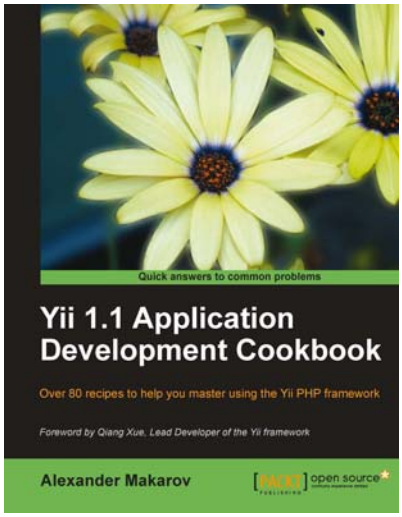
[PACKT] open source
community experience distilled
PUBLISHING

Yii 1.1 Application Development Cookbook

Alexander Makarov



Chapter No. 8
"Extending Yii"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.8 "Extending Yii"

A synopsis of the book's content

Information on where to buy this book

About the Author

Alexander Makarov graduated from Voronezh State University in 2007 with a master degree in computer science. During his study, he started working on homegrown PHP frameworks and projects trying various design patterns and techniques.

During the last year of his study, he spent a year working for Siemens mainly doing Java coding and complex SQL reports and also did many small and medium freelance projects in his free time.

In 2007, he joined an outsourcing company, Murano Software, and had a lot of experience with web development in general, J2EE, PHP, and client-side technologies working on projects such as `wrike.com` and `docufide.com`. As in previous years he did some notable freelance jobs, including social network for Russia Today built with Yii in 2009 and heavy loaded NNM.ru portal in 2008.

Between 2008 and 2010, he helped the Russian CodeIgniter community to grow and started actively to contribute to open source projects.

In 2009, Alexander joined Yii camp and started growing the Russian Yii community, translated documentation into Russian and, since May 2010, has become a passionate Yii framework core developer.

He has published several articles in Smashing Magazine and a lot more in his Russian blog `http://rmcreative.ru/`, and has presented numerous talks on Yii and web development in general at various conferences.

Alexander currently resides in Voronezh, Russia, with his beloved wife and daughter. Besides the web, he enjoys movies, rock music, travelling, photography, and languages.

I would like to thank Qiang Xue, Maurizio Domba, Sebastián Thierer, Alexander Kochetov, Antonio Ramirez Cobos, and all people who reviewed the RAW book. Your suggestions and critics helped to improve this book a lot.

I would like to thank Qiang Xue and Wei Zhuo for creating Yii.

I would also like to thank Packt Publishing for inviting me to write this book and helping me to actually get it done. I would like to thank all the past and current Yii core team members for keeping Yii in a good shape and making it better and better. You guys rock!

Yii 1.1 Application Development Cookbook

Yii is a very flexible and high-performance application development framework written in PHP. It helps building web applications from small ones to large-scale enterprise applications. The framework name stands for Yes It Is. This is often the accurate and most concise response to inquires from those new to Yii:

Is it fast? ... Is it secure? ... Is it professional? ... Is it right for my next project? ... The answer is Yes, it is!

This cookbook contains 13 independent chapters full of recipes that will show you how to use Yii efficiently. You will learn about the hidden framework gems, using core features, creating your own reusable code base, using test-driven development, and many more topics that will bring your knowledge to a new level!

What This Book Covers

Chapter 1, Under the Hood provides information about the most interesting Yii features hidden under the hood: events, import, autoloading, exceptions, component, and widget configuration, and more.

Chapter 2, Router, Controller, and Views is about handy things concerning the Yii URL router, controllers, and views: URL rules, external actions and controllers, view clips, decorators, and more.

Chapter 3, AJAX and jQuery focuses on the Yii's client side that is built with jQuery—the most widely used JavaScript library out there. It is very powerful and easy to learn and use. This chapter focuses on Yii-specific tricks rather than jQuery itself.

Chapter 4, Working with Forms. Yii makes working with forms a breeze and documentation on it is almost complete. Still, there are some areas that need clarification and examples. Some of the topics covered in this chapter are creating own validators and input widgets, uploading files, using, and customizing CAPTCHA.

Chapter 5, Testing Your Application covers both unit testing, functional testing, and generating code coverage reports. Recipes follow a test driven development approach. You will write tests for several small applications and then will implement functionality.

Chapter 6, Database, Active Record, and Model Tricks is about working with databases efficiently, when to use models and when not to, how to work with multiple databases, how to automatically pre-process Active Record fields, and how to use powerful database criteria.

Chapter 7, Using Zii Components covers data providers, grids, and lists: How to configure sorting and search, how to use grids with multiple related models, how to create your own column types, and more.

Chapter 8, Extending Yii shows not only how to implement your own Yii extension but also how to make your extension reusable and useful for the community. In addition, we will focus on many things you should do to make your extension as efficient as possible.

Chapter 9, Error Handling, Debugging, and Logging reviews logging, analyzing the exception stack trace, and own error handler implementation.

Chapter 10, Security provides information about keeping your application secure according to the general web application security principle "filter input escape output". We will cover topics such as creating your own controller filters, preventing XSS, CSRF, and SQL injections, escaping output, and using role-based access control.

Chapter 11, Performance Tuning shows how to configure Yii to gain extra performance. You will learn a few best practices of developing an application that will run smoothly until you have very high loads.

Chapter 12, Using External Code focuses on using the third party code with Yii. We will use Zend Framework, Kohana, and PEAR but you will be able to use any code after learning how it works.

Chapter 13, Deployment covers various tips that are especially useful on application deployment, when developing an application in a team, or when you just want to make your development environment more comfortable.

8
Extending Yii

In this chapter, we will cover:

- Creating model behaviors
- Creating components
- Creating reusable controller actions
- Creating reusable controllers
- Creating a widget
- Creating CLI commands
- Creating filters
- Creating modules
- Creating a custom view renderer
- Making extensions distribution-ready

Introduction

In this chapter, we will show not only how to implement your own Yii extension but how to make your extension reusable and useful for the community. In addition, we will focus on many things you should do in order to make your extension as efficient as possible.

Creating model behaviors

There are many similar solutions in today's web applications. Leading products such as Google's Gmail are defining nice UI patterns. One of these is soft delete. Instead of a permanent deletion with tons of confirmations, Gmail allows to immediately mark messages as deleted and then easily undo it. The same behavior can be applied to any object such as blog posts, comments, and so on.

Let's create a behavior that will allow marking models as deleted, restoring models, selecting not yet deleted models, deleted models, and all models. In this recipe we'll follow a test driven development approach to plan the behavior and test if the implementation is correct.

Getting ready

Carry out the following steps:

- Create a database and add a post table to your database:

```
CREATE TABLE `post` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  `title` varchar(255) default NULL,
  `is_deleted` tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (`id`)
)
```

- Configure Yii to use this database in your primary application (`protected/config/main.php`).
- Make sure the `test` application have the same settings (`protected/config/test.php`).
- Uncomment the `fixture` component in the `test` application settings.
- Use Gii to generate the `Post` model.

How to do it...

1. Let's prepare a test environment first starting with defining fixtures for the Post model in `protected/tests/fixtures/post.php`:

```
<?php
return array(
  array(
    'id' => 1,
    'title' => 'post1',
    'text' => 'post1',
    'is_deleted' => 0,
  ),
  array(
    'id' => 2,
    'title' => 'post2',
    'text' => 'post2',
    'is_deleted' => 1,
  ),
  array(
```

```
    'id' => 3,
    'title' => 'post3',
    'text' => 'post3',
    'is_deleted' => 0,
  ),
  array(
    'id' => 4,
    'title' => 'post4',
    'text' => 'post4',
    'is_deleted' => 1,
  ),
  array(
    'id' => 5,
    'title' => 'post5',
    'text' => 'post5',
    'is_deleted' => 0,
  ),
);
```

2. Then, we need to create a test case `protected/tests/unit/soft_delete/SoftDeleteBehaviorTest.php`:

```
<?php
class SoftDeleteBehaviorTest extends CDbTestCase
{
  protected $fixtures = array(
    'post' => 'Post',
  );

  function testRemoved()
  {
    $postCount = Post::model()->removed()->count();
    $this->assertEquals(2, $postCount);
  }

  function testNotRemoved()
  {
    $postCount = Post::model()->notRemoved()->count();
    $this->assertEquals(3, $postCount);
  }

  function testRemove()
  {
    $post = Post::model()->findByPk(1);
    $post->remove()->save();
```

```
    $this->assertNull(Post::model()->notRemoved()->findByPk(1));
  }

  function testRestore()
  {
    $post = Post::model()->findByPk(2);
    $post->restore()->save();

    $this->assertNotNull(Post::model
      ()->notRemoved()->findByPk(2));
  }

  function testIsDeleted()
  {
    $post = Post::model()->findByPk(1);
    $this->assertFalse($post->isRemoved());

    $post = Post::model()->findByPk(2);
    $this->assertTrue($post->isRemoved());
  }
}
```

3. Now we need to implement `behavior`, attach it to the model, and make sure the test passes. Create a new directory under `protected/extensions` named `soft_delete`. Under this directory, create `SoftDeleteBehavior.php`. Let's attach the behavior to `Post` model first:

```
class Post extends CActiveRecord
{
  // …

  public function behaviors()
  {
    return array(
      'softDelete' => array(
        'class' => 'ext.soft_delete.SoftDeleteBehavior'
      ),
    );
  }

  // …
```

4. Now let's implement `protected/extensions/soft_delete/SoftDeleteBehavior.php`:

```
<?php
class SoftDeleteBehavior extends CActiveRecordBehavior
{
  public $flagField = 'is_deleted';

  public function remove()
  {
    $this->getOwner()->{$this->flagField} = 1;
    return $this->getOwner();
  }

  public function restore()
  {
    $this->getOwner()->{$this->flagField} = 0;
    return $this->getOwner();
  }

  public function notRemoved()
  {
    $criteria = $this->getOwner()->getDbCriteria();
    $criteria->compare($this->flagField, 0);
    return $this->getOwner();
  }

  public function removed()
  {
    $criteria = $this->getOwner()->getDbCriteria();
    $criteria->compare($this->flagField, 1);
    return $this->getOwner();
  }

  public function isRemoved()
  {
    return (boolean)$this->getOwner()->{$this->flagField};
  }
}
```

   Run the test and make sure it passes.

5. That's it. We've created reusable behavior and can use it for all future projects by just connecting it to a model.

How it works...

Let's start with the test case. Since we want to use a set of models, we are defining fixtures. Fixture set is put into the DB each time the test method is executed. To use fixtures, the test class should be inherited from `CDbTestCase` and have public `$fixtures` declared:

```
protected $fixtures = array(
  'post' => 'Post',
);
```

In the preceding definition, `post` is the name of the file with fixture definitions and `Post` is the name of the model that fixtures will be applied to.

First, we are testing `removed` and `notRemoved` custom named scopes. First, we should limit the find result to removed items only, and second to non-removed items. Since we know which data we will get from fixtures, we can test for count of removed and non-removed items like the following:

```
$postCount = Post::model()->removed()->count();
$this->assertEquals(2, $postCount);
```

Then we are testing the `remove` and `restore` methods. The following is remove method test:

```
$post = Post::model()->findByPk(1);
$post->remove()->save();

$this->assertNull(Post::model()->notRemoved()->findByPk(1));
```

We are getting the item by `id`, removing it, and then trying to get it again using the `notRemoved` named scope. Since it's removed we should get `null` as result.

Finally, we are testing the `isRemoved` method that just returns the corresponding column value as Boolean.

Now let's move to the interesting implementation details. Since we are implementing the Active Record model behavior, we need to extend from `CActiveRecordBehavior`. In behavior, we can add our own methods that will be mixed into the model that behavior is attached to. We are using it to add `remove/restore/isRemoved` methods and `removed/notRemoved` named scopes:

```
public function remove()
{
  $this->getOwner()->{$this->flagField} = 1;
  return $this->getOwner();
}
public function removed()
{
```

```
    $criteria = $this->getOwner()->getDbCriteria();
    $criteria->compare($this->flagField, 1);
    return $this->getOwner();
}
```

In both methods, we are using the `getOwner` method to get the object the behavior is attached to. In our case it's a model so we can work with its data or change its finder criteria. We are returning the model instance to allow chained method calls like:

```
$post->remove()->save();
```

There's more...

There are more things that should be mentioned in this recipe.

CActiveRecordBehavior and CModelBehavior

Sometimes we need to get some more flexibility in a behavior such as reacting to model events. Both `CActiveRecordBehavior` and `CModelBehavior` are adding event-like methods we can override to handle model events. For example, if we need to handle cascade delete in a behavior we can do it by overriding the `afterDelete` method.

More behavior types

Behavior can be attached not only to a model but to any component. Each behavior inherits from `CBehavior` class so we can use its methods:

- `getOwner` to get the component that the behavior is attached to.
- `getEnabled` and `setEnabled` to check if behavior is enabled and set its state.
- `attach` and `detach` can be correspondingly used to initialize behavior and clean up temporary data created during behavior usage.

Further reading

To learn more about behaviors refer to the following API pages:

- `http://www.yiiframework.com/doc/api/CActiveRecordBehavior`
- `http://www.yiiframework.com/doc/api/CModelBehavior`
- `http://www.yiiframework.com/doc/api/CBehavior`

See also

- *Making extensions distribution-ready in this chapter*

Creating components

If you have some code that looks like it can be reused but you don't know if it's a behavior, widget, or something else, most probably it's a component. Component should be inherited from `CComponent` or `CApplicationComponent`. Later on the component can be attached to the application and configured using `protected/config/main.php` configuration file. That's the main benefit compared to using just a plain PHP class. Additionally we are getting behaviors, events, getters, and setters support.

For our example, we'll implement a simple `EImageManager` application component that will be able to resize images using the GD library, attach it to the application, and use it.

Getting ready

You will need to install the GD PHP extension to see image resizing in action.

How to do it...

Carry out the following steps:

1. Create `protected/components/EImageManager.php`:

```
<?php
class EImageManager extends CApplicationComponent
{
  protected $image;
  protected $width;
  protected $height;

  protected $newWidth;
  protected $newHeight;

  public function resize($width = false, $height = false){
    if($width!==false) $this->newWidth = $width;
    if($height!==false) $this->newHeight = $height;

    return $this;
  }

  public function load($filePath)
  {
    list($this->width, $this->height, $type) =
      getimagesize($filePath);

    switch ($type)
    {
```

```
      case IMAGETYPE_GIF:
        $this->image = imagecreatefromgif($filePath);
      break;
      case IMAGETYPE_JPEG:
        $this->image = imagecreatefromjpeg($filePath);
      break;
      case IMAGETYPE_PNG:
        $this->image = imagecreatefrompng($filePath);
      break;
      default:
        throw new CException('Unsupported image type ' . $type);
    }

    return $this;
  }

  public function save($filePath)
  {

    $ext = pathinfo($filePath, PATHINFO_EXTENSION);

    $newImage = imagecreatetruecolor($this->newWidth,
      $this->newHeight);
    imagecopyresampled($newImage, $this->image, 0, 0, 0, 0,
      $this->newWidth, $this->newHeight, $this->width,
      $this->height);

    switch($ext)
    {
      case 'jpg':
      case 'jpeg':
        imagejpeg($newImage, $filePath);
      break;
      case 'png':
        imagepng($newImage, $filePath);
      break;
      case 'gif':
        imagegif($newImage, $filePath);
      break;
      default:
        throw new CException("Unsupported image type ", $ext);
    }
```

```
        imagedestroy($newImage);

        if(!is_file($filePath))
          throw new CException("Failed to write image.");
      }

      function __destruct()
      {
        imagedestroy($this->image);
      }
    }
```

2. Now we need to attach our component to the application. In `protected/config/main.php` we need to add the following:

```
…
// application components
'components'=>array(
  'image' => array(
    'class' => 'EImageManager',
  ),
…
```

3. Now we can use it like the following:

```
Yii::app()->image
  ->load(Yii::getPathOfAlias('webroot').'/src.png')
  ->resize(100,100)
  ->save(Yii::getPathOfAlias('webroot').'/dst.png');
```

How it works...

To be able to attach a component to an application it needs to be extended from `CApplicationComponent`. Attaching is as simple as adding a new array to the `components` section of configuration. There, a `class` value specifies the component's class and all other values are set to a component through the corresponding component's public properties and setter methods.

Implementation itself is very straightforward: We are wrapping GD calls into comfortable API with `save`, `load`, and `resize` methods. To be able to chain API calls, `load` and `resize` are returning the component itself.

We can access our class by its component name using `Yii::app()`. In our case it will be `Yii::app()->image`.

There's more...

Besides creating your own components, you can do more.

Overriding existing application components

Most of the time there will be no need to create your own application components since other types of extensions such as widgets or behaviors are covering almost all types of reusable code. However, overriding core framework components is a common practice and can be used to customize the framework's behavior for your specific needs without hacking into the core.

For example, to be able to get the user's role from the database using `Yii::app()->user->role` you can extend the `CWebUser` component like the following:

```
<?php
class WebUser extends CWebUser {
    private $_model = null;

    function getRole() {
        if($user = $this->getModel()){
            return $user->role;
        }
        else return 'guest';
    }

    private function getModel(){
        if($this->_model === null){
            if($this->id === null) return null;
            $this->_model = User::model()->findByPk($this->id);
        }
        return $this->_model;
    }
}
```

To replace the standard user component `main.php` configuration should be customized:

```
…
// application components
'components'=>array(
  'user'=>array(
    'class' => 'WebUser',
    // other properties
  ),
…
```

In the preceding code, we specified a new `class` for the `user` component.

Further reading

In order to learn more about components, refer to the following API pages:

- `http://www.yiiframework.com/doc/api/CComponent/`
- `http://www.yiiframework.com/doc/api/CApplicationComponent/`

See also

- The recipe named *Making extensions distribution-ready* in this chapter

Creating reusable controller actions

Common actions such as deleting the AR model by the primary key or getting data for AJAX autocomplete could be moved into reusable controller actions and later attached to controllers as needed.

In this recipe, we will create reusable `delete` action that will delete the specified AR model by its primary key.

Getting ready

- Create a fresh Yii application using `yiic webapp`.
- Create a new database and configure it.
- Execute the following SQL:

```
CREATE TABLE `post` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  `title` varchar(255) default NULL,
  PRIMARY KEY  (`id`)
);

CREATE TABLE `comment` (
  `id` int(11) NOT NULL auto_increment,
  `text` text,
  PRIMARY KEY  (`id`)
);
```

- Generate models for `post` and `comment` using Gii.

How to do it...

Carry out the following steps:

1. Create `protected/extensions/actions/EDeleteAction.php`:

```
<?php
class EDeleteAction extends CAction
{
  public $modelName;
  public $redirectTo = array('index');

  /**
   * Runs the action.
   * This method is invoked by the controller owning this action.
   */
  public function run($pk)
  {
    CActiveRecord::model($this->modelName)->deleteByPk($pk);
    if(Yii::app()->getRequest()->getIsAjaxRequest())
    {
      Yii::app()->end(200, true);
    }
    else
    {
      $this->getController()->redirect($this->redirectTo);
    }
  }
}
```

2. Now we need to attach it to the controller `protected/controllers/DeleteController.php`:

```
<?php
class DeleteController extends CController
{
  public function actions()
  {
    return array(
      'deletePost' => array(
        'class' => 'ext.actions.EDeleteAction',
        'modelName' => 'Post',
        'redirectTo' => array('indexPosts'),
      ),
      'deleteComment' => array(
        'class' => 'ext.actions.EDeleteAction',
        'modelName' => 'Comment',
```

```
            'redirectTo' => array('indexComments'),
          ),
        );
    }

    public function actionIndexPosts()
    {
      echo "I'm index action for Posts.";
    }

    public function actionIndexComments()
    {
      echo "I'm index action for Comments.";
    }
}
```

3. That is it. Now you can delete a post by visiting `/delete/deletePost/pk/<pk>` and delete a comment by visiting `/delete/deleteComment/pk/<pk>`. After the deletion, you will be redirected to a corresponding index action.

How it works...

To create an external controller action you need to extend your class from `CAction`. The only mandatory method to implement is `run`. In our case, it accepts the parameter named `pk` from `$_GET` using the automatic parameter binding feature of Yii and tries to delete a corresponding model.

To make it customizable we've created two public properties configurable from the controller. These are `modelName` that holds a name of the model we are working with, and `redirectTo` that specifies a route the user will be redirected to.

Configuration itself is done by implementing the `actions` method in your controller. There, you can attach the action once or multiple times and configure its public properties.

There's more...

There are two usable methods implemented in `CAction`. First is `getController` that we can use to get an instance of the controller that the action is attached to. You will need it to redirect to another action or, for example, generate URL.

Another method is `getId` that returns action name specified in the controller's `actions` method.

Further reading

To learn more about external controller action refer to the following API pages:

- `http://www.yiiframework.com/doc/api/CAction/`
- `http://www.yiiframework.com/doc/api/CController/#actions-detail`

See also

- The recipe named *Creating reusable controllers* in this chapter
- The recipe named *Making extensions distribution-ready* in this chapter

Creating reusable controllers

In Yii, you can create reusable controllers. If you are creating a lot of applications or controllers that are of the same type, moving all common code into a reusable controller will save you a lot of time.

In this recipe, we will create a simple reusable `api` controller that will implement a simple JSON CRUD API for a model. It will take input data from POST and GET and will respond with JSON data and a corresponding HTTP response code.

Getting ready

- Create a fresh Yii application using `yiic webapp`.
- Generate a model using Gii. In our example we'll use the `Post` model but you can use any model you want.

How to do it...

Carry out the following steps:

1. Create `protected/extensions/json_api/JsonApiController.php`:

```
<?php
class JsonApiController extends CController
{
  const RESPONSE_OK = 'OK';
  const RESPONSE_NO_DATA = 'No data';
  const RESPONSE_NOT_FOUND = 'Not found';
  const RESPONSE_VALIDATION_ERRORS = 'Validation errors';

  public $modelName;

  public function init()
  {
```

```
    parent::init();
    if(empty($this->modelName))
      throw new CException("You should set modelName before
        using JsonApiController.");
  }

  public function actionCreate()
  {
    if(empty($_POST))
      $this->respond(400, self::RESPONSE_NO_DATA);

    $model = new $this->modelName;
    $model->setAttributes($_POST);

    if($model->save())
      $this->respond(200, self::RESPONSE_OK);
    else
      $this->respond(400, self::RESPONSE_VALIDATION_ERRORS,
        $model->getErrors());
  }

  public function actionGet($pk)
  {
    $model = CActiveRecord::model
      ($this->modelName)->findByPk($pk);
    if(!$model)
      $this->respond(404, self::RESPONSE_NOT_FOUND);

    $this->respond(200, self::RESPONSE_OK,
      $model->getAttributes());
  }

  public function actionUpdate($pk)
  {
    if(empty($_POST))
      $this->respond(400, self::RESPONSE_NO_DATA);

    $model = CActiveRecord::model
      ($this->modelName)->findByPk($pk);
    if(!$model)
      $this->respond(404, self::RESPONSE_NOT_FOUND);

    $model->setAttributes($_POST);
    if($model->save())
      $this->respond(200, self::RESPONSE_OK);
```

```
    else
      $this->respond(400, self::RESPONSE_VALIDATION_ERRORS,
        $model->getErrors());
  }

  public function actionDelete($pk)
  {
    if(CActiveRecord::model($this->modelName)->deleteByPk($pk))
    {
      $this->respond(200, self::RESPONSE_OK);
    }
    else {
      $this->respond(404, self::RESPONSE_NOT_FOUND);
    }
  }

  protected function respond($httpCode, $status, $data = array())
  {
    $response['status'] = $status;
    $response['data'] = $data;

    echo CJSON::encode($response);

    Yii::app()->end($httpCode, true);
  }
}
```

2. Now we need to connect it to our application via `protected/config/main.php`. It can be done by adding controller configuration to the `controllerMap` property of `CWebApplication` so we need to place the following right after the configuration array opening:

```
…
'controllerMap' => array(
  'api' => array(
    'class' => 'ext.json_api.JsonApiController',
    'modelName' => 'Post',
  ),
),
…
```

3. That is it. We have connected the controller and specified that it should work with the `Post` model.

You will need forms to post data but if you have some data already you can use get methods right from a URL like `/api/get/pk/1`. Applications should return data like the following:

```
{"status":"OK","data":{"id":"1","text":"post1",
  "title":"post1","is_deleted":"0"}}
```

How it works...

When you are running an application and passing a route such as `api/get`, prior to executing `ApiController::actionGet` Yii checks if there is `controllerMap` defined. Since we have an `api` controller defined there, Yii executes it instead of going the usual way.

In the controller itself we've defined the `modelName` property to be able to connect the controller multiple times and have an API for multiple models. We are setting it when attaching the controller.

The controller itself isn't that much different from a regular one with only a few tricks:

- When working with Active Record we are creating a new class like the following:

  ```
  $model = new $this->modelName;
  ```

- And getting model finder like the following:

  ```
  $model = CActiveRecord::model($this->modelName);
  ```

- This allows us not to be bound to a specific model but to specify a model name.
- We're using `CJSON` to encode arrays and models to JSON.
- `Yii::app()->end()` is used to end application execution with HTTP response code specified.

There's more...

In order to learn more about the controllers map, refer to the following API page:

`http://www.yiiframework.com/doc/api/CWebApplication#controllerMap-detail`

See also

- The recipe named *Creating reusable controller actions* in this chapter
- The recipe named *Making extensions distribution-ready* in this chapter

Creating a widget

A widget is a reusable part of a view that not only renders some data but does it according to some logic. It can even get data from models and use its own views, so it is like a reduced reusable version of a module.

Let's create a widget that will draw a pie chart using Google APIs.

Getting ready

Create a fresh Yii application using `yiic`.

How to do it...

1. Create `protected/extensions/chart/EChartWidget.php`:

```
<?php
class EChartWidget extends CWidget
{
  public $title;
  public $data=array();
  public $labels=array();

  public function run()
  {
    echo "<img
      src=\"http://chart.apis.google.com/chart?chtt=".urlencode
      ($this->title)."&cht=pc&chs=300x150&chd=".
      $this->encodeData($this->data)."&chl=".implode
      ('|', $this->labels)."\">";
  }

  protected function encodeData($data)
  {
    $maxValue=max($data);

    $chars='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx
      yz0123456789';

    $chartData="s:";
    for($i=0;$i<count($data);$i++)
    {
      $currentValue=$data[$i];
```

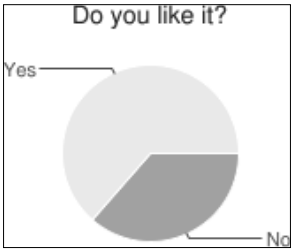
```
        if($currentValue>-1)
          $chartData.=substr($chars,61*($currentValue/$maxValue),1);
        else
          $chartData.='_';
      }

      return $chartData."&chxt=y&chxl=0:|0|".$maxValue;
    }
}
```

2. Now create a controller `protected/controllers/ChartController.php`:

```
<?php
class ChartController extends CController
{
  public function actionIndex()
  {
    $value = rand(10, 90);
    $this->widget('ext.chart.EChartWidget', array(
      'title' => 'Do you like it?',
      'data' => array(
        $value, 100-$value
      ),
      'labels' => array(
        'No',
        'Yes',
      ),
    ));
  }
}
```

3. Now try to run the index action of the controller. You should see a pie chart like the following:



How it works...

Like in every other type of extension, we are creating some public properties we can configure when calling a widget using `CController::widget`. In this case, we are configuring title, data set, and data labels.

The main method of a widget is `run()`. In our widget, we are generating a URL pointing to the Google charting API, and then echoing `<img>` tag.

There's more...

To learn more about widgets refer to the following API pages:

- `http://www.yiiframework.com/doc/api/CWidget/`
- `http://www.yiiframework.com/doc/api/CCaptcha/`

See also

- The recipe named *Configuring components* in *Chapter 1*, *Under the Hood*
- The recipe named *Configuring widget defaults* in *Chapter 1*
- The recipe named *Creating custom input widget with CWidget* in *Chapter 4*, *Working with Forms*
- The recipe named *Making extensions distribution-ready* in this chapter

Creating CLI commands

Yii has a good command-line support and allows creating reusable console commands. Console commands are faster to create than web GUIs. If you need to create some kind of utility for your application that will be used by developers or administrators, console commands are the right tool. To show how to create a console command we'll create a simple command that will clean up various things such as cache, temp directories, and so on.

Getting ready

Create a fresh Yii application using `yiic webapp`.

How to do it...

Carry out the following steps:

1. Create `protected/extensions/clean_command/ECleanCommand.php`:

```
<?php
class ECleanCommand extends CConsoleCommand
{
```

```
public $webRoot;
public function actionCache()
{
  $cache=Yii::app()->getComponent('cache');
      if($cache!==null){
          $cache->flush();
          echo "Done.\n";
      }
      else {
          echo "Please configure cache component.\n";
      }
}

public function actionAssets()
{
  if(empty($this->webRoot))
  {
    echo "Please specify a path to webRoot in command
      properties.\n";
    Yii::app()->end();
  }

  $this->cleanDir($this->webRoot.'/assets');

      echo "Done.\n";
}

public function actionRuntime()
{
  $this->cleanDir(Yii::app()->getRuntimePath());
  echo "Done.\n";
}

private function cleanDir($dir)
{
  $di = new DirectoryIterator ($dir);
      foreach($di as $d)
      {
          if(!$d->isDot())
          {
              echo "Removed ".$d->getPathname()."\n";
              $this->removeDirRecursive($d->getPathname());
          }
      }
}
```

```
  private function removeDirRecursive($dir)
  {
        $files = glob($dir.'*', GLOB_MARK);
        foreach ($files as $file)
        {
                if (is_dir($file))
                        $this-
        >removeDirRecursive($file);
                else
                        unlink($file);
                }
                if (is_dir($dir))
                        rmdir($dir);
  }
}
```

2. Now we need to attach a command to the console application. By default the console application uses a separate bootstrap file `yiic.php`:

```
<?php

// change the following paths if necessary
$yiic=dirname(__FILE__).'/../../framework/yiic.php';
$config=dirname(__FILE__).'/config/console.php';

require_once($yiic);
```

3. Therefore, the configuration is `protected/config/console.php`. Let's add our console command to the `commandMap` property:

```
// This is the configuration for yiic console application.
// Any writable CConsoleApplication properties can be configured
here.
return array(
  'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..',
  'name'=>'My Console Application',

  'commandMap' => array(
    'clean' => array(
      'class' => 'ext.clean_command.ECleanCommand',
      'webRoot' => 'path/to/your/application/webroot',
    ),
  ),
);
```

4. Don't forget to change `webRoot` to the actual path. That's it. Now go to protected directory and try running these commands:

```
yiic clean
yiic clean cache
yiic clean assets
yiic clean runtime
```

How it works...

All console commands should be extended from the `CConsoleCommand` class. Since all console commands are run in `CConsoleApplication` instead of `CWebApplication`, we don't have a way to determine the web application server root. For this purpose we are creating a configurable public property called `webRoot`.

The console command structure itself is like a typical controller. We are defining several actions we can run via `yiic <console command> <command action>`.

As you can see, there are no views used so we can focus on programming tasks instead of design, markup, and so on. Still, you need to provide some useful output so users will know what is going on. This is done through simple PHP echo statements.

There's more...

Custom help

If your command is relatively complex such as `message` or `migrate` bundled with Yii, it's a good decision to provide some extra description of the available options and actions. It can be done by overriding the `getHelp` method:

```
public function getHelp()
{
  $out = "Clean command allows you to clean up various
    temporary data Yii and an application are generating.\n\n";
  return $out.parent::getHelp();
}
```

Running just `yiic clean` will output:

```
Clean command allows you to clean up various temporary data Yii and an applicati
on are generating.

Usage: W:\home\server.local\www\app\yiic.php clean <action>
Actions:
    cache
    assets
    runtime
```

Further reading

To learn more about creating console applications and commands refer to the following:

- The recipe named `http://www.yiiframework.com/doc/guide/topics.console`
- `http://www.yiiframework.com/doc/api/CConsoleCommand/`
- `http://www.yiiframework.com/doc/api/CConsoleApplication/`

See also

- The recipe named *Making extensions distribution-ready* in this chapter

Creating filters

A filter is a class that can run before/after an action is executed. It can be used to modify execution context or decorate output. In our example we'll implement a simple output filter that will compress HTML output removing all optional formatting.

Don't use this filter in production: GZIP will do a better job in reducing bandwidth. Moreover, it is not safe to remove formatting for pre, textarea, among others.

Getting ready

Create a fresh Yii application using `yiic`.

How to do it...

Carry out the following steps:

1. Create `protected/extensions/compress_html/ECompressHtmlFilter.php`:

```
<?php
class ECompressHtmlFilter extends CFilter
{
  protected function preFilter($filterChain)
  {
    ob_start();
    return parent::preFilter($filterChain);
  }

  protected function postFilter($filterChain)
  {
    $out = ob_get_clean();
```

```
        echo preg_replace("~>(\s+|\t+|\n+)<~", "><", $out);
        parent::postFilter($filterChain);
    }
}
```

2. That's it. Now we need to connect it to application. Since we already have `SiteController` let's add filter to it by overriding `filters` method:

```
public function filters()
{
  return array(
    array(
      'ext.compress_html.ECompressHtmlFilter'
    ),
  );
}
```

3. Now run application and check HTML source code. It should be a single line of text such as:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.
dtd"><html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en"><head><meta http-equiv="Content-Type" content="text/
html; charset=utf-8" /><meta name="language" content="en" /><!--
blueprint CSS framework -->
…
```

How it works...

A filter should at minimum implement the `IFilter` interface but since we want to do post- and pre-filtering, we can directly extend `CFilter`. The plan is to somehow get all of the content generated by an action and do some processing prior to sending it to a browser. PHP's output buffering fits well for this task, so we override `preFilter` to start buffering by calling `ob_start` there and `postFilter` to get the buffer content with `ob_get_clean`, and do processing with a regular expression and echo result.

Note that we are calling parent methods when overriding `preFilter` and `postFilter`. It allows the developer to build a chain of filters when configuring controller.

There's more...

In order to learn more about filters, refer to the following API pages:

- `http://www.yiiframework.com/doc/api/CFilter`
- `http://www.yiiframework.com/doc/api/IFilter`

See also

- The recipe named *Making extensions distribution-ready* in this chapter

Creating modules

If you have created a complex application part and want to use it with some degree of customization in your next project, most probably you need to create a module.

In this recipe, we will see how to create a wiki module. For simplicity, we will not focus on the user and permissions management and let everyone edit everything.

Getting ready

Carry out the following steps:

1. Create a fresh Yii application using `yiic webapp`.
2. Configure MySQL database and execute the following SQL:

```
CREATE TABLE `wiki` (
  `id` varchar(255) NOT NULL,
  `text` text NOT NULL,
  PRIMARY KEY  (`id`)
)
```

3. Generate `Wiki` model using Gii.
4. Generate `Wiki` module using Gii.
5. Move `protected/models/Wiki.php` to `protected/modules/wiki/models/`.
6. Add `wiki` to `modules` section of `protected/config/main.php`:

```
'modules'=>array(
  // uncomment the following to enable the Gii tool
  'gii'=>array(

    'class'=>'system.gii.GiiModule',
    'password'=>false,
  ),
  'wiki'
),
```

How to do it...

Let's do some planning first:

- A wiki is a set of pages where one page can link to another using its name
- Typically wiki uses simpler human-readable markup instead of HTML
- If a user goes to a page that doesn't yet exist he's prompted to create one
- To delete a page a user needs to save it with an empty body

Based on this planning we'll use markdown as markup language. We will add a way to link to another page by name. Let's say it will be `[[page name]]` or `[[Custom title|page name]]` if you need a custom titled link.

1. First, let's add wiki links to `CMarkdownParser`. Create `protected/modules/wiki/components/WikiMarkdownParser.php`:

```
<?php
class WikiMarkdownParser extends CMarkdownParser
{
  public function transform($text)
  {
    $text = preg_replace_callback('~\[\[(.*?)(?:\|(.*?))?\]\]~',
      array($this, 'processWikiLinks'), $text);
    return parent::transform($text);
  }

  protected function processWikiLinks($matches)
  {
    $page = $matches[1];
    $title = isset($matches[2]) ? $matches[2] : $matches[1];
    return CHtml::link(CHtml::encode($title), array(
      'view', 'id' => $page,
    ));
  }
}
```

2. Now let's use it by adding the `getHtml` method to `protected/modules/wiki/models/Wiki.php` model:

```
public function getHtml()
{
  $parser = new WikiMarkdownParser();
  return $parser->transform($this->text);
}
```

We're moving to customizing the `protected/modules/wiki/controller/DefaultController.php` controller. We'll need just two actions there: `view` action and `edit` action. Additionally we'll create an `index` action that will just call view with `id = index`. Since we don't need a view for the `index` action it can be deleted safely.

1. Now edit `protected/modules/wiki/controller/DefaultController.php`:

```
class DefaultController extends Controller
{
  public function actionIndex()
  {
    $this->actionView('index');
  }

  public function actionView($id)
  {
    $model = Wiki::model()->findByPk($id);
    if(!$model)
    {
      $this->actionEdit($id);
      Yii::app()->end();
    }

    $this->render('view', array(
      'model' => $model,
    ));
  }

  public function actionEdit($id)
  {
    $model = Wiki::model()->findByPk($id);
    if(!$model)
    {
      $model = new Wiki();
      $model->id = $id;
    }

    if(!empty($_POST['Wiki']))
    {
      if(!empty($_POST['Wiki']['text']))
      {
        $model->text = $_POST['Wiki']['text'];
        if($model->save())
          $this->redirect(array('view', 'id' => $id));
      }
```

```
        else
        {
          Wiki::model()->deleteByPk($id);
        }
      }

      $this->render('edit', array(
        'model' => $model
      ));
    }
  }
```

2. In addition, we will need two views. `protected/modules/wiki/views/default/view.php`:

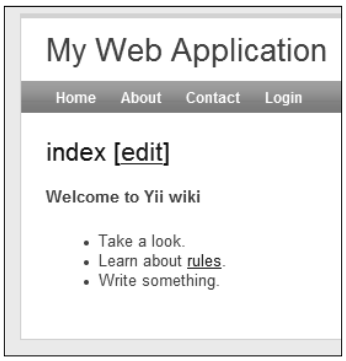
```
<h2>
  <?php echo CHtml::encode($model->id)?>
  [<?php echo CHtml::link('edit', array('edit', 'id' =>
    $model->id))?>]
</h2>
<?php echo $model->html ?>
```

3. Another one `protected/modules/wiki/views/default/edit.php`:

```
<h2>Editing <?php echo CHtml::encode($model->id)?></h2>

<?php echo CHtml::beginForm()?>
  <?php echo CHtml::activeTextArea($model, 'text',
    array('cols' => 100, 'rows' => 20))?>
  <br /><br />
  <?php echo CHtml::submitButton('Done')?>
<?php echo CHtml::endForm()?>
```

4. That is it. Now run the wiki module and check it out entering some text. Don't forget to add internal links like `[[rules]]`:



5. The preceding page will look like the following when being edited:

6. Since we moved the `Wiki` model from application to module, the module doesn't have any dependencies on the application itself, so we can move it to an extension package like `protected/extensions/wiki/`, changing the module configuration `protected/config/main.php` to:

```
'modules'=>array(
  // uncomment the following to enable the Gii tool
  'gii'=>array(
    'class'=>'system.gii.GiiModule',
    'password'=>false,
  ),
  'wiki'=>array(
    'class' => 'ext.wiki.WikiModule'
  ),
),
```

How it works...

Each module created contains a main module class like `WikiModule` where we can define configurable properties, define imports, change paths, attach controllers, and so on. By default, a module generated with Gii runs `index` action of the `default` controller:

```
public function actionIndex()
{
  $this->actionView('index');
}
```

In our `wiki` module `index` we are just calling `view` action passing `id = index` to it:

```
$model = Wiki::model()->findByPk($id);
if(!$model)
{
  $this->actionEdit($id);
  Yii::app()->end();
}

$this->render('view', array(
  'model' => $model,
));
```

If there is a model with such an ID we are displaying it using a view.

If there is no page with such an ID, we are, again, delegating processing to another action. This time it's `edit`:

```
$model = Wiki::model()->findByPk($id);
if(!$model)
{
  $model = new Wiki();
  $model->id = $id;
}

if(!empty($_POST['Wiki']))
{
  if(!empty($_POST['Wiki']['text']))
  {
    $model->text = $_POST['Wiki']['text'];
    if($model->save())
      $this->redirect(array('view', 'id' => $id));
  }
  else
  {
    Wiki::model()->deleteByPk($id);
  }
}

$this->render('edit', array(
  'model' => $model
));
```

If there is no model with an ID passed we're creating new one; if there is a model, we are editing it. Edit form data comes from POST and if a text is empty we are deleting a model. If there is a text we are saving a model.

There's more...

To learn more about modules refer to the following:

- `http://www.yiiframework.com/doc/guide/basics.module`
- `http://www.yiiframework.com/doc/api/CWebModule/`
- `http://www.yiiframework.com/doc/api/CModule/`
- `http://www.yiiframework.com/doc/api/GiiModule/`

See also

- The recipe named *Making extensions distribution-ready* in this chapter

Creating a custom view renderer

There are many PHP template engines out there. Yii offers only two template types out of the box: native PHP and Prado-like templates. If you want to use one of the existing template engines or create your own one you have to implement it. Of course, if it's not yet implemented by the Yii community

In this recipe we'll implement Smarty templates support.

Getting ready

- Create a fresh Yii application using `yiic webapp`.
- Get the latest Smarty 3 release from `http://www.smarty.net/`.
- Extract contents of `libs` directory to `protected/vendors/smarty/`.

How to do it...

Carry out the following steps:

1. Create `protected/extensions/smarty/ESmartyViewRenderer.php`:

```
<?php
class ESmartyViewRenderer extends CApplicationComponent
  implements IViewRenderer
{
    public $fileExtension='.tpl';
    public $filePermission=0755;

    private $smarty;
```

```
    function init()
  {
        Yii::import('application.vendors.smarty.*');

        spl_autoload_unregister(array('YiiBase','autoload'));
        require_once('Smarty.class.php');
        spl_autoload_register(array('YiiBase','autoload'));

        $this->smarty = new Smarty();

        $this->smarty->template_dir = '';
        $compileDir = Yii::app()->getRuntimePath
          ().'/smarty/compiled/';

        if(!file_exists($compileDir)){
            mkdir($compileDir, $this->filePermission, true);
        }

        $this->smarty->compile_dir = $compileDir;
        $this->smarty->assign('Yii', Yii::app());
    }

    /**
   * Renders a view file.
   * This method is required by {@link IViewRenderer}.
   * @param CBaseController the controller or widget who is
       rendering the view file.
   * @param string the view file path
   * @param mixed the data to be passed to the view
   * @param boolean whether the rendering result should be
       returned
   * @return mixed the rendering result, or null if the rendering
       result is not needed.
   */
  public function renderFile($context,$sourceFile,$data,$return) {
        // current controller properties will be accessible as
{this.property}
        $data['this'] = $context;

        if(!is_file($sourceFile) || ($file=realpath($sourceFile))=
          ==false)
            throw new CException(Yii::t('ext','View file
"$sourceFile" does not exist.', array('{file}'=>$sourceFile)));

    $this->smarty->assign($data);

    if($return)
        return $this->smarty->fetch($sourceFile);
```

```
    else
      $this->smarty->display($sourceFile);
  }
}
```

2. Now we need to connect view rendered to application. In `protected/config/main.php` we need to override the `viewRenderer` component:

```
…
// application components
'components'=>array(
  …
  'viewRenderer'=>array(
    'class'=>'ext.smarty.ESmartyViewRenderer',
  ),
),
…
```

3. Now let's test it. Create `protected/controllers/SmartyController.php`:

```
<?php
class SmartyController extends Controller
{
  function actionNative()
  {
    $this->render('native', array(
      'username' => 'Alexander',
    ));
  }

  function actionSmarty()
  {
    $this->render('smarty', array(
      'username' => 'Alexander',
    ));
  }
}
```

4. Now we need views. `protected/views/smarty/native.php`:

```
Hello, <?php echo $username?>!
```

`protected/views/smarty/smarty.tpl`:

```
Hello, {$username}!
```

5. Now try running controller actions. In both cases you should get:

Hello, Alexander!

How it works...

A view renderer is a child of `CApplicationComponent` that implements the `IViewRenderer` interface with only one method called `renderFile`:

```
/**
 * Renders a view file.
 * @param CBaseController $context the controller or widget who is
     rendering the view file.
 * @param string $file the view file path
 * @param mixed $data the data to be passed to the view
 * @param boolean $return whether the rendering result should be
returned
 * @return mixed the rendering result, or null if the rendering result
is
    not needed.
 */
public function renderFile($context,$file,$data,$return);
```

Therefore, we are getting context, template path, data, and return flag that determines if we need to echo the processing result immediately or return it. In our case processing itself is done by the Smarty template engine so we need to properly initialize it and call its processing methods.

Importing and initializing Smarty is a bit tricky:

```
Yii::import('application.vendors.smarty.*');

spl_autoload_unregister(array('YiiBase','autoload'));
require__once('Smarty.class.php');
spl_autoload_register(array('YiiBase','autoload'));

$this->smarty = new Smarty();

$this->smarty->template_dir = '';
$compileDir = Yii::app()->getRuntimePath().'/smarty/compiled/';

if(!file_exists($compileDir)){
  mkdir($compileDir, $this->filePermission, true);
}

$this->smarty->compile_dir = $compileDir;
$this->smarty->assign('Yii', Yii::app());
```

First we are using `Yii::import` to tell Yii we want to use classes from `protected/vendors/smarty/`. Since Yii autoloader conflicts with the one included in Smarty and we can't control how Smarty does it, we need to unregister Yii's one, require Smarty and then register Yii autoloader back. Since the Yii template directory can vary, we are resetting the Smarty default option to an empty string. It is a good practice to store Yii temporary files in the application runtime directory. That is why we are setting the compile directory, where Smarty stores its templates compiled into PHP, to `runtime/smarty/compiled`. Not to disturb developers too much we are creating this directory automatically if it does not exist. Also, we are creating a special Smarty template variable named Yii that points to `Yii::app()` and allows to get application properties inside of a template.

Rendering itself is a bit simpler:

```
// current controller properties will be accessible as {this.property}
$data['this'] = $context;

if(!is_file($sourceFile) || ($file=realpath($sourceFile))===false)
  throw new CException(Yii::t('ext','View file "$sourceFile" does not
exist.', array('{file}'=>$sourceFile)));

$this->smarty->assign($data);

if($return)
  return $this->smarty->fetch($sourceFile);
else
  $this->smarty->display($sourceFile);
```

We are assigning another Smarty variable called `this` that will allow getting controller properties such as page title.

To ease debugging, the template is checked for existence and if it doesn't exist an error message with the template name is displayed.

All data set via `$this->render` is passed to the Smarty template as is. Then we are either rendering templates or returning it depending on the `$return` argument.

There's more...

You can get ready to use Smarty view renderer with plugins and configuration support at `http://www.yiiframework.com/extension/smarty-view-renderer`.

It's OK to use a custom template engine in your application. However, if you are creating a reusable extension try to avoid using template engines other than the native PHP.

Further reading

To learn more about Smarty and view renderers in general refer to the following:

- `http://www.smarty.net/`
- `http://www.yiiframework.com/doc/api/IViewRenderer/`
- `http://www.yiiframework.com/doc/api/CViewRenderer/`
- `http://www.yiiframework.com/doc/api/CPradoViewRenderer/`

See also

- The recipe named *Making extensions distribution-ready* in this chapter

Making extensions distribution-ready

In this chapter, you have learned how to create various types of Yii extensions. Now we'll talk about how to share your results with people and why it's important.

Getting ready

Let's form a checklist for a good extension first. A good programming product should follow these points:

- Consistent, easy to read and use API
- Good documentation
- People should be able to find it
- Extension should apply to the most common use cases
- Should be maintained
- Well-tested code; ideally with unit tests
- You need to provide support for it

Of course, having all these requires a lot of work but these are necessary to create a good product.

How to do it...

Let's review our list in more detail starting with API. API should be consistent, easy to read and use. Consistent means that overall style should not change so no different variable naming such as `camelCasedVariableNames` and `underscored_variable_names`, no inconsistent names such as `isFlag1()` and `isNotFlag2()`, and so on. Everything should obey the rules you've defined for your code. This allows less checking of documentation and focusing on coding.

Code without any documentation is almost useless. An exception is relatively simple code but even if it's only a few lines it doesn't feel right if there is not a single word about how to install and use it. What makes a good documentation?

The purpose of the code and its pros. It should be as visible as possible and should be written loud and clear. For example, if an extension is sending an e-mail, the description should be something like:

> *EMailer allows you to send an e-mail in an easy and convenient way. It handles non-English text and title properly. Moreover you can use Yii views as e-mail templates.*

Code is useless if developers don't know where to put it and what should be in application configuration. Don't expect that people know how to do framework-specific things. Installation guide should be verbose. Step-by-step form is preferred by a majority of developers. If code needs SQL schema to work, provide it.

Even if your API methods and properties are named properly you still need to document them with phpDoc comments specifying argument types and return types, providing a brief description for each method. Don't forget protected and private methods and properties since sometimes it's necessary to read these to understand the details of how code works. Also, consider listing public methods and properties in documentation so it can be used as a reference.

Provide use case examples with well-commented code. Try to cover most common ways of extension usage. In an example don't try to solve multiple problems at a time since it can be confusing.

Extension should have a version number and a change log. It will allow the community to check if they have the latest version and check what is changed before upgrading.

It's important to make your code flexible so it will apply to many use cases. But, since it's not possible to create code for every possible use case, try to cover the most common ones.

It's important to make people feel comfortable. Providing a good documentation is a first step. The second is providing a proof your code works as expected and will work with further updates. The best way to do it is a set of unit tests.

Extension should be maintained. At least until it's stable and there are no more feature requests and bug reports. So expect questions, reports, and reserve some time to work on the code further. If you can't devote more time to maintain extensions, but it's very innovative and no one did it before, it's still worth sharing it. If the community likes it, someone will definitely offer his or her help.

Finally, you need to make extensions available. A common place for Yii extensions is `http://www.yiiframework.com/extensions/`. You need to be a registered forum member with a few posts before you will be allowed to upload your code. Try to find a good descriptive name, write descriptive summary, and specify relevant category and tags. Don't use archives other than ZIP or GZIP since these are the most common archive formats and you can be sure anyone can unpack it.

Even if your extension is relatively simple and documentation is good, there could be questions and, for the first time, the only man who can answer them is you. Typically, questions are asked at official forums, so it is better to create a topic where people can discuss your code and provide a link at the extension page.

How it works...

If you want to share an extension with the community and be sure it will be useful and popular, you need to do more than just write code. Making extensions distribution-ready is much more work to do. It can be even more than creating an extension itself. So why is it good to share extensions with the community in the first place?

Comparing to a code you use in your own projects open source has its pros. You are getting people, a lot more people than you can get to test your closed source project. People who are using your extension are testing it, giving valuable feedback, and reporting bugs. If your code is popular, there will be passionate developers who will try to improve your code, to make it more extensive, more stable, and reusable. Moreover, it just feels good because of doing a good thing.

There's more...

We have covered the most important things. Still there are more to check out. Try existing extensions before writing your own. If an extension almost fits try contacting the extension author and contribute ideas you have. Reviewing existing code helps to find out useful tricks, dos, and don'ts. Also check wiki articles and the official forum from time to time, there is a lot of useful information about creating extensions and developing using Yii in general.

Where to buy this book

You can buy Yii 1.1 Application Development Cookbook from the Packt Publishing website: `packtpub.com/yii-1-1-using-php-framework-application-development-cookbook`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/yii-1-1-using-php-framework-application-development-cookbook/book