The Basics

Requests

Retrieving Input

You may access all input from the global Request instance.

Retrieving Data

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

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

The DEFAULT Parameter

In the event that 'address' is not in the request or is unset, you may specify a default value as the second argument:

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

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 = Request::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

Cookies

Cookie values may be accessed in the same manner as user input with the one change of method name from input to cookie.

```
$value = Request::cookie('name');
```

Other

The Request class provides many methods for examining the HTTP request for your application.

Request Method

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

Current URL

```
$current = Request::current();
```

Root URL for the application

This differs from base() in that it will always return the root URI for the application, regardless of the sub-directory the request was called from.

```
$base = Request::root();
```

Base URL for the application

By default, this method will return the full base path for the current request, including scheme, host, and port. To return the path only, pass true.

```
// Gets http://myhub.org/
$base = Request::base();

// Gets /
$base = Request::base(true);
```

User IP Address

```
$ip = Request::ip();
```

Responses

Overview

The CMS application contains a HubzeroHttpResponse instance that all extension output (component, template, etc) is attached to. The response instance allows for customizing the response's HTTP status code, content, and headers. The response instance inherits from the SymfonyComponentHttpFoundationResponse class, providing a variety of methods for building HTTP responses.

Note: For a full list of available Response methods, check out the Symfony API documentation.

Response Object

The creation, setting of content and headers, and sending of the response is handled automatically by the application. But, in some cases, it is beneficial to access and manipulate the response as needed. The response instance may be accessed via its Facade, available in all application types (site, admin, muse, etc), or by retrieving the object directly from the application container.

Facade:

```
Response::header('Content-Type', 'application/json');

echo json_encode($data);
```

Direct access:

```
$response = App::get('response');
$response->header('Content-Type', 'application/json');

echo json_encode($data);
```

Attaching Headers

Most response methods are chainable, allowing for the fluent building of responses. For example, you may use the header method to add a series of headers to the response before sending it back to the user:

```
$response->header('Content-Type', $type)
            ->header('X-Header-One', 'Header Value')
            ->header('X-Header-Two', 'Header Value');
```

Setting Content

To set the content of the response, use the setContent method.Note that the value passed must be of type string.

```
$response->setContent($output);
```

Sending a Response

This will send the set content and headers to the client.

```
$response->send();
```

Redirects

One may also generate redirects by calling the redirect() method on the App. That method accepts three arguments: 1) a URL to predict to, 2) an optional message to display, and 3) an optional message type.

```
App::redirect(
    Route::url('index.php?option=com_support')
);
```

Note that a redirect call is immediate meaning no code immediately after the redirect will be executed.

```
App::redirect(
    Route::url('index.php?option=com_support')
);

// This will not be executed
```

```
die('Hello');
```

The redirect() method is instantiating a new instance of a a HubzeroHttpRedirectResponse class which is a specialized, extended instance of the HubzeroHttpResponse class. If need be, the class can be directly instantiated:

```
$redirect = new HubzeroHttpRedirectResponse($url);
$redirect->setRequest(App::get('request'));
$redirect->send();
```

Config

Global Configuration

Global (site) configuration values can be directly accessed via the get() method of the global Config facade:

```
$value = Config::get('sitename');
```

Alternatively, one may grab the entire configuration object from the application:

```
$config = App::get('config');

$value = $config->get('sitename');
```

Component Configuration

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

```
$config = Component::params('com_mycomponent');
```

Retrieving a value from the configuration:

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

Plugin Configuration

A fairly common task is accessing plugin-specific configurations. This can be done by accessing the public params property on all plugins.

```
class plgSystemExample extends Plugin
{
```

```
    public function onDoSomething()
    {
        $config = $this->params;
    }
}
```

If the configuration for a specific plugin is needed from elsewhere (e.g., another extension), this can be done through the global Plugin facade. Call the params() method, passing in the type of plugin (e.g., authentication) and the name (e.g., facebook) of the plugin:

```
$config = Plugin::params('authentication', 'facebook');
```

Retrieving a value from the configuration:

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

Module Configuration

Module-specific configurations can be accessed via the public params property on any modules that extend the HubzeroModuleModule class.

```
class Example extends Module
{
    public function display()
    {
        $config = $this->params;
    }
}
```

Retrieving a value from the configuration:

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

Languages

Overview

To create your own language file it is necessary that you use the exact contents of the default language file and translate the contents of the define statements. Language files are INI files which are readable by standard text editors and are set up as key/value pairs.

Working With INI Files

INI files have several restrictions. If a value in the ini file contains any non-alphanumeric characters it needs to be enclosed in double-quotes ("). There are also reserved words which must not be used as keys for ini files. These include: NULL, yes, no, TRUE, and FALSE. Values NULL, no and FALSE results in "", yes and TRUE results in 1. Characters {}|&~![()" must not be used anywhere in the key and have a special meaning in the value. Do not use them as it will produce unexpected behavior.

Files are named after their internationally defined standard abbreviation and may include a locale suffix, written as language_REGION. Both the language and region parts are abbreviated to alphabetic, ASCII characters. A user from the USA would expect the language English and the region USA, yielding the locale identifier "en_US". However, a user from the UK may expect a region of UK, yielding "en_UK".

Setup

As previously mentioned, language files are setup as key/value pairs. A key is used within the widget's view and the translator retrieves the associated string for the given language. The following code is an extract from a typical widget language file.

```
; Module - Example (en_US)
MOD_EXAMPLE_HERE_IS_LINE_ONE = "Here is line one"
MOD_EXAMPLE_HERE_IS_LINE_TWO = "Here is line two"
MOD_EXAMPLE_MYLINE = "My Line"
```

Translation keys can be upper or lowercase or a mix of the two and may contain underscores but no spaces. HUBzero convention is to have keys all uppercase with words separated by underscores, following a pattern of {ExtensionPrefix}_{WidgetName}_{TextName} for naming.

Table 1: Translation key prefixes for the various extensionsExtension Type Key Prefix
Component

Extension Type	Key Prefix
Module	
Plugin	
Template	

Adhering to this naming convention is not required but is strongly recommended as it can help avoid potential translation collisions. Since a component can potentially have modules loaded into it, the possibility of a widget and a module having the same translation key arises. To illustrate this, we have the following example of a component named mycomponent that loads a module named mymodule.

The language files for both:

```
; mymodule en_US.ini
MYLINE = "Your Line"
```

```
; mycomponent en_US.ini
MYLINE = "My Line"
```

The layout files for both:

```
<!-- mymodule layout -->
<strong><php echo Lang::txt('MYLINE'); ?></strong>
```

```
<!-- mycomponent layout -->
<div>
 <!-- Load the module -->
 <php echo Module::render('mymodule'); ?>
 <!-- Translate some component text -->
 <php echo Lang::txt('MYLINE'); ?>
</div>
```

Outputs:

```
<div>
 <!-- Load the module -->
 <strong>Your Line</strong>
 <!-- Translate some component text -->
 Your Line
</div>
```

Since the module is loaded in the component view, i.e. *after* the component's translation files have been loaded, the module's instance of MYLINE overwrites the existing MYLINE from the component. Thus, the view outputs "Your Line" for the component translation instead of the expected "My Line". Using the HUBzero naming convention of adding component and module name prefixes helps avoid such errors:

The language files for both:

```
; mymodule en-US.ini
MOD_MYMODULE_MYLINE = "Your Line"
```

```
; mycomponent en-US.ini
COM_MYCOMPONENT_MYLINE = "My Line"
```

The view files for both:

```
<!-- mymodule view -->
<strong><php echo Lang::txt('MOD_MYMODULE_MYLINE'); ?></strong>
```

```
<!-- mycomponent view -->
<div>
 <!-- Load the module -->
 <php echo $this->Widgets()->renderWidget('mywidget'); ?>
 <!-- Translate some module text -->
 <php echo Lang::txt('COM_MYCOMPONENT_MYLINE'); ?>
```

```
</div>
```

Outputs:

```
<div>
 <!-- Load the widget -->
 <strong>Your Line</strong>
 <!-- Translate some module text -->
 My Line
</div>
```

To Further avoid potential collisions as it is possible to have a component and module with the same name, module translation keys are prefixed with MOD_ and component translation keys with COM_.

Translating Text

A translate helper (Lang) is available in all views and the appropriate language file for an extension is preloaded when the extension is instantiated. This is all done automatically and requires no extra work on the developer's part to load and parse translations.

Below is an example of accessing the translate helper:

```
<p><?php echo Lang::txt("MOD_EXAMPLE_MY_LINE"); ?></p>
```

Strings or keys not found in the current translation file will output as is.

Overrides

In order to accommodate rewording across hub deployments, we provide a mechanism for overriding language constants. Web developers are highly encouraged to make use of language constants throughout extension development as language overrides are generally simpler and easier to maintain than view overrides when wording simply needs to be updated.

Administrator Interface

1. On the administrative backend, navigate to the Language Manager through the Extensions menu.

2. Click on the Overrides sub-menu.

3. To add a new Language Override, click the “New” button.

To replace a constant, you can search for a value using the right-hand “*Search Text You Want to Change*” functionality. You will need to provide the wording which you wish to replace. The utility will provide you with the Language Constant to be used in the *Create a New Override* interface.

4. Fill out the *Create A New Override* section appropriately.

a. Unless you have another language installed, leave the Language field to its default setting.

5. The Location field refers to which CMS application this override applies. Possible values are: *Site*, *Administrator*, *Cli*, or *Api*.

6. The File field refers to the location of the language override. The format of this would be <webroot>/<hubname>/app/bootstrap/<application>/language/overrides/<language>-override.ini where values for <application> are: *site, administrator, cli,* or *api.*

File Structure

While the administrative backend offers a user-friendly mechanism to add language overrides, system administrators or developers can directly modify the language file.

Language Overrides are application specific. The HUBzero CMS is comprised of four different application types:

- Site - the public-facing interface of the CMS.
- Administrator - the administrative interface of the CMS.
- Cli - The command line interface (muse is a member of this type).
- Api - The REST-ful-esque API for the HUBzero CMS.

When each application is initiated, a set of parameters are “bootstrapped” to the application. These parameters can be defined in the app/bootstrap directory. One of these parameters is

language.

To change a Language Constant on the public-facing Site:

1. Search for the string which you would like to replace.
    1. Using the command `grep` is useful:
        a. Example: grep -i -H -r -n "one more thing" ./
            i. Returns ./core/components/com_projects/site/language/en-GB/en-GB.com_projects.ini:193:COM_PROJECTS_SETUP_BEFORE_COMPLETE="Just one more thing before you get started... (our lawyers made us do it)"
            ii. The constant is: COM_PROJECTS_SETUP_BEFORE_COMPLETE
2. Edit the file: /www/<webroot>/app/bootstrap/site/language/overrides/en-GB.override.ini using your favorite text editor.
3. Insert the constant and language override into this file.
4. Save the file.

Users & Profiles

User Object

Current User

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

```
$user = User::getRoot();
```

Other Users

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

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

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

```
$id = $user->get('id');
$name = $user->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. Jane Doe)
- **username** - The login/screen name of the user. (e.g. janedoe2015)
- **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 the CMS. (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'.

- **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.

```
$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 by calling the isGuest() method on the global User facade:

```
// If true, they are logged OUT
// If false, they are logged IN
if (User::isGuest())
{
 return false;
}
```

Alternatively, one may need to work with a user object more directly:

```
// Get the root object behind the facade
$user = User::getRoot();

// ... Do some processing on the $user

if ($user->isGuest())
{
 return false;
}
```

The isGuest() method checks the guest property on the user object. This property can be directly accessed, if desired:

```
// If true, they are logged OUT
// If false, they are logged IN
if (User::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 groups of the current logged-in user
$user_groups = HubzeroUserHelper::getGroups(User::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

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
namespace ComponentsExampleModels;

use ComponentsTagsModelsCloud;

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

class Tags extends Cloud
```

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

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 = Request::getVar('tags', '');

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

// Tag the object
// $user_id will typically be the current logged in user or User::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
$et = new ComponentsExamplesModelsTags($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
$et = new ComponentsExamplesModelsTags($object_id);

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

will give:

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

render('array')

Returns an array of associative arrays.

```
// Instantiate the tagging class
$et = new ComponentsExamplesModeslTags($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:

```
// Trigger the event
$tf = Event::trigger( 'hubzer.onGetMultiEntry', array(array('tags','ta
gs','actags','',$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.

Debugging

Debug Mode

To turn on Debug mode:

- Login to the administration area e.g. http:/YOURSITE/administrator/
- At the top under the **Site** menu click **Global Configuration**.
- Click the **System** tab.
- Under the **Debug Settings** section change **Debug System** to *Yes*.
- Click the **Save** button.

Debug mode will output a list of all queries that were executed in order to generate the page. This will also turn on a stack trace output for error and warning pages. Hubzero components will also have PHP error reporting turned on, allowing one to see any PHP errors that may be present.

Note: Turning on debugging mode for production (live) sites is strongly discouraged and it is recommended to be avoided if at all possible.

Restricting who sees debug output

Since debug mode can contain potentially sensitive, it is **strongly** recommended that access to debug output is restricted to the administrator or super administrator user access levels and/or a defined list of users.

To restrict:

- Login to the administration area e.g. http:/YOURSITE/administrator/
- At the top under the **Extensions** menu click **Plugin Manager**.
- Select **System** from the "Select Type" drop-down.
- Find the debug plugin, typically titled "System - Debug", and click to edit.
- Under the **Parameters** section select the **Allowed Groups** and/or enter a comma-seprated list of usernames into the Allows Users box.
- Click the **Save** button.

Inspecting Variables

Hubzero provides the utility class HubzeroUtilityDebug for dumping variables.

dump()
: This will perform a print_r on the variable passed, wrapping the output in HTML <pre> tags.

ddie()

Short for "dump and die", this will perform a print_r on the variable passed, wrapping the output in HTML <pre> tags and die();.

dlog()

This method allows developers to dump variables to the debug toolbar, allowing data to be inspected without interrupting the flow or process of the code or output. **Note:** This feature requires the global Debug mode and system debug plugin to be enabled.

Example

```
$myvar = array(
    'one' => 'foo',
    'two' => 'bar',
);

HubzeroUtilityDebug::dump($myvar);
```

Illegal variable ... passed to script.

One encounters the following error:

Illegal variable _files or _env or _get or _post or _cookie or _server or _session or globals passed to script.

This error is generated when the key of a key-value pair is numeric in one of the following variables: _files or _env or _get or _post or _cookie or _server or _session or globals. An example of this would be $_POST[5] = 'value'. This is most often generated by having form elements with numeric values as names. For example:

```
<input type="text" name="5" />
```

As the error indicates, this is not allowed. Element names must include at least one non-numeric character. Examples:

```
<input type="text" name="n5" />

<input type="text" name="n_5" />
```

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:

```
/app
.. /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 HubzeroPluginPlugin
{
 /**
  * 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'  => Lang::txt('PLG_CRON_MYTASKS_DOSOMETHING'),  // Nice la
bel
    '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 HubzeroPluginPlugin
{
 /**
  * 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'  => Lang::txt('PLG_CRON_SUPPORT_CLOSE_PENDING'),
    'params' => 'ticketpending'
   ),
   array(
    'name'   => 'sendTicketsReminder',
    'label'  =>  Lang::txt('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.

Dates

The Date class

To help working with dates the framework provides the Hubzero\Utility\Date class. Since that can be a bit much to type every time when instantiating a new instance, a global Date facade can be used instead. To get a Date object that represents the current date and time do the following:

```
$now = Date::getRoot();
```

The first thing to note is that we do NOT use the =& assignment operator. The static getRoot() method does not return references to globally accessible instances of Date. This means each time Date is used it is retrieving a new object.

It is also possible to specify the date and time we want the Date object to represent. A likely source for this would be a DATETIME field extracted from the database.

```
$created = Date::of($row->created);
```

Since Hubzero\Utility\Date extends PHP's DateTime object, the method used to parse date and time values is relatively robust. Formats other than the MySQL DATETIME representation of YYYY-MM-DD HH:MM:SS can be used. The table below describes the acceptable formats.

Format	Example	Notes
Timestamp	1254497100	Seconds since the Unix Epoch
RFC 2822	Fri, 2 Oct 2009 15:25:00 +0000	Name of day and UTC offset is optional. Date does not support all of the obsolete RFC 822 time zone identifiers. Date support numeric time zone identifiers, UT, GMT, and military time zones.
RFC 3339	2009-10-02 T15:25:00+00:00	RFC 3339 time zone offset can be expressed numerically or as the time zone alpha identifier Z (Zulu, UTC+0). RFC 3339 *is also known as* ISO 8601.
US English date format	2 October 2009	For more information about US

Format	Example	Notes
		English date formats refer to http://php.net/strtotime.

In the table above both the RFC 2822 and RFC 3339 examples include a UTC offset in the value. In the examples the offset is 0. Date always internally represents the date and time in the UTC+0 time zone. Had the offsets in the examples been non zero values, and had we used these to create new Date objects, we would have found that the date and time within the Date objects would have been adjusted to represent a timezone of UTC+0.

Outputting Dates

The Date class includes five handy methods for retrieving formatted date and time strings. The most versatile of these methods is format(). This method allows for explicitly defining the format in which the date and time are to be described. The format can be defined in the same way as when using the PHP function strftime().

```
$string = $myDate->format('%Y-%m-%d');
```

The remaining four methods to retrieve formatted date and time strings are used to extract specific representations of the date and time. These representations are RFC 2822 (successor to RFC 822), ISO 8601 (also known as RFC 3339), Unix timestamp, and SQL (determined by the specific database connector used).

```
// D, d M Y H:i:s
// Tuesday, 06 October 2009 12:54:37+0000
$rfc2822 = $myDate->toRFC822();
```

```
// Y-m-dTH:i:s
// 2009-10-06T12:54:37Z
$iso8601 = $myDate->toISO8601();
```

```
// Unix timestamp
// 1254833677
```

```
$unix = $myDate->toUnix();



// The format is determined by the databse being used. The following e
xample is for MySQL.
// Y-m-d H:i:s
// 2009-10-06 12:54:37
$mysql = $myDate->toSql();
```

Outputting dates in different time zones

As mentioned above, Date internally stores dates and times in the UTC time zone. In conjunction with that it is good practice to store dates and times in the database in the UTC time zone. For end users however, this is not necessarily easy to read. To aid with this, Date can output dates and times in different time zones.

In addition to the date and time that a Date object represents, a Date object can also record a time zone in which to output formatted dates. This value can be set and retrieved with the setTimezone and getTimezone methods, respectively.

The timezone being discussed in this section is separate from the timezone specified when *creating* a new Date object.