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 perfect sense as a stand-alone system.

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}.

Examples

In the com_drwho example component, we demonstrate working with an MVC structure, basic usage of the database ORM, and more. The admin and site examples show how to output a listing (with pagination) and a form for entering new items and how to save 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.

Example usage of the API is also included.

Download: Doctor Who component (single model, single controller)
This is a basic example including a single database table, one model, and a single controller. Included are examples of the front-end, admin interface, and API.

Download: Doctor Who component (multi-model, multi-controller)
This is a more complex example including multiple database tables, multiple inter-dependent models, and a couple controllers to demonstrate division and organization of code. Included are examples of the front-end, admin interface, and API.

Structure

Naming Conventions

The model, view and controller files use classes from the framework, HubzeroBaseModel, HubzeroComponentView and HubzeroComponentSiteController, respectively. Each class is then extended with a new class specific to the component.

Administrative controllers extend HubzeroComponentAdminController which in turn extends HubzeroComponentSiteController and adds a few extra methods frequently used throughout the administrative portion of the site.

All components must be under the Components namespace and follow PSR-0 naming scheme with one exception: files and folders may be lowercase even when their class names are not.

Directories & 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 #__extensions table will not appear in the administrative list of available components.

To illustrate the typical component directory structures and files:

```
/app
.. /components
.. .. /com_example
.. .. .. /admin
.. .. .. /api
.. .. .. /helpers
.. .. .. /migrations
.. .. .. /models
.. .. .. /site
.. .. .. .. /assets
.. .. .. .. .. /css
```

```
.. .. .. .. .. /js
.. .. .. .. .. /img
.. .. .. .. /controllers
.. .. .. .. .. example.php
.. .. .. .. /views
.. .. .. .. .. /example
.. .. .. .. .. .. /tmpl
.. .. .. .. .. .. .. display.php
.. .. .. .. .. .. .. display.xml
.. .. .. .. example.php
.. .. .. .. router.php
.. .. .. /tests
```

Files are contained within directories titled "com_example". Some directories and files are optional but, for this example, we've included a more common setup.

All client-specific files and sub-directories are split between the respective client directories, such as admin and site. Since controllers and views are specific to a client, they reside within those client directories. Shared files, typically models and helpers, are within directories at the same level as the client folders.

Directory & File Explanation

/com_{componentname}/{client}/{componentname}.php
This is the component's entry point for the admin and site clients. API and Cli (console) are special cases and don't require this file.
/com_{componentname}/{client}/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}/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}/migrations

Database migrations for this component. The first or oldest migration will typically be the initial installation, creating any necessary database tables or seeding required data. Subsequent migrations will handle any schema or global data changes needed as the code evolves.

/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 "HubzeroBaseModel". 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}/{client}/controllers

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

/com_{componentname}/{client}/controllers/{controllername}.php
This file holds the controller class {ComponentName}Controller{ControllerName}. This class must extend the base class HubzeroComponentsSiteController.

/com_{componentname}/tests

Unit tests for your code. These tests can be run through the command line **muse** application that comes with a HUBzero install.

Entry Point

The CMS 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:

```
For search engine friendly URLs:
/hello

For non-SEF URLs:
/index.php?option=com_hello
```

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

Implementation

```
<?php
// Define the namespace
// Components{ComponentName}{ClientName};
namespace ComponentsHelloSite;

// Get the requested controller
$controllerName = Request::getCmd('controller', Request::getCmd('view'
, 'one'));

// Ensure the controller exists
if (!file_exists(__DIR__ . DS . 'controllers' . DS . $controllerName .
 '.php'))
{
 App::abort(404, Lang::txt('Controller not found'));
}
require_once(__DIR__ . DS . 'controllers' . DS . $controllerName . '.p
hp');
$controllerName = __NAMESPACE__ . '\Controllers\' . ucfirst(strtolower
($controllerName));

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

The first statement is defining the namespace. All component namespaces **must** be under the Components namespace.

__DIR__ is a pre-defined PHP constant that evaluates to the absolute path to the current directory, in our case /webroot/app/components/com_hello/site.

DS is and alias of PHP's DIRECTORY_SEPARATOR constant and produces 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 OSes.

Most modern systems can now handle paths designated with a forward slash "/" so use of the DS constant is no longer strictly required or necessary.

First 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 namespaced camel-cased pattern of Components{Component name}{Client name}Controllers{Controller name}

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 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 and will be used for all php-only files. Please see the coding Style Guide for further details.

Configuration

Overview

The framework allows the use of parameters stored in each component.

Defining Options

Configuration options can also be defined in a separate file named config.xml located in the /config sub-directory of the component directory.

```
/app
.. /components
.. .. /com_hello
.. .. .. /config
.. .. .. .. config.xml
```

The XML file's root element should be <config>. Fields are then added and grouped by fieldsets. These fieldsets correspond to the tabs located in the admin side when viewing the component's options.

```
<?xml version="1.0" encoding="utf-8"?>
<config>
 <fieldset
  name="greetings"
  label="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_LABEL"
  description="COM_HELLOWORLD_CONFIG_GREETING_SETTINGS_DESC"
 >
  <field
   name="greeting"
   type="text"
   label="COM_HELLOWORLD_FIELD_GREETING_LABEL"
   description="COM_HELLOWORLD_FIELD_GREETING_DESC"
   default=""
  />
 </fieldset>
</config>
```

It is good practice to use the component's language file to define all the appropriate strings.

Retrieving Values

One may quickly retrieve the options for any component by calling the params() method on the Component facade or directly accessing the method on the underlying HubzeroComponentLoader class. This method returns a HubzeroConfigRegistry object.

```
$params = Component::params('com_hello');

echo $param->get('greeting');
```

Routing

Overview

All components 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 "Blog" component, you could type http://yourhub.org/index.php?option=com_blog.

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/blog, the first portion after the slash translates to the component com_blog. 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.

The Router

Every router.php file has a class with two methods: build() which takes a query string and turns it into a SEF URL and parse() which deconstructs a SEF URL back into a query string to be passed to the component.

```
<?php
namespace ComponentsExampleSite;

use HubzeroComponentRouterBase;

class Router extends Base
{
 public function build(&$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;
 }

 public function parse($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;
 }
}
```

The build() Method

This method is called when using Route::url(). Route::url() 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.

```
// $query = 'task=view&id=123&format=rss'
public function build(&$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 Route::url() which will construct the final SEF URL of example/view/123/rss.

The parse() Method

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 parse() which then associates each segment with an appropriate variable name based on the segment's position in the array.

```
public function parse($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.

Assets

Overview

Frequently, components will make use of their styles, images, and scripts to further enhance the interface and user experience. There are a number of helpers to make adding CSS and Javascript to the document a quick and easy process.

Directory Structure & Files

Assets are stored in the same directory as the entry point, views, and controllers for each client type of a component. This means, for example, the administrative side and front-end of a component may make use of completely different assets.

While there are no hard rules on the placement and organization of the files, it is highly recommended to follow the structure detailed below as it helps keep both small and large projects clean, organized, and allows for several helper methods (detailed in the "Helpers" section) to function, eliminating the tedious need for path building and file existence checking before attaching to the document.

All assets are stored within an assets folder, which is further sub-divided by asset type. The most common types being js (javascript), css (cascading stylesheets), and img (images) but may also contain any other asset such as fonts, less, and so on.

```
/app
.. /components
.. .. /{ComponentName}
.. .. .. /{ClientName}
.. .. .. .. /assets
.. .. .. .. .. /css
.. .. .. .. .. /img
.. .. .. .. .. /js
```

Helpers

The HubzeroComponentSiteController and HubzeroComponentView classes bring with them some useful methods for pushing StyleSheets and JavaScript assets to the document and building paths to images. These methods can be called from within a controller or a component view.

Cascading Stylesheets

The css() method provides a quick and convenient way to attach stylesheets. It accepts two arguments:

1. The name of the stylesheet to be pushed to the document (file extension is optional). If no name is provided, the name of the component (without the com_ prefix) will be used. For instance, if called within a view of the members component com_members, the system will look for a stylesheet named members.css.
2. The name of the extension to look for the stylesheet. This accepts either module, component or plugin name and will follow the same naming conventions used for extension directories (e.g. "com_tags", "mod_login", etc). Passing an extension name of "system" will retrieve assets from the core system assets (/core/assets).

For the defined stylesheet to be found, the assets **must** be organized as described in the "Directory Structure & Files" section.

Method chaining is also allowed.

```
<?php
// Push a stylesheet to the document
$this->css()
      ->css('another');
?>
... view HTML ...
```

Javascript

Similarly, a js() method is available for pushing javascript assets to the document. The arguments accepted are the same as the css() method described above.

```
<?php
// Push some javascript to the document
$this->js()
      ->js('another');
?>
... view HTML ...
```

Images

Finally, a img() method is available for building paths to images within the component's assets directory. Unlike the css() and js() methods, this helper does not add anything to the global document object and, instead, simply returns an absolute file path.

Given the following directory structure:

```
/app
.. /components
.. .. /{ComponentName}
.. .. .. /{ClientName}
.. .. .. .. /assets
.. .. .. .. .. /img
.. .. .. .. .. .. picture.png
```

From a component view:

```
<!-- Generate the path to the image -->
<img src="<?php echo $this->img('picture.png'); ?>" alt="My picture" /
>
```

Views

Directory Structures & Files

Views are written in PHP and HTML and have a .php file extension. View scripts are placed in /com_{component name}/{client}/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 display.php.

```
/app
  /components
    /com_{componentname}
      /{client [site, admin]}
        /views
          /{viewname}
            /tmpl
              default.php
```

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

Creating A View

The task of the view is very simple: It retrieves the data to be displayed and pushes it into the template.

```
// Instantiate a new view
$view = new HubzeroComponentView(array(
    'name' => $this->_controller,
    'layout' => 'foo'
));

// Assign data to the view
$view->greetings = 'Hello';

// Echo out the results
$view->display();
```

In the above example, the view constructor is passed an array of options. The two most important options are listed: name, which is the folder to look for the view file in and will typically correspond to the current controller's name, and layout, which is the specific view file to load. If no layout is specified, the layout is typically auto-assigned to the current task name. So, if the controller in the example code is one, the directory structure would look as follow:

```
/com_example
    /views
        /one
            /tmpl
                /foo.php
```

Method Chaining

All Hubzero view objects support method chaining for brevity and ease of use.

```
// Instantiate a new view
$view = new HubzeroComponentView(array(
    'name' => $this->_controller,
    'layout' => 'foo'
));

$view->set('greetings', 'Hello')
      ->setLayout('bar')
      ->display();
```

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 Lang::txt("COM_EXAMPLE_MY_LINE"); ?></p>
```

Lang::txt is used for both simple strings and strings that require dynamic data passed to them for variable replacement.

```
<p><?php echo Lang::txt('Hello %s. How are you?', $name); ?></p>
```

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

See the Languages overview for details.

Models

Overview

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

Creating A Model

All HUBzero models extend the HubzeroBaseModel class. The naming convention for models in the 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 ComponentsHelloModelsHello.

```
<?php
namespace ComponentsHelloModels;

use HubzeroBaseModel;

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

You will notice a lack of include, require, or import calls. Hubzero classes are autoloaded and map to files located in the /core/libraries/Hubzero directory. See more on naming conventions.

Using A Model

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

```
<?php
namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;
use ComponentsHelloModelsHello;

/**
 * Controller for the HelloWorld Component
 */
class Greetings extends SiteController
{
 public function display()
 {
  $model = new Hello();
  $greeting = $model->getGreeting();

  $this->set('greeting', $greeting)
       ->display();
 }
}
```

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 helpers sub-directory of your component folder. As with all other classes, naming follows the PSR-0 convention and are within the Components namespace. Therefore, our helper class is called ComponentsHelloHelpersOutput.

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

```
<?php

namespace ComponentsHelloHelpers;

/**
 * Hello World Component Helper
 */
class Output
{
 /**
  * Method to make all text upper case
  *
  * @param  string  $txt
  * @return  string
  */
 public static function shout($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:

```
<?php

namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;
use ComponentsHelloHelpersOutput;

class Greetings extends SiteController
{
 public function displayTask()
 {
  include_once(dirname(dirname(__DIR__)) . DS . 'helpers' . DS . 'outp
ut.php');

  $greeting = Output::shout('Hello World');

  $this->view
                 ->set('greeting', $greeting)
                 ->display();
 }
}
```

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.

Site Controller

```
<?php
namespace ComponentsHelloSiteControllers;

use HubzeroComponentSiteController;

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

  // Set any errors
  $view->setErrors($this->getErrors());

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

The first, and most important part to note is that we're extending HubzeroComponentSiteController which brings several tools and some auto-setup for us.

Note: HubzeroComponentSiteController extends HubzeroBaseObject, so all its methods and properties are available.

In the execute() 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
Route::url('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:

```
class One extends SiteController
{
 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 HubzeroComponentSiteController will have the following properties available:

- _option - String, component name (e.g., com_example)
- _controller - String, controller name
- view - Object (View)
- config - Object (Registry), component config

```
<?php
```

```
class One extends SiteController
{
 public function displayTask()
 {
  $this->view->userName = User::get('name');
  $this->view->display();
 }
}
```

Auto-generation of views

The HubzeroComponentSiteController automatically instantiates a new HubzeroComponentView 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}
    /site
      /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 One extends SiteController
{
 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.

Admin Controller

Administrator component controls are built and function the same as the Front-end (site) controllers with one key difference: they extends HubzeroComponentAdminController.

```
<?php

class One extends AdminController
{
    ...
}
```

The primary difference between SiteController and AdminController is the pre-defining of a few

tasks commonly used in administrator components.

API Controller

API controllers extend HubzeroComponentApiController. Functionally, API controllers are very similar to site and admin controllers in that defining executable tasks is done by creating public methods with a "Task" suffix. They differ, however, in two key ways:

1) Controllers follow a naming convention unique to the API. [TODO: fill in]

2) The API has no concept of views and thus no View object to render data. Instead, data is sent back to the application via the send method which, in turn, prepares the response before delivering to the user.

```
<?php
namespace ComponentsExampleApiControllers;

use HubzeroComponentApiController;

class Greetings extend ApiController
{
    public function listTask()
    {
        $model = new Archive();
        $data = $model->all();

        $this->send($data);
    }
}
```

Packaging

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 a composer.json document that will allow the Installer to do this for you. This package file resides in the top-level of your component's directory and contains a variety of information:

- basic descriptive details about your component (i.e. name), and optionally, a description, copyright and license information.
- the extension type (component, module, plugin, template)
- optionally, a destined install directory

Structure

Packaging a component for distribution is relatively easy. The file and directory structure is exactly as it would be after installation. Here's what a typical package will look like:

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

```
{controllername}.php
```

Migrations

All the common extension types for HUBzero can include their own migrations directory. Migrations are used for installing the extension into the required tables for the CMS to know about said extension's existence, installing any needed tables, installing sample data, etc.

To illustrate the typical component directory structures and files:

```
/app
.. /components
.. .. /com_example
.. .. .. /admin
.. .. .. /api
.. .. .. /helpers
.. .. .. /migrations
.. .. .. .. /Migration20190301102219ComExample.php
.. .. .. .. /Migration20190301102256ComExample.php
.. .. .. .. /Migration20190301102301ComExample.php
.. .. .. /models
.. .. .. /site
.. .. .. /tests
```

See the Migrations documentation for more about naming conventions, setup, etc.

Components typically have one to three initial migrations: one for registering the component with the CMS, one for installing any tables specific to the component, and one for installing any default or sample data. While all of this can be done in one migration, it's typically broken into the three to allow for easier (re-)running of the migrations in steps. Next, we'll go through examples of each common migration.

A migration for registering the component with the CMS typically just involves calling the addComponentEntry helper method:

```
<?php

use Hubzero\Content\Migration\Base;

// No direct access
defined('_HZEXEC_') or die();

/**
 * Migration script for registering the example component
```

```
 **/
class Migration20190301102219ComExample extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        // Register the component Note the 'com_' prefix is optional.
        //
        // @param   string  $name           (required) Component name
        // @param   string  $option         (optional) com_xyz
        // @param   int     $enabled        (optional, default: 1) Wh
ether or not the component should be enabled
 // @param   string  $params         (optional) Component params (if
already known)
        // @param   bool    $createMenuItem  (optional, default: true)
 Create an admin menu item for this component
        $this->addComponentEntry('example');
    }

    /**
     * Down
     **/
    public function down()
    {
        // Provide the name of the component. Note the 'com_' prefix i
s optional.
        $this->deleteComponentEntry('example');
    }
}
```

That's all there is to it! The addComponentEntry() adds the necessary entries to the needed database tables for the CMS to be aware of the component's existence. Next, we'll look at a typical table installation migration.

```
<?php

use Hubzero\Content\Migration\Base;

// No direct access
defined('_HZEXEC_') or die();
```

```
/**
 * Migration script for registering the example component
 **/
class Migration20190301102256ComExample extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        if (!$this->db->tableExists('#__example_entries'))
        {
            $query = "CREATE TABLE `#__example_entries` (
                `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
                `title` varchar(255) NOT NULL DEFAULT '',
                `alias` varchar(255) NOT NULL DEFAULT '',
                `content` text NOT NULL,
                `created` datetime DEFAULT NULL,
                `created_by` int(11) unsigned NOT NULL DEFAULT '0',
                `state` tinyint(2) NOT NULL DEFAULT '0',
                `params` tinytext NOT NULL,
                `access` tinyint(3) NOT NULL DEFAULT '0',
                PRIMARY KEY (`id`),
                KEY `idx_created_by` (`created_by`),
                KEY `idx_alias` (`alias`)
                ) ENGINE=InnoDB DEFAULT CHARSET=utf8;";

            $this->db->setQuery($query);
            $this->db->query();
        }
    }

    /**
     * Down
     **/
    public function down()
    {
        if ($this->db->tableExists('#__example_entries'))
        {
            $query = "DROP TABLE IF EXISTS `#__example_entries`;";
            $this->db->setQuery($query);
            $this->db->query();
        }
    }
}
```

In the up() method, we check if the table exists first and, if not, we add it. Existence checks for tables, keys, columns, etc. are essential to ensure migrations can be run and re-ran without conflicts. Also note that in all our migrations we undo everything from the up() in the down() method. This allows for rolling back a migration should the need arise.

Finally, we'll take a look at a migration that installs sample data.

```
<?php

use Hubzero\Content\Migration\Base;

// No direct access
defined('_HZEXEC_') or die();

/**
 * Migration script for registering the example component
 **/
class Migration20190301102301ComExample extends Base
{
    /**
     * Up
     **/
    public function up()
    {
        if ($this->db->tableExists('#__example_entries'))
        {
            // Check if the entry already exists
            $query = "SELECT * FROM `#__example_entries` WHERE `alias`
='sample' AND `created_by`='1000';"
            $this->db->setQuery($query);
            $id = $this->db->loadResult();

            if (!$id)
            {
                $now = new HubzeroUtilityDate('now');

                $query = "INSERT INTO `#__example_entries` (
                `title`, `alias`, `content`,  `created`, `created_by`,
 `state`, `params`, `access`) VALUES ('Sample', 'sample', 'Sample cont
ent!', $now->toSql(), '1000', '1', '', '1');

                $this->db->setQuery($query);
                $this->db->query();
            }
```

```
        }
    }

    /**
     * Down
     **/
    public function down()
    {
        if ($this->db->tableExists('#__example_entries'))
        {
            $query = "DELETE FROM `#__example_entries` WHERE `alias`='
sample' AND `created_by`='1000';
            $this->db->setQuery($query);
            $this->db->query();
        }
    }
}
```