The Model-View-Controller Packages

Introduction

Version 12.1 of the platform introduced a new format for model-view-controller paradigm. Principly, the classes JModel, JView and JController are now interfaces and the base abstract classes are now JModelBase, JViewBase and JControllerBase respectively. In additional, all classes have been simplified removing a lot of coupling with the Joomla CMS that is unnecessary for standalone Joomla Platform applications.
All the API for controllers, models and views has moved from the Application package into separate Controller, Model and View packages respectively. Much of the API previously devoted to adding include paths for each of the classes has been removed because of improvements in the auto-loader or by registering or discovering classes explicitly using JLoader.
Controllers only support one executable task per class via the execute method. This differs from the legacy JController class which mapped tasks to methods in a single class. Messages and redirection are not always required so have been dropped in this base class. They can be provided in a downstream class to suit individual applications. Likewise, methods to create models and views have been dropped in favor of using application or package factory classes.
Models have been greatly simplified in comparison to their legacy counterpart. The base model is nothing more than a class to hold state. All database support methods have been dropped except for database object support in JModelDatabase. Extended model classes such as JModelAdmin, JModelForm, JModelItemand JModelList are part of the legacy platform. Most of their function has been replaced by API availble in the Content package also new in 12.1.
Views have also been greatly simplified. Views are now injected with a single model and a controller. Magic get methods have been dropped in favor of using the model directly. Similarly, assignment methods have also been dropped in favor of setting class properties explicitly. The JViewHtml class still implements layout support albeit in a simplified manner.

JController

JController is an interface that requires a class to be implemented with an execute, a getApplication and a getInput method.

JControllerBase

Construction

The constructor for JControllerBase takes an optional JInput object and an optional JApplciationBase object. If either is omitted, the constructor defers to the protected loadInput and loadApplication methods respectively. These methods can be overriden in derived classes if the default application and request input is not appropriate.

Usage

The JControllerBase class is abstract so cannot be used directly. The derived class must implement the execute method to satisfy the interface requirements. Note that the execute method no longer takes a "task" argument as each controller class. Multi-task controllers are still possible but not recommended. Each controller class should do just one sort of 'thing', just as saving, deleting, checking in, checking out and so on. However, controllers, or even models and views, have the liberty of invoking other controllers to allow for HMVC architectures.

Example 2.22. Example controller

```
	/**
	 * My custom controller.
	 *
	 * @package  Examples
	 *
	 * @since   12.1
	 */
class MyController extends JControllerBase
{
	/**
	 * Method to execute the controller.
	 *
	 * @return  void
	 *
	 * @since   12.1
	 * @throws  RuntimeException
	 */
	public function execute()
	{
		echo time();
	}
}

// Instantiate the controller.
$controller = new MyController;

// Print the time.
$controller->execute();
```

Serialization

The JControllerBase class implements Serializable. When serializing, only the input property is serialized. When unserializing, the input variable is unserialized and the internal application property is loaded at runtime.

JModel

JModel is an interface that requires a class to be implemented with a getState and a setState method.

JModelBase

Construction

The contructor for JModelBase takes an optional JRegistry object that defines the state of the model. If omitted, the contructor defers to the protectedloadState method. This method can be overriden in a derived class and takes the place of the populateState method used in the legacy model class.

Usage

The JModelBase class is abstract so cannot be used directly. All requirements of the interface are already satisfied by the base class.

Example 2.23. Example model

```
	/**
	 * My custom model.
	 *
	 * @pacakge  Examples
	 *
	 * @since   12.1
	 */
class MyModel extends JModelBase
{
	/**
	 * Get the time.
	 *
	 * @return  integer
	 *
	 * @since   12.1
	 */
	public function getTime()
	{
		return time();
	}
}
```

JModelDatabase

Construction

JModelDatabase is extended from JModelBase and the contructor takes an optional JDatabaseDriver object and an optional JRegistry object (the same one that JModelBase uses). If the database object is omitted, the contructor defers to the protected loadDb method which loads the database object from the platform factory.

Usage

The JModelDatabase class is abstract so cannot be used directly. It forms a base for any model that needs to interact with a database.

Example 2.24. Example database model

```
	/**
	 * My custom database model.
	 *
	 * @package  Examples
	 *
	 * @since   12.1
	 */
class MyDatabaseModel extends JModelDatabase
{
	/**
	 * Get the content count.
	 *
```

```
	 * @return  integer
	 *
	 * @since   12.1
	 * @throws  RuntimeException on database error.
	 */
	public function getCount()
	{
		// Get the query builder from the internal database object.
		$q = $this->db->getQuery(true);

		// Prepare the query to count the number of content records.
		$q->select('COUNT(*)')
			->from($q->qn('#__content'));

		$this->db->setQuery($q);

		// Execute and return the result.
		return $this->db->loadResult();
	}
}

try
{
	$model = new MyDatabaseModel;
	$count = $model->getCount();
}
catch (RuntimeException $e)
{
	// Handle database error.
}
```

JView

JView is an interface that requires a class to be implemented with an escape and a render method.

JViewBase

Construction

The contructor for JViewBase takes a JModel object and a JController object. Both are mandatory.
Note that these are interfaces so the objects do no necessarily have to extend from JModelBase or JControllerBase classes. Given that, the view should only rely on the API that is exposed by the interface and not concrete classes unless the contructor is changed in a derived class to take more explicit classes or interaces as required by the developer.

Usage

The JViewBase class is abstract so cannot be used directly. It forms a simple base for rendering any kind of data. The class already implements the escapemethod so only a render method

need to be added. Views derived from this class would be used to support very simple cases, well suited to supporting web services returning JSON, XML or possibly binary data types. This class does not support layouts.

Example 2.25. Example view

```
	/**
	 * My custom view.
	 *
	 * @package  Examples
	 *
	 * @since   12.1
	 */
class MyView extends JViewBase
{
	/**
	 * Render some data
	 *
	 * @return  string  The rendered view.
	 *
	 * @since   12.1
	 * @throws  RuntimeException on database error.
	 */
	public function render()
	{
		// Prepare some data from the model.
		$data = array(
			'count' => $this->model->getCount()
		);

		// Convert the data to JSON format.
		return json_encode($data);
	}
}

try
{
	$view = new MyView(new MyDatabaseModel, new MyController);
	echo $view->render();
}
catch (RuntimeException $e)
{
	// Handle database error.
}
```

JViewHtml

Construction

JViewHtml is extended from JViewBase. The constructor, in addition to the model and controller arguments, take an optional SplPriorityQueue object that serves as a lookup for layouts. If omitted, the view defers to the protected loadPaths method.

Usage

The JViewHtml class is abstract so cannot be used directly. This view class implements render. It will try to find the layout, include it using output buffering and return the result. The following examples show a layout file that is assumed to be stored in a generic layout folder not stored under the web-server root.

Example 2.26. Example HTML layout

```
<?php
/**
 * Example layout "layouts/count.php".
 *
 * @package  Examples
 * @since    12.1
 */

// Declare variables to support type hinting.

/** @var $this MyHtmlView */
?>

<dl>
	<dt>Count</dt>
	<dd><?php echo $this->model->getCount(); ?></dd>
</dl>
```

Example 2.27. Example HTML view

```
	/**
	 * My custom HTML view.
	 *
	 * @package  Examples
	 * @since    12.1
	 */
class MyHtmlView extends JViewHtml
{
	/**
	 * Redefine the model so the correct type hinting is available in the layout.
	 *
	 * @var     MyDatabaseModel
	 * @since   12.1
	 */
	protected $model;
}

try
{
	$paths = new SplPriorityQueue;
	$paths->insert(__DIR__ . '/layouts');

	$view = new MyView(new MyDatabaseModel, new MyController, $paths);
	$view->setLayout('count');
	echo $view->render();
```

```
}
catch (RuntimeException $e)
{
        // Handle database error.
}
```