Building a Blog System using Yii

Qiang Xue

Contents

License of Yii

The Yii framework is free software. It is released under the terms of the following BSD License.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Yii Software LLC nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Chapter 1

Getting Started

1.1 Building a Blog System using Yii

This tutorial describes how to use Yii to develop a blog application shown as the blog demo. It explains in detail every step to be taken during the development, which may also be applied in developing other Web applications. As a complement to the Guide and the Class Reference of Yii, this tutorial aims to show practical usage of Yii instead of thorough and definitive description.

Readers of this tutorial are not required to have prior knowledge about Yii. However, basic knowledge of object-oriented programming (OOP) and database programming would help readers to understand the tutorial more easily.

1.2 Testdriving with Yii

In this section, we describe how to create a skeleton application that will serve as our starting point. For simplicity, we assume that the document root of our Web server is `/wwwroot` and the corresponding URL is `http://www.example.com/`.

1.2.1 Installing Yii

We first install the Yii framework. Grab a copy of the Yii release file (version 1.0.3 or above) from www.yiiframework.com and unpack it to the directory `/wwwroot/yii`. Double check to make sure that there is a directory `/wwwroot/yii/framework`.

> **Tip:** The Yii framework can be installed anywhere in the file system. Its `framework` directory contains all framework code and is the only directory needed when deploying an Yii application. A single installation of Yii can be used by multiple Yii applications.

After installing Yii, open a browser window and access the URL `http://www.example.com/yii/requirements/index.php`. It shows the requirement checker provided in the Yii release. Make sure our Web server and PHP installation reaches the minimal requirement by Yii. In particular, we should enable both the `pdo` and `pdo_sqlite` PHP extensions which are required by our blog application to access the SQLite database.

1.2.2 Creating Skeleton Application

We then use the `yiic` tool to create a skeleton application under the directory `/wwwroot/blog`. The `yiic` tool is a command line tool provided in the Yii release. It can be used to generate code for certain tasks.

Open a command window and execute the following command:

```
% /wwwroot/yii/framework/yiic webapp /wwwroot/blog
Create a Web application under '/wwwroot/blog'? [Yes|No]y
......
```

> **Tip:** In order to use the `yiic` tool as shown above, the CLI PHP program must be on the command search path. If not, the following command may be used instead:
>
> `path/to/php /wwwroot/yii/framework/yiic.php webapp /wwwroot/blog`

To try out the application we just created, open a Web browser and navigate to the URL `http://www.example.com/blog/index.php`. We shall see that our application has three fully functional pages: the homepage, the contact page and the login page.

In the following, we briefly describe what we have in this skeleton application.

Entry Script

We have an entry script file `/wwwroot/blog/index.php` which has the following content:

```php
<?php
$yii='/wwwroot/framework/yii.php';
$config=dirname(__FILE__).'/protected/config/main.php';

// remove the following line when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);

require_once($yii);
Yii::createWebApplication($config)->run();
```

This is the only script that Web users can directly access. The script first includes the Yii bootstrap file `yii.php`. It then creates an application instance with the specified configuration and executes the application.

Base Application Directory

We also have an application base directory `/wwwroot/blog/protected`. The majority of our code and data will be placed under this directory, and it should be protected from being accessed by Web users. For Apache httpd Web server, we place under this directory a `.htaccess` file with the following content:

```
deny from all
```

For other Web servers, please refer to the corresponding manual on how to protect a directory from being accessed by Web users.

1.2.3 Application Workflow

To help understand how Yii works, we describe the main workflow in our skeleton application when a user is accessing its contact page:

1. The entry script is executed by the Web server to process the request;
2. An application instance is created and configured with initial property values specified in the application configuration file `/wwwroot/blog/protected/config/main.php`;
3. The application resolves the request into a controller and a controller action. For the contact page request, it is resolved as the `site` controller and the `contact` action;
4. The application creates the `site` controller in terms of a `SiteController` instance and then executes it;
5. The `SiteController` instance executes the `contact` action by calling its `actionContact()` method;
6. The `actionContact` method renders a view named `contact` to the Web user. Internally, this is achieved by including the view file `/wwwroot/blog/protected/views/site/contact.php` and embedding the result into the layout file `/wwwroot/blog/protected/views/layouts/main.php`.

1.3 Requirements Analysis

The blog system that we are going to develop is a single user system. The owner of the system will be able to perform the following actions:

- Login and logout
- Create, update and delete posts
- Publish, unpublish and archive posts
- Approve and delete comments

All other users are guest users who can perform the following actions:

- Read posts
- Create comments

Additional Requirements for this system include:

- The homepage of the system should display a list of the most recent posts.
- If a page contains more than 10 posts, they should be displayed in pages.
- The system should display a post together with its comments.
- The system should be able to list posts with a specified tag.
- The system should show a cloud of tags indicating their use frequencies.
- The system should show a list of most recent comments.
- The system should be themeable.
- The system should use SEO-friendly URLs.

1.4 Overall Design

Based on the analysis of the requirements, we identify that our blog application requires four database tables to store data: `User`, `Post`, `Comment` and `Tag`:

- `User` stores the user information, including username and password.

- `Post` stores post information. It mainly consists of the following columns:
  - `title`: required, title of the post;
  - `content`: required, body content of the post which uses the Markdown format;
  - `status`: required, status of the post, which can be one of following values:
    - `draft`: the post is in draft and is not visible to public;
    - `published`: the post is published to public;
    - `archived`: the post is outdated and is not visible to public.
  - `tags`: optional, a list of comma-separated words categorizing the post.
- `Comment` stores post comment information. Each comment is associated with a post and mainly consists of the following columns:
  - `name`: required, the author name;
  - `email`: required, the author email;
  - `website`: optional, the author website URL;
  - `content`: required, the comment content which uses the Markdown format.
  - `status`: required, status of the comment, which indicates whether the comment is approved (value 1) or not (value 0).
- `Tag` stores post tag information. Each post can have multiple tags, while each tag can also be attached to multiple posts. The `Tag` table is mainly used by the tag cloud portlet which needs to calculate the use frequency of each tag.

The following entity-relation (ER) diagram shows the table structure and relationships about the above tables. Note that the relationship between `Post` and `Tag` is many-to-many, we use the `PostTag` table to decouple this relationship into two one-to-many relationships.

Complete SQL statements corresponding to the above ER diagram may be found in the blog demo. In our Yii installation, they are in the file `/wwwroot/yii/demos/blog/protected/data/schema.sqlite.sql`.

We divide the development of our blog application into the following milestones.

- Milestone 1: creating a prototype of the blog system. It should consist of most of the required functionalities.
- Milestone 2: completing post management. It includes creating, listing, showing, updating and deleting posts.

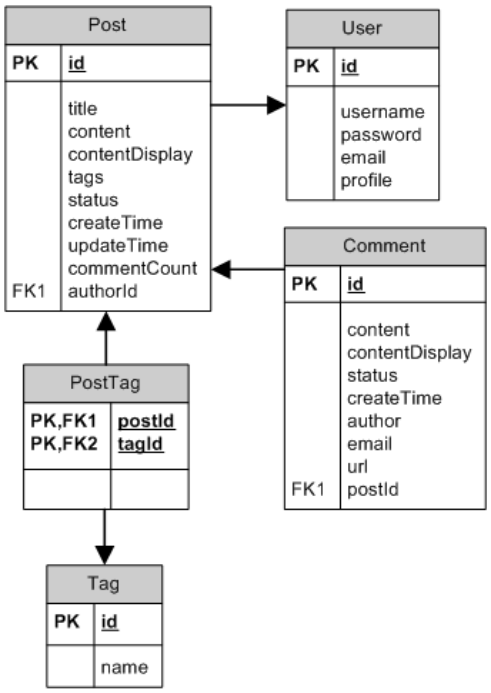

Figure 1.1: Entity-Relation Diagram of the Blog Database

- Milestone 3: completing comment management. It includes creating, listing, approving, updating and deleting post comments.
- Milestone 4: implementing portlets. It includes user menu, login, tag cloud and recent comments portlets.
- Milestone 5: final tune-up and deployment.

Chapter 2

Initial Prototyping

2.1 Setting Up Database

Having created a skeleton application and finished database design, in this section we will create the blog database and establish the connection to it in the skeleton application.

2.1.1 Creating Database

We choose to create a SQLite database. Because the database support in Yii is built on top of PDO, we can easily switch to use a different type of DBMS (e.g. MySQL, PostgreSQL) without the need to change our application code.

We create the database file `blog.db` under the directory `/wwwroot/blog/protected/data`. Note that both the directory and the database file have to be writable by the Web server process, as required by SQLite. We may simply copy the database file from the blog demo in our Yii installation which is located at `/wwwroot/yii/demos/blog/protected/data/blog.db`. We may also generate the database by executing the SQL statements in the file `/wwwroot/yii/demos/blog/protected/data/schema.sqlite.sql`.

> **Tip:** To execute SQL statements, we may use the `sqlite3` command line tool that can be found in the SQLite official website.

2.1.2 Establishing Database Connection

To use the blog database in the skeleton application we created, we need to modify its application configuration which is stored as a PHP script `/wwwroot/blog/protected/config/main.php`. The script returns an associative array consisting of name-value pairs, each of which is used to initialize a property of the application instance.

We configure the `components` property of the application by adding a new entry named `db` shown as follows,

```
return array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'sqlite:/wwwroot/blog/protected/data/blog.db',
        ),
    ),
    ......
);
```

The above configuration says that we have a `db` application component whose class is CDbConnection and whose `connectionString` property should be initialized as `sqlite:/wwwroot/blog/protected/data/blog.db`.

With this configuration, we can access the DB connection object using `Yii::app()->db` at any place in our code. Note that `Yii::app()` returns the application instance that we create in the entry script. If you are interested in possible methods and properties that the DB connection has, you may refer to its class reference. However, in most cases we are not going to use this DB connection directly. Instead, we will use the so-called ActiveRecord to access the database.

2.2 Scaffolding

Create, read, update and delete (CRUD) are the four basic operations of persistent storage. In our blog application, the major task is to implement the CRUD operations for both posts and comments. In this section, we will use the `yiic` tool to accomplish this task. This process is also known as *scaffolding*.

Open a command window and run the following commands:

```
% /wwwroot/yii/framework/yiic shell /wwwroot/blog/index.php
Yii Interactive Tool v1.0
Please type 'help' for help. Type 'exit' to quit.
>> model User
......
>> model Post
......
>> model Tag
......
>> model Comment
......
>> crud Post
```

```
......
>> crud Comment
......
>> exit
```

> **Info:** Some PHP installations may use a different `php.ini` file for command line (CLI) PHP parser. As a result, when running the above `yiic` commands, you may encounter errors like "YiiBase::include(PDO.php): failed to open stream..." or "...could not find driver". Please double check your CLI PHP configuration by executing the following command:
>
> ```
> php -r "phpinfo();"
> ```
>
> The result of the above command will show which `php.ini` file is being used and which extensions are loaded. If a wrong `php.ini` file is used, you may use the following command to explicitly specify the correct `php.ini` to use:
>
> ```
> php -c php.ini /wwwroot/yii/framework/yiic.php shell /wwwroot/blog/index.php
> ```

The commands above accomplish two tasks. First, the `model` commands generate a model class file for each database table. Second, the `crud` commands generate the code needed by the CRUD operations for the `Post` and `Comment` models.

We can test the generated code by accessing the following URLs:

```
http://www.example.com/blog/index.php?r=post
http://www.example.com/blog/index.php?r=comment
```

Notice that the post and comment features implemented by the generated code are completely independent of each other. Also, when creating a new post or comment, we are required to enter information, such as `authId` and `createTime`, which in real application should be set by the program. Don't worry. We will fix these problems in the next milestones. For now, we should be fairly satisfied as this prototype already contains most features that we need to implement for the blog application.

To prepare for the next milestones, let's take a closer look at the files generated by the above commands. All the files are generated under `/wwwroot/blog/protected`. For convenience, we group them into model files, controller files and view files:

- model files:

  - `models/User.php` contains the `user` class that extends from CActiveRecord and can be used to access the `User` database table;
  - `models/Post.php` contains the `Post` class that extends from CActiveRecord and can be used to access the `Post` database table;
  - `models/Tag.php` contains the `Tag` class that extends from CActiveRecord and can be used to access the `Tag` database table;
  - `models/Comment.php` contains the `Comment` class that extends from CActiveRecord and can be used to access the `Comment` database table;

- controller file:

  - `controllers/PostController.php` contains the `PostController` class which is the controller in charge of all CRUD operations about posts;
  - `controllers/CommentController.php` contains the `CommentController` class which is the controller in charge of all CRUD operations about comments;

- view files:

  - `views/post/create.php` is the view file that shows an HTML form to create a new post;
  - `views/post/update.php` is the view file that shows an HTML form to update an existing post;
  - `views/post/show.php` is the view file that displays the detailed information of a post;
  - `views/post/list.php` is the view file that displays a list of posts;
  - `views/post/admin.php` is the view file that displays posts in a table with administrative commands.
  - `views/post/_form.php` is the partial view file that displays the HTML form for collecting post information. It is embedded in the `create` and `update` views.
  - a similar set of view files are also generated for comment.

In order to understand better how the above files are used, we show in the following the workflow that occurs in the blog application when displaying a list of posts:

1. The entry script is executed by the Web server which creates and initializes an application instance to handle the request;
2. The application creates an instance of `PostController` and executes it;

3. The `PostController` instance executes the requested `list` action by calling its `actionList()` method;
4. The `actionList()` method queries database to bring back the list of recent posts;
5. The `actionList()` method renders the `list` view with the post data.

2.3 Authenticating User

Our blog application needs to differentiate between the system owner and guest users. Therefore, we need to implement the user authentication feature.

As you may have found that the skeleton application already provides user authentication by checking if the username and password are both `demo` or `admin`. In this section, we will modify the corresponding code so that the authentication is done against the `User` database table.

User authentication is performed in a class implementing the IUserIdentity interface. The skeleton application uses the `UserIdentity` class for this purpose. The class is stored in the file `/wwwroot/blog/protected/components/UserIdentity.php`.

> **Tip:** By convention, the name of a class file must be the same as the corresponding class name suffixed with the extension `.php`. Following this convention, one can refer to a class using a path alias. For example, we can refer to the `UserIdentity` class with the alias `application.components.UserIdentity`. Many APIs in Yii can recognize path aliases (e.g. Yii::createComponent()), and using path aliases avoids the necessity of embedding absolute file paths in the code. The existence of the latter often causes trouble when we deploy an application.

We modify the `UserIdentity` class as follows,

```
<?php
class UserIdentity extends CUserIdentity
{
    private $_id;

    public function authenticate()
    {
        $username=strtolower($this->username);
        $user=User::model()->find('LOWER(username)=?',array($username));
        if($user===null)
            $this->errorCode=self::ERROR_USERNAME_INVALID;
        else if(md5($this->password)!==$user->password)
```

```
            $this->errorCode=self::ERROR_PASSWORD_INVALID;
        else
        {
            $this->_id=$user->id;
            $this->username=$user->username;
            $this->errorCode=self::ERROR_NONE;
        }
        return !$this->errorCode;
    }

    public function getId()
    {
        return $this->_id;
    }
}
```

In the `authenticate()` method, we use the `User` class to look for a row in the `User` table whose `username` column is the same as the given username in a case-insensitive manner. Remember that the `User` class was created using the `yiic` tool in the prior section. Because the `User` class extends from CActiveRecord, we can exploit the ActiveRecord feature and access the `User` table in an OOP fashion.

In the `UserIdentity` class, we also override the `getId()` method which returns the `id` value of the user found in the `User` table. The parent implementation would return the username, instead. Both the `username` and `id` properties will be stored in the user session and may be accessed via `Yii::app()->user` from anywhere in our code.

Tip: In the `UserIdentity` class, we reference the class CUserIdentity without explicitly including the corresponding class file. This is because CUserIdentity is a core class provided by the Yii framework. Yii will automatically include the class file for any core class when it is referenced for the first time. We also do the same with the `User` class. This is because the `User` class file is placed under the directory `/wwwroot/blog/protected/models` which has been added to the PHP `include_path` according to the following lines found in the application configuration:

```
return array(
    ......
    'import'=>array(
        'application.models.*',
        'application.components.*',
    ),
    ......
);
```

The above configuration says that any class whose class file is located under either `/wwwroot/blog/protected/models` or `/wwwroot/blog/protected/components` will be automatically included when the class is referenced for the first time.

The `UserIdentity` class is mainly used by the `LoginForm` class to authenticate a user based on the username and password input collected from the login page. The following code fragment shows how `UserIdentity` is used:

```
$identity=new UserIdentity($username,$password);
$identity->authenticate();
switch($identity->errorCode)
{
    case UserIdentity::ERROR_NONE:
        Yii::app()->user->login($identity);
        break;
    ......
}
```

Info: People often get confused about identity and the `user` application component. The former represents a way of performing authentication, while the latter is used to represent the information related with the current user. An application can only have one `user` component, but it can have one or several identity classes, depending on what kind of authentication it supports. Once authenticated, an identity instance may pass its state information to the `user` component so that they are globally accessible via `user`.

To test the modified `UserIdentity` class, we can browse the URL `http://www.example.com/blog/index.php` and try logging in with the username and password that we store in the `User` table. If we use the database provided by the blog demo, we should be able to login with username `demo` and password `demo`. Note that this blog system does not provide the user management feature. As a result, a user cannot change his account or create a new one through the Web interface. The user management feature may be considered as a future enhancement to the blog application.

2.4 Summary

We have completed the milestone 1. Let's summarize what we have done so far:

1. We identified the requirements to be fulfilled;
2. We installed the Yii framework;
3. We created a skeleton application;
4. We designed and created the blog database;
5. We modified the application configuration by adding the database connection;
6. We generated the code that implements the basic CRUD operations for both posts and comments;
7. We modified the authentication method to check against the `User` table.

For a new project, most of the time will be spent in step 1 and 4 for this first milestone.

Although the code generated by the `yiic` tool implements fully functional CRUD operations for a database table, it often needs to be modified in practical applications. For this reason, in the next two milestone, our job is to customize the generated CRUD code about posts and comments so that it reaches our initial requirements.

In general, we first modify the model class file by adding appropriate validation rules and declaring relational objects. We then modify the controller action and view code for each individual CRUD operation.

Chapter 3

Post Management

3.1 Customizing Post Model

The `Post` model class generated by the `yiic` tool mainly needs to be modified in three places:

- the `rules()` method: specifies the validation rules for the model attributes;
- the `relations()` method: specifies the related objects;
- the `safeAttributes()` method: specifies which attributes can be massively assigned (mainly used when passing user input to the model);

> **Info:** A model consists of a list of attributes, each associated with a column in the corresponding database table. Attributes can be declared explicitly as class member variables or implicitly without any declaration.

3.1.1 Customizing `rules()` Method

We first specify the validation rules which ensure the attribute values populated by user inputs are correct before they are saved to the database. For example, the `status` attribute of `Post` should be an integer 0, 1 or 2. The `yiic` tool also generates validation rules for each model. However, these rules are based on the table column information and may not be appropriate.

Based on the requirements analysis, we modify the `rules()` method as follows:

```
public function rules()
{
    return array(
```

```
            array('title, content, status', 'required'),
            array('title', 'length', 'max'=>128),
            array('status', 'in', 'range'=>array(0, 1, 2)),
            array('tags', 'match', 'pattern'=>'/^[\w\s,]+$/',
                'message'=>'Tags can only contain word characters.'),
        );
    }
```

In the above, we specify that the `title`, `content` and `status` attributes are required; the length of `title` should not exceed 128; the `status` attribute value should be 0 (draft), 1 (published) or 2 (archived); and the `tags` attribute should only contain word characters and commas. All other attributes (e.g. `id`, `createTime`) will not be validated because their values do not come from user input.

After making these changes, we can visit the post creation page again to verify that the new validation rules are taking effect.

> **Info:** Validation rules are used when we call the validate() or save() method of the model instance. For more information about how to specify validation rules, please refer to the Guide.

3.1.2 Customizing `safeAttributes()` Method

We then customize the `safeAttributes()` method to specify which attributes can be massively assigned. When passing user inputs to the model instance, we often use the following massive assignment to simplify our code:

```
$post->attributes=$_POST['Post'];
```

Without using the above massive assignment, we would end up with the following lengthy code:

```
$post->title=$_POST['Post']['title'];
$post->content=$_POST['Post']['content'];
......
```

Although massive assignment is very convenient, it has a potential danger that a malicious user may attempt to populate an attribute whose value should remain read only or should only be changed by developer in code. For example, the `id` of the post currently being updated should not be changed.

To prevent from such danger, we should customize the `safeAttributes()` as follows, which states only `title`, `content`, `status` and `tags` attributes can be massively assigned:

```
public function safeAttributes()
{
    return array('title', 'content', 'status', 'tags');
}
```

> **Tip:** An easy way to identity which attributes should be put in the safe list is by observing the HTML form that is used to collect user input. Model attributes that appear in the form to receive user input may be declared as safe. Since these attributes receive input from end users, they usually should be associated with some validation rules.

3.1.3 Customizing `relations()` Method

Lastly we customize the `relations()` method to specify the related objects of a post. By declaring these related objects in `relations()`, we can exploit the powerful Relational ActiveRecord (RAR) feature to access the related object information of a post, such as its author and comments, without the need to write complex SQL JOIN statements.

We customize the `relations()` method as follows:

```
public function relations()
{
    return array(
        'author'=>array(self::BELONGS_TO, 'User', 'authorId'),
        'comments'=>array(self::HAS_MANY, 'Comment', 'postId',
            'order'=>'??.createTime'),
        'tagFilter'=>array(self::MANY_MANY, 'Tag', 'PostTag(postId, tagId)',
            'together'=>true,
            'joinType'=>'INNER JOIN',
            'condition'=>'??.name=:tag'),
    );
}
```

The above relations state that

- A post belongs to an author whose class is `User` and the relationship is established based on the `authorId` attribute value of the post;

- A post has many comments whose class is `Comment` and the relationship is established based on the `postId` attribute value of the comments. These comments should be sorted according to their creation time.

The `tagFilter` relation is a bit complex. It is used to explicitly join the `Post` table with the `Tag` table and choose only the rows with a specified tag name. We will show how to use this relation when we implement the post display feature.

With the above relation declaration, we can easily access the author and comments of a post like the following:

```
$author=$post->author;
echo $author->username;

$comments=$post->comments;
foreach($comments as $comment)
    echo $comment->content;
```

For more details about how to declare and use relations, please refer to the Guide.

3.1.4 Representing Status in Text

Because the status of a post is stored as an integer in the database, we need to provide a text representation so that it is more intuitive when being displayed to end users. For this reason, we modify the `Post` model as follows,

```
class Post extends CActiveRecord
{
    const STATUS_DRAFT=0;
    const STATUS_PUBLISHED=1;
    const STATUS_ARCHIVED=2;

    ......

    public function getStatusOptions()
    {
        return array(
            self::STATUS_DRAFT=>'Draft',
            self::STATUS_PUBLISHED=>'Published',
            self::STATUS_ARCHIVED=>'Archived',
        );
    }

    public function getStatusText()
```

```
    {
        $options=$this->statusOptions;
        return isset($options[$this->status]) ? $options[$this->status]
            : "unknown ({$this->status})";
    }
}
```

In the above, we define class constants to represent the possible status values. These constants are mainly used in the code to make it more maintainable. We also define the `getStatusOptions()` method which returns a mapping between status integer values and text display. And finally, we define the `getStatusText()` method which simply returns the textual status display of the current post.

3.2 Creating and Updating Posts

With the `Post` model ready, we need to fine-tune the actions and views for the controller `PostController`. In this section, we first customize the access control of CRUD operations; we then modify the code implementing the `create` and `update` operations; and finally we implement the preview feature for both operations.

3.2.1 Customizing Access Control

The first thing we want to do is to customize the access control because the code generated by `yiic` does not fit our needs.

We modify the `accessRules()` method in the file `/wwwroot/blog/protected/controllers/PostController.php` as follows,

```
public function accessRules()
{
    return array(
        array('allow',  // allow all users to perform 'list' and 'show' actions
            'actions'=>array('list', 'show'),
            'users'=>array('*'),
        ),
        array('allow', // allow authenticated users to perform any action
            'users'=>array('@'),
        ),
        array('deny',  // deny all users
            'users'=>array('*'),
        ),
    );
}
```

The above rules state that all users can access the `list` and `show` actions, and authenticated users can access any actions, including the `admin` action. The user should be denied access in any other scenario. Note that these rules are evaluated in the order they are listed here. The first rule matching the current context makes the access decision. For example, if the current user is the system owner who tries to visit the post creation page, the second rule will match and it will give the access to the user.

3.2.2 Customizing `create` and `update` Operations

The `create` and `update` operations are very similar. They both need to display an HTML form to collect user inputs, validate them, and save them into database. The main difference is that the `update` operation will pre-populate the form with the existing post data found in the database. For this reason, the `yiic` tool generates a partial view `/wwwroot/blog/protected/views/post/_form.php` that is embedded in both the `create` and `update` views to render the needed HTML form.

We first change the `_form.php` file so that the HTML form only collects the inputs we want: `title`, `content` and `status`. We use plain text fields to collect inputs for the first two attributes, and a dropdown list to collect input for `status`. The dropdown list options are the text displays of the possible post statuses:

```
<?php echo CHtml::activeDropDownList($post,'status',Post::model()->statusOptions); ?>
```

> **Tip:** In the above, we can also use `Post::model()->getStatusOptions()` instead of `Post::model()->statusOptions` to return the possible status options. The reason that we can use the latter expression is because `Post` is a component which allows us to access properties defined in terms of getter methods.

We then modify the `Post` class so that it can automatically set some attributes (e.g. `createTime`, `authorId`) before a post is saved to the database. We override the `beforeValidate()` method as follows,

```
protected function beforeValidate($on)
{
    $parser=new CMarkdownParser;
    $this->contentDisplay=$parser->safeTransform($this->content);
    if($this->isNewRecord)
    {
        $this->createTime=$this->updateTime=time();
        $this->authorId=Yii::app()->user->id;
```

```
    }
    else
        $this->updateTime=time();
    return true;
}
```

In this method, we use CMarkdownParser to convert the content from Markdown format into HTML and save the result to `contentDisplay`. This avoids repeated format conversion when we display a post. If the post is new, we set its `createTime` and `authorId` attributes; otherwise we set its `updateTime` to be the current time. Note that this method will be invoked automatically when we call `validate()` or `save()` method of the model.

Because we want to save post tags to the `Tag` table, we also need the following method in the `Post` class, which is invoked automatically after a post is saved to the database:

```
protected function afterSave()
{
    if(!$this->isNewRecord)
        $this->dbConnection->createCommand(
            'DELETE FROM PostTag WHERE postId='.$this->id)->execute();

    foreach($this->getTagArray() as $name)
    {
        if(($tag=Tag::model()->findByAttributes(array('name'=>$name)))===null)
        {
            $tag=new Tag(array('name'=>$name));
            $tag->save();
        }
        $this->dbConnection->createCommand(
            "INSERT INTO PostTag (postId, tagId) VALUES ({$this->id},{$tag->id})")->execute();
    }
}

public function getTagArray()
{
    // break tag string into a set of tags
    return array_unique(
        preg_split('/\s*,\s*/',trim($this->tags),-1,PREG_SPLIT_NO_EMPTY)
    );
}
```

In the above, we first clean up the `PostTag` table for rows related with the current post. We then insert new tags into the `Tag` table and add a reference in the `PostTag` table. The logic here is a bit complex. Instead of using ActiveRecord, we write raw SQL statements and execute them with the database connection.

> **Tip:** It is good practice to keep business logic, such as the above `beforeValidate()` and `afterSave()` code, in models instead of controllers.

3.2.3 Implementing Preview Feature

Besides the above customizations, we also want to add the preview feature that would allow us to preview a post before we save it to the database.

We first change the `_form.php` view file to add a `preview` button and a preview display. The preview is only displayed when the preview button is clicked and there is not validation error.

```
<?php echo CHtml::submitButton('Preview',array('name'=>'previewPost')); ?>
......
<?php if(isset($_POST['previewPost']) && !$post->hasErrors()): ?>
...display preview of $post here...
<?php endif; ?>
```

We then change the `actionCreate()` and `actionUpdate()` methods of `PostController` to respond to the preview request. Below we show the updated code of `actionCreate()`, which is very similar to that in `actionUpdate()`:

```
public function actionCreate()
{
    $post=new Post;
    if(isset($_POST['Post']))
    {
        $post->attributes=$_POST['Post'];
        if(isset($_POST['previewPost']))
            $post->validate();
        else if(isset($_POST['submitPost']) && $post->save())
            $this->redirect(array('show','id'=>$post->id));
    }
    $this->render('create',array('post'=>$post));
}
```

In the above, if the preview button is clicked, we call `$post->validate()` to validate the user input; otherwise if the submit button is clicked, we try to save the post by calling `$post->save()` which implicitly performs validation. If the saving is successful (no validation errors and the data is saved to the database without error), we redirect the user browser to show the newly created post.

3.3 Displaying Posts

In our blog application, a post may be displayed among a list of posts or by itself. The former is implemented as the `list` operation while the latter the `show` operation. In this section, we customize both operations to fulfill our initial requirements.

3.3.1 Customizing `show` Operation

The `show` operation is implemented by the `actionShow()` method in `PostController`. Its display is generated by the `show` view with the view file `/wwwroot/blog/protected/views/post/show.php`.

Below is the relevant code implementing the `show` operation in `PostController`:

```
public function actionShow()
{
    $this->render('show',array(
        'post'=>$this->loadPost(),
    ));
}

private $_post;

protected function loadPost($id=null)
{
    if($this->_post===null)
    {
        if($id!==null || isset($_GET['id']))
            $this->_post=Post::model()->findbyPk($id!==null ? $id : $_GET['id']);
        if($this->_post===null || Yii::app()->user->isGuest &&
            $this->_post->status!=Post::STATUS_PUBLISHED)
            throw new CHttpException(404,'The requested post does not exist.');
    }
    return $this->_post;
}
```

Our change mainly lies in the `loadPost()` method. In this method, we query the `Post` table according to the `id` GET parameter. If the post is not found or if it is not published (when the user is a guest), we will throw a 404 HTTP error. Otherwise the post object is returned to `actionShow()` which in turn passes the post object to the `show` view for further display.

> **Tip:** Yii captures HTTP exceptions (instances of CHttpException) and displays them in error pages using some predefined templates. These templates can be customized per application, which we will describe in detail at the end of this tutorial.

The change in the `show` view is mainly about ajdusting the formatting and styles of the post display. We will not go into details here.

3.3.2 Customizing `list` Operation

Like the `show` operation, we customize the `list` operation in two places: the `actionList()` method in `PostController` and the view file `/wwwroot/blog/protected/views/post/list.php`. We mainly need to add the support for displaying a list of posts that are associated with a specified tag.

Below is the modified `actionList()` method in `PostController`:

```
public function actionList()
{
    $criteria=new CDbCriteria;
    $criteria->condition='status='.Post::STATUS_PUBLISHED;
    $criteria->order='createTime DESC';

    $withOption=array('author');
    if(!empty($_GET['tag']))
    {
        $withOption['tagFilter']['params'][':tag']=$_GET['tag'];
        $postCount=Post::model()->with($withOption)->count($criteria);
    }
    else
        $postCount=Post::model()->count($criteria);

    $pages=new CPagination($postCount);
    $pages->applyLimit($criteria);

    $posts=Post::model()->with($withOption)->findAll($criteria);

    $this->render('list',array(
        'posts'=>$posts,
        'pages'=>$pages,
    ));
}
```

In the above, we first create a query criteria which specifies only published posts should be listed and they should be sorted according to their creation time in descending order.

We then compute the total number of posts satisfying the criteria. The number is used by the pagination component to correctly compute how many pages the posts should be displayed in. Finally, we retrieve the post data from the database and send them to the `list` view for display.

Notice that when there is `tag` GET parameter, we would query with the `tagFilter` using the corresponding GET parameter value. Including `tagFilter` in the relational query will ensure that only a single SQL JOIN statement is used to retrieve the posts with the specified tag. Without this call, Yii would break the query into two separate SQL statements (for efficiency concern) and would return incorrect results.

Two variables are passed to the `list` view: `$posts` and `$pages`. The former refers to the list of posts be displayed, while the latter contains pagination information (e.g. how many pages in total, what is the current page). The `list` view contains a pagination widget that can automatically display posts in separate pages if there are too many of them.

3.4 Managing Posts

Managing posts mainly refers to listing posts in an administrative view and deleting posts. They are accomplished by the `admin` operation and the `delete` operation, respectively. The code generated by `yiic` does not need much modification. Below we mainly explain how these two operations are implemented.

3.4.1 Listing Posts in Tabular View

The `admin` operation shows all posts (including both published and unpublished) in a tabular view. The view supports multi-column sorting and pagination. The following is the `actionAdmin()` method in `PostController`:

```
public function actionAdmin()
{
    $criteria=new CDbCriteria;

    $pages=new CPagination(Post::model()->count());
    $pages->applyLimit($criteria);

    $sort=new CSort('Post');
    $sort->defaultOrder='status ASC, createTime DESC';
    $sort->applyOrder($criteria);

    $posts=Post::model()->findAll($criteria);

    $this->render('admin',array(
```

```
        'posts'=>$posts,
        'pages'=>$pages,
        'sort'=>$sort,
    ));
}
```

The above code is very similar to that in `actionList()`. The main difference is that here we use a CSort object to represent the sorting informatin (e.g. which columns are being sorted in which directions). The CSort object is used by the `admin` view to generate appropriate hyperlinks in the table head cells. Clicking on a link would cause the current page to be refreshed and the data to be sorted along that column.

Below is the code for the `admin` view:

```
<h2>Manage Posts</h2>

<table class="dataGrid">
  <tr>
    <th><?php echo $sort->link('status'); ?></th>
    <th><?php echo $sort->link('title'); ?></th>
    <th><?php echo $sort->link('createTime'); ?></th>
    <th><?php echo $sort->link('updateTime'); ?></th>
  </tr>
<?php foreach($posts as $n=>$post): ?>
  <tr class="<?php echo $n%2?'even':'odd';?>">
    <td><?php echo CHtml::encode($post->statusText); ?></td>
    <td><?php echo CHtml::link(CHtml::encode($post->title),
        array('show','id'=>$post->id)); ?></td>
    <td><?php echo date('F j, Y',$post->createTime); ?></td>
    <td><?php echo date('F j, Y',$post->updateTime); ?></td>
  </tr>
<?php endforeach; ?>
</table>

<br/>
<?php $this->widget('CLinkPager',array('pages'=>$pages)); ?>
```

The code is very straight-forward. We iterate through the list of posts and display them in a table. In the head cells of the table, we use the CSort object to generate the hyperlinks for sorting purpose. And at the end, we embed a CLinkPager widget to display pagination buttons if needed.

> **Tip:** When displaying text, we call CHtml::encode() to encode HTML entities in it. This prevents from cross-site scripting attack.

3.4.2 Deleting Posts

When a post is displayed using the `show` operation, we display a `delete` link if the current user is the system owner. Clicking on this button would cause the deletion of the post. Since the post deletion is causing the change of the server-side data, we use a POST request to trigger the deletion. We thus use the following code to generate the `delete` button:

```
<?php echo CHtml::linkButton('Delete',array(
   'submit'=>array('post/delete','id'=>$post->id),
   'confirm'=>"Are you sure to delete this post?",
)); ?>
```

The CHtml::linkButton() method generates a link button that is like the normal push button. Clicking on the link would cause the submission of the enclosing HTML form in POST method. Here we specify that the form should be submitted to the URL generated according to `array('post/delete','id'=>$post->id)`. In our blog application, the generated URL would be `/blog/index.php?r=post/delete&id=1`, which refers to the `delete` action of `PostController`. We also specify that a confirmation dialog should pop up when clicking on this link. This gives the user a chance to re-consider his deletion request.

The code for the `delete` operation is self-explanatory. We are not going to explain here.

```
public function actionDelete()
{
    if(Yii::app()->request->isPostRequest)
    {
        // we only allow deletion via POST request
        $this->loadPost()->delete();
        $this->redirect(array('list'));
    }
    else
        throw new CHttpException(400,'Invalid request...');
}
```

Chapter 4

Comment Management

4.1 Customizing Comment Model

Like the `Post` model, we need to customize the `rules()`, `relations()` and `safeAttributes()` methods of the `Comment` model. In addition, we also need to modify the `attributeLabels()` to declare the customized labels for some attributes.

4.1.1 Customizing `rules()` Method

We first customize the validation rules generated by the `yiic` tool. The following rules are used for comments:

```
public function rules()
{
    return array(
        array('author,email,content', 'required'),
        array('author,email,url','length','max'=>128),
        array('email','email'),
        array('url','url'),
        array('verifyCode', 'captcha', 'on'=>'insert',
            'allowEmpty'=>!Yii::app()->user->isGuest),
    );
}
```

In the above, we specify that the `author`, `email` and `content` attributes are required; the length of `author`, `email` and `url` cannot exceed 128; the `email` attribute must be a valid email address; the `url` attribute must be a valid URL; and the `verifyCode` attribute should be validated as a CAPTCHA code.

The `verifyCode` attribute in the above is mainly used to store the verification code that a user enters in order to leave a comment. Because it is not present in the `Comment` table, we need to explicitly declare it as a public member variable. Its validation is using a special validator named `captcha` which refers to the CCaptchaValidator class. Moreover,

the validation will only be performed when a new comment is being inserted (see the `on` option). And for authenticated users, this is not needed (see the `allowEmpty` option).

4.1.2 Customizing `safeAttributes()` Method

We then customize the `safeAttributes()` method to specify which attributes can be massively assigned.

```
public function safeAttributes()
{
    return array('author', 'email', 'url', 'content', 'verifyCode');
}
```

This also indicates that the comment form would consist of fields to collect the information about author, email, URL, content and verification code.

4.1.3 Customizing `relations()` Method

When we develop the "recent comments" portlet, we need to list the most recent comments together with their corresponding post information. Therefore, we need to customize the `relations()` method to declare the relation about post.

```
public function relations()
{
    return array(
        'post'=>array(self::BELONGS_TO, 'Post', 'postId',
            'joinType'=>'INNER JOIN'),
    );
}
```

Notice that the join type for the `post` relation is `INNER JOIN`. This is because a comment has to belong to a post.

4.1.4 Customizing `attributeLabels()` Method

Finally, we need to customize the `attributeLabels()` method to declare the customized labels for the attributes. The method returns an array consisting of name-label pairs. When we call CHtml::activeLabel() to display an attribute label, it will first check if a customized label is declared. If not, it will use an algorithm to generate the default label.

```
public function attributeLabels()
```

```
{
    return array(
        'author'=>'Name',
        'url'=>'Website',
        'content'=>'Comment',
        'verifyCode'=>'Verification Code',
    );
}
```

> **Tip:** The algorithm for generating the default label is based on the attribute name. It first breaks the name into words according to capitalization. It then changes the first character in each word into upper case. For example, the name `verifyCode` would have the default label `Verify Code`.

4.1.5 Customizing Saving Process

Because we want to keep the comment count in each post, when we add or delete a comment, we need to adjust the corresponding comment count for the post. We achieve this by overriding the `afterSave()` method and the `afterDelete()` method of the `Comment` model. We also override its `beforeValidate()` method so that we can convert the content from the Markdown format to HTML format and record the creation time.

```
protected function beforeValidate($on)
{
    $parser=new CMarkdownParser;
    $this->contentDisplay=$parser->safeTransform($this->content);
    if($this->isNewRecord)
        $this->createTime=time();
    return true;
}

protected function afterSave()
{
    if($this->isNewRecord && $this->status==Comment::STATUS_APPROVED)
        Post::model()->updateCounters(array('commentCount'=>1), "id={$this->postId}");
}

protected function afterDelete()
{
    if($this->status==Comment::STATUS_APPROVED)
        Post::model()->updateCounters(array('commentCount'=>-1), "id={$this->postId}");
}
```

4.2 Creating and Displaying Comments

In this section, we implement the comment display and creation features.

4.2.1 Displaying Comments

Instead of displaying and creating comments individual pages, we use the post display page. Below the post content display, we display first a list of comments belonging to that post and then a comment creation form.

In order to display comments on the post page, we modify the `actionShow()` method of `PostController` as follows,

```
public function actionShow()
{
    $post=$this->loadPost();
    $this->render('show',array(
        'post'=>$post,
        'comments'=>$post->comments,
    ));
}
```

Note that the expression `$post->comments` is valid because we have declared a `comments` relation in the `Post` class. Evaluating this expression would trigger an implicit JOIN database query to bring back the comments belonging to the current post. This feature is known as lazy relational query.

We also modify the `show` view by appending the comment display at the end of the post display, which we are not going to elaborate here.

4.2.2 Creating Comments

To handle comment creation, we first modify the `actionShow()` method of `PostController` as follows,

```
public function actionShow()
{
    $post=$this->loadPost();
    $comment=$this->newComment($post);
    $this->render('show',array(
        'post'=>$post,
        'comments'=>$post->comments,
        'newComment'=>$comment,
    ));
```

```
}

protected function newComment($post)
{
    $comment=new Comment;
    if(isset($_POST['Comment']))
    {
        $comment->attributes=$_POST['Comment'];
        $comment->postId=$post->id;
        $comment->status=Comment::STATUS_PENDING;

        if(isset($_POST['previewComment']))
            $comment->validate('insert');
        else if(isset($_POST['submitComment']) && $comment->save())
        {
            Yii::app()->user->setFlash('commentSubmitted','Thank you...');
            $this->refresh();
        }
    }
    return $comment;
}
```

In the above, we call the `newComment()` method before we render the `show` view. In the `newComment()` method, we generate a `Comment` instance and check if the comment form is submitted. The form may be submitted by clicking either the submit button or the preview button. If the former, we try to save the comment and display a flash message. The flash message is displayed only once, which means if we refresh the page again, it will disappear.

We also modify the `show` view by appending the comment creation form:

```
......
<?php $this->renderPartial('/comment/_form',array(
    'comment'=>$newComment,
    'update'=>false,
)); ?>
```

Here we embed the comment creation form by rendering the partial view `/wwwroot/blog/protected/views/comment/_form.php`. The variable `$newComment` is passed by the `actionShow` method. Its main purpose is to store the user comment input. The variable `update` is set as false, which indicates the comment form is being used to create a new comment.

In order to support comment preview, we add a preview button to the comment creation form. When the preview button is clicked, the comment preview is displayed at the

bottom. Below is the updated code of the comment form:

```
...comment form with preview button...

<?php if(isset($_POST['previewComment']) && !$comment->hasErrors()): ?>
<h3>Preview</h3>
<div class="comment">
  <div class="author"><?php echo $comment->authorLink; ?> says:</div>
  <div class="time"><?php echo date('F j, Y \a\t h:i a',$comment->createTime); ?></div>
  <div class="content"><?php echo $comment->contentDisplay; ?></div>
</div><!-- post preview -->
<?php endif; ?>
```

4.3 Managing Comments

Comment management includes updating, deleting and approving comments. These operations are implemented as actions in the `CommentController` class.

4.3.1 Updating and Deleting Comments

The code generated by `yiic` for updating and deleting comments remains largely unchanged. Because we support comment preview when updating a comment, we only need to change the `actionUpdate()` method of `CommentController` as follows,

```
public function actionUpdate()
{
    $comment=$this->loadComment();

    if(isset($_POST['Comment']))
    {
        $comment->attributes=$_POST['Comment'];
        if(isset($_POST['previewComment']))
            $comment->validate('update');
        else if(isset($_POST['submitComment']) && $comment->save())
            $this->redirect(array('post/show',
                'id'=>$comment->postId,
                '#'=>'c'.$comment->id));
    }

    $this->render('update',array('comment'=>$comment));
}
```

It is very similar to that in `PostController`.

4.3.2 Approving Comments

When comments are newly created, they are in pending approval status and need to be approved in order to be visible to guest users. Approving a comment is mainly about changing the status column of the comment.

We create an `actionApprove()` method in `CommentController` as follows,

```
public function actionApprove()
{
    if(Yii::app()->request->isPostRequest)
    {
        $comment=$this->loadComment();
        $comment->approve();
        $this->redirect(array('post/show',
            'id'=>$comment->postId,
            '#'=>'c'.$comment->id));
    }
    else
        throw new CHttpException(400,'Invalid request...');
}
```

In the above, when the `approve` action is invoked via a POST request, we call the `approve()` method defined in the `Comment` model to change the status. We then redirect the user browser to the page displaying the post that this comment belongs to.

We also modify the `actionList()` method of `Comment` to show a list of comments pending approval.

```
public function actionList()
{
    $criteria=new CDbCriteria;
    $criteria->condition='Comment.status='.Comment::STATUS_PENDING;

    $pages=new CPagination(Comment::model()->count($criteria));
    $pages->pageSize=self::PAGE_SIZE;
    $pages->applyLimit($criteria);

    $comments=Comment::model()->with('post')->findAll($criteria);

    $this->render('list',array(
        'comments'=>$comments,
        'pages'=>$pages,
    ));
}
```

In the `list` view, we display the detail of every comment that is pending approval. In particular, we show an `approve` link button as follows,

```
<?php if($comment->status==Comment::STATUS_PENDING): ?>
    <span class="pending">Pending approval</span> |
    <?php echo CHtml::linkButton('Approve', array(
        'submit'=>array('comment/approve','id'=>$comment->id),
    )); ?> |
<?php endif; ?>
```

We use CHtml::linkButton() instead of CHtml::link() because the former would trigger a POST request while the latter a GET request. It is recommended that a GET request should not alter the data on the server. Otherwise, we face the danger that a user may inadvertently change the server-side data several times if he refreshes the page.

Chapter 5

Portlets

5.1 Creating Portlet Architecture

Features like "the most recent comments", "tag cloud" are better to be implemented in portlets. A portlet is a pluggable user interface component that renders a fragment of HTML code. In this section, we describe how to set up the portlet architecture for our blog application.

Based on the requirements analysis, we need four different portlets: the login portlet, the "user menu" portlet, the "tag cloud" portlet and the "recent comments" portlet. These portlets will be placed in the side bar section of every page.

5.1.1 Creating `Portlet` Class

We define a class named `Portlet` to serve as the base class for all our portlets. The base class contains the common properties and methods shared by all portlets. For example, it defines a `title` property that represents the title of a portlet; it defines how to decorate a portlet using a framed box with colored background.

The following code shows the definition of the `Portlet` base class. Because a portlet often contains both logic and presentation, we define `Portlet` by extending CWidget, which means a portlet is a widget and can be embedded in a view using the widget() method.

```
class Portlet extends CWidget
{
    public $title; // the portlet title
    public $visible=true; // whether the portlet is visible
    // ...other properties...

    public function init()
    {
        if($this->visible)
        {
```

```
            // render the portlet starting frame
            // render the portlet title
        }
    }

    public function run()
    {
        if($this->visible)
        {
            $this->renderContent();
            // render the portlet ending frame
        }
    }

    protected function renderContent()
    {
        // child class should override this method
        // to render the actual body content
    }
}
```

In the above code, the `init()` and `run()` methods are required by CWidget, which are called automatically when the widget is being rendered in a view. Child classes of `Portlet` mainly need to override the `renderContent()` method to generate the actual portlet body content.

5.1.2 Customizing Page Layout

It is time for us to adjust the page layout so that we can place portlets in the side bar section. The page layout is solely determined by the layout view file `/wwwroot/blog/protected/views/layouts/main.php`. It renders the common sections (e.g. header, footer) of different pages and embeds at an appropriate place the dynamic content that are generated by individual action views.

Our blog application will use the following layout:

```
<html>
<head>
......
<?php echo CHtml::cssFile(Yii::app()->baseUrl.'/css/main.css'); ?>
<title><?php echo $this->pageTitle; ?></title>
</head>

<body>

...header...
```

```
<div id="sidebar">
...list of portlets...
</div>

<div id="content">
<?php echo $content; ?>
</div>

...footer...

</body>
</html>
```

Besides customizing the layout view file, we also need to adjust the CSS file `/wwwroot/blog/css/main.css` so that the overall appearance would look like what we see in the blog demo. We will not go into details here.

5.2 Creating User Menu Portlet

In this section, we will develop our first concrete portlet - the user menu portlet which displays a list of menu items that are only available to authenticated users. The menu contains four items:

- Approve Comments: a hyperlink that leads to a list of comments pending approval;
- Create New Post: a hyperlink that leads to the post creation page;
- Manage Posts: a hyperlink that leads to the post management page;
- Logout: a link button that would log out the current user.

5.2.1 Creating `UserMenu` Class

We create the `UserMenu` class to represent the logic part of the user menu portlet. The class is saved in the file `/wwwroot/blog/protected/components/UserMenu.php` which has the following content:

```
<?php
class UserMenu extends Portlet
{
    public function init()
    {
        $this->title=CHtml::encode(Yii::app()->user->name);
```

```
        parent::init();
    }

    protected function renderContent()
    {
        $this->render('userMenu');
    }
}
```

The `UserMenu` class extends from the `Portlet` class that we created previously. It overrides both the `init()` method and the `renderContent()` method of `Portlet`. The former sets the portlet title to be the name of the current user; the latter generates the portlet body content by rendering a view named `userMenu`.

> **Tip:** Notice that we do not explicitly include the class file for `Portlet` even though we reference it in the code. This is due to the reason we explained in the previous section.

5.2.2 Creating `userMenu` View

Next, we create the `userMenu` view which is saved in the file `/wwwroot/blog/protected/components/views/userMenu.php`:

```
<ul>
<li><?php echo CHtml::link('Approve Comments', array('comment/list'))
    . ' (' . Comment::model()->pendingCommentCount . ')'; ?></li>
<li><?php echo CHtml::link('Create New Post',array('post/create')); ?></li>
<li><?php echo CHtml::link('Manage Posts',array('post/admin')); ?></li>
<li><?php echo CHtml::linkButton('Logout',array(
    'submit'=>'',
    'params'=>array('command'=>'logout'),
)); ?></li>
</ul>
```

> **Info:** By default, view files for a widget should be placed under the `views` subdirectory of the directory containing the widget class file. The file name must be the same as the view name.

In the view, we call CHtml::link to create the needed hyperlinks; we also call CHtml::linkButton to create a link button which works like a normal push button. When the button is clicked,

it submits an implicit form to the current page with the parameter `command` whose value is `logout`.

In order to respond to the clicking of the `logout` hyperlink, we need to modify the `init()` method of `UserMenu` as follows:

```
public function init()
{
    if(isset($_POST['command']) && $_POST['command']==='logout')
    {
        Yii::app()->user->logout();
        $this->controller->redirect(Yii::app()->homeUrl);
    }

    $this->title=CHtml::encode(Yii::app()->user->name);
    parent::init();
}
```

In the `init()` method, we check if there is a `command` POST variable whose value is `logout`. If so, we log out the current user and redirect the user browser to the application's home page. Note that the `redirect()` method will implicitly terminate the execution of the current application.

5.2.3 Using `UserMenu` Portlet

It is time for us to make use of our newly completed `UserMenu` portlet. We modify the layout view file `/wwwroot/blog/protected/views/layouts/main.php` as follows:

```
......

<div id="sidebar">

<?php $this->widget('UserMenu',array('visible'=>!Yii::app()->user->isGuest)); ?>

</div>

......
```

In the above, we call the `widget()` method to generate and execute an instance of the `UserMenu` class. Because the portlet should only be displayed to authenticated users, we toggle its `visible` property according to the `isGuest` property of the current user.

5.2.4 Testing `UserMenu` Portlet

Let's test what we have so far.

1. Open a browser window and enter the URL `http://www.example.com/blog/index.php`. Verify that there is nothing displayed in the side bar section of the page.
2. Click on the `Login` hyperlink and fill out the login form to login. If successful, verify that the `UserMenu` portlet appears in the side bar and the portlet has the username as its title.
3. Click on the 'Logout' hyperlink in the `UserMenu` portlet. Verify that the logout action is successful and the `UserMenu` portlet disappears.

5.2.5 Summary

What we have created is a portlet that is highly reusable. We can easily reuse it in a different project with little or no modification. Moreover, the design of this portlet follows closely the philosophy that logic and presentation should be separated. While we did not point this out in the previous sections, such practice is used nearly everywhere in a typical Yii application.

5.3 Creating Login Portlet

The skeleton application we created already contains a login page. In this section, we will convert this page into a login portlet named `UserLogin`. The portlet will be displayed in the side bar section of pages when the current user is a guest user who is not authenticated. If he logs in successfully, the portlet will disappear and the previously developed user menu portlet will show up.

5.3.1 Creating `UserLogin` Class

Like the user menu portlet, we create the `UserLogin` class to contain the logic of the user login portlet and save it in the file `/wwwroot/blog/protected/components/UserLogin.php`. The file has the following content:

```
<?php
class UserLogin extends Portlet
{
    public $title='Login';

    protected function renderContent()
```

```
    {
        $form=new LoginForm;
        if(isset($_POST['LoginForm']))
        {
            $form->attributes=$_POST['LoginForm'];
            if($form->validate())
                $this->controller->refresh();
        }
        $this->render('userLogin',array('form'=>$form));
    }
}
```

The code in the `renderContent()`method is copied from the `actionLogin()` method of `SiteController` that we generated at the beginning using the `yiic` tool. We mainly change the `render()` method call by rendering a view named `userLogin`. Notice also that we create an object of the `LoginForm` class in this method. The class represents the user input that we collect from the login form. It is in the file `/wwwroot/blog/protected/models/LoginForm.php` and is generated by the `yiic` tool when we create the skeleton application.

5.3.2 Creating `userLogin` View

The content of the `userLogin` view also comes largely from the `login` view for the `SiteController`'s `login` action. The view is saved in the file `/wwwroot/blog/protected/components/views/userLogin.php` and has the following content:

```
<?php echo CHtml::beginForm(); ?>
<div class="row">
<?php echo CHtml::activeLabel($form,'username'); ?>
<br/>
<?php echo CHtml::activeTextField($form,'username') ?>
<?php echo CHtml::error($form,'username'); ?>
</div>
<div class="row">
<?php echo CHtml::activeLabel($form,'password'); ?>
<br/>
<?php echo CHtml::activePasswordField($form,'password') ?>
<?php echo CHtml::error($form,'password'); ?>
</div>
<div class="row">
<?php echo CHtml::activeCheckBox($form,'rememberMe'); ?>
<?php echo CHtml::label('Remember me next time',CHtml::getActiveId($form,'rememberMe')); ?>
</div>
<div class="row">
<?php echo CHtml::submitButton('Login'); ?>
<p class="hint">You may login with <b>demo/demo</b></p>
</div>
```

```
<?php echo CHtml::endForm(); ?>
```

In the login form, we display a username text field and a password field. We also display a check box indicating whether the user login status should be remembered even if the browser is closed. The view has a local variable named `$form` which comes from the data passed to the `render()` method call in `UserLogin::renderContent()`.

Because `LoginForm` data model contains validation rules (like in the `Post` model), when a user submits the form, the model will perform data validation. If there is any validation error, the form will display it next to the incorrect input field via CHtml::error().

5.3.3 Using `UserLogin` Portlet

We use `UserLogin` like we do with `UserMenu` by modifying the layout file `/wwwroot/blog/protected/views/layouts/main.php` as follows,

```
......
<div id="sidebar">

<?php $this->widget('UserLogin',array('visible'=>Yii::app()->user->isGuest)); ?>

<?php $this->widget('UserMenu',array('visible'=>!Yii::app()->user->isGuest)); ?>

</div>
......
```

Notice that `UserLogin` is visible only when the current user is a guest, which is contrary to `UserMenu`.

5.3.4 Testing `UserLogin` Portlet

To test the `UserLogin` portlet, follow the steps below:

1. Access the URL `http://www.example.com/blog/index.php`. If the current user is not logged in, we should be able to see the `UserLogin` portlet.
2. Without entering anything in the login form, if we click the `Login` button, we should see error messages.
3. Try logging in with username `demo` and password `demo`. The current page will be refreshed, the `UserLogin` portlet disappears, and the `UserMenu` portlet appears.
4. Click on the `Logout` menu item in the `UserMenu` portlet, we should see that the `UserMenu` portlet disappears while the `UserLogin` portlet appears again.

5.3.5 Summary

The `UserLogin` portlet is a typical example that follows the MVC design pattern. It uses the `LoginForm` model to represent the data and business rules; it uses the `userLogin` view to generate user interface; and it uses the `UserLogin` class (a mini controller) to coordinate the model and the view.

5.4 Creating Tag Cloud Portlet

Tag cloud displays a list of post tags with visual decorations hinting the popularity of each individual tag.

5.4.1 Creating `TagCloud` Class

We create the `TagCloud` class in the file `/wwwroot/blog/protected/components/TagCloud.php`. The file has the following content:

```
<?php
class TagCloud extends Portlet
{
    public $title='Tags';

    public function getTagWeights()
    {
        return Tag::model()->findTagWeights();
    }

    protected function renderContent()
    {
        $this->render('tagCloud');
    }
}
```

In the above we invoke the `findTagWeights` method which is defined in the `Tag` class. The method returns a list of tags with their relative frequency weights. If a tag is associated with more posts, it receives higher weights. We will use the weights to control how the tags are displayed.

5.4.2 Creating `tagCloud` View

The `tagCloud` view is saved in the file `/wwwroot/blog/protected/components/views/tagCloud.php`. For each tag returned by `TagCloud::getTagWeights()`, it displays a hyperlink which would lead to the page listing the posts with that tag. The font size of the link is deter-

mined according to the weight value of the tag. The higher the weight, the bigger the fone size.

5.4.3 Using `TagCloud` Portlet

Usage of the `TagCloud` portlet is very simple. We modify the layout file `/wwwroot/blog/protected/views/layouts/main.php` as follows,

```
......
<div id="sidebar">

<?php $this->widget('UserLogin',array('visible'=>Yii::app()->user->isGuest)); ?>

<?php $this->widget('UserMenu',array('visible'=>!Yii::app()->user->isGuest)); ?>

<?php $this->widget('TagCloud'); ?>

</div>
......
```

5.5 Creating Recent Comments Portlet

In this section, we create the last portlet that displays a list of comments recently published.

5.5.1 Creating `RecentComments` Class

We create the `RecentComments` class in the file `/wwwroot/blog/protected/components/RecentComments.php`. The file has the following content:

```
<?php
class RecentComments extends Portlet
{
    public $title='Recent Comments';

    public function getRecentComments()
    {
        return Comment::model()->findRecentComments();
    }

    protected function renderContent()
    {
        $this->render('recentComments');
    }
}
```

In the above we invoke the `findRecentComments` method which is defined in the `Comment` class as follows,

```
class Comment extends CActiveRecord
{
    ......

    public function findRecentComments($limit=10)
    {
        $criteria=array(
            'condition'=>'Comment.status='.self::STATUS_APPROVED,
            'order'=>'Comment.createTime DESC',
            'limit'=>$limit,
        );
        return $this->with('post')->findAll($criteria);
    }
}
```

5.5.2 Creating `recentComments` View

The `recentComments` view is saved in the file `/wwwroot/blog/protected/components/views/recentComments.php`. The view simply displays every comment returned by the `RecentComments::getRecentComments()` method.

5.5.3 Using `RecentComments` Portlet

We modify the layout file `/wwwroot/blog/protected/views/layouts/main.php` to embed this last portlet,

```
......
<div id="sidebar">

<?php $this->widget('UserLogin',array('visible'=>Yii::app()->user->isGuest)); ?>

<?php $this->widget('UserMenu',array('visible'=>!Yii::app()->user->isGuest)); ?>

<?php $this->widget('TagCloud'); ?>

<?php $this->widget('RecentComments'); ?>

</div>
......
```

Chapter 6

Final Work

6.1 Beautifying URLs

The URLs linking various pages of our blog application currently look ugly. For example, the URL for the page showing a post looks like the following:

```
/index.php?r=post/show&id=1
```

In this section, we describe how to beautifying these URLs and make them SEO-friendly. Our goal is to be able to use the following URLs in the application:

- `/index.php/tag/yii`: leads to the page showing a list of posts with tag `yii`;
- `/index.php/posts`: leads to the page showing the latest posts;
- `/index.php/post/1`: leads to the page showing the detail of the post with ID 1;
- `/index.php/post/update/1`: leads to the page that allows updating the post with ID 1.

To achieve our goal, we modify the application configuration as follows,

```
return array(
    ......
    'components'=>array(
        ......
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
                'tag/<tag>'=>'post/list',
                'posts'=>'post/list',
                'post/<id:\d+>'=>'post/show',
```

```
                'post/update/<id:\d+>'=>'post/update',
            ),
        ),
    ),
);
```

In the above, we configure the urlManager component by setting its `urlFormat` property to be `path` and adding a set of `rules`.

The rules are used by `urlManager` to parse and create the URLs in the desired format. For example, the first rule says that if a URL `/index.php/tag/yii` is requested, the `urlManager` component should be responsible to dispatch the request to the route `post/list` and generate a `tag` GET parameter with the value `yii`. On the other hand, when creating a URL with the route `post/list` and parameter `tag`, the `urlManager` component will also use this rule to generate the desired URL `/index.php/tag/yii`. For this reason, we say that `urlManager` is a two-way URL manager.

The `urlManager` component can further beautify our URLs, such as hiding `index.php` in the URLs, appending suffix like `.html` to the URLs. We can obtain these features easily by configuring various properties of `urlManager` in the application configuration. For more details, please refer to the Guide.

6.2 Logging Errors

A production Web application often needs sophisticated logging for various events. In our blog application, we would like to log the errors occurring when it is being used. Such errors could be programming mistakes or users' misuse of the system. Logging these errors will help us to improve the blog application.

We enable the error logging by modifying the application configuration as follows,

```
return array(
    'preload'=>array('log'),

    ......

    'components'=>array(
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error, warning',
                ),
```

```
            ),
        ),
        ......
    ),
);
```

With the above configuration, if an error or warning occurs, detailed information will be logged and saved in a file located under the directory `/wwwroot/blog/protected/runtime`.

The `log` component offers more advanced features, such as sending log messages to a list of email addresses, displaying log messages in JavaScript console window, etc. For more details, please refer to the Guide.

6.3 Customizing Error Display

Our blog application is using the templates provided by Yii to display various errors. Because the style and wording are different from what we want, we would like to customize these templates. To do so, we create a set of view files under the directory `/wwwroot/blog/protected/views/system`.

We first create a file named `error.php`. This is the default view that will be used to display all kinds of errors if a more specific error view file is not available. Because this view file is used when an error occurs, it should not contain very complex PHP logic that may cause further errors. Note also that error view files do not use layout. Therefore, each view file should have complete page display.

We also create a file named `error403.php` to display 403 (unauthenticated) HTTP errors, and a file named `error404.php` to display 404 (page not found) HTTP errors.

To learn more details about the naming of these error view files, please refer to the Guide.

6.4 Final Tune-up and Deployment

We are close to finish our blog application. Before deployment, we would like to do some tune-ups.

6.4.1 Changing Home Page

We change to use the post list page as the home page. We modify the application configuration as follows,

```
return array(
```

```
    ......
    'defaultController'=>'post',
    ......
);
```

> **Tip:** Because `PostController` already declares `list` to be its default action, when we access the home page of the application, we will see the result generated by the `list` action of the post controller.

6.4.2 Enabling Schema Caching

Because ActiveRecord relies on the metadata about tables to determine the column information, it takes time to read the metadata and analyze it. This may not be a problem during development stage, but for an application running in production mode, it is a total waste of time if the database schema does not change. Therefore, we should enable the schema caching by modifying the application configuration as follows,

```
return array(
    ......
    'components'=>array(
        ......
        'cache'=>array(
            'class'=>'CDbCache',
        ),
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:/wwwroot/blog/protected/data/blog.db',
            'schemaCachingDuration'=>3600,
        ),
    ),
);
```

In the above, we first add a `cache` component which uses a default SQLite database as the caching storage. If our server is equipped with other caching extensions, such as APC, we could change to use them as well. We also modify the `db` component by setting its schemaCachingDuration property to be 3600, which means the parsed database schema data can remain valid in cache for 3600 seconds.

6.4.3 Disabling Debugging Mode

We modify the entry script file `/wwwroot/blog/index.php` by removing the line defining the constant `YII_DEBUG`. This constant is useful during development stage because it allows

Yii to display more debugging information when an error occurs. However, when the application is running in production mode, displaying debugging information is not a good idea because it may contain sensitive information such as where the script file is located, and the content in the file, etc.

6.4.4 Deploying the Application

The final deployment process manly involves copying the directory `/wwwroot/blog` to the target directory. The following checklist shows every needed step:

1. Install Yii in the target place if it is not available;
2. Copy the entire directory `/wwwroot/blog` to the target place;
3. Edit the entry script file `index.php` by pointing the `$yii` variable to the new Yii bootstrap file;
4. Edit the file `protected/yiic.php` by setting the `$yiic` variable to be the new Yii `yiic.php` file;
5. Change the permission of the directories `assets` and `protected/runtime` so that they are writable by the Web server process.

6.5 Future Enhancements

6.5.1 Using a Theme

Without writing any code, our blog application is already themeable. To use a theme, we mainly need to develop the theme by writing customized view files in the theme. For example, to use a theme named `classic` that uses a different page layout, we would create a layout view file `/wwwroot/blog/themes/classic/views/layouts/main.php`. We also need to change the application configuration to indicate our choice of the `classic` theme:

```
return array(
    ......
    'theme'=>'classic',
    ......
);
```

6.5.2 Internationalization

We may also internationalize our blog application so that its pages can be displayed in different languages. This mainly involves efforts in two aspects.

First, we may create view files in different languages. For example, for the `list` page of `PostController`, we can create a view file `/wwwroot/blog/protected/views/post/zh_cn/list.php`. When the application is configured to use simplified Chinese (the language code is `zh_cn`), Yii will automatically use this new view file instead of the original one.

Second, we may create message translations for those messages generated by code. The message translations should be saved as files under the directory `/wwwroot/blog/protected/messages`. We also need to modify the code where we use text strings by enclosing them in the method call `Yii::t()`.

For more details about internationalization, please refer to the Guide.

6.5.3 Improving Performance with Cache

While the Yii framework itself is very efficient, it is not necessarily true that an application written in Yii efficient. There are several places in our blog application that we can improve the performance. For example, the tag clould portlet could be one of the performance bottlenecks because it involves complex database query and PHP logic.

We can make use of the sophisticated caching feature provided by Yii to improve the performance. One of the most useful components in Yii is COutputCache, which caches a fragment of page display so that the underlying code generating the fragment does not need to be executed for every request. For example, in the layout file `/wwwroot/blog/protected/views/layouts/main.php`, we can enclose the tag cloud portlet with COutputCache:

```
<?php if($this->beginCache('tagCloud', array('duration'=>3600))) { ?>

<?php $this->widget('TagCloud'); ?>

<?php $this->endCache(); } ?>
```

With the above code, the tag cloud display will be served from cache instead of being generated on-the-fly for every request. The cached content will remain valid in cache for 3600 seconds.

6.5.4 Adding New Features

Our blog application only has very basic functionalities. To become a complete blog system, more features are needed, for example, calendar portlet, email notifications, post categorization, archived post portlet, and so on. We will leave the implementation of these features to interested readers.