Joomla/Mambo

Community Builder

Version 1.0 RC2

Plugin Framework

API

Beat B.

Contact: PMS to “Beat” on www.joomlapolis.com

document version build 10 – 10 November 2005

Table of Contents

List of Figures

1 Introduction

Community Builder 1.0 RC2 introduces the possibility to add functionality through extensions called plug-ins. This is a combination of component, module and mambots functions as known in **Joomla/Mambo**.

1.1 Plug-in types

Community Builder supports currently:

- Language plug-ins
- User plug-ins
- Template plug-ins

In the future:

- Field-types plug-ins

Will also be supported.

There are special classes of plug-ins:

- Core plug-ins (starting with "**cb.**")
- Private Messaging plug-ins (starting with "**pms.**")

1.2 Plug-in naming

To avoid plugins name conflicts and duplicate developmentts, plugin-authors are encouraged to contact the authors of Community Builder before starting development and choosing a plugin name.

Names from the English dictionary or reflecting other Joomla or Mambo components, as well as composed of two such names, as well as any name starting with «cb» are reserved either for Community Builder itself, or for coordinated developments.

1.3 Installation process

Plug-in installation is documented in the user manual.

Standard plug-in installers are a simple zip file, as usual in Joomla!.

1.4 Structure of plug-ins

The plug-in installer is a zip file containing:

- An **XML** file (with **`.xml`** extension) including all information for installation and uninstall.
- An empty index**`.html`** file for each folder created by the installer.
- At least one **php** (with **`.php`** extension) file containing the plug-in main code (or language definitions).
- Further files and folders can be contained in the plug-in, as required by the plug-in functions (images, php files, language files folder etc).

1.5 XML file

The XML file is the first and most important file for a CB plug-in. It contains all "instructions" for CB to handle correctly the plug-in installation, settings and un-installation.

It must be XML-language compliant (otherwise the simplified XML handler used by Joomla! will not flag errors in the file, but just stop working or flag other error).

It is divided into following main parts (some of which are optional):

- Header
- Files
- Plugin-parameters
- Tabs section with:
  - Tab definition
  - Tab parameters
  - Fields
- Installation SQL queries
- Un-installation SQL queries
- Installation file for PHP function
- Uninstall file for PHP function.

1.5.1 Header

The XML header looks as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<cbinstall version="4.5.3" type="plugin" group="user">
	<name>pms.MyPMSPro</name>
	<author>JoomlaJoe and Beat</author>
	<creationDate>September 2005</creationDate>
	<copyright>(C) 2005 MamboJoe.com</copyright>
	<license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
	<authorEmail>mambojoe@mambojoe.com</authorEmail>
	<authorUrl>www.mambojoe.com</authorUrl>
	<version>1.0 RC 2</version>
	<description>Provides the MyPMS and PMS Pro integration for Community
Builder.</description>
```

Figure 1 – Header for XML Plug-in Installation File

The file character-set should be set right from the begin to UTF-8 for future compatibility with Joomla! 1.1, but accented characters escaped properly in html coding:

```
<?xml version="1.0" encoding="utf-8"?>
```

The installation file is included in this flag:

```
<cbinstall version="4.5.3" type="plugin" group="user">
```

The version indicates the used installer version for compatibility check.

The Type must be “plug-in” for CB.

The Group indicates the type of plug-in and may be:

- “user” for user-type plug-ins
- “templates” for templates-plug-ins
- “language” for language-plug-ins
- other words are reserved for future use.

```
        <name>pms.MyPMSPro</name>
```

The name is used for two purposes:

1. To differentiate special kinds of plug-ins (first part separated by a dot “.”):
   - “**cb.**” Indicates core plug-ins (reserved for CB’s own use only)
   - “**pms.**” Indicates PMS-capable plugins

   The kinds of plug-ins also set the class from which the plug-in object is derived.
2. To name the folder to create for installation (prefixed by “**plug**_” in case of user-plug-ins). Dots “.” And spaces “ “ are converted to underscores “_” for the folder name.

```
        <author>JoomlaJoe and Beat</author>
        <creationDate>September 2005</creationDate>
        <copyright>(C) 2005 MamboJoe.com</copyright>
        <license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
        <authorEmail>mambojoe@mambojoe.com</authorEmail>
        <authorUrl>www.mambojoe.com</authorUrl>
```

are free fields. Email and URL must be valid for future use.

```
<version>1.0 RC 2</version>
```

This is important: The version must strictly match the CB version. It is expected that the API may evolve until 1.0 final release without full backwards compatibility, if needed.

```
        <description>Provides the MyPMS and PMS Pro integration for Community
Builder.</description>
```

This description is shown after successful install and when displaying the plug-in. It may contain instructions for the admin.

1.5.2 Files

The next section is about files to install:

```
<files>
	<filename plugin="pms.mypmspro">pms.mypmspro.php</filename>
	<filename>index.html</filename>
</files>
```

Figure 2 – PMS plug-in File section for XML Plug-in Installation File

The XML file should not be included; it is copied by the CB installer at the end if there are no XML errors.

The main file containing the main classes should have the tag argument "**`plugin=`**" with the name of the plug-in as above.

Don't forget the empty index.html file for web server settings allowing directory listings.

A language plug-in needs to have following files:

```
<files>
  <filename plugin="english">default_language.php</filename>
  <filename>index.html</filename>
  <filename>calendar-locals.js</filename>
  <filename>images/index.html</filename>
  <filename>images/nophoto.jpg</filename>
  <filename>images/pendphoto.jpg</filename>
  <filename>images/tnnophoto.jpg</filename>
  <filename>images/tnpendphoto.jpg</filename>
</files>
```

Figure 3 - Language plug-in File section for XML Plug-in Installation File

we see that directories can be created and copied from the zip file.

1.5.3 Plug-in Parameters

Next tag is the **`params`** tag for the plug-in. We will see later that tabs can also have their own parameters.

```
<params>
  <param name="pmsType" type="list" default="1" label="PMS Component type"
description="Choose type of component installed. &lt;strong&gt;IMPORTANT: Component
configuration must also be done!&lt;/strong&gt;">
       <option value="1">MyPMS Open Source</option>
       <option value="2">PMS Pro</option>
  </param>
  <param name="@spacer" type="spacer" default="" label="" description="" />
  <param name="pmsMenuText" type="text" size="25" default="_UE_PM_USER" label="PMS Send
Menu/Link text" description="Default is _UE_PM_USER, the local translation of &quot;Send
Private Message&quot;" />
  <param name="@spacer" type="spacer" default="only for PMS Pro" label="Following
parameters:" description="" />
  <param name="online" type="radio" default="1" label="_UE_ONLINESTATUS"
description="IMPORTANT: General Community Builder configuration must also allow to show
this!">
       <option value="0">Hide</option>
       <option value="1">Show</option>
  </param>
  <param name="examplefieldsel" type="field" size="" default="" label="Example Field"
description="For example" />
  <param name="newslettersRegList" type="custom" class="getNewslettersTab"
method="loadNewslettersList" default="" label="_UE_NEWSLETTERSREGLIST"
description="_UE_NEWSLETTERSREGLIST_DESC" />
</params>
```

Figure 4 - Params section for XML Plug-in Installation File

Most/all Joomla! modules xml parameter types can be used also for plug-ins. A few extensions have been made:

- The “spacer” type has been extended to be also used as separator with text or as section header.
- The “field” type has been added to be able to select a user or CB field from the list of published fields, sorted by tab ordering & field ordering. The parameter value stored and passed to the plug-in is the fieldid, not the field name, so that in case of future change of name, the field stays referenced.
- The “**`custom`**” type has been added. This allows a plug-in to display his own controls, as for instance the list of newsletters in the **YaNC** plug-in.

Parameters have default values that must be matched when fetching the parameters in the plug-in.

Parameters are optional, but following tags need to be left if there are NO parameters:

```
<params>
</params>
```

Otherwise a text area will be displayed for typing in freely parameters.

1.5.4 Tabs

Next section is about tabs included in user plug-ins:

```
<tabs>
  <tab name="_UE_PMSTAB" description="" class="getMyPMSProTab" fields="0"
position="cb_right" displaytype="html">
    <params>
      <param name="@spacer" type="spacer" default="Quick PMS Settings" label=""
description="" />
      <param name="showTitle" type="list" default="1" label="Show Tab title"
description="Show the title of the tab inside this tab. The description is also shown, if
present. &lt;strong&gt;IMPORTANT: The title is the tab title here.&lt;/strong&gt;">
             <option value="0">Hide</option>
             <option value="1">Show</option>
      </param>
      <param name="width" type="text" size="10" default="30" label="Width (chars)"
description="" />
    </params>
    <fields>
    </fields>
  </tab>
</tabs>
```

Figure 5 - Tabs section for XML Plug-in Installation File

A user plug-in may include multiple tabs with a "<**tab**> tag within the "<**tabs**>" tag.

1.5.4.1 Tab definition

The tab parameters are as follows:

name	Name appearing as title on tab
description	Html text appearing usually at begin of tab.
class	Name of the class in the main php file for handling the tab
fields	0: tab does not accept normal cb fields on it. Any fields defined under <fields> tag will be created and treated as private plug-in fields and will not appear in front-end or backend. 1: tab does accept normal cb fields on it.
position	Default position of tab on user profile. Top-down / left to right: cb_head, cb_left, cb_middle, cb_right, cb_tabmain, cb_underall .
displaytype	Default type of display of this tab: tab, div, html, overlib, overlibfix, overlibsticky .

1.5.4.2 Tab parameters

All parameters rules listed above for plug-ins apply also to tab parameters.

1.5.4.3 Fields

Tabs can define their own fields.

The fields section within tabs allows the plug-in to add its own CB-standard fields to the tab.

If fields="1" in the <tab> tag, they will display after the plug-in-specific tab content, and appear as any other field in front-end on user profile, update profile, and in the backend under fields management.

If fields="0" in the <tab> tag, these fields will not display in the front-end and not appear in the backend. They will be stored as usual in the **#__comprofiler** table along with other CB fields. This is the preferred way to store user-specific data for this tab.

The XML would look like this:

```
<tabs>
  <tab name="Example" description="" class="getExampleTab" fields="1"
position="cb_tabmain" displaytype="tab">
    <params>
    </params>
    <fields>
      <field title="Your first company" name="cb_firstcompany" description="First company
you worked in" type="text" registration="0" profile="1" readonly="0" params="" />
    </fields>
  </tab>
</tabs>
```

Figure 6 – Tabs with Fields section for XML Plug-in Installation File

The <**field**> tag attributes correspond to the **#__comprofiler_fields** table fields and can be easily figured out from existing CB fields.

On uninstall plug-in, tabs and field definitions are removed, but the corresponding fields data in the user's Community Builder **#__comprofiler** table are kept, so on re-install, they are conserved.

On re-install, if the **type of field changes** compared to previous version of the plugin, the field type in the database **will be automatically updated**, which can potentially **lead to data-loss, if the field-type are different**.

1.5.5 SQL queries (install and un-install)

On plug-in install and un-install, the xml file can specify SQL queries to be performed as follows:

```
<install>
  <queries>
    <query>
       CREATE TABLE IF NOT EXISTS `#__comprofiler_plug_example_test` (
         `id` int(11) NOT NULL default '0',
         `user_id` int(11) NOT NULL default '0',
         `example_text` mediumtext,
        PRIMARY KEY  (`id`)
       ) TYPE=MyISAM;
    </query>
  </queries>
</install>
<uninstall>
  <queries>
    <query>
       DROP TABLE IF EXISTS `#__comprofiler_plug_example_test`
    </query>
  </queries>
</uninstall>
```

Figure 7 - Install section for XML Plug-in Installation File

Multiple queries can be made on install and un-install using multiple “<query>” tags. Tables specific to a plug-in should be named “#__comprofiler_plug_NAMEofPLUG”.

1.5.6 Install code (also un-install code)

On plug-in install and un-install, the xml file can specify PHP code to be called as follows:

```
<installfile>example.userplugin.php</installfile>
<uninstallfile>example.userplugin.php</uninstallfile>
```

In this case, the filename is same as the main php file, but the files may differ. If these files are omitted from the files list, they will be still copied into the plug-in directory.

On Install, the function:

```
string plug_XXXX_install();
```

will be called. The “XXXX” above will be replaced by the plugin system name: that is the the main file plug attribute: in the XML file: <file plug=”XXXX”>, with spaces removed (there shouldn’t be any spaces there anyway!) and dots “.” Changed to underscores “_”.

On Un-install, the function:

```
string plug_XXXX_uninstall();
```

will be called. The “XXXX” being replaced same as for install.

Both functions must return a string (can be empty) with html-formatted results to be displayed on the admin interface upon completion of install or uninstall.

1.6 Language plugins

Language plug-ins are a type of CB plug-ins.

Their XML file typically looks as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<cbinstall version="4.5.3" type="plugin" group="language">
       <name>English</name>
       <author>MamboJoe</author>
       <creationDate>August 2005</creationDate>
       <copyright>(C) 2005 MamboJoe.com</copyright>
       <license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
       <authorEmail>mambojoe@mambojoe.com</authorEmail>
       <authorUrl>www.mambojoe.com</authorUrl>
       <version>1.0 RC 2</version>
       <description>Provides the default (English) language for Community Builder core
functions.</description>
       <files>
              <filename plugin="english">english.php</filename>
              <filename>index.html</filename>
              <filename>calendar-locals.js</filename>
        <filename>images/index.html</filename>
              <filename>images/nophoto.jpg</filename>
        <filename>images/pendphoto.jpg</filename>
        <filename>images/tnnophoto.jpg</filename>
        <filename>images/tnpendphoto.jpg</filename>
       </files>
    <params>
    </params>
</cbinstall>
```

Figure 8 – Language File Example for XML Plug-in Installation File

The **bold underlined** elements above must be changed in the xml file for each other language to correspond to Joomla!'s language name.

A "**default_language**" core plug-in is the default plug-in for the default American English. An "**English**" plug-in can be loaded, and will be used if existing instead of the **default_language** plug-in.

Important: the language file character set of the plugin should match the character-set of the language file of the site (and if multiple languages are used, all should match the same character set: this is a Mambo/Joomla! constraint, since Joomla! does not support character sets transcodings).

The only universal solution to this problem is the use of the UTF-8 character-set (variable lenght-byte encodings from 1-8 bytes per character). As Joomla is moving towards UTF-8 being mandatory in Joomla! 1.1, **translators are encouraged to use UTF-8 from the start, and site-admins to install corresponding UTF-8 core language files.**

For performance reasons, the published status is not taken into account for language plug-ins.

The translation/localization of the calendar JavaScript program used to choose dates is in the calendar-locals.js file. The author site of the calendar application gives incomplete translations/localizations, which can also be used as example.

1.7 User plugins

User-type plug-ins are the functions-extending plug-ins for adding functions to CB. These can include tabs and userbots.

1.7.1 Parameters passing

Parameters are defined in the xml file, as shown above.

They are stored CB-internally with the plug-ins SQL table.

They can be used in the plug-in methods as follows:

```
$params = $this->params;
$pmsType     = $params->get('pmsType', '1');
$showTitle   = $params->get('showTitle', "1");
```

The first argument of **`$params->get`** is the name of the parameter, which must correspond to the "**`name=`**" attribute of the corresponding "**`<param>`**" tag in the XML file.

The second optional argument is the default value of that parameter, which should correspond to the "**`default=`**" attribute of the corresponding "**`<param>`**" tag in the XML file.

The third optional parameter of this method is the name of the search/paging/sorting sorter (see paragraph 1.9.4 on page 29).

```
/**
* gets the name input parameter for search and other functions
* @param string name of parameter of plugin
* @param string postfix for identifying multiple pagings/search/sorts (optional)
* @returns string value of the name input parameter
*/
```

```
        function _getPagingParamName($name="search", $postfix="")
```

In case of tabs, the parameters in **`$this->params`** correspond to the parameters of the tab **and** of the parameters of the plug-in. Later user-specific parameters/preference settings may also be available probably this way.

1.7.2 Error management

Plug-ins user bots and tabs can give multiple error messages, calling each time this method:

```
        /**
        * PRIVATE method: sets the text of the last error
        * @param string error message
        * @return boolean true
        */
        function _setErrorMSG($msg)
```

They must also raise an error condition to stop the normal CB workflow, and make CB display an error message (for now using a JS alert):

```
        /**
        * PRIVATE method: sets the error condition and priority (for now 0)
        * @param error priority
        * @return boolean true
        */
        function raiseError($priority)
```

1.7.3 Objects

All interactions between CB and plug-ins is done with objects. Some functions are non-object, but the rule is to have objects, and as CB is developed further, it will become more and more object-oriented.

Normally, the objects of plug-ins are instantiated only once, and object variables are kept throughout the whole execution of the Community Builder call (page generation).

So for instance, if a given page generation calls a userbot method, then later a tab display method, it's the same object which is created only once, and the variables of the object will remain.

Each object should call the constructor (in php4 compliant manner) of its parent class.

Please keep in mind that in the future, not all tabs will be loaded and generated at each display.

CB developers reserve the possibility to either draw tabs only one by one and do page reloads on each tab switch, or to download tab content dynamically as needed. Therefore, the tab getDisplayTab() method may not be called upfront, even if the tab title is drawn, but not visible.

1.7.4 Tabs

All tabs are derived from this class:

```
/**
* Tab Class for handling the CB tab api
* @package Community Builder
* @author JoomlaJoe and Beat
*/
class cbTabHandler extends cbPluginHandler  {
	/**
	* Constructor
	*/
	function cbTabHandler() {
		$this->cbPluginHandler();
	}
	/**
	* Generates the menu and user status to display on the user profile by calling
back $this->addMenu
	* @param object tab reflecting the tab database entry
	* @param object mosUser reflecting the user being displayed
	* @param int 1 for front-end, 2 for back-end
	* @returns boolean : either true, or false if ErrorMSG generated
	*/
	function getMenuAndStatus($tab,$user,$ui) {
	}
	/**
	* Generates the HTML to display the user profile tab
	* @param object tab reflecting the tab database entry
	* @param object mosUser reflecting the user being displayed
	* @param int 1 for front-end, 2 for back-end
	* @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
	*/
	function getDisplayTab($tab, $user, $ui) {
	}
	/**
	* Generates the HTML to display the user edit tab
	* @param object tab reflecting the tab database entry
	* @param object mosUser reflecting the user being displayed
	* @param int 1 for front-end, 2 for back-end
	* @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
```

```
        */
        function getEditTab($tab, $user, $ui) {
        }
        /**
        * Saves the user edit tab postdata into the tab's permanent storage
        * @param object tab reflecting the tab database entry
        * @param object mosUser reflecting the user being displayed
        * @param int 1 for front-end, 2 for back-end
        * @param array _POST data for saving edited tab content as generated with
getEditTab
        * @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
        */
        function saveEditTab($tab, $user, $ui, $postdata) {
        }
        /**
        * Generates the HTML to display the registration tab/area
        * @param object tab reflecting the tab database entry
        * @param object mosUser reflecting the user being displayed
        * @param int 1 for front-end, 2 for back-end
        * @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
        */
        function getDisplayRegistration($tab, $user, $ui) {
        }
        /**
        * Saves the registration tab/area postdata into the tab's permanent storage
        * @param object tab reflecting the tab database entry
        * @param object mosUser reflecting the user being displayed
        * @param int 1 for front-end, 2 for back-end
        * @param array _POST data for saving edited tab content as generated with
getEditTab
        * @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
        */
        function saveRegistrationTab($tab, $user, $ui, $postdata) {
        }
```

PMS and Menu tabs definitions are derived from this base class, and are documented with comments in their implementation files.

Each tab in the **`getDisplayTab`** should include the code to display the description set by the admin in backend:

```
            if($tab->description != null) $return .= "\t\t<div
class=\"tab_Description\">".unHtmlspecialchars(getLangDefinition($tab-
>description))."</div>\n";
```

Please **note that all user parameters are unescaped in all cases** independently of the magic_quotes_gpc PHP setting, and that if you want to use those in SQL queries, they need to be properly escaped, e.g. with **`$database->getEscaped($row->username)`** as escapings may vary between database types (either \' or '' for ' e.g.).

1.7.5 User bots

User bots are methods or functions called upon particular CB events. They are allowed to handle those events, and to output messages, errors and to block operations of CB.

Great care must be placed when modifying data passed to them.

A userbot must first be registered.

The CB API for this is:

```
        /**
        * Registers a function to a particular event group
        * @param string The event name
        * @param string The function name
        */
        function registerFunction( $event, $method, $class=null )

called as:
global $_PLUGINS
$_PLUGIN->registerFunction
```

This is done in the main plug-in php file as follows for a function **`pluginExampleBeforeSaveUser()`**:

```
$_PLUGINS->registerFunction( 'onBeforeUserUpdate', 'pluginExampleBeforeSaveUser' );
```

and as follows for a method **`pluginExampleBeforeSaveUser()`** of a class **`'getExampleTab'`**:

```
$_PLUGINS->registerFunction('onBeforeUserUpdate',
'pluginExampleBeforeSaveUser','getExampleTab' );
```

Please **note that all user parameters are unescaped in all cases** independently of the magic_quotes_gpc PHP setting, and that if you want to use those in SQL queries, they need to be properly escaped, e.g. with **`$database->getEscaped($row->username)`** as escapings may vary between database types (either \' or '' for ' e.g.).

Here is a list of events that can be registered and the arguments received:

1.7.5.1 User management events

CB triggers many user management events that can use to sync the user data and events of CB with third party applications or to extend the functionality of CB.

Much of the plug-in framework is leverages off of the core mambo Mambot code.

At a very minimum you must register the functions you want to leverage with the available CB events.

Following table shows the event name identical with function name, and the parameters passed to the function/method:

```
//Add user via Admin events
function onBeforeNewUser (&$row, &$rowExtras, false)
function onAfterNewUser ($row, $rowExtras, false, true)

//New user Registration events
function onBeforeUserRegistration (&$row,&$rowExtras, false)
function onAfterUserRegistration ($row, $rowExtras, true)

//Confirm User events
function onBeforeUserConfirm ($userObject)
function onAfterUserConfirm ($userObject, true)

//Approve/Reject User events
function onBeforeUserApproval ($row, $approved)
function onAfterUserApproval ($row, $approved, $success)
function onUserActive ($row, $activated=true)

//Update User events
function onBeforeUserUpdate (&$row,&$rowExtras)
function onAfterUserUpdate ($row, $rowExtras, true)

//Delete User events
function onBeforeDeleteUser ($row->id)
function onAfterDeleteUser ($row->id,true)
```

1.7.5.2 User session events

CB triggers two user session events u can use to authenticate users with external services, like for example LDAP or 3PD forums.

```
//Login User event
function onBeforeLogin ($username, $passwd2)
function onAfterLogin ($row, true)

//Logout User event
function onLogout ($row, true)
```

1.7.5.3 View profile events

When profiles are viewed, events are generated for plug-ins:

- Before to allow the profile view and to alter if necessary profile content.
- After to perform other actions if necessary.

```
// View profile events
function onBeforeUserProfileDisplay ($user, $ui, $cbUserIsModerator, $cbMyIsModerator) //
ui=1 front, ui=2 backend, 2* boolean

function onAfterUserProfileDisplay ($user, $succes=true)
```

1.7.5.4 Connection events

Connection events are generated to enable plug-ins to perform particular actions on those changes of connections states.

```
// Connection events
function onBeforeAddConnection ($userid,$connectionid, $ueConfig['useMutualConnections'],
$ueConfig['autoAddConnections'],&$userMessage)

function onAfterAddConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections'])

function onBeforeRemoveConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections'])

function onAfterRemoveConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections'])

function onBeforeDenyConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections']))

function onAfterDenyConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections']))

function onBeforeAcceptConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections']))

function onAfterAcceptConnection ($userid,$connectionid,
$ueConfig['useMutualConnections'],$ueConfig['autoAddConnections']))
```

1.7.6 Generating HTML output

HTML can be generated by plug-ins on following occasions:

- User Profile Display (method **`getDisplayTab`**)
- User Edit (method **`getEditTab`**, saving output with method **`saveEditTab`**)
- Registration Page (method **`getDisplayRegistration`**, saving output with method **`saveRegistrationTab`**)

These methods are described in the Tab Object paragraph above.

1.7.6.1 Accessibility

Plug-ins should output accessible HTML code (user profile of CB already uses a table-less design), allowing disabled people to use reading browsers.

1.7.6.2 W3C Compliance

Code generated by CB aims to be compliant with W3C xhtml 1.0 transitional standards and be displayed correctly on all popular modern browsers including:

- Firefox 1.0.6+
- Internet Explorer 5.5 and 6.0
- Opera 8.5+
- Safari 1.3.1+
- Lynx (this is to help accessibility assertion and lightweight browsing)

It should display as correctly as expected (degrading gracefully) in following circumstances:

- JavaScript on/off
- Images on/off
- CSS on/off

The RC2 release achieves this for user profiles, all front-end management views, and started doing it for the registration and user edit views.

1.7.6.3 Separating logic from output

One sound design practice is to separate the business logic from output, so each method should first compute all variables to be displayed, and then output html code including those variables.

1.7.6.4 patTemplates

`patTemplates` are the output generation chosen by Joomla! 1.1 and 1.2. CB will follow this after mambo 4.5.0 support is dropped.

Plug-ins may explore this before, but core plug-in can't for backwards compatibility.

1.7.7 User profile

See **`getDisplayTab`** method.

1.7.8 User edit

See **`getEditTab`** and **`saveEditTab`** methods.

1.7.9 Registration

See **`getDisplayRegistration`** and **`saveRegistrationTab`** methods.

1.7.10 Fields Validation

Fields on registration and user edit can be validated by plug-ins. Minimum validation is with PHP in the plug-in. Optionally, JavaScript can also be used.

1.7.10.1 In PHP in the plugin

See user bots:

- **`onAfterUserUpdate`** example:

```
$_PLUGINS->registerFunction( 'onBeforeUserUpdate', 'pluginExampleBeforeSaveUser' );

/**
* Example store user method
* Method is called before user data is stored in the database
* @param     array         holds the core mambo user data
* @param     array         holds the community builder user data
* @param     boolean              true if a new user is stored
*/
function pluginExampleBeforeSaveUser(&$user,&$cbUser) {
       global $_POST, $_PLUGINS;

       // crash();

       if ($_POST['username'] == $_POST['password']) {
              $_PLUGINS->raiseError(0);
              $_PLUGINS->_setErrorMSG("Password must be different from username!");
       }
       return true;
}
```

- **`onAfterUserRegistrationSave`**

```
$_PLUGINS->registerFunction( 'onBeforeUserRegistration',
'pluginExampleBeforeUserRegistration' );


/**
* Example registration verify user method
* Method is called before user data is stored in the database
* @param     array         holds the core mambo user data
* @param     array         holds the community builder user data
* @param     boolean              false
*/
function pluginExampleBeforeUserRegistration(&$user,&$cbUser, $stored) {
       global $_POST, $_PLUGINS;
```

```
	if ($_POST['username'] == $_POST['password']) {
		$_PLUGINS->raiseError(0);
		$_PLUGINS->_setErrorMSG("Password has to be different from username!");
	}
	return true;
}
```

1.7.10.2 In JavaScript generated by the plug-in

CB API has a method for tabs to add user-side JavaScript validation (keep in mind that these do not operate when the user disabled his browser JavaScript, so PHP validation is also needed):

```
	/**
	* adds a validation JS code for the Edit Profile and Registration pages
	* @param string Javascript code ready for HTML output, with a tab \t at the begin
and a newline \n at the end.
	*/
	function _addValidationJS($js)
```

This function can be called from the plug-in from the **`getEditTab`** method (example):

```
	/**
	* Generates the HTML to display the user edit tab
	* @param object tab reflecting the tab database entry
	* @param object mosUser reflecting the user being displayed
	* @param int 1 for front-end, 2 for back-end
	* @returns mixed : either string HTML for tab content, or false if ErrorMSG
generated
	*/
	function getEditTab($tab,$user,$ui) {
		$params = $this->params;
		$exampleText	= $params->get('exampletext', 'Text Parameter not
set!');

		$ret = "<p>Hello ".$user->name." !</p>";
		$ret .= "<p>ParameterText: ".$exampleText."</p>";
		$ret .= "<p>Be carefull: don't set password same as username !</p>";
		$this->_addValidationJS( "\tif (me['username'].value ==
me['password'].value) {\n"
							."\t  errorMSG +=
\"".html_entity_decode("Password must differ from username!")."\\n\"\n"
							."\t
me['password'].style.background = \"red\";\n"
							."\t  iserror=1;\n"
							."\t}\n");
		// also see event: 'onBeforeUserUpdate', implemented here in function:
pluginExampleBeforeSaveUser().
		return $ret;
	}
```

and/or from the **`getDisplayRegistration`** methods as follows (example):

```
	/**
	* Generates the HTML to display the registration tab/area
	* @param object tab reflecting the tab database entry
```

```
	* @param object mosUser reflecting the user being displayed (here null)
	* @param int 1 for front-end, 2 for back-end
	* @return mixed : either string HTML for tab content, or false if ErrorMSG
generated
	*/
	function getDisplayRegistration($tab, $user, $ui) {
		$params = $this->params;
		$exampleText	= $params->get('exampletext', 'Text Parameter not
set!');

		$ret = "\t<tr>\n";
		$ret .= "\t\t<td class='titleCell'>"."Example plugin warnings:"."</td>\n";
		$ret .= "\t\t<td class='fieldCell'>";

		$ret .= "ParameterText: ".$exampleText;

		$ret .= "<p>Be carefull: don't set password same as username !</p>";
		$ret .= "</td>";
		$ret .= "\t</tr>\n";

		$this->_addValidationJS( "\tif (me['username'].value ==
me['password'].value) {\n"
							."\t  errorMSG +=
\"".html_entity_decode("Password must differ from username!")."\\n\"\n"
							."\t
me['password'].style.background = \"red\";\n"
							."\t  iserror=1;\n"
							."\t}\n");
		// also see event: 'onBeforeUserRegistration', implemented here in
function: pluginExampleBeforeUserRegistration().
		return $ret;
	}
```

1.8 Special user plug-ins

There are two special classes of user plug-ins:

- Private Messaging Systems – capable plug-ins
- Menu plug-ins (core)

Special plug-ins have a dot “.” in their name. Do not use a dot in normal plug-ins.

1.8.1 PMS

CB supports multiple PMS plug-ins. In case more than one is published, they will be used each to send the PMS.

PMS plug-ins must start with “**pms.**”

1.8.2 Menu

The Menu plug-in is core CB stuff. Please don’t touch this, except for the CSS-setable parts, which are part of the template plug-ins.

1.9 CB API

CB provides a User-Interface (UI) Application Programming Interface (API) for plug-ins.

Various displays and user interactions are provided in a generic way by CB. These are detailed in the next paragraphs.

1.9.1 Menus

CB provides natively for a CB menu (this can be switched off by the admin). The admin can choose in the backend between a menu bar and a menu list format.

CB RC2 provides for one level of submenu only, but in the future more levels will be supported.

As displaying tabs method may in the future not systematically be called, there is a separate plug-in method in tab objects used by CB to ask plug-ins to register menus:

```
	/**
	* Generates the menu and user status to display on the user profile by calling
back $this->addMenu
	* @param object tab reflecting the tab database entry
	* @param object mosUser reflecting the user being displayed
	* @param int 1 for front-end, 2 for back-end
	* @returns boolean : either true, or false if ErrorMSG generated
	*/
	function getMenuAndStatus($tab,$user,$ui) {
	}
```

Plug-ins can add their own menu items using following CB API:

```
	/**
* Registers a menu or status item to a particular menu position
* @param array a menu item like:
	// Test example:
	$mi = array(); $mi["_UE_MENU_CONNECTIONS"]["duplique"]=null;
	$this->addMenu( array(
		"position"	=> "menuBar" ,		// "menuBar", "menuList"
		"arrayPos"	=> $mi ,
		"caption"	=> _UE_MENU_MANAGEMYCONNECTIONS ,
		"url"		=> sefRelToAbs($ue_manageConnection_url) ,		// can
also be "<a ....>" or "javascript:void(0)" or ""
		"target"	=> "" ,		// e.g. "_blank"
		"img"		=> null ,	// e.g. "<img
src='plugins/user/myplugin/images/icon.gif' width='16' height='16' alt='' />"
		"alt"		=> null ,	// e.g. "text"
		"tooltip"	=> _UE_MENU_MANAGEMYCONNECTIONS_DESC ,
		"keystroke"	=> null ) ); // e.g. "P"
	// Test example: Member Since:
	$mi = array(); $mi["_UE_MENU_STATUS"]["_UE_MEMBERSINCE"]["dupl"]=null;
	$dat = cbFormatDate($user->registerDate);
	if (!$dat) $dat="?";
	$this->addMenu( array(
		"position"	=> "menuList" ,	// "menuBar", "menuList"
		"arrayPos"	=> $mi ,
		"caption"	=> $dat ,
		"url"		=> "" ,		// can also be "<a ....>" or
```

```
                                        // "javascript:void(0)" or ""
            "target"     => "" ,        // e.g. "_blank"
            "img"        => null ,      // e.g. "<img
src='plugins/user/myplugin/images/icon.gif' width='16' height='16' alt='' />"
            "alt"        => null ,      // e.g. "text"
            "tooltip"    => _UE_MEMBERSINCE_DESC ,
            "keystroke"  => null ) ); // e.g. "P"
*/
function addMenu( $menuItem )
```

Menu additions sort themselves on a first come first served base, but get hierarchically sorted, depending on main menu name, for a submenu.

As example, this adds a menu for PMS if it's not the user's profile which is displayed:

```
global $my;
if ($my->id!=$user->id && $my->id > 0) {
       $pmsurl=...
       $mi = array();
       $mi["_UE_MENU_MESSAGES"]["_UE_PM_USER"]=null;
       $this->menuBar->addObjectItem($mi,_UE_PM_USER,sefRelToAbs($pmsurl), "",
       "","", _UE_MENU_PM_USER_DESC,"");
}
```

It's important for SEF-compatibility to call **`sefRelToAbs`** for all internal URLs.

1.9.2 Status display

Status displays are similar to menus. They are setup using the same API:

Example:

```
if ($sbConfig['showranking'] && ($params->get('statRanking', '1') == 1) &&
($sbUserDetails !== false)) {
       $mi = array(); $mi["_UE_MENU_STATUS"][$params->get('statRankingText',
"_UE_FORUM_FORUMRANKING")]["_UE_FORUM_FORUMRANKING"]=null;
       $this->addMenu( array(
         "position" => "menuList" ,           // "menuBar", "menuList"
         "arrayPos" => $mi ,
         "caption"  => $sbUserDetails->msg_userrank.($params->get('statRankingImg',
'1')==1 ? $sbUserDetails->msg_userrankimg : "") ,
         "url"           => "" ,              // can also be "<a ....>" or
"javascript:void(0)" or ""
         "target"   => "" ,             // e.g. "_blank"
         "img"           => null ,      // e.g. "<img
src='plugins/user/myplugin/images/icon.gif' width='16' height='16' alt='' />"
         "alt"           => null ,      // e.g. "text"
         "tooltip"  => "") );
}
```

1.9.3 Forms

Forms in plug-ins must be coded using CB's forms support, so that a form in a plug-in returns to that plug-in and parameter names get coded for the plug-in.

For now, forms are supported only on user profiles.

Forms can be sent using either GET or POST HTTP methods.

The parameters are prefixed with the name of the plug-in so that multiple plug-ins do not collide on parameters with same name (e.g. "search" will become "**`simpleboardtabsearch`**").

All parameters, when passed through GET are correctly sefed.

A complete example for forms is the **JoomlaBoard/SimpleBoard** plug-in (user profile tab): it needs to be activated in the backend Joomlaboard/simpleboard tab parameters, to activate search and pagination and sorting functions.

1.9.3.1 URLs support

All URLs should be generated whenever possible in lower-case, but **should always be decoded also in lower-case**, as some SEF and URL rewriting tools allow the lowercasing of them.

URLs referring to the plug-in must always be generated using the CB API method:

```
	/**
	* gives the URL of a link with plugin parameters.
	* @param array of string with key name of parameters
	* @param string cb task to link to (default: userProfile)
	* @param boolean TRUE to call sefRelToAbs (default), FALSE to leave URL unsefed
	* @param array of string with keys of parameters to not include
	* @return string value of the parameter
	*/
	function _getAbsURLwithParam($paramArray, $task="userProfile", $sefed=true,
$excludeParamList=array())
```

Parameters received can be taken back using following CB API:

```
	/**
	* gets an ESCAPED and urldecoded request parameter for the plugin
	* @param string name of parameter in REQUEST URL
	* @param string default value of parameter in REQUEST URL if none found
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @return string value of the parameter (urldecode processed for international and
special chars) and ESCAPED! and ALLOW HTML!
	*            you need to call cbUnEscapeSQL to remove escapes, and htmlentities
before displaying.
	*/
	function _getReqParam($name, $def=null, $postfix="")
```

Please note that the parameters (e.g. $paramArray array) are escaped using addslashes() . So stripslashes() should be used if unescaping is needed, and htmlentities(stripslashes()) for outputing into html.

The developer is **strongly encouraged** to test his plugin with both ON and OFF of gpc_magic_quotes PHP setting.

Characters to test on URLs, fields and SQL accesses include at least:

- single-quote ‘
- and double-quote “,
- escaped quotes \’ \”
- as well as single and double backslash \ \\,
- as well as few accented characters,
- 1-byte UTF-8 characters like à è à ü ö ä,
- and multi-byte ones as well (e.g. ñ which is %C3%B1 in UTF-8),
- and also newline escapes \n.

One of our test-string is **`héllo’'world”"bàck\släsh\\ñ\n\’\'co\”\"`** . This kind of test-string should make a safe and uncorrupted journey through your plugin workflow...

These tests are not only for beauty and usability, **but also for SQL injection-safety**...

The following gives any GET/POST parameter name using the CB prefixing:

```
	/**
	* gets the name input parameter for search and other functions
	* @param string name of parameter of plugin
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @returns string value of the name input parameter
	*/
	function _getPagingParamName($name="search", $postfix="")
```

An example from the PMS plug-in tab:

```
if (isset($_POST[$this->_getPagingSearchName("sndnewmsg")]) && $_POST[$this-
>_getPagingSearchName("sndnewmsg")] == _UE_PM_SENDMESSAGE) {
	$sender = $this->_getReqParam("sender", null);
	$recip = $this->_getReqParam("recip", null);
	if ($sender && $recip) {
		$newsub = htmlentities($this->_getReqParam("newsub", null));
	//urldecode done in _getReqParam
```

```
            $newmsg = htmlentities($this->_getReqParam("newmsg", null));       //don't
allow html input on user profile!
```

The developer is strongly advised to protect his forms from SQL injection attacks. Same also from bots by implementing a protection code, as done in the PMS plug-in.

The following API allows generating a proper URL for forms and links:

```
      /**
      * gives the URL of a link with plugin parameters.
      * @param array of string with key name of parameters
      * @param string cb task to link to (default: userProfile)
      * @param boolean TRUE to call sefRelToAbs (default), FALSE to leave URL unsefed
      * @param array of string with keys of parameters to not include
      * @return string value of the parameter
      */
      function _getAbsURLwithParam($paramArray, $task="userProfile", $sefed=true,
$excludeParamList=array())
```

In Quick PMS tab for instance this is used as follows:

```
$base_url = $this->_getAbsURLwithParam(array());
$ret .= '<form method="post" action="'.$base_url.'">';
...
$ret .= '<input type="text" name="'.$this->_getPagingSearchName("newsub") .'"
size="'.$width.'" value="'.$newsub.'" class="inputbox" />
...
```

1.9.4 Generic list support

The CB API supports a list in the plug-in tab with paging, sorting and searching facilities.

A generic API method allows getting all relevant parameters for paging, sorting and searching, along with other plug-in specific parameters into an array of strings:

```
      /**
            * gets the paging limitstart, search and sortby parameters, as well as
additional parameters
      * @param array of string : keyed additional parameters as "Param-name" =>
"default-value"
      * @param mixed : array of string OR string : postfix for identifying multiple
pagings/search/sorts (optional)
      * @return array("limitstart" => current list-start value (default: null), "search"
=> search-string (default: null), "sortby" => sorting by, +additional parameters as keyed
in $additionalParams)
      */
      function _getPaging($additionalParams=array(), $postfixArray=null)
```

This is called for instance as follows:

```
	$pagingParams = $this->_getPaging();
```

These can then be used and set simply like this:

```
if ($pagingParams["limitstart"] === null) $pagingParams["limitstart"] = "0";
```

or when multiple lists are displayed on the plugin tab:

```
	$pagingParams = $this->_getPaging(array(), array(“posts_”,”subscriptions_”);
```

These can then be used and set simply like this:

```
if ($pagingParams["posts_limitstart"] === null) $pagingParams["posts_limitstart"] = "0";
```

The whole array can be initialized using this function:

```
	/**
	* sets the paging limitstart, search and sortby parameters, as well as additional
parameters
	* @param array of string : keyed additional parameters as "Param-name" => "value"
	* @param string search string
	* @param string sorting parameter added as &sortby=... if NOT NULL
	* @param array of string : keyed additional parameters as "Param-name" =>
"default-value"
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @return array("limitstart" => current list-start value (default: null), "search"
=> search-string (default: null), "sortby" => sorting by, +additional parameters as keyed
in $additionalParams)
	*/
	function
_setPaging($limitstart="0",$search=null,$sortBy=null,$additionalParams=array(),
$postfix="")
```

1.9.4.1 Pagination

Part of the list framework, CB API supports a powerful, yet simple, paging API using this function:

```
	/**
	* Writes the html links for pages inside tabs, eg, previous 1 2 3 ... x next
	* @param array: paging parameters. ['limitstart'] is the record number to start
dislpaying from will be ignored
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @param int Number of rows to display per page
	* @param int Total number of rows
	* @param string cb task to link to (default: userProfile)
	* @return string html text displaying paging
	*/
	function _writePaging($pagingParams, $postfix, $limit, $total,
$task="userProfile")
```

A pagination line is simply written as follows for Joomlaboard/SimpleBoard tab:

```
if ($pagingEnabled && ($postsNumber < $total)) {
	$return .= "<div style='width:95%;text-align:center;'>"
	.$this->_writePaging($pagingParams,”posts_”,$postsNumber,$total)
	."</div>";
}
```

The total number of post is found with a separate SQL query taking in account the search criteria, and maximum amount of posts shown.

1.9.4.2 Search

Together with pagination functions (but also independently), the CB API provides a simple search box to be drawn by a plug-in:

```
	/**
	* Writes the html search box as <form><div><input ....
	* @param array: paging parameters. ['limitstart'] is the record number to start
dislpaying from will be ignored
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @param string the class/style for the div
	* @param string the class/style for the input
	* @param string cb task to link to (default: userProfile)
	* @return string html text displaying a nice search box
	*/
	function
_writeSearchBox($pagingParams,$postfix="",$divClass="style=\"float:right;\"",$inputClass=
"class=\"inputbox\"",$task="userProfile")
```

This is implemented for instance as follows (example is Joomlaboard/SimpleBoard):

```
if ($searchEnabled) {
	$searchForm = $this->_writeSearchBox( $pagingParams, “posts_”,
$divClass="style='float:right;'", $inputClass="class='inputbox'");
} else {
	$pagingParams["search"] = "0";
}
```

1.9.4.3 Sorting

Together with pagination and search functions (but also independently), the CB API provides a simple sorting fields header to be drawn by a plug-in, calling for each header this method:

```
	/**
	* Writes the html links for sorting as headers
	* @param array: paging parameters. ['limitstart'] is the record number to start
dislpaying from will be ignored
	* @param string postfix for identifying multiple pagings/search/sorts (optional)
	* @param string sorting parameter added as &sortby=... if NOT NULL
```

```
        * @param string Name to display as column heading/hyperlink
        * @param boolean true if it is the default sorting field to not output sorting
        * @param string class/style html for the unselected sorting header
        * @param string class/style html for the selected sorting header
        * @param string cb task to link to (default: userProfile)
        * @return string html text displaying paging
        */
        function _writeSortByLink($pagingParams,$postfix,$sortBy,$sortName,
$defaultSort=false,$unselectedClass='class="cbSortHead"',
$selectedClass='class="cbSortHeadSelected"',$task="userProfile")
```

The table header is typically written as follows in Joomlaboard/SimpleBoard tab to be sortable:

```
$return .= "\n\t<tr class=\"sectiontableheader\">";
$return .= "<th>".$this->_writeSortByLink($pagingParams, "posts_", "date",
_UE_FORUMDATE,true)."</th>";
$return .= "<th>".$this->_writeSortByLink($pagingParams, "posts_", "subject",
_UE_FORUMSUBJECT)."</th>";
$return .= "<th>".$this->_writeSortByLink($pagingParams, "posts_", "category",
_UE_FORUMCATEGORY)."</th>";
$return .= "<th>".$this->_writeSortByLink($pagingParams, "posts_", "hits",
_UE_FORUMHITS)."</th>";
$return .= "</tr>";
```

The user clicking on such a link will generate a POST/GET HTTP request, giving a **`PLUGINTABNAMEsortby`** parameter, to be collected using **`_getPaging`** method.

Parameters for sorting are used as follows (example in Joomlaboard/SimpleBoard tab) to generate a "**`ORDER BY`**" SQL request:

```
            switch ($pagingParams['posts_sortby']) {
                  case "subject":
                        $order = "a.subject ASC, a.time DESC";
                        break;
                  case "category":
                        $order = "b.id ASC, a.time DESC";
                        break;
                  case "hits":
                        $order = "c.hits DESC, a.time DESC";
                        break;
                  case "date":
                  default:
                        $order = "a.time DESC";
                        break;
            }
```

1.9.4.4 Other form inputs

Other forms can be built using any combination of the above methods. A typical example is shown in the PMS plug-in Quick PMS tab.

1.9.5 User lists support

This is not supported yet. But will in the future.

1.9.6 User search support

This is not supported yet. But will in the future.

1.9.7 Language support for plug-ins

Plug-ins can use their own language files. The recommended way is to have a language directory with inside a **`default_language.php`** file, and language files for each supported language.

Uploading plug-in language files is not supported yet. We are waiting for Joomla! new language support before doing more work in this area.

1.10 Integrating with other components

To integrate well with other components, several ways are possible.

1.10.1 Talk with the others

A basic rule to integrate is to talk with the other dev team, and get their commitment if possible, so that future interoperability and backwards compatibility are maintained.

1.10.2 Prefered way: clean API

We recommend clean, abstract, encapsulated, object-oriented interfaces.

The **YaNC** interface is an example written by the CB authors in cooperation with the **YaNC** maintainer.

1.10.3 Other way: through SQL tables

An alternate way is through SQL tables accesses, if this structure is known to be critical by both teams (plug-in and the other components' team).

1.11 Conclusions

The Community Builder 1.0 RC2 Plugin API is a generic API made for helping integrating user-centric components .

It's a first release, which will be further developed, with backwards compatibility in mind. The authors can't guarrantee that this will be the case.

Your feed-backs, improvement proposals and plugin developments are highly welcome in the Community Builder Community.

Have fun developing plugins — as we had fun developing the new:

Community Builder Plugin Framework.

Of which you have the API Documentation in your hands.

– The Community Builder Team.