WIZ – The web portal

PLUGINS AND EXTENSIONS

Table of Contents

Introduction

Adding new functionalities to a software during its life cycle is a common practice. This is also true for the WIZ portal, which has been provided with several functionalities to allow an IT developer with advanced computing skills to increase the potentialities of the portal with the use of plugins.

It is also possible to modify, or in general, to customise existing functionalities, not only to add new ones. This operation can not be executed by means of the plugin mechanism, but through a code "extension" tapping into the potentialities offered by the Yii framework.

In the first case, you need a basic knowledge of the framework but a good knowledge of the source code of the portal. In the second case, on the other hand, you need excellent knowledge of both the framework and the source code.

Plugins

A plugin allows to add new functionalities to the system in a simple and fast way. It is made of an archive (.tar or .zip), and its name acts as a unique ID. Below follows the structure of the plugin Foo:

```
Foo.tar                the plugin archive file
   Foo.version          ext file containing information about the plugin version
   Foo.sql                       sql file to create new tables (optional)
   Foo.php                       model class file
   FooController.php             controller class file
   views/                        folder containing view files
```

The IT developer can upload the archives that form the plugin through a simple graphical interface and enable or disable the ones previously uploaded.

The archive's structure must be necessarily like the one illustrated above, otherwise the system can not install the plugin correctly.

The subsequent paragraphs show how the structures of the several files that build the archive need to look like.

Version

The file version is a text file with a .txt extension. It is subdivided into sections, put in square brackets. The main sections is *version*, which is compulsory, and contains the number of plugin versions while the other two sections are optional:

- *Description*: contains a description of the plugin and the offered functionalities
- *Changelog*: contains information on modifications carried out in the last version (and possibly in the previous versions).

Below follows an example of a hypothetical file version for the plugin Foo.

```
[version]
1.30-r9 Beta
```

```
[description]
A short or detailed description of Foo plugin

[changelog]
The changelog
```

Sql

In the event the plugin requires new tables in the database, it is necessary to deliver a file containing a sql to create them. The portal is not concerned with the content of the file; it simply performs the sql code. It is hence the IT developer's responsibility to deliver valid and formally correct sql files, considering also the DBMS used. Below follows an example of sql files.

```
CREATE TABLE foo_table_one
(
id integer NOT NULL,
something character varying(255) NOT NULL,
CONSTRAINT id_pkey PRIMARY KEY(id)
)
```

Model

The model is a class, which has to extend CFormModel or CactiveRecord. It is used to manipulate and save data. It represents a single object, which can be either a row in a table or a html form with input fields. Every object's field is represented by a model attribute.

The name of the class has to be the same as the one used for the model file (except for the .php extension).

```
class Foo extends CActiveRecord
{
    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }

    public function tableName()
    {
        return 'foo_table_one';
    }
}
```

Controller

The controller needs to extend the CExtController class or any other controller already available in the portal. The controller performs the various actions foreseen, using possibly the model's data and displaying information through the views.

```
class FooController extends CExtController
{
    public function actionIndex()
    {
        // ...
    }
    // other actions
}
```

Views

All views need to be organised in the views folder. A view is simply a html file, which may contain a code in php.

```
<div class="top">
     <h1>Title</h1>
    <?php echo $content; ?>
</div>
```

The views are used, inside of the controller, by the different actions to display data and information to a user. The following example shows how to invoke the edit view from whatever FooController action.

```
$this->render('edit', array(
    'var1'=>$value1,
    'var2'=>$value2,
));
```

The operation bellow shows how to invoke an already existing view, which is related to the bar component.

```
$this->render('//bar/view', array(
    'var1'=>$value1,
    'var2'=>$value2,
));
```

Extensions

An extension usually serves for a single purpose. It can be classified as follows,

- application component
- behavior
- widget
- action
- filter
- controller
- validator
- helper
- module

An extension can also be a component that does not fall into any of the above categories. As a matter of fact, Yii is carefully designed such that nearly every piece of its code can be extended and customized to fit for individual needs.

Application Component

An application component should implement the interface IApplicationComponent or extend from CApplicationComponent. The main method that needs to be implemented is IApplicationComponent::init in which the component performs some initialization work. This method is invoked after the component is created and the initial property values (specified in application configuration) are applied.

By default, an application component is created and initialized only when it is accessed for the first time during request handling. If an application component needs to be created right after the application instance is created, it should require the user to list its ID in the CApplication::preload property.

To use an application component, we first need to change the application configuration by adding a new entry to its `components` property, like the following:

```
return array(
    // 'preload'=>array('xyz',...),
    'components'=>array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // other component configurations
    ),
);
```

Then, we can access the component at any place using `Yii::app()->xyz`. The component will be lazily created (that is, created when it is accessed for the first time) unless we list it the `preload` property.

Behavior

To create a behavior, one must implement the IBehavior interface. For convenience, Yii provides a base class CBehavior that already implements this interface and provides some additional convenient methods. Child classes mainly need to implement the extra methods that they intend to make available to the components being attached to.

When developing behaviors for CModel and CActiveRecord, one can also extend CModelBehavior and CActiveRecordBehavior, respectively. These base classes offer additional features that are specifically made for CModel and CActiveRecord.

The following code shows an example of an ActiveRecord behavior. When this behavior is attached to an AR object and when the AR object is being saved by calling `save()`, it will automatically set the `create_time` and`update_time` attributes with the current timestamp.

```
class TimestampBehavior extends CActiveRecordBehavior
{
    public function beforeSave($event)
    {
        if($this->owner->isNewRecord)
            $this->owner->create_time=time();
        else
```

```
            $this->owner->update_time=time();
    }
}
```

Behavior can be used in all sorts of components. Its usage involves two steps. In the first step, a behavior is attached to a target component. In the second step, a behavior method is called via the target component. For example:

```
// $name uniquely identifies the behavior in the component
$component->attachBehavior($name,$behavior);
// test() is a method of $behavior
$component->test();
```

More often, a behavior is attached to a component using a configurative way instead of calling the `attachBehavior` method. For example:

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'CDbConnection',
            'behaviors'=>array(
                'xyz'=>array(
                    'class'=>'ext.xyz.XyzBehavior',
                    'property1'=>'value1',
                    'property2'=>'value2',
                ),
            ),
        ),
        //....
    ),
);
```

The above code attaches the `xyz` behavior to the `db` application component.

For CController, CFormModel and CActiveRecord classes which usually need to be extended, attaching behaviors can be done by overriding their `behaviors()` method. For example:

```
public function behaviors()
{
    return array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzBehavior',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
    );
}
```

Widget

A widget should extend from CWidget or its child classes.

The easiest way of creating a new widget is extending an existing widget and overriding its methods or changing its default property values. For example, if you want to use a nicer CSS style for CTabView, you could configure its CTabView::cssFile property when using the widget. You can also extend CTabView as follows so that you no longer need to configure the property when using the widget.

```
class MyTabView extends CTabView
{
    public function init()
    {
        if($this->cssFile===null)
        {
            $file=dirname(__FILE__).DIRECTORY_SEPARATOR.'tabview.css';
            $this->cssFile=Yii::app()->getAssetManager()->publish($file);
        }
        parent::init();
    }
}
```

In the above, we override the CWidget::init method and assign to CTabView::cssFile the URL to our new default CSS style if the property is not set. We put the new CSS style file under the same directory containing the `MyTabView` class file so that they can be packaged as an extension. Because the CSS style file is not Web accessible, we need to publish it.

To create a new widget from scratch, we mainly need to implement two methods CWidget::init and CWidget::run. The first method is called when we use `$this->beginWidget` to insert a widget in a view, and the second method is called when we call `$this->endWidget`. If we want to capture and process the content displayed between these two method invocations, we can start output buffering in CWidget::init and retrieve the buffered output in CWidget::run for further processing.

A widget often involves including CSS, JavaScript or other resource files in the page that uses the widget. We call these files *assets* because they stay together with the widget class file and are usually not accessible by Web users. In order to make these files Web accessible, we need to publish them usingCWebApplication::assetManager, as shown in the above code snippet. Besides, if we want to include a CSS or JavaScript file in the current page, we need to register it using CClientScript

```
class MyWidget extends CWidget
{
    protected function registerClientScript()
    {
        // ...publish CSS or JavaScript file here...
        $cs=Yii::app()->clientScript;
        $cs->registerCssFile($cssFile);
        $cs->registerScriptFile($jsFile);
    }
}
```

A widget may also have its own view files. If so, create a directory named `views` under the directory containing the widget class file, and put all the view files there. In the widget class, in order to render a view, use`$this->render('ViewName')`.

Given a widget class `XyzClass` belonging to the `xyz` extension, we can use it in a view as follows,

```
// widget that does not need body content
<?php $this->widget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

// widget that can contain body content
<?php $this->beginWidget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>
```

```
...body content of the widget...

<?php $this->endWidget(); ?>
```

Action

An action should extend from CAction or its child classes. The main method that needs to be implemented for an action is IAction::run.

Actions are used by a controller to respond specific user requests. Given an action class `XyzClass` belonging to the `xyz` extension, we can use it by overriding the CController::actions method in our controller class:

```
class TestController extends CController
{
    public function actions()
    {
        return array(
            'xyz'=>array(
                'class'=>'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // other actions
        );
    }
}
```

Then, the action can be accessed via route `test/xyz`.

Filter

A filter should extend from CFilter or its child classes. The main methods that need to be implemented for a filter are CFilter::preFilter and CFilter::postFilter. The former is invoked before the action is executed while the latter after.

```
class MyFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // logic being applied before the action is executed
        return true; // false if the action should not be executed
    }

    protected function postFilter($filterChain)
    {
        // logic being applied after the action is executed
    }
}
```

The parameter `$filterChain` is of type CFilterChain which contains information about the action that is currently filtered.

Given a filter class `XyzClass` belonging to the `xyz` extension, we can use it by overriding theCController::filters method in our controller class:

```
class TestController extends CController
{
    public function filters()
    {
        return array(
            array(
                'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // other filters
        );
    }
}
```

Controller

A controller distributed as an extension should extend from CExtController, instead of CController. The main reason is because CController assumes the controller view files are located under `application.views.ControllerID`, while CExtController assumes the view files are located under the `views` directory which is a subdirectory of the directory containing the controller class file.

A controller provides a set of actions that can be requested by users. In order to use a controller extension, we need to configure the CWebApplication::controllerMap property in the application configuration:

```
return array(
    'controllerMap'=>array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // other controllers
    ),
);
```

Then, an action `a` in the controller can be accessed via route `xyz/a`.

Validator

A validator should extend from CValidator and implement its CValidator::validateAttribute method.

```
class MyValidator extends CValidator
{
    protected function validateAttribute($model,$attribute)
    {
        $value=$model->$attribute;
        if($value has error)
            $model->addError($attribute,$errorMessage);
    }
}
```

A validator is mainly used in a model class (one that extends from either CFormModel or CActiveRecord). Given a validator class `XyzClass` belonging to the `xyz` extension, we can use it by overriding the CModel::rules method in our model class:

```
class MyModel extends CActiveRecord // or CFormModel
{
    public function rules()
    {
        return array(
            array(
                'attr1, attr2',
                'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // other validation rules
        );
    }
}
```

Helper

A helper is a class with only static methods. It is like global functions using the class name as their namespace.

```
class MyHelper
{
    public static function utility($param)
    {
        // ...code here...
    }
}
```

```
// ......
```

```
MyHelper::utility($value)
// ......
```

Module

A module must have a module class that extends from CWebModule; it consists of models, views and controllers.

A general guideline for developing a module is that it should be self-contained. Resource files (such as CSS, JavaScript, images) that are used by a module should be distributed together with the module. And the module should publish them so that they can be Web-accessible.

A module is organized as a directory whose name serves as its unique ID. The following shows the typical directory structure of a module named `forum`:

```
forum/
   ForumModule.php            the module class file
   components/                containing reusable user components
      views/                  containing view files for widgets
   controllers/               containing controller class files
      DefaultController.php   the default controller class file
   extensions/                containing third-party extensions
   models/                    containing model class files
   views/                     containing controller view and layout files
      layouts/                containing layout view files
      default/                containing view files for DefaultController
         index.php            the index view file
```

To use a module, first place the module directory under `modules` of the application base directory. Then declare the module ID in the modules property of the application. For example, in order to use the above `forum` module, we can use the following application configuration:

```
return array(
    ......
    'modules'=>array('forum',...),
    ......
);
```

A module can also be configured with initial property values. For example, the `forum` module may have a property named `postPerPage` in its module class which can be configured in the application configuration as follows:

```
return array(
    ......
    'modules'=>array(
        'forum'=>array(
            'postPerPage'=>20,
        ),
    ),
    ......
);
```

The module instance may be accessed via the module property of the currently active controller. Through the module instance, we can then access information that are shared at the module level. For example, in order to access the above `postPerPage` information, we can use the following expression:

```
$postPerPage=Yii::app()->controller->module->postPerPage;
// or the following if $this refers to the controller instance
// $postPerPage=$this->module->postPerPage;
```

A controller action in a module can be accessed using the route `moduleID/controllerID/actionID`. For example, assuming the above `forum` module has a controller named `PostController`, we can use the route `forum/post/create` to refer to the `create` action in this controller. The corresponding URL for this route would be `http://www.example.com/index.php?r=forum/post/create`.