WebAccess Plugin Architecture

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

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.

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 (client-side only)**
  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.

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 about descriptive information like name of the author, name of the plugin and a description.

It als 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.

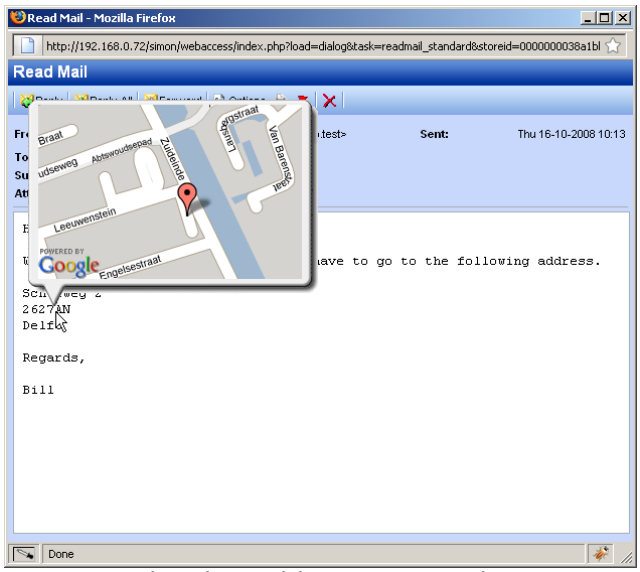

Image 1: Showing address on Google Maps

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 create the features he

wishes to have 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.

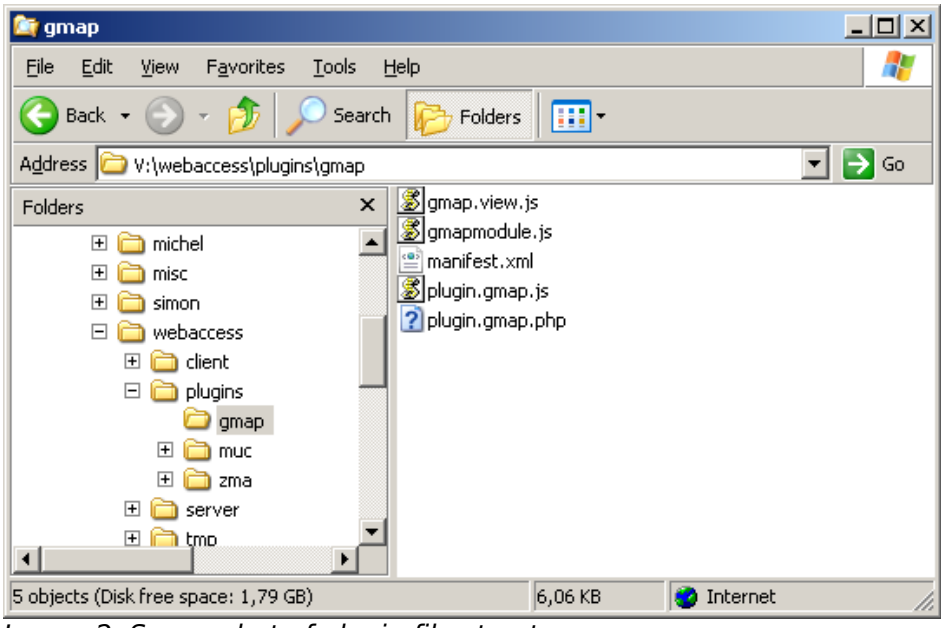

Image 2: Screenshot of plugin file structure

Example

Just to give you a bit of a peak into the plugin system, here is some random example code to get a feeling of how the plugin system will look like code-wise.

The following bit of code is a manifest file of a Google Maps plugin.

```
<plugin version="1">
	<info>
		<version>0.1</version>
		<name>Gmap</name>
		<title>Zarafa Google Maps View</title>
		<author>S.A.Koster</author>
		<authorURL>http://www.zarafa.com</authorURL>
		<description>Zarafa Google Maps View</description>
	</info>
	<resources>
		<server>
			<serverfile>plugin.gmap.php</serverfile>
		</server>
		<client>
			<clientfile type="js">plugin.gmap.js</clientfile>
			<clientfile type="module">gmapmodule.js</clientfile>
			<clientfile type="js">gmap.view.js</clientfile>
			<clientfile type="css">gmap.css</clientfile>
		</client>
	</resources>
</plugin>
```

The next bit of code is an example of a Google Maps plugin class.

```
<?php
class Plugingmap extends Plugin {
	var $APIKEY;

	function Plugingmap(){}


	function init(){
		$this->registerHook('main.load.include.jsfiles');
		$this->APIKEY = 'ABQIAAAABANe1LKxmYdkK8B9fovJHRSUEys-ALfvZz4Rtjy-
```

```
18vMsnutWhTs9jG_vy0v_k4kR5gnz1XvHuZM7g';
	}

	function execute($eventID, &$data){
		switch($eventID){
			case 'main.load.include.jsfiles':
				$this->onMainLoadIncludeJsFiles($data);
				break;
		}
	}

	function onMainLoadIncludeJsFiles(&$data){
		$data['files'][] = 'http://maps.google.com/maps?file=api&amp;v=2&amp;key='.
$this->APIKEY;
	}
}
?>
```