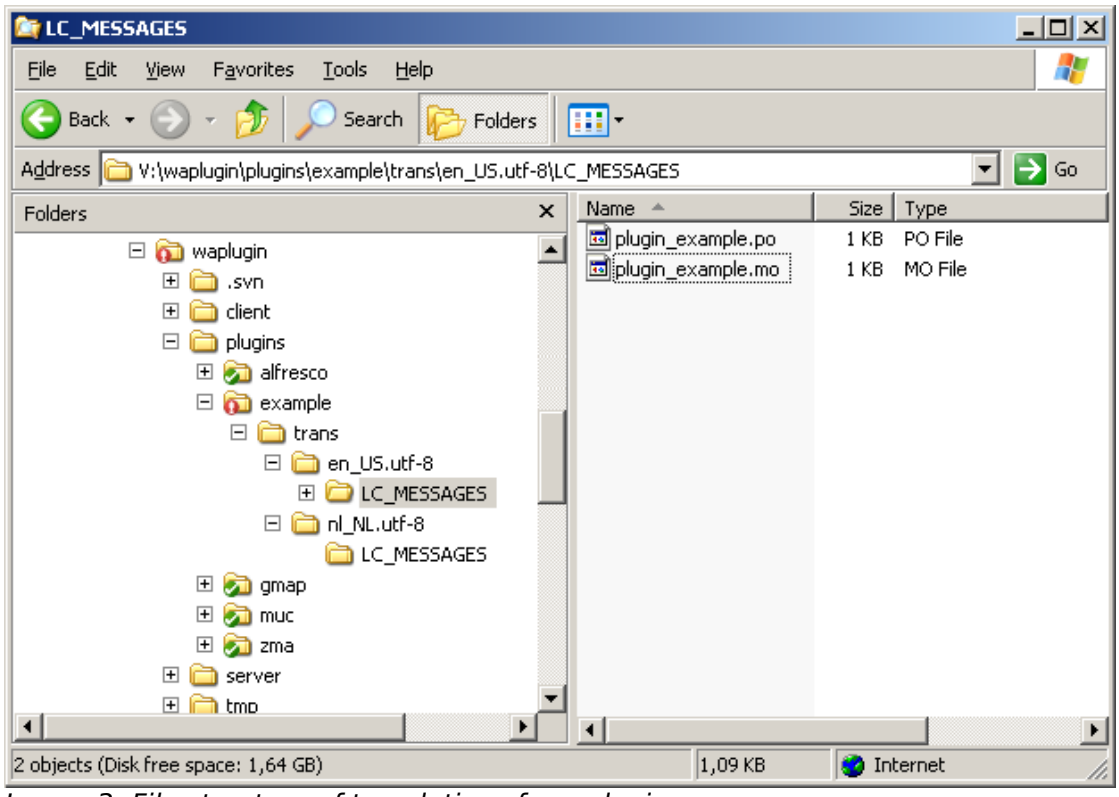

Image 3: File structure of translations for a plugin

2.7.2 Using the translations in the code

In order to use the translations in the webaccess you have to use specific functions. In the client-side webaccess you can use the _() function. This function accepts two arguments. The first one is the translatable text and the second argument will have to the be the translation domain. The following Javascript code example shows what this will look like for the plugin named “example”.

```
alert(_("Your text here? Call 555-1234", "plugin_example"));
```

On the server-side the normal _() can only be used by the native webaccess code as it looks up a message in the current domain. This means that it will always select the native Webaccess translations. In order for the plugin to use their own translations it has to use the standard gettext() function. When calling this function you have to supply the translation domain as the first argument and the translatable text as the second argument. The following PHP code example shows what this will look like for the plugin named “example”.

```
<?php
echo dgettext('plugin_example', 'Your text here? Call 555-1234');
?>
```

The reason that the gettext function on the client-side accepts the arguments in reverse order is to keep it backwards compatible. If you really hate to have those different sets of arguments, you can also use the dgettext() alias for _() on the client-side. It is created specifically to match the server-side PHP function. The following Javascript code example shows what this will look like for the plugin named “example”.

```
alert(dgettext("plugin_example", "Your text here? Call 555-1234"));
```

3 Debugging

In order to debug plugins in the webaccess you need to have the debug.php file and place it in the root of the webaccess. Create two files in this folder as well: debug.txt and errors.txt. Make sure the apache user has the proper permissions to write to these files. The debug.php file will trigger several debug mechanisms that could cause the webaccess to run a bit slower. Therefor it is not advised to use the debug.php in production environments.

You can put the Plugin System into debug mode by defining the following debug.php configuration option.

Configuration key	Default value	Description
DEBUG_PLUGINS	true	Enables or disables the debug mode of the Plugin System

The debug mode of the Plugin System provides the plugin developer with the following options.

- **Plugin Manager reports on manifest validation.**
  The Plugin Manager will output error messages when the manifest of a plugin does not validate and why. These messages will be outputted to the debug.txt file.
- **No caching of the plugin data in the session.**
  The data the Plugin Manager retrieves from the manifests is stored in the session. Normally the session data will be used in each request improving the performance of the Plugin System. In debug mode the Plugin Manager is forced to always use the data from the manifests. Every update to the plugin is then immediately available to the next request to the server.

When you do not have a debug.php file you can also define the debug configuration option in the config.php file. The downside to this is that the server will not report of errors in your PHP code and will not report on the validation of the manifest. The not caching of the plugin data will still be available.

3.1 Invalid manifest debug errors

The following error messages can be outputted when the manifest of a plugin is invalid.

- **[PLUGIN ERROR] Plugin "PLUGINNAME" has an invalid manifest.**
  This message is always outputted when the manifest is invalid. This message will mention the plugin name and is printed after a more detailed error messages indicating where the manifest is invalid.
- **[PLUGIN ERROR] No plugin info and/or resources were found.**
  This message is printed when there are no <info> or <resources> nodes defined in the manifest.
- **[PLUGIN ERROR] Server file XML is invalid.**
  This message is printed when a <serverfile> node is invalid.
- **[PLUGIN ERROR] Client file XML is invalid.**
  This message is printed when a <clientfile> node is invalid.
- **[PLUGIN ERROR] Dialog XML is invalid (contains invalid name and file info).**
  This message is printed when a <dialog> node contains non-string input for the dialog name and the path to the setup file.
- **[PLUGIN ERROR] Dialog XML is invalid (does not contain name and file info).**
  This message is printed when a <dialog> node does not contain a dialog name or a path to the setup file.

4 Documentation

Still to be written...

4.1 Hooks

4.2 Webaccess documentation and API

4.3 Best practices