# Components Front-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 /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_hello". 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 exactly the same as in the front side. Note that the view, models, controllers etc. of the front and admin parts are completely separated, and have nothing to do with each other - the front part and the admin part can be thought of as two different components! A view in the /administrator/components/com_hello folder may have a counterpart with the same name in the /components/com_hello folder, yet the two views have nothing in common but their name.

```
/{YourHub}
    /administrator
        /components
            /com_hello
    ...
    /components
        /com_hello
            /models
                foo.php
            /views
                /index
                    /tmpl
                        default.php
                        default.xml
            controller.php
            hello.php
            router.php
```

Joomla! is always accessed through a single point of entry: index.php for the Site Application or 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/index.php?option=com_hello&view=hello

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

1. First, we added some lines that check if site debugging is turned on. If so, we turn on PHP error reporting. This can aid greatly in development.

2. Next, we added the jimport call to include the component JView class.

3. Then we include our controller and create a new instance of it.

4. 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

5. 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
// No direct access
 defined( '_JEXEC' ) or die( 'Restricted access' );

// Check if debugging is turned on
// Turn on error reporting so we can more clearly see our PHP bugs
$config = JFactory::getConfig();
if ($config->getValue('config.debug')) {
      error_reporting(E_ALL);
      @ini_set('display_errors','1');
}

// Include the JView class
jimport('joomla.application.component.view');

// Require the base controller
require_once( JPATH_COMPONENT.DS.'controller.php' );

// Create the controller
$controller = new HelloController( );

// Perform the Request task
$controller->execute();

// Redirect if set by the controller
$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 hello() 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 hello method, we're instantiating a new view, assigning it some data, and then displaying the output. When we instantiate the view, we’re explicitly telling it the name of the view we wish to load. We can also pass it a specific layout to load, if needed. Otherwise, the layout loaded will be named “default.” This tells the code to look for the view in /components/com_hello/views/hello/tmpl/default.php

/components/com_hello/controller.php

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

		switch ($this->_task)
		{
			default: $this->hello(); break;
		}
	}
	...
	protected function hello()
	{
		// Instantiate a new view
		$view = new JView( array('name'=>'hello') );

		// Pass the view any data it may need
		$view->greeting = 'Hello, World!';

		// Output the HTML
		$view->display();
	}
}
```

Views are written in PHP and HTML and have a .php file extension. View scripts are placed in /com_{componentname}/views/, where they are further categorized by the /{viewname}/tmpl. Within these subdirectories, you will then find and create view scripts that correspond to each controller action exposed; in the "hello" case, we have the view script default.php.

The variables assigned in the controller can be accessed from the template using $this->{propertyname} (see the template code for an example).

Our template is very simple: we only want to display the greeting that was passed in from the hello() method of our controller - this file is: views/hello/tmpl/default.php

Note: For a information on view overrides see the “Component Overrides” section of the documentation: http://hubzero.org/documentation/0.9.0/webdevs/templates.overrides

/components/com_hello/views/hello/tmpl/default.php

```
<?php
// No direct access
 defined('_JEXEC') or die('Restricted access'); ?>
<h1><?php echo $this->greeting; ?></h1>
```

1. The first step is to obtain a reference to a database object. Since Joomla! uses the database for its normal operation, a database connection already exists; therefore, it is not necessary to create your own.

2. Now that we have obtained a reference to the database object, we can retrieve our data. We do this in two steps:

   A. Store our query in the database object.

   B. Load the result. The $db->loadResult() method will execute the stored database query and return the first field of the first row of the result.

   In the example, we're passing the result directly to our view to be used for displaying.

/components/com_hello/controller.php

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

        switch ($this->_task)
        {
            case 'world': $this->world(); break;
            default: $this->hello(); break;
        }
    }
    ...
    protected function world()
    {
        $view = new JView( array('name'=>'world') );

        $db =& JFactory::getDBO();                                  (1)
        $db->setQuery( 'SELECT count(*) FROM #__components' );      (2A)
        $view->component_count = $db->loadResult();                 (2B)

        // Output the HTML
        $view->display();
    }
}
```

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

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

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

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');
```

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', 'Address is empty', 'post');
```

/components/com_hello/controller.php

```
class HelloController extends JObject
{
    ...
    protected function world()
    {
        $view = new JView( array('name'=>'world') );

        $view->multiplier = JRequest::getInt( 'multiplier', 5 );

        $db =& JFactory::getDBO();
        $db->setQuery( 'SELECT count(*) FROM #__components' );
        $view->component_count = $db->loadResult();

        // Output the HTML
        $view->display();
    }
}
```

1

/components/com_hello/views/world/tmpl/default.php

```
<?php
// No direct access
defined('_JEXEC') or die('Restricted access'); ?>
<h1><?php echo ($this->component_count*$this->multiplier); ?></h1>
```

2

1. Here we're retrieving a variable named "multiplier" and have the default value set to 5. We're using getInt instead of getVar to ensure any data retrieved is an integer.
2. Now we're outputting the multiplication of our component count by the multiplier we retrieved.

Joomla! has its very own way of dealing with errors. Our gateway to this is the static JError class. There are three error levels defined by Joomla! Error, Warning, and Notice.

This example explains how to raise a JError of the level error. We should use this type of JError when we encounter a fatal problem, that is, when the problem cannot be overcome and execution needs to stop. The nice thing about this approach is that although the JError will be fatal, the termination of the script will be graceful and will provide the user with an understandable error page.

To raise a fatal JError, we use the static JError::raiseError() method. This method requires two parameters, an error code and a message.

Other available methods:

JError::raiseNotice(); - A notice is informational. It should only indicate trivial problems that are expected to occur periodically, but do not affect the flow of the script execution.

JError::raiseWarning(); - These are specifically for use when an error occurs that is not fatal, but alters the actions taken by the script.

/components/com_hello/controller.php

```
class HelloController extends JObject
{
    ...
    protected function world()
    {
        $view = new JView( array('name'=>'world') );

        $view->multiplier = JRequest::getInt( 'multiplier', 5 );

        if ($view->multiplier < 2) {
            JError::raiseError(
                500,
                'The multiplier must be greater than 1'
            );
            return;
        }

        $db =& JFactory::getDBO();
        $db->setQuery( 'SELECT count(*) FROM #__components' );
        $view->component_count = $db->loadResult();

        // Output the HTML
        $view->display();
    }
}
```