# Common Tasks & Objects

# Config

## Global Configuration

Accessing the global site configuration:

```
$jconfig = JFactory::getConfig();
```

Retrieving a value from the configuration:

```
echo $config->getValue('config.sitename');
```

## Component Configuration

Although rarer than accessing the global site configuration, sometimes it is necessary to access component-specific configurations. This can be done as follows:

```
$config = JComponentHelper::getParams('com_mycomponent');
```

Retrieving a value from the configuration:

```
echo $config->get('paramName');
```

# Users & Profiles

## Joomla User Object

### Current User

Accessing the Joomla! User object for the current user can be done as follows:

```
$juser = JFactory::getUser();
```

### Other Users

To access user info for anyone not the current user (accepts user ID number or username):

```
$otheruser = JUser::getInstance($id);
```

Any field from the user database table may then be accessed through the get('fieldname') method:

```
$id = $juser->get('id');
$name = $juser->get('name');
```

### Object Member Variables and Parameters

These are the relevant member variables automatically generated on a call to getUser():

- **id** - The unique, numerical user id. Use this when referencing the user record in other database tables.
- **name** - The name of the user. (e.g. Vint Cerf)
- **username** - The login/screen name of the user. (e.g. shmuffin1979)
- **email** - The email address of the user. (e.g. crashoverride@hackers.com)
- **password** - The encrypted version of the user's password
- **password_clear** - Set to the user's password only when it is being changed. Otherwise, remains blank.
- **usertype** - The role of the user within Joomla!. (Super Administrator, Editor, etc...)
- **gid** - Set to the user's group id, which corresponds to the usertype.
- **block** - Set to '1' when the user is set to 'blocked' in Joomla!.

- **registerDate** - Set to the date when the user was first registered.
- **lastvisitDate** - Set to the date the user last visited the site.
- **guest** - If the user is not logged in, this variable will be set to '1'. The other variables will be unset or default values.

In addition to the member variables (which are stored in the database in columns), there are parameters for the user that hold preferences. To get one of these parameters, call the getParam() member function of the user object, passing in the name of the parameter you want along with a default value in case it is blank.

```
$user = JFactory::getUser();
$language = $user->getParam('language', 'the default');

echo "<p>Your language is set to {$language}.</p>";
```

## HUBzero Extended Profile

HUBzero comes with extended user profiles that allow for considerably more information than the standard Joomla! User. Extended fields include information about disability, gender, race, bios, picture, etc. To access an extended profile, use the Profile object and load() method (accepts user ID number or username).

```
// Instantiate a new profile object
$profile = new HubzeroUserProfile();

// Load the profile
$profile->load( $id );
```

Alternatively, you may the getInstance() method. This can save on calls to the database as it stores any previously called profiles in memory.

```
// Load the profile
$profile = HubzeroUserProfile::getInstance($id);
```

Any field from the user database table may then be accessed through the get('fieldname') method:

```
$email = $profile->get('email');
```

```
$name = $profile->get('name');
```

Multi-option fields such as disability will return arrays.

## Checking if a User is logged in

Checking if a user is currently logged in can be done as follows:

```
// Call the user object
$juser = JFactory::getUser();

// If 'guest' is true, they are logged OUT
// If 'guest' is false, they are logged IN
if ($juser->get('guest'))
{
 return false;
}
```

## Group Memberships

Sometimes you may have a component or plugin that is meant to be accessed by members of a certain group or displays specific data based on membership in certain groups.

```
// Get the user object
$juser = JFactory::getUser();

// Get the groups of the current logged-in user
$user_groups = HubzeroUserHelper::getGroups($juser->get('id'));
```

The getGroups() method is passed a user ID and returns an array of objects if any group memberships are found. It will return false if no group memberships are found. Each object contains data specifying the user's status within the group, among other things.

```
Array (
    [0] => stdClass Object (
  [published] => 1
```

```
  [cn] => greatgroup
  [description] => A Great Group
  [registered] => 1
  [regconfirmed] => 1
  [manager] => 0
 )
    [1] => stdClass Object (
  [published] => 1
  [cn] => mygroup
  [description] => My Group
  [registered] => 1
  [regconfirmed] => 1
  [manager] => 1
 )
)
```

- **published** - 0 or 1, the published state of the group
- **cn** - string, the group alias
- **description** - string, the group title
- **registered** - 0 or 1, if the user applied for membership to this group (only 0 if the user was invited)
- **regconfirmed** - 0 or 1, if the user's membership application has been accepted (automatically 1 for invitees)
- **manager** - 0 or 1, if the user is a manager of this group

# Database

## Overview

HUBzero has been built with the ability to use several different kinds of SQL-database-systems and to run in a variety of environments with different table-prefixes. In addition to these functions, the class automatically creates the database connection. Besides instantiating the object, at a minimum, you only need 2 lines of code to get a result from the database in a variety of formats. Using the database layer ensures a maximum of compatibility and flexibility for your extension.

This tutorial looks at how to set and execute various queries.

## Preparing The Query

```
// Get a database object
$db = JFactory::getDBO();

$query = "SELECT * FROM #__example_table WHERE id = 999999;";
$db->setQuery($query);
```

First we instantiate the database object, then we prepare the query. You can use the normal SQL-syntax, the only thing you have to change is the table-prefix. To make this as flexible as possible, Joomla! uses a placeholder for the prefix, the "#__". In the next step, the $db->setQuery(), this string is replaced with the correct prefix.

Now, if we don't want to get information from the database, but insert a row into it, we need one more function. Every string-value in the SQL-syntax should be quoted. For example, MySQL uses back-ticks `` for names and single quotes '' for values. Joomla! has some functions to do this for us and to ensure code compatibility between different databases. We can pass the names to the function $db->nameQuote($name) and the values to the function $db->Quote($value).

A fully quoted query example is:

```
$query = "
  SELECT *
    FROM ".$db->nameQuote('#__example_table')."
    WHERE ".$db->nameQuote('id')." = ".$db->quote('999999').";
  ";
```

Whatever we want to do, we have to set the query with the $db->setQuery() function. Although you could write the query directly as a parameter for $db->setQuery(), it's commonly done by first saving it in a variable, normally $query, and then handing this variable over. This helps writing clean, readable code.

**setQuery($query)**

The setQuery($query) method sets up a database query for later execution either by the query() method or one of the Load result methods.

```
$db = JFactory::getDBO();
$query = "/* some valid sql string */";
$db->setQuery($query);
```

**Note**: The parameter $query must be a valid SQL string, it can either be added as a string parameter or as a variable; generally a variable is preferred as it leads to more legible code and can help in debugging.

setQuery() also takes three other parameters: $offset, $limit - both used in list pagination; and $prefix - an alternative table prefix. All three of these variables have default values set and can usually be ignored.

## Executing The Query

To execute the query, Joomla! provides several functions, which differ in their return value.

**Basic Query Execution**

The query() method is the the basic tool for executing sql queries on a database. In the CMS it is most often used for updating or administering the database and not seen often for loading data. This largely because the various load methods detailed on this page have the query step built in to them.

The syntax is very straightforward:

```
$db = JFactory::getDBO();
$query = "/* some valid sql string */";
$db->setQuery($query);
$result = $db->query();
```

**Note:** $db->query() returns an appropriate database resource if successful, or FALSE if not.

**Query Execution Information**

- getAffectedRows()
- explain()
- insertid()

**Insert Query Execution**

- insertObject()

## Query Results

The database class contains many methods for working with a query's result set.

**Single Value Result**

loadResult()

Use loadResult() when you expect just a single value back from your database query.

| id | name | email | username |
|---|---|---|---|
| 1 | John Smith | johnsmith@example.com | johnsmith |
| 2 | Magda Hellman | magda_h@example.com | magdah |
| 3 | Yvonne de Gaulle | ydg@example.com | ydegaulle |

This is often the result of a 'count' query to get a number of records:

```
$db = JFactory::getDBO();
$query = "
  SELECT COUNT(*)
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('name')." = ".$db->quote($value).";
  ";
$db->setQuery($query);
$count = $db->loadResult();
```

or where you are just looking for a single field from a single row of the table (or possibly a single field from the first row returned).

```
$db = JFactory::getDBO();
$query = "
  SELECT ".$db->nameQuote('field_name')."
    FROM ".$db->nameQuote('#__my_table')."
    WHERE ".$db->nameQuote('some_name')." = ".$db->quote($some_va
lue).";
  ";
$db->setQuery($query);
$result = $db->loadResult();
```

**Single Row Results**

Each of these results functions will return a single record from the database even though there may be several records that meet the criteria that you have set. To get more records you need to call the function again.

| id | name | email | username |
|---|---|---|---|
| 1 | John Smith | johnsmith@example.com | johnsmith |
| 2 | Magda Hellman | magda_h@example.com | magdah |
| 3 | Yvonne de Gaulle | ydg@example.com | ydegaulle |

loadRow()

loadRow() returns an indexed array from a single record in the table:

```
...
$db->setQuery($query);
$row = $db->loadRow();
print_r($row);
```

will give:

```
Array (
 [0] => 1
 [1] => John Smith
```

```
 [2] => johnsmith@example.com
 [3] => johnsmith
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

**Note:**

1. The array indices are numeric starting from zero.
2. Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

loadAssoc()

loadAssoc() returns an associated array from a single record in the table:

```
$db->setQuery($query);
$row = $db->loadAssoc();
print_r($row);
```

will give:

```
Array (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row['name'] // e.g. $row['name']
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

loadObject()

loadObject() returns a PHP object from a single record in the table:

```
$db->setQuery($query);
$result = $db->loadObject();
print_r($result);
```

will give:

```
stdClass Object (
 [id] => 1
 [name] => John Smith
 [email] => johnsmith@example.com
 [username] => johnsmith
)
```

You can access the individual values by using:

```
$row->index // e.g. $row->email
```

Whilst you can repeat the call to get further rows, one of the functions that returns multiple rows might be more useful

**Single Column Results**

Each of these results functions will return a single column from the database.

| id | name | email | username |
|---|---|---|---|
| 1 | John Smith | johnsmith@example.com | johnsmith |
| 2 | Magda Hellman | magda_h@example.com | magdah |
| 3 | Yvonne de Gaulle | ydg@example.com | ydegaulle |

loadResultArray()

loadResultArray() returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadResultArray();
print_r($column);
```

will give:

```
Array (
 [0] => John Smith
 [1] => Magda Hellman
 [2] => Yvonne de Gaulle
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

**Note:**

1. The array indices are numeric starting from zero.
2. loadResultArray() is equivalent to loadResultArray(0)

loadResultArray($index)

loadResultArray($index) returns an indexed array from a single column in the table:

```
$query = "
  SELECT name, email, username
    FROM . . . ";

$db->setQuery($query);
$column= $db->loadResultArray(1);
print_r($column);
```

will give:

```
Array (
 [0] => johnsmith@example.com
 [1] => magda_h@example.com
 [2] => ydg@example.com
)
```

You can access the individual values by using:

```
$column['index'] // e.g. $column['2']
```

loadResultArray($index) allows you to iterate through a series of columns in the results

```
$db->setQuery($query);
for ( $i = 0; $i loadResultArray($i);
  print_r($column);
}
```

will give:

```
Array ( [0] => John Smith [1] => Magda Hellman [2] => Yvonne de G
aulle )
Array ( [0] => johnsmith@example.com [1] => magda_h@example.com [
2] => ydg@example.com )
Array ( [0] => johnsmith [1] => magdah [2] => ydegaulle )
```

The array indices are numeric starting from zero.

### Multi-Row Results

Each of these results functions will return multiple records from the database.

| id | name | email | username |
|---|---|---|---|
| 1 | John Smith | johnsmith@example.com | johnsmith |
| 2 | Magda Hellman | magda_h@example.com | magdah |
| 3 | Yvonne de Gaulle | ydg@example.com | ydegaulle |

loadRowList()

loadRowList() returns an indexed array of indexed arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadRowList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [0] => 1 [1] => John Smith [2] => johnsmith@examp
le.com [3] => johnsmith )
 [1] => Array ( [0] => 2 [1] => Magda Hellman [2] => magda_h@exam
ple.com [3] => magdah )
 [2] => Array ( [0] => 3 [1] => Yvonne de Gaulle [2] => ydg@examp
le.com [3] => ydegaulle )
)
```

You can access the individual values by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['index'] // e.g. $row['2']['3']
```

The array indices are numeric starting from zero.

loadAssocList()

loadAssocList() returns an indexed array of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList();
print_r($row);
```

will give:

```
Array (
 [0] => Array ( [id] => 1 [name] => John Smith [email] => johnsmi
th@example.com [username] => johnsmith )
 [1] => Array ( [id] => 2 [name] => Magda Hellman [email] => magd
a_h@example.com [username] => magdah )
 [2] => Array ( [id] => 3 [name] => Yvonne de Gaulle [email] => y
dg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']['column_name'] // e.g. $row['2']['email']
```

loadAssocList($key)

loadAssocList($key) returns an associated array - indexed on 'key' - of associated arrays from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadAssocList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => Array ( [id] => 1 [name] => John Smith [email] =>
 johnsmith@example.com [username] => johnsmith )
 [magdah] => Array ( [id] => 2 [name] => Magda Hellman [email] =>
 magda_h@example.com [username] => magdah )
 [ydegaulle] => Array ( [id] => 3 [name] => Yvonne de Gaulle [ema
il] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']['column_name'] // e.g. $row['johnsmith']['email
']
```

**Note:** Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

loadObjectList()

loadObjectList() returns an indexed array of PHP objects from the table records returned by the query:

```
$db->setQuery($query);
$result = $db->loadObjectList();
print_r($result);
```

will give:

```
Array (
 [0] => stdClass Object ( [id] => 1 [name] => John Smith
     [email] => johnsmith@example.com [username] => johnsmith )
 [1] => stdClass Object ( [id] => 2 [name] => Magda Hellman
     [email] => magda_h@example.com [username] => magdah )
 [2] => stdClass Object ( [id] => 3 [name] => Yvonne de Gaulle
     [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['index'] // e.g. $row['2']
```

and you can access the individual values by using:

```
$row['index']->name // e.g. $row['2']->email
```

loadObjectList('key')

loadObjectList('key') returns an associated array - indexed on 'key' - of objects from the table records returned by the query:

```
$db->setQuery($query);
$row = $db->loadObjectList('username');
print_r($row);
```

will give:

```
Array (
 [johnsmith] => stdClass Object ( [id] => 1 [name] => John Smith
     [email] => johnsmith@example.com [username] => johnsmith )
 [magdah] => stdClass Object ( [id] => 2 [name] => Magda Hellman
     [email] => magda_h@example.com [username] => magdah )
 [ydegaulle] => stdClass Object ( [id] => 3 [name] => Yvonne de G
aulle
     [email] => ydg@example.com [username] => ydegaulle )
)
```

You can access the individual rows by using:

```
$row['key_value'] // e.g. $row['johnsmith']
```

and you can access the individual values by using:

```
$row['key_value']->column_name // e.g. $row['johnsmith']->email
```

**Note:** Key must be a valid column name from the table; it does not have to be an Index or a Primary Key. But if it does not have a unique value you may not be able to retrieve results reliably.

### Misc Result Set Methods

getNumRows()

getNumRows() will return the number of result rows found by the last query and waiting to be read. To get a result from getNumRows() you have to run it after the query and before you have retrieved any results.

```
$db->setQuery($query);
$db->query();
$num_rows = $db->getNumRows();
print_r($num_rows);
$result = $db->loadRowList();
```

will give:

3

**Note:** if you run getNumRows() after loadRowList() - or any other retrieval method - you may get a PHP Warning.

# JTable

## Overview

The JTable class is an implementation of the Active Record design pattern. It is used throughout Joomla! for creating, reading, updating, and deleting records in the database table.

When properly extended, JTable gives you all of the basic functions you need for managing and retrieving records in a database table. Member functions take care of the rest when you add member variables, the table name, and the key column.

## Writing an extension of JTable

To use JTable, create an extension of the class. In this example, we have a database table containing recipes.

```
<?php

defined('_JEXEC') or die();

class KitchenTableRecipes extends JTable
{
 public function __construct(&$db)
 {
  parent::__construct( '#__recipes', 'id', $db );
 }
}
```

When naming your class extension, the convention is to prefix it with 'Table', then follow with a CamelCased version of the table's name. Unlike previous versions, it is NOT necessary to list all of the member variables of your class that match the column names in the database. The table columns are defined from the database schema.

Finally, create a constructor for the class that accepts a reference to the current database instance. This will call the parent constructor which needs the name of the table, the name of the primary key column, and the database instance. The name of the table uses #__ instead of jos_, as the administrator can pick any table prefix desired during Joomla! installation.

If you were using this class as a part of a component called 'Kitchen', you would place this code in the file /administrator/components/com_kitchen/tables/recipes.php.

## Using a JTable class extension

Once the table class is in place, you can use it in any Joomla! extension. To include the file, place this line in your extension's source code (use com_nameofyourcomponent in place of com_recipes):

```
JTable::addIncludePath(JPATH_ADMINISTRATOR.DS.'components'.DS.'com_rec
ipes'.DS.'tables');
```

To get an instance of the object, use this code:

```
$row =& JTable::getInstance('recipes', 'Table');
```

Notice that the lowercase version of the suffix of your class name is used as the first parameter, with the prefix 'Table' as the second. Also, the getInstance() member function of JTable returns the object by reference instead of value.

In a model class (extends JModel) you can also use:

```
$row =& $this->getTable('recipes');
```

Notice that if you have not used the standard naming convention, you can supply the class prefix as the optional second parameter.

**Create/Update**

In a typical situation, you will have an HTML form submitted by the user which PHP will interpret for you as an associative array. The JRequest class in Joomla! has functions ready to assist with retrieving this data safely. Use JRequest::get('post') to retrieve all of the elements in the HTTP POST request as a sanitized array.

Once you have this array, you can pass it into the bind() method of JTable. Doing this will match the associated items of the array with member variables of the class. In the following example, the array is retrieved from JRequest::get('post') and immediately passed into bind().

```
if (!$row->bind( JRequest::get( 'post' ) ))
{
```

```
 return JError::raiseWarning( 500, $row->getError() );
}
```

If bind() fails, you want to stop the application and explain the failure before your extension attempts to send the data. The raiseWarning() function of JError allows you to stop Joomla!, while the getError() function returns the error message stored in the JTable object.

When binding succeeds and your object is ready, call the store() function. Again, if something goes wrong, stop the application and explain why.

```
if (!$row->store())
{
 JError::raiseError(500, $row->getError() );
}
```

**Note:**

- If any member variables of your JTable object are null when store() is called, they are ignored by default. This allows you to update specific columns of your table, while leaving the others untouched. If you wish to override this behavior to ensure that all columns have a value, pass true into store().
- The JTable::bind() and JRequest::get() functions do not enforce data types. If you need a column to be a specific type (for instance, integer), you need to add this logic to your code before calling store().

**Read**

To load a specific row of the database with JTable, pass the key into the load() member function.

```
$row->load( $id );
```

This relies on the key column you specified in the second parameter of parent::__construct()

when you extended JTable.

**Delete**

Like read(), delete() allows you to destroy a specific row in the table based on the key specified earlier.

```
$row->delete( $id );
```

If you want to delete multiple rows at once, you will need to write the query manually.

# Tags

## Overview

The Tag class is a set of tools for adding, removing, editing, and displaying tags on objects. It is used throughout HUB installations for adding tags to such things as resources, users, events, and more.

When properly extended, Tags gives you all of the basic functions you need for managing and retrieving tag records in the database table.

All tags are stored within a single table called "#__tags". The information that associates a particular tag to a specific user, event or group, is stored in a table called "#__tags_object". Storing the association data separate from the tag itself allows for a tag to be represented once but be connected to multiple items. If that tag is ever changed for any reason, it will be represented the same regardless of what object it is attached to.

The #__tags_object table stores, among other things, such data as the unique ID of the tag, the unique ID of the object being tagged, and what component (or, potentially, table) that object belongs to.

| id | objectid | tagid | tbl |
|---|---|---|---|
| 1 | 77 | 6 | resources |
| 2 | 77 | 6 | events |

Here we have two entries that both link to a tag with an ID of "6" and both with object IDs of "77". One entry is a resource and the other is an event. The "tbl" field is the most important distinguishing factor; This allows us to have multiple objects with the same object ID, linking to the same tag but not create a conflict.

## Writing an extension of Tags

To use Tags, create an extension of the class. In this example, we're adding tags to our "com_example" objects.

```
<?php
// Check to ensure this file is included in Joomla!
defined('_JEXEC') or die( 'Restricted access' );

require_once(JPATH_ROOT . DS . 'components' . DS . 'com_tags ' . DS .
'models' . DS . 'cloud.php');

class ExampleModelTags extends TagsModelCloud
{
```

```
 /**
  * Object type, used for linking objects (such as resources) to tags
  *
  * @var string
  */
 protected $_scope = 'example';
}
```

When naming your class extension, the convention is to have a CamelCased version of the component's name suffixed with "Tags".

Finally, assign to $_scope the name to be used to uniquely identify tag data as belonging to your specific component.

## Using a Tag class extension

Once the class is created and in place, it can be included and instantiated

**Create/Update**

```
// Retrieve posted tags (comma delimitated string)
$tags = JRequest::getVar('tags', '');

// Instantiate the tagging class
$cloud = new ExamplesModelTags($object_id);

// Tag the object
// $user_id will typically be the current logged in user or JFactory::
getUser()->get('id);
$cloud->setTags($tags, $user_id);
```

This method is the same for both adding tags to a previously untagged object and updating the existing list of tags on an object.

**Read**

render('string')

Returns a string of comma-separated tags.

```
// Instantiate the tagging class
$et = new ExamplesModelTags($object_id);

// Get a tag cloud (HTML List)
echo $et->render('string');
```

will give:

```
My Tag, Your Tag, Their Tag
```

render()

Returns a tag cloud, derived of a an HTML list. Each tag is linked to the Tags component and comprises one list item. A CSS class of "tags" on the list allows for styling.

```
// Instantiate the tagging class
$et = new ExamplesModelTags($object_id);

// Get a tag cloud (HTML List)
echo $et->render();
```

will give:

```
<ol class="tags">
 <li><a href="/tags/mytag">My Tag</a></li>
 <li><a href="/tags/yourtag">Your Tag</a></li>
 <li><a href="/tags/theirtag">Their Tag</a></li>
</ol>
```

render('array')

Returns an array of associative arrays.

```
// Instantiate the tagging class
$et = new ExamplesModelTags($object_id);

// Get a tag cloud (HTML List)
$tags = $et->render('array');
print_r($tags);
```

will give:

```
Array (
 [0] => Array (
   [tag] => 'mytag'
   [raw_tag] => 'My Tag'
   [tagger_id] => 32
   [admin] => 0
 )
 [1] => Array (
   [tag] => 'yourtag'
   [raw_tag] => 'Your Tag'
   [tagger_id] => 32
   [admin] => 0
 )
 [2] => Array (
   [tag] => 'theirtag'
   [raw_tag] => 'Their Tag'
   [tagger_id] => 32
   [admin] => 0
  )
)
```

## Using the Tag Editor plugin

To make adding tags and editing a list of existing tags in a form, HUBzero offers a Tag Editor plugin. To use the plugin in a view, do the following:

```
// Load the plugin
JPluginHelper::importPlugin( 'hubzero' );
$dispatcher = JDispatcher::getInstance();

// Trigger the event
```

```
$tf = $dispatcher->trigger( 'onGetMultiEntry', array(array('tags','tag
s','actags','',$tags)) );

// Output
if (count($tf) > 0) {
 echo $tf[0];
} else {
 echo '<input type="text" name="tags" value="'. $tags .'" />';
}
```

The first parameter passed ('tags') tells the plugin that you wish to display a tags autocompleter. The nest parameter is the name of the input field. The third is the ID of the input field. The fourth is any CSS class you wish to assign to the input. The $tags variable here must be a string of comma-separated tags.

# Retrieving GET & POST data

## JRequest 'getVar' method

To retrieve GET/POST request data, use the getVar method of the JRequest class (JRequest::getVar()).

### Retrieving Data

If you have a form variable named 'address', you would want to use this code to get it:

```
$address = JRequest::getVar('address');
```

Unless other parameters are set, all HTML and trailing whitespace will be filtered out.

### The DEFAULT Parameter

If you want to specify a default value in the event that 'address' is not in the request or is unset, use this code:

```
$address = JRequest::getVar('address', 'Address is empty');
echo $address;  // Address is empty
```

### The SOURCE Parameter

Frequently, you will expect your variable to be found in a specific portion of the HTTP request (POST, GET, etc...). If this is the case, you should specify which portion; this will slightly increase your extension's security. If you expect 'address' to only be in POST, use this code to enforce that:

```
$address = JRequest::getVar('address', 'default value goes here', 'pos
t');
```

### The VARIABLE TYPE Parameter

The fourth parameter of getVar() can be used to specify certain filters to force validation of specific value types for the variable.

```
$address = JRequest::getVar(
    'address',
    'default value goes here',
    'post',
    'variable type'
);
```

Here is a list of types you can validate:

- INT
- INTEGER
- FLOAT
- DOUBLE
- BOOL
- BOOLEAN
- WORD
- ALNUM
- CMD
- BASE64
- STRING
- ARRAY
- PATH
- USERNAME

**The FILTER MASK Parameter**

Finally, there are some mask constants you can pass in as the fifth parameter that allow you to bypass portions of the filtering:

```
$address = JRequest::getVar(
    'address',
    'default value goes here',
    'post',
    'validation type',
    'mask type'
);
```

- JREQUEST_NOTRIM - prevents trimming of whitespace
- JREQUEST_ALLOWRAW - bypasses filtering
- JREQUEST_ALLOWHTML - allows most HTML. If this is not passed in, HTML is stripped out by default.

# Constants

## System Constants

These constants are defined for use in the CMS and extensions:

| | |
|---|---|
| DS | Directory separator. "/" |
| JPATH_ADMINISTRATOR | The path to the administrator folder. |
| JPATH_BASE | The path to the installed Joomla! site. |
| JPATH_CACHE | The path to the cache folder. |
| JPATH_COMPONENT | The path to the current component being executed. |
| JPATH_CONFIGURATION | The path to folder containing the configuration.php file. |
| JPATH_INSTALLATION | The path to the installation folder. |
| JPATH_LIBRARIES | The path to the libraries folder. |
| JPATH_PLUGINS | The path to the plugins folder. |
| JPATH_ROOT | The path to the installed Joomla! site. |
| JPATH_SITE | The path to the installed Joomla! site. |
| JPATH_THEMES | The path to the templates folder. |
| JPATH_XMLRPC | The path to the XML-RPC Web service folder. |

**Note:** These paths are the absolute paths of these locations within the file system, NOT the path you'd use in a URL.

For URL paths, try using JURI::base.

# Scheduled Tasks

## Plugins

A set of tasks can be registered with the Cron component by making a plugin. Each plugin must respond to the "onCronEvents" trigger. The response from that trigger is an object (stdClass) that returns the plugin's name and an array of callable tasks (event triggers).

### Registering Tasks

Plugins should be placed within the cron plugins folder:

```
/myhub
    /plugins
        /cron
```

Here is an example of a cron plugin that registers a set of "mytasks" events.

```
/**
 * Cron plugin for my tasks
 */
class plgCronMytasks extends JPlugin
{
 /**
  * Return a list of events
  *
  * @return     array
  */
 public function onCronEvents()
 {
  // Load the plugin's language file
  $this->loadLanguage();

  // Create the return object
  $obj = new stdClass();

  // Assign the plugin's name
  $obj->plugin = $this->_name;

  // Build the list of callable events
  $obj->events = array(
   array(
```

```
    'name'   => 'doSomething',  // The name of your task
    'label'  => JText::_('PLG_CRON_MYTASKS_DOSOMETHING'),  // Nice lab
el
    'params' => ''  // Name of the params group to load (optional)
   )
  );

  // Return the data
  return $obj;
 }
}
```

As shown in the previous example, each event consist of an array containing three keys: name, label, and params.

name

> The plugin must implement a method with the same name as whatever is specified for the name key and the names should match *exactly*. That is, if a name of 'onJumpUpAndDown' is specified, then the plugin **must** have a method of onJumpUpAndDown();.

label

> This is a nice, human readable name for the event trigger. It should be a language key with an associated string in the plugin's language file.

params

> This is an optional value for specifying a params group (Joomla 1.5) or fieldset (Joomla 1.6+) containing parameters associated with the specific plugin event. This allows for multiple cron jobs calling the same event but with varying values. An example of this can be found in the support tickets cron plugin where the event sendTicketsReminder has a specified params group of 'ticketreminder'. Changing those params would allow, for instance, a job that sends ticket reminders one a month for all open tickets and a ticket reminder once a week for all open and *status: critical* tickets.
>
> A snippet from the support plugin, specifying the list of available tasks:
>
> ```
> /**
> ```

```
 * Cron plugin for support tickets
 */
class plgCronSupport extends JPlugin
{
 /**
  * Return a list of events
  *
  * @return     array
  */
 public function onCronEvents()
 {
  $this->loadLanguage();

  $obj = new stdClass();
  $obj->plugin = $this->_name;

  $obj->events = array(
   array(
    'name'   => 'onClosePending',
    'label'  => JText::_('PLG_CRON_SUPPORT_CLOSE_PENDING'),
    'params' => 'ticketpending'
   ),
   array(
    'name'   => 'sendTicketsReminder',
    'label'  =>  JText::_('PLG_CRON_SUPPORT_EMAIL_REMINDER'),
    'params' => 'ticketreminder'
   )
  );

  return $obj;
 }
 ...
}
```

In the support plugin's manifest:

```
...
<fieldset group="ticketreminder">
 <field name="support_ticketreminder_severity" type="list" defaul
t="all" label="Tickets with severity" description="Ticket severit
y to message users about.">
  <option value="all">All</option>
  <option value="critical,major">High</option>
```

```
  <option value="normal">Normal</option>
  <option value="minor,trivial">Low</option>
 </field>
 <field name="support_ticketreminder_group" type="text" menu="hid
e" label="For users in group" default="" description="Only users
within the group specified will be messaged." />
</fieldset>
...
```

### Running Tasks

All tasks are run as standard plugin events. Tasks should return a boolean of true upon completion.

See the managers documentation on how to create and schedule jobs.