Components Back-end

Version	HUBzero 1.0
Online	http://hubzero.org/documentation/0.9.0/webdevs/components

Components follow the Model-View-Controller (MVC) design pattern. This pattern separates the data gathering (Model), presentation (View) and user interaction (Controller) activities of a module. Such separation allows for expanding or revising properties and methods of one section without requiring additional changes to the other sections.

In its barest state, no database entry or other setup is required to "install" a component. Simply placing the component into the /administrator/components directory will make it available for use. However, if a component requires the installation of database tables or configuration (detailed in a config.xml file), then an administrator must install the component using one of the installation options in the administrative back-end.

In the example, all component related files and sub-directories are split between the administrator components and front-end components. In both cases, the files are contained within directories titled "com_example". Some directories and files are optional but, for this example, we've included the most common setup.

The file structure in the administrative portion of the component is fairly simple compared to the front-end counterpart. This is because, as of Joomla! 1.5.x, MVC has not been fully implemented for administrative components.

```
/{YourHub}
    /administrator
        /components
            /com_hello
                admin.controller.php
                admin.hello.html.php
                admin.hello.php
                hello.class.php
                toolbar.hello.html.php
                toolbar.hello.php
                hello.xml
```

Joomla! is always accessed through a single point of entry: administrator/index.php for the Administrator Application. The application will then load the required component, based on the value of 'option' in the URL or in the POST data. For our component, the URL would be:

http://yourhub.org/administrator/index.php?option=com_hello

This will load our main file, which can be seen as the single point of entry for our component: /administrator/components/com_hello/admin.hello.php.

1. First, we turn on PHP error reporting. This can aid greatly in development.

2. Next, we set the access levels for the component.

3. Then we check if the current user has the appropriate authority to access this component.

4. Then we include some needed classes and instantiate our controller.

5. After the controller is created, we instruct the controller to execute the task, as defined in the URL: index.php?option=com_hello&task=some

6. The controller might decide to redirect the page, usually after a task like 'save' has been completed. This last statement takes care of the actual redirection.

The main entry point (hello.php) essentially passes control to the controller, which handles performing the task that was specified in the request.

/components/com_hello/hello.php

```php
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

error_reporting(E_ALL);
@ini_set('display_errors','1');

// Set access levels
$jacl =& JFactory::getACL();
$jacl->addACL( $option, 'admin', 'users', 'super administrator' );
$jacl->addACL( $option, 'manage', 'users', 'administrator' );
$jacl->addACL( $option, 'manage', 'users', 'manager' );

// Authorization check
$user = & JFactory::getUser();
if (!$user->authorize( 'com_hello', 'manage' )) {
    $mainframe->redirect( 'index.php', JText::_('ALERTNOTAUTH') );
}

// Include some needed code
require_once( JPATH_COMPONENT.DS.'hello.class.php' );
require_once( JPATH_COMPONENT.DS.'admin.hello.html.php' );
require_once( JPATH_COMPONENT.DS.'admin.controller.php' );

// Create the controller
$controller = new HelloController( );
$controller->execute();
$controller->redirect();
```

The base controller class for the front-end component is named {ComponentName}Controller. So, the controller class for com_hello would be “HelloController”

The two methods of note in this example are the execute() and entries() methods.

In the execute method, we're explicitly declaring a list of available tasks and what those tasks execute via the switch statement.

In our entries method, we're simply echoing “Hello.” to the page.

/administrator/components/com_hello/admin.controller.php

```
class HelloController extends JObject
{
	...
	public function execute()
	{
		$this->_task = JRequest::getString('task', '');

		switch ($this->_task)
		{
			default: $this->entries(); break;
		}
	}
	...
	protected function entries()
	{
		echo ‘Hello.’;
	}
}
```

Components Back-end

Version	HUBzero 1.0
Online	http://hubzero.org/documentation/0.9.0/webdevs/components

We could write individual functions to take care of the queries necessary to add, update, and delete data. Fortunately, the Joomla! team has ready done this for you. The JTable class provides functions for performing modify data from single table in the database. We just extend the class.

1. Here we're creating variables for each field in the database table.
2. In the constructor, we set the name of the table and the name of the primary key.
3. Finally, we have a simple method for retrieving all the rows in our database table. We're returning the data as an array of objects. Joomla! has a variety of other methods available that can return a single row, return an array of arrays, return a single result (such as a record count), etc.

/administrator/components/com_hello/hello.class.php

```
class HelloEntry extends JTable
{
    var $id = NULL;  // @var int(11) Primary key                    1
    var $greeting = NULL;  // @var varchar(255)

    public function __construct( &$db )
    {
        parent::__construct( '#__hellos', 'id', $db );            2
    }

    public function getEntries()
    {
        $q = "SELECT * FROM $this->_tbl ORDER BY greeting ASC";
3
        $this->_db->setQuery( $q );
        return $this->_db->loadObjectList();
    }
}
```

In our entries method, we're simply echoing "Hello." to the page. Not very useful as is. What we want to do is display a list of all the entries in our database table.

1. First we'll instantiate a new "HelloEntry" object. The HelloEntry class is a representation of our database table and contains all the SQL and methods used to access data in the table.

2. Then we simply call the getEntries() method of the object. This will return an array of objects, each one representing a single row of our database table. If there are no rows, it will return an empty array.

3. Finally, we pass the data to a method of our HelloHtml class. This will output the data as a table of entries.

/administrator/components/com_hello/admin.controller.php

```
protected function entries()
{
    // Get the database object
    $database =& JFactory::getDBO();

    // Instantiate a HelloEntry object
    $obj = new HelloEntry( $database );         1

    // Get the records
    $rows = $obj->getEntries();             2

    // Output HTML
    HelloHtml::entries( $rows );             3
}
```

In our entries method, we're simply looping through all the rows returned from the database and echoing the data in a table. Later we'll add the ability to select an entry to edit or delete it. We'll also look into adding pagination.

But, before we can edit or page through any data, we need to have data to work with! Next, we'll create methods for adding, editing, saving, and deleting entries.

/administrator/components/com_hello/admin.hello.html.php

```
class HelloHtml
{
    public function entries($rows)
    {
?>
        <form action="index.php" method="post" name="adminForm">
            <table class="adminlist">
                <tbody>
<?php
        $k = 0;
        for ($i=0, $n=count( $rows ); $i < $n; $i++)
        {
            $row =& $rows[$i];
?>
                    <tr class="<?php echo "row$k"; ?>">
                        <td><?php echo $row->greeting; ?></td>
                    </tr>
<?php
            $k = 1 - $k;
        }
?>
                </tbody>
            </table>
        </form>
<?php
    }
}
```

We want to add the ability to create, edit, and save entries so we can start populating our database table.

1. First, we'll add the tasks to our switch statement in the execute() method.

2. Then we'll add the methods to our class. They won't do anything for the moment. We'll get to that later.

/components/com_hello/controller.php

```
class HelloController extends JObject
{
    ...
    public function execute()
    {
        $this->_task = JRequest::getString('task', '');

        switch ($this->_task)
        {
            case 'add': $this->edit(); break;
            case 'edit': $this->edit(); break;
            case 'save': $this->save(); break;            1
            case 'delete': $this->delete(); break;
            case 'cancel': $this->cancel(); break;
            default: $this->entries(); break;
        }
    }
    ...
    protected function edit()
    {
        // Blank for now
    }                                                     2
    protected function save()
    {
        // Blank for now
    }
    ...
```

Throughout the back end, all the core components implement toolbars with similar buttons for saving, deleting, editing, and publishing items. You can use these buttons in your component so that frequent administrators will have a seamless experience.

To start, create the toolbar.reviews.html.php file in the administrator/components/com_hello folder and enter in the example code:

Files containing output codes are usually organized into classes, like the code here with HelloToolbar. Each member function here represents a different toolbar. The class JToolBarHelper contains functions that generate all the HTML necessary to build toolbars.

Joomla! allows you to override any button with your own task and label, passing them as the first and second parameters respectively.

/administrator/components/com_hello/toolbar.hello.html.php

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

class HelloToolbar
{
    public function _EDIT()
    {
        JToolBarHelper::save();
        JToolBarHelper::apply();
        JToolBarHelper::cancel();
    }

    public function _DEFAULT()
    {
        JToolBarHelper::title( JText::_( 'Hello' ),'generic.png' );
        JToolBarHelper::deleteList();
        JToolBarHelper::editList();
        JToolBarHelper::addNew();
    }
}
```

The toolbars are now defined, but you need to add some code that will decide which one to display. In the back end, Joomla! automatically loads the file beginning with the component name and ending in .hello.php in the upper right-hand portion of the screen. Add the following code into toolbar.hello.php in the administrator/components/com_hello folder:

/administrator/components/com_hello/toolbar.hello.php

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

require_once( JApplicationHelper::getPath( 'toolbar_html' ) );

switch ($task)
{
    case 'edit':
    case 'add': HelloToolbar::_EDIT(); break;
    default: HelloToolbar::_DEFAULT(); break;
}
```

The line containing require_once(...) uses the getPath() member function of the JApplicationHelper class. The call to getPath() allows you to call up the toolbar.reviews.html.php file without committing to a component name. Later, even if you change the name to 'Restaurants' and also change the filenames, this line of code will still load the output code for the toolbar with no modification.

After toolbar.hello.php loads the file with the output class, you need to decide which toolbar should be displayed. The request variable $task is automatically registered in global scope by Joomla! and is used to direct the logic flow within the component.

1. First we're retrieving any ID passed to the method. We're expecting an array (ex: index.php?option=com_hello&id[]=3) and need to reduce it down to one item.

2. We instantiate a HelloEntry and use the load() method provided by extending Joomla's JTable class. We're passing it the $id variable. If $id is 0, it'll simply create a new object with no data. If the $id isn't 0, it will attempt to pull a row from the database with the matching primary key and populate the class variables. This means we can use the same method for creating and editing entries!

3. Finally, we output a simple form for inputting/editing data.

/administrator/components/com_hello/admin.controller.php

```
class HelloController extends JObject
{
	...
	protected function edit()
	{
		// Incoming
		$ids = JRequest::getVar( 'id', array(0) );          1
		if (is_array($ids) && !empty($ids)) {
			$id = $ids[0];
		}

		// Get the database object
		$database =& JFactory::getDBO();

		// Load the article
		$row = new HelloEntry( $database );                 2
		$row->load( $id );

		// Output HTML                                      3
		HelloHtml::edit( $row, $this->_option );
	}
}
```

Now we add a method to admin.hello.html to display our editing form.

NOTE: Some of the code has been removed in the example presented here for the sake of page room. Please see the corresponding example downloads available at http://hubzero.org

/administrator/components/com_hello/admin.hello.html.php

```
	public function edit( $row, $option )
	{
		?>
		<script type="text/javascript">
		function submitbutton(pressbutton)
		{
			if (pressbutton == 'cancel') {
				submitform( pressbutton );
				return;
			}
			...
		}
		</script>
		<form action="index.php" method="post" name="adminForm" 
class="editform">
			<fieldset class="adminform">
				...
			</fieldset>
			<input type="hidden" name="id" value="<?php echo $row-
>id; ?>" />
			<input type="hidden" name="option" value="<?php echo 
$option; ?>" />
			<input type="hidden" name="task" value="save" />
		</form>
		<?php
	}
```

We need to be able to save the submitted data. So we'll add to the empty save() method we created earlier.

1. First, we instantiate a new HelloEntry object and then we bind the incoming POST data to it. This method, provided by extending the JTable class, will retrieve any data from POST that has a matching name to one of the public variables in our HelloEntry class. So, $_POST['greeting'] will map to HelloEntry's "greeting" variable. If the bind fails in some way, we notify the user with a javascript alert and back the page up to the edit form.

2. Next we call a check() method for data validation. This is an empty method by default when extending JTable and any validation you want performed must be set up in the HelloEntry class.

3. Next, we store() the data. If the primary key (id, in this case) is 0 or NULL, JTable will perform an INSERT statement with all the supplied data. If the primary key exist, JTable will perform an UPDATE.

4. Finally, barring no errors, we set the private _redirect variable to return to the main listing of entries and the private _message variable to give us a success message.

/administrator/components/com_hello/admin.controller.php

```
protected function save()
{
    $database =& JFactory::getDBO();

    // Initiate extended database class
    $row = new HelloEntry( $database );
    if (!$row->bind( $_POST )) {
        echo HelloHtml::alert( $row->getError() );          1
        exit();
    }

    // Check content
    if (!$row->check()) {                                    2
        echo HelloHtml::alert( $row->getError() );
        exit();
    }

    // Store new content
    if (!$row->store()) {                                    3
        echo HelloHtml::alert( $row->getError() );
        exit();
    }

    // Set the redirect
    $this->_redirect = 'index.php?option='.$this->_option;  4
    $this->_message = JText::_('Entry saved!');
}
```

We may want to delete entries every so often. Thanks to methods that come with the JTable we're extending, this is quite easy:

1. First, we retrieve any incoming IDs of entries we want to delete.
2. Next, we instantiate a new HelloEntry object. We only need one object to delete multiple entries.
3. Next, we loop through all the $ids and delete() the entry with the $id we passed to it. If no $id is passed, the method will attempt to use $entry->id.
4. Finally, barring no errors, we set the private _redirect variable to return to the main listing of entries and the private _message variable to give us a success message.

/administrator/components/com_hello/admin.controller.php

```
protected function delete()
{
    // Incoming
    $ids = JRequest::getVar( 'id', array() );                          1

    if (!empty($ids)) {
        // Get the database object
        $database =& JFactory::getDBO();

        // Create a category object
        $entry = new HelloEntry( $database );                          2

        // Loop through all the IDs
        foreach ($ids as $id)
        {
            // Delete the entry
            $entry->delete( $id );                                     3
        }
    }

    // Set the redirect
    $this->_redirect = 'index.php?option='.$this->_option;             4
    $this->_message = JText::_('Entries deleted!');
}
```

Sometimes we click on the wrong entry or change our minds about saving data. Earlier, we added a “Cancel” button to our toolbar and an empty method to our controller. Now need to fill out the cancel() method.

All we do is add a line for setting the private _redirect variable to take us back to the main listing of entries. That’s it!

/administrator/components/com_hello/admin.controller.php

```
protected function cancel()
{
    // Set the redirect
    $this->_redirect  = 'index.php?option='.$this->_option;
}
```

As the data grows in our table, it can be come unwieldy to view all records at once. Especially if your table has thousands of records! So, we want to add paging.

1. First we retrieve any paging data that may have been passed. “limit” and “limitstart” are standard variables used throughout Joomla!. If no data was passed, we start at zero and limit the records to 25.

2. We instantiate our HelloEntry object as before.

3. We get a count of ALL records. This is used by the pagination object to determine how many “pages” of data there will be and to display the appropriate number of links.

4. Then we call the getEntries() method, passing it the pagination filters. That method will then construct a query with the appropriate limit and start point for returning data.

5. Then we create the pagination object. We pass it the total, start point, and limit.

6. Finally, we output some HTML.

/administrator/components/com_hello/admin.controller.php

```
	protected function entries()
	{
		// Get paging variables
		$filters = array();
		$filters['limit'] = JRequest::getInt('limit', 25);                1
		$filters['start'] = JRequest::getInt('limitstart', 0);

		// Get the database object
		$database =& JFactory::getDBO();

		// Instantiate a HelloEntry object
		$obj = new HelloEntry( $database );                              2

		$total = $obj->getEntryCount();  // Get record count             3

		$rows = $obj->getEntries( $filters );  // Get the records        4

		// Initiate paging
		jimport('joomla.html.pagination');
		$pageNav = new JPagination( $total, $filters['start'], $fil-     5
ters['limit'] );

		// Output HTML
		HelloHtml::entries( $rows, $pageNav, $this->_option );           6
	}
```

Here we've added the getEntryCount() method to our HelloEntry class. The query simply returns a count of all records in the database table.

/administrator/components/com_hello/hello.class.php

```
    public function getEntryCount()
    {
        $q = "SELECT count(*) FROM $this->_tbl";

        $this->_db->setQuery( $q );
        return $this->_db->loadResult();
    }
```

Finally, we update our HelloHtml::entries() method to add paging, link entries to an edit form, and add checkboxes so we can select multiple entries for deletion.

NOTE: Some of the code has been removed in the example presented here for the sake of page room. Please see the corresponding example downloads available at http://hubzero.org

/administrator/components/com_hello/admin.hello.html.php

```
...
<tfoot>
  <tr>
    <td colspan="3">
        <?php echo $pageNav->getListFooter(); ?>
    </td>
  </tr>
</tfoot>
...
```