Components

Overview

The largest and most complex of the extension types, a component is in fact a separate application. You can think of a component as something that has its own functionality, its own database tables and its own presentation. So if you install a component, you add an application to your website. Examples of components are a forum, a blog, a community system, a photo gallery, etc. You could think of all of these as being a separate application. Everyone of these would make perfectly sense as a stand-alone system. A component will be shown in the main part of your website and only one component will be shown. A menu is then in fact nothing more then a switch between different components.

Throughout these articles, we will be using {ComponentName} to represent the name of a component that is variable, meaning the actual component name is chosen by the developer. Notice also that case is important. {componentname} will refer to the lowercase version of {ComponentName}, eg. "CamelCasedController" -> "camelcasedcontroller". Similarly, {ViewName} and {viewname}, {ModelName} and {modelname}, {ControllerName} and {controllername}.

We also strongly encourage developers to take a look at Joomla!'s documentation.

Directory Structures & Files

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 the config.xml file), then an administrator must install the component using one of the installation options in the administrative back-end.

Note: Components not installed via one of the installation options or without a database entry in the #__components table will not appear in the administrative list of available components.

To illustrate the typical component directory structures and files:

```
/hubzero
  /administrator
    /components
      /com_example
```

```
...
/components
  /com_example
      /controllers
        example.php
      /models
        foo.php
      /views
        /index
          /tmpl
            display.php
            display.xml
    example.php
    router.php
```

In the above 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 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_example folder may have a counterpart with the same name in the /components/com_example folder, yet the two views have nothing in common but their name.

Directory & File Explanation

/com_{componentname}/{componentname}.php
This is the component's main file and entry point *for the front-end part*.
/com_{componentname}/controller.php
This file holds the default frontend controller, which is a class called "{ComponentName}Controller". This class must extend the base class "JController".
/com_{componentname}/views

This folder holds the different views for the component.

/com_{componentname}/views/{viewname}
This folder holds the files for the view {ViewName}.
/com_{componentname}/views/{viewname}/view.html.php

This file is the entry point for the view {ViewName}. It should declare the class {ComponentName}View{ViewName}. This class must extend the base class "JView".

/com_{componentname}/views/{viewname}/tmpl

This folder holds the template files for the view {ViewName}.

/site/views/{viewname}/tmpl/default.php

This is the default template for the view {ViewName}.

/com_{componentname}/models

This folder holds additional models, if needed by the application.

/com_{componentname}/models/{modelname}.php

This file holds the model class {ComponentName}Model{ModelName}. This class must extend the base class "JModel". Note that the view named {ViewName} will by default load a model called {ViewName} if it exists. Most models are named after the view they are intended to be used with.

/com_{componentname}/controllers

This folder holds additional controllers, if needed by the application.

/com_{componentname}/controllers/{controllername}.php

This file holds the controller class {ComponentName}Controller{ControllerName}. This class must extend the base class "JController".

Naming Conventions

Classes

The model, view and controller files use the jimport function to load classes from the Joomla! framework, JModel, JView and JController, respectively. Each class is then extended with a new class specific to the component.

The base controller class for the site is named {ComponentName}Controller. For the administrative section, an "s" is added to the ComponentName, giving {ComponentName}sController. Classnames for additional controllers found within the controllers/ subdirectory are {ComponentName}Controller{ControllerName} for site/ and {ComponentName}sController{ControllerName} for admin/.

The view class is named {ComponentName}View{ViewName}.

The model class is named {ComponentName}Model{ModelName}. Remember that the {ModelName} and the {ViewName} should be the same.

Reserver Words

There are reserved words, which can't be used in names of classes and components.

An example is word "view" (in any case) for view class (except "view" that must be second part of that class name). Because first part of view class name is the same as controller class name, controller class name also can't contain word "view". And because of convension (although violating of it won't produce an error) controller class name must contain component name, so component name also can't contain word "view". So components can't be named "com_reviews", or if thay are, thay must violate naming convention and have different base controller class name (or have some other hacks).

Examples

Here we have a basic front-end component that simply displays a "Hello, World!" message. We present it using standard HUBzero methods, which differ from Joomla! in a few key ways. Note, however, that despite any differences from standard Joomla! methods, all HUBzero methods will still work on a stock Joomla! 1.5+ install.

Download: Hello World component

In the com_drwho example component, we demonstrate working with a few simple database tables. The example shows how to output a listing (with pagination), a form for entering new items, and saving to the database.

Other examples included are using multiple controllers, using models, handling errors, adding some security, and pushing assets (e.g., CSS) to the document.

Download: Dr Who front-end (site) component

Installation

Installing

See Installing Extensions for details.

Uninstalling

See Uninstalling Extensions for details.

Manifests

Overview

It is possible to install a component manually by copying the files using an SFTP client and modifying the database tables. It is more efficient to create a package file in the form on an XML document that will allow the Joomla! Installer to do this for you. This package file contains a variety of information:

- basic descriptive details about your component (i.e. name), and optionally, a description, copyright and license information.
- a list of files that need to be copied.
- optionally, a PHP file that performs additional install and uninstall operations.
- optionally, an SQL file which contains database queries that should be executed upon install/uninstall

Note: All components must be prefixed with com_.

Structure

This XML file just lines out basic information about the template such as the owner, version, etc. for identification by the Joomla! installer and then provides optional parameters which may be set in the Module Manager and accessed from within the module's logic to fine tune its behavior. Additionally, this file tells the installer which files should be copied and installed.

A typical component manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<install type="component" version="1.5.0">
 <name>hello_world</name>
 <!-- The following elements are optional and free of formatting contt
raints -->
 <creationDate>2007 01 17</creationDate>
 <author>John Doe</author>
 <authorEmail>john.doe@example.org</authorEmail>
 <authorUrl>http://www.example.org</authorUrl>
 <copyright>Copyright Info</copyright>
 <license>License Info</license>
 <!--  The version string is recorded in the components table -->
 <version>Component Version String</version>
 <!-- The description is optional and defaults to the name -->
 <description>Description of the component ...</description>

 <!-- Custom Install Script to execute -->
```

```
 <!-- Note: This will be copied from the root of the installation pack
age to the administrator directory automatically -->
 <installfile>install.eventlist.php</installfile>

 <!-- Custom Uninstall Script to execute -->
 <!-- Note: This will be copied from the root of the installation pack
age to the administrator directory automatically -->
 <uninstallfile>uninstall.eventlist.php</uninstallfile>

 <!-- Install Database Section -->
 <install>
  <sql>
   <file driver="mysql" charset="utf8">install.mysql.utf8.sql</file>
   <file driver="mysql">install.mysql.nonutf8.sql</file>
  </sql>
 </install>

 <!-- Uninstall Database Section -->
 <uninstall>
  <sql>
   <file driver="mysql" charset="utf8">uninstall.mysql.utf8.sql</file>
   <file driver="mysql">uninstall.mysql.nonutf8.sql</file>
  </sql>
 </uninstall>

 <!-- Site Main File Copy Section -->
 <files>
  <filename>index.html</filename>
  <filename>test.php</filename>
  <folder>views</folder>
 </files>

 <!-- Site Main Language File Copy Section -->
 <languages>
  <language tag="en-GB">en-GB.com_test.ini</language>
  <language tag="de-DE">de-DE.com_test.ini</language>
  <language tag="nl-NL">nl-NL.com_test.ini</language>
 </languages>

 <!-- Site Main Media File Copy Section -->
 <media destination="com_test">
  <filename>image.png</filename>
  <filename>flash.swf</filename>
 </media>

 <administration>
```

```
  <!-- Administration Menu Section -->
  <menu img="components/com_test/assets/test-16.png">EventList</menu>
  <submenu>
   <!-- Note that all & must be escaped to & for the file to be valid
XML and be parsed by the installer -->
   <menu link="option=com_helloworld&task=hello&who=world">Hello World
!</menu>
   <!-- Instead of link you can specify individual link attributes -->
   <menu img="icon" task="hello" controller="z" view="a" layout="b" su
b="c">Hello Again!</menu>
   <menu view="test" layout="foo">Testing Foo Layout</menu>
  </submenu>

  <!-- Administration Main File Copy Section -->
  <!-- Note the folder attribute: This attribute describes the folder
    to copy FROM in the package to install therefore files copied
    in this section are copied from /admin/ in the package -->
  <files folder="admin">
   <filename>index.html</filename>
   <filename>admin.test.php</filename>
  </files>

  <!-- Administration Language File Copy Section -->
  <languages folder="admin">
   <language tag="en-GB">en-GB.com_test.ini</language>
   <language tag="de-DE">de-DE.com_test.ini</language>
   <language tag="nl-NL">nl-NL.com_test.ini</language>
  </languages>

  <!-- Administration Main Media File Copy Section -->
  <media folder="admin" destination="com_test">
   <filename>admin-image.png</filename>
   <filename>admin-flash.swf</filename>
  </media>
 </administration>
</install>
```

Entry Point

Overview

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:

```
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.

Implementation

(preferred) HUBzero Methodology

HUBzero components differ in subtle, but key ways from standard Joomla! components. This is, in part, due to legacy issues. Some changes are made to aid in development while others may simply be a difference in philosophy. Note, however, that **no** differences require hacking or altering Joomla! in any way and HUBzero methodologies will run on any stock Joomla! install.

```
<?php
// No direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

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

if (version_compare(JVERSION, '1.6', 'lt'))
{
 $jacl = JFactory::getACL();
 $jacl->addACL($option, 'manage', 'users', 'super administrator');
 $jacl->addACL($option, 'manage', 'users', 'administrator');
 $jacl->addACL($option, 'manage', 'users', 'manager');
```

```
}

jimport('joomla.application.component.helper');

// Get the requested controller
$controllerName = JRequest::getCmd('controller', 'one');
// Ensure the controller exists
if (!file_exists(JPATH_COMPONENT . DS . 'controllers' . DS . $controll
erName . '.php'))
{
 $controllerName = 'one';
}
require_once(JPATH_COMPONENT . DS . 'controllers' . DS . $controllerNa
me . '.php');
$controllerName = 'ExampleController' . ucfirst(strtolower($controller
Name));

// Instantiate controller
$controller = new $controllerName();
// Execute whatever task(s)
$controller->execute();
// Redirect as needed
$controller->redirect();
```

Here, you can see we've added a few things and made one subtle change in calling the execute() method. 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.

Next, we added the jimport call to include some Joomla component helpers. This is done specifically because HUBzero controllers do **not** extend JController. JController does some autoloading and initial setup for Joomla! components and since we're not employing it, we need to do some class loading and setup of our own.

Then we look for a requested controller name. There is a default set in case none has been passed or if the requested controller is not found. With the controller name, we build the class name for the controller following the standard camel-cased pattern of {Component name}Controller{Controller name}

Finally, we removed the JRequest::getWord('task') being passed to the execute() method. HUBzero controllers handle the task request within the execute() method, rather than passing the task to it.

Joomla! 1.5 Methodology

The code for this file is fairly typical across components.

```
<?php
// No direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

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

// Require specific controller if requested
if ($controller = JRequest::getWord('controller')) {
 $path = JPATH_COMPONENT.DS.'controllers'.DS.$controller.'.php';
 if (file_exists($path)) {
  require_once $path;
 } else {
  $controller = '';
 }
}

// Create the controller
$classname = 'HelloController'.$controller;
$controller = new $classname( );

// Perform the Request task
$controller->execute( JRequest::getWord( 'task' ) );

// Redirect if set by the controller
$controller->redirect();
```

The first statement is a security check.

JPATH_COMPONENT is the absolute path to the current component, in our case components/com_hello. If you specifically need either the Site component or the Administrator component, you can use JPATH_COMPONENT_SITE or JPATH_COMPONENT_ADMINISTRATOR.

DS is the directory separator of your system: either '/' or '\'. This is automatically set by the framework so the developer doesn't have to worry about developing different versions for different server OSs. The 'DS' constant should always be used when referring to files on the local server.

After loading the base controller, we check if a specific controller is needed. In this component,

the base controller is the only controller, but we will leave this conditional check "in place" for future use.

JRequest:getWord() finds a word variable in the URL or the POST data. So if our URL is index.php?option=com_hello&controller=controller_name, then we can retrieve our controller name in our component using: echo JRequest::getWord('controller');

Now we have our base controller 'HelloController' in com_hello/controller.php, and, if needed, additional controllers like 'HelloControllerController1' in com_hello/controllers/controller1.php. Using this standard naming scheme will make things easy later on: '{Componentname}{Controller}{Controllername}'

After the controller is created, we instruct the controller to execute the task, as defined in the URL: index.php?option=com_hello&task=sometask. If no task is set, the default task 'display' will be assumed. When display is used, the 'view' variable will decide what will be displayed. Other common tasks are save, edit, new...

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.

Note that we don't use a closing php tag in this file: ?>. The reason for this is that we will not have any unwanted whitespace in the output code. This is default practice since Joomla! 1.5, and will be used for all php-only files.

Controllers

Overview

The controller is responsible for responding to user actions. In the case of a web application, a user action is (generally) a page request. The controller will determine what request is being made by the user and respond appropriately by triggering the model to manipulate the data appropriately and passing the model into the view. The controller does not display the data in the model, it only triggers methods in the model which modify the data, and then pass the model into the view which displays the data.

Most components have two controllers: one for the front-end and one for the back-end.

Creating the Front-end Controller

Most HUBzero component controllers will differ from Joomla! in some important ways. Some changes are made to aid in development while others may simply be a difference in philosophy. Note, however, that no differences require hacking or altering Joomla! in any way and HUBzero methodologies will run on any stock Joomla! install.

```
<?php
// No direct access
defined('_JEXEC') or die('Restricted access');

ximport('Hubzero_Controller');

class HelloControllerOne extends Hubzero_Controller
{
 public function displayTask()
 {
  // Pass the view any data it may need
  $this->view->greeting = 'Hello, World!';

  // Set any errors
  if ($this->this->getError())
  {
   foreach ($this->getErrors() as $error)
   {
    $view->setError($error);
   }
  }

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

```
 }
}
```

There doesn't appear to be much going on here due to some of the auto-setup the Hubzero_Controller class brings.

The first, and most important, difference to note is that we're extending Hubzero_Controller rather than JController. Since we're not employing JController, how tasks and views are determined, built, and executed differ from standard Joomla components.

Note: Hubzero_Controller extends JObject, so all its methods and properties are available.

One key difference is how the execute() method is handled. In Joomla!, any public method is assumed to be an executable task. In the HUBzero method, the list of available tasks is built from only methods that are 1) public and 2) end in "Task". When calling a task, the "Task" suffix should be left off. For example:

```
// This route
JRoute::_('index.php?option=com_example&task=other');

// Refers to
....
public function otherTask()
{
 ...
}
....
```

If no task is supplied, the controller will default to a task of "display". The default task can be set in the controller:

```
<?php
// No direct access
defined('_JEXEC') or die('Restricted access');

ximport('Hubzero_Controller');

class HelloControllerOne extends Hubzero_Controller
{
 public function execute()
```

```
 {
  // Set the default task
  $this->registerTask('__default', 'mydefault');

  // Set the method to execute for other tasks
  // The following can be called by task=delete and will execute the r
emoveTask method
  $this->registerTask('delete', 'remove');  // (task, method name);

  parent::execute();
 }
 ...
}
```

Each controller extending Hubzero_Controller will have the following properties available:

- _option - String, component name (e.g., com_example)
- _controller - String, controller name
- view - Object (JView)
- config - Object (JParamter), component config
- database - Object (JDatabase)
- juser - Object (JUser)

```
class HelloControllerOne extends Hubzero_Controller
{
 public function displayTask()
 {
  $this->view->userName = $this->juser->get('name');
  $this->view->display();
 }
}
```

Auto-generation of views

The Hubzero_Controller automatically instantiates a new JView object for each task and assigns the component ($option) and controller ($controller) names as properties for use in your view. Controller names map to view directory and task names directly map to view names.

```
  /{component}
    /views
```

```
      /one  (controller name)
        /tmpl
          /display.php
          /remove.php
```

Example usage within a view:

```
<p>This is component <?php echo $this->option; ?> using controller: <?
php echo $this->controller; ?></p>
```

Changing view layout

As mentioned above, the view object is auto-generated with the same layout as the current $task. There are times, however, when you may want to use a different layout or are executing a task after directing through from a previous task (example: saveTask encountering an error and falling through to the editTask to display the edit form with error message). The layout can easily be switched with the setLayout method.

```
  /{component}
    /views
      /one  (controller name)
        /tmpl
          /display.php
          /world.php

//-------------
//-------------

class HelloControllerOne extends Hubzero_Controller
{
 public function displayTask()
 {
  // Set the layout to 'world.php'
  $this->view->setLayout('world');

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

Any assigned data or vars to the view will not be effected.

Helpers

Overview

A helper class is a class filled with static methods and is usually used to isolate a "useful" algorithm. They are used to assist in providing some functionality, though that functionality isn't the main goal of the application. They're also used to reduce the amount of redundancy in your code.

Implementation

Helper classes are stored in the helper sub-directory of your component directory. Naming convention typically follows a pattern of {ComponentName)Helper(HelperName}. Therefore, our helper class is called HelloHelperOutput.

Here's our com_hello/helpers/output.php helper class:

```
<?php
// No direct access

defined( '_JEXEC' ) or die( 'Restricted access' );

jimport('joomla.application.component.helper');

/**
 * Hello World Component Helper
 *
 * @package    Joomla.Tutorials
 * @subpackage Components
 */
class HelloHelperOutput
{
 /**
  * Method to make all text upper case
  *
   * @access    public
  */
 public function shout($txt='')
 {
  return strToUpper($txt).'!';
 }
}
```

We have one method in this class that takes all strings passed to it and returns them uppercase with an exclamation point attached to the end. To use this helper, we do the following:

```
class HelloViewHello extends JView
{
 function display($tpl = null)
 {
  include_once(JPATH_COMPONENT.DS.'helpers'.DS.'output.php');

  $greeting = HelloHelperOutput::shout("Hello World");
  $this->assignRef( 'greeting', $greeting );

  parent::display($tpl);
 }
}
```

Models

Overview

The concept of model gets its name because this class is intended to represent (or 'model') some entity.

Creating A Model

All Joomla! models extend the JModel class. The naming convention for models in the Joomla! framework is that the class name starts with the name of the component, followed by 'model', followed by the model name. Therefore, our model class is called HelloModelHello.

```
<?php
// No direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.application.component.model' );

/**
 * Hello Model
 */
class HelloModelHello extends JModel
{
    /**
    * Gets the greeting
    * @return string The greeting to be displayed to the user
    */
    function getGreeting()
    {
        return 'Hello, World!';
    }
}
```

You will notice a line that starts with jimport. The jimport function is used to load files from the Joomla! framework that are required for our component. This particular statement will load the file /libraries/joomla/application/component/model.php. The '.'s are used as directory separators and the last part is the name of the file to load. All files are loaded relative to the libraries directory. This particular file contains the class definition for the JModel class, which is necessary because our model extends this class.

Using A Model

The Joomla! framework is setup in such a way that the controller will automatically load the model that has the same name as the view and will push it into the view. We can easily retrieve a reference to our model using the JView::getModel() method. If the model had not followed this convention, we could have passed the model name to JView::getModel().

Here's an example of using a model with our Hello component (com_hello).

```
<?php
// No direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.application.component.view');

/**
 * HTML View class for the HelloWorld Component
 *
 * @package    HelloWorld
 */

class HelloViewHello extends JView
{
 function display($tpl = null)
 {
  $model = &$this->getModel();
  $greeting = $model->getGreeting();
  $this->assignRef( 'greeting', $greeting );

  parent::display($tpl);
 }
}
```

Languages

Setup

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

```
; Module - Hellow World (en-US)
COM_HELLOWORLD_LABEL_USER_COUNT = "User Count"
COM_HELLOWORLD_DESC_USER_COUNT = "The number of users to display"
COM_HELLOWORLD_RANDOM_USERS = "Random Users for Hello World"
COM_HELLOWORLD_USER_LABEL = "%s is a randomly selected user"
```

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 COM_{ComponentName}_{Text} for naming. Adhering to this naming convention is not required but is strongly recommended as it can help avoid potential translation collisions.

See the Languages overview for details.

Translating Text

Below is an example of accessing the translate helper:

```
<p><?php echo JText::_("COM_EXAMPLE_MY_LINE"); ?></p>
```

JText::_ is used for simple strings.
JText::sprintf is used for strings that require dynamic data passed to them for variable replacement.

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

See the Languages overview for details.

Layouts

Directory Structures & Files

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 default case, we have the view script default.php.

```
/hubzero
  /components
    /com_{componentname}
      /views
        /{viewname}
          view.html.php
          /tmpl
            default.php
```

Overriding module and component presentation in templates is further explained in the Templates: Overrides section.

Creating A View

Joomla! 1.5 Method

The task of the view is very simple: It retrieves the data to be displayed and pushes it into the template. Data is pushed into the template using the JView::assignRef method.

```
<?php

// no direct access
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.application.component.view');

/**
 * HTML View class for the HelloWorld Component
 *
 * @package    HelloWorld
 */

class HelloViewHello extends JView
```

```
{
    function display($tpl = null)
    {
        $greeting = "Hello World!";
        $this->assignRef( 'greeting', $greeting );

        parent::display($tpl);
    }
}
```

HUBzero Method

Not necessary. Data retrieval and template assignment is handled in the controller.

Creating the Template

Joomla! templates/layouts are regular PHP files that are used to layout the data from the view in a particular manner. The variables assigned by the JView::assignRef method can be accessed from the template using $this->{propertyname} (see the template code below for an example).

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

```
<?php

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

Routing

Overview

All components in Joomla! can be accessed through a query string by using the option parameter which will equate to the name of the component. For example, to access the "Contacts" component, you could type http://yourhub.org/index.php?option=com_contact.

When SEF URLs are being employed, the first portion after the site name will almost always be the name of a component. For the URL http://yourhub.org/contact, the first portion after the slash translates to the component com_contact. If a matching component cannot be found, routing will attempt to match against an article section, category, and/or page alias.

While not required, most components will have more detailed routing instructions that allow SEF URLs to be made from and converted back into query strings that pass necessary data to the component. This is done by the inclusion of a file called router.php.

router.php

Every router.php file has two methods: {ComponentName}BuildRoute() which takes a query string and turns it into a SEF URL and {ComponentName}ParseRoute() which deconstructs a SEF URL back into a query string to be passed to the component.

```
function ExampleBuildRoute(&$query)
{
 $segments = array();

 if (!empty($query['task'])) {
  $segments[] = $query['task'];
  unset($query['task']);
 }
 if (!empty($query['id'])) {
  $segments[] = $query['id'];
  unset($query['id']);
 }
 if (!empty($query['format'])) {
  $segments[] = $query['format'];
  unset($query['format']);
 }

 return $segments;
}

function ExampleParseRoute($segments)
```

```
{
 $vars = array();

 if (empty($segments)) {
  return $vars;
 }
 if (isset($segments[0])) {
  $vars['task'] = $segments[0];
 }
 if (isset($segments[1])) {
  $vars['id'] = $segments[1];
 }
 if (isset($segments[2])) {
  $vars['format'] = $segments[2];
 }

    return $vars;
}
```

{ComponentName}BuildRoute()

This method is called when using JRoute::_(). JRoute::_() passes the query string (minus the option={componentname} portion) to the method which returns an array containing the necessary portions of the URL to be constructed *in the order* they need to appear in the final SEF URL.

```
// $query = 'task=view&id=123&format=rss'
function ExampleBuildRoute(&$query)
{
 $segments = array();

 if (!empty($query['task'])) {
  $segments[] = $query['task'];
  unset($query['task']);
 }
 if (!empty($query['id'])) {
  $segments[] = $query['id'];
  unset($query['id']);
 }
 if (!empty($query['format'])) {
  $segments[] = $query['format'];
  unset($query['format']);
 }
```

```
 return $segments;
}
```

Will return:

```
Array(
 'view',
 '123',
 'rss'
);
```

This will in turn be passed back to JRoute::_() which will construct the final SEF URL of example/view/123/rss.

{ComponentName}ParseRoute()

This method is automatically called on each page view. It is passed an array of segments of the SEF URL that called the page. That is, a URL of example/view/123/rss would be separated by the forward slashes with the first segment automatically being associated with a component name. The rest are stored in an array and passed to {ComponentName}ParseRoute() which then associates each segment with an appropriate variable name based on the segment's position in the array.

```
function ExampleParseRoute($segments)
{
 $vars = array();

 if (empty($segments)) {
  return $vars;
 }
 if (isset($segments[0])) {
  $vars['task'] = $segments[0];
 }
 if (isset($segments[1])) {
  $vars['id'] = $segments[1];
 }
 if (isset($segments[2])) {
  $vars['format'] = $segments[2];
 }
```

```
    return $vars;
}
```

Note: Position of segments is very important here. A URL of example/view/123/rss could yield completely different results than a URL of example/rss/view/123.

Packaging

Overview

Packaging a component for distribution is relatively easy. All front-end files are places within a directory called /site and all administration files are placed within a directory called /admin. Here's what a typical package will look like:

```
/com_{componentname}
  {componentname}.xml
  /site
    {componentname}.php
    controller.php
    /views
      /{viewname}
        view.html.php
        /tmpl
          default.php
    /models
      {modelname}.php
    /controllers
      {controllername}.php
  /admin
    {componentname}.php
    controller.php
    /views
      /{viewname}
        view.html.php
        /tmpl
          default.php
    /models
      {modelname}.php
    /controllers
      {controllername}.php
```

Just "zip" up the primary directory into a compressed archive file. When the ZIP file is installed, the language file is copied to /language/{LanguageName}/{LanguageName}.{ComponentName}.ini and is loaded each time the module is loaded. All of the other files are copied to the /components/{ComponentName} and /administrator/components/{ComponentName}directories of the Joomla! installation.