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 which can be found in the Yii release files. 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.

> **Note:** This tutorial isn't a complete step by step guide. You will have to fix errors popping up, check API and read the definitive guide while following it.

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 1.1.1 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, not necessarily under a Web folder. Its `framework` directory contains all framework code and is the only framework 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. For our blog application, besides the minimal requirements needed by Yii, we also need to enable both the `pdo` and `pdo_sqlite` PHP extensions so that we can access SQLite databases.

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 to reduce certain repetitive coding tasks.

Open a command window and execute the following command:

```
% /wwwroot/yii/framework/yiic webapp /wwwroot/blog
Create a Web application under &#039;/wwwroot/blog&#039;? [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 should see that our skeleton application already has four fully functional pages: the homepage, the about page, 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 user requests the URL `http://www.example.com/blog/index.php?r=site/contact`;
2. The entry script is executed by the Web server to process the request;
3. An application instance is created and configured with initial property values specified in the application configuration file `/wwwroot/blog/protected/config/main.php`;
4. 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 (the `actionContact` method in `/wwwroot/blog/protected/controllers/SiteController.php`);

5. The application creates the `site` controller in terms of a `SiteController` instance and then executes it;
6. The `SiteController` instance executes the `contact` action by calling its `actionContact()` method;
7. 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/column1.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 decide to use the following database tables to store the persistent data for our blog application:

- `tbl_user` stores the user information, including username and password.
- `tbl_post` stores the blog 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:
    - 1, meaning the post is in draft and is not visible to public;
    - 2, meaning the post is published to public;
    - 3, meaning the post is outdated and is not visible in the post list (still accessible individually, though).
  - `tags`: optional, a list of comma-separated words categorizing the post.
- `tbl_comment` stores the post comment information. Each comment is associated with a post and mainly consists of the following columns:
  - `author`: required, the author name;
  - `email`: required, the author email;
  - `url`: optional, the author website URL;
  - `content`: required, the comment content in plain text format.
  - `status`: required, status of the comment, which indicates whether the comment is approved (value 2) or not (value 1).
- `tbl_tag` stores post tag frequency information that is needed to implement the tag cloud feature. The table mainly contains the following columns:
  - `name`: required, the unique tag name;
  - `frequency`: required, the number of times that the tag appears in posts.
- `tbl_lookup` stores generic lookup information. It is essentially a map between integer values and text strings. The former is the data representation in our code, while the latter is the corresponding presentation to end users. For example, we use integer 1 to represent the draft post status and string `Draft` to display this status to end users. This table mainly contains the following columns:

- `name`: the textual representation of the data item that is to be displayed to end users;
- `code`: the integer representation of the data item;
- `type`: the type of the data item;
- `position`: the relative display order of the data item among other items of the same type.

The following entity-relation (ER) diagram shows the table structure and relationships about the above tables.

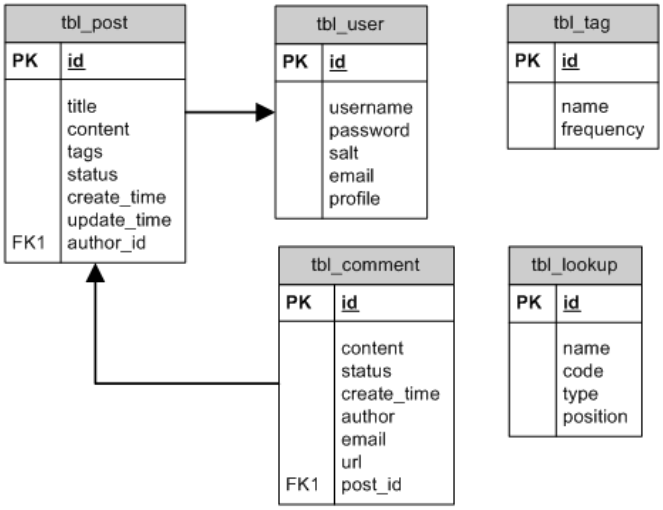

Figure 1.1: Entity-Relation Diagram of the Blog Database

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`.

> **Info:** We name all our table names and column names in lower case. This is because different DBMS often have different case-sensitivity treatment and we want to avoid troubles like this.
>
> We also prefix all our tables with `tbl_`. This serves for two purposes. First, the prefix introduces a namespace to these tables in case when they need to coexist with other tables in the same database, which often happens in a shared hosting environment where a single database is being used by multiple applications. Second, using table prefix reduces the possibility of having some table names that are reserved keywords in DBMS.

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.
- 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 the 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 in the 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 writable property of the application instance.

We configure the `db` component as follows,

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

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

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.

We would like to explain a bit more about the `tablePrefix` property that we set in the configuration. This tells the `db` connection that it should respect the fact we are using `tbl_` as the prefix to our database table names. In particular, if in a SQL statement there is a token enclosed within double curly brackets (e.g. {{`post`}}), then the `db` connection should translate it into a name with the table prefix (e.g. `tbl_post`) before sending it to DBMS for execution. This feature is especially useful if in future we need to modify the table name prefix without touching our source code. For example, if we are developing a generic content management system (CMS), we may exploit this feature so that when it is being installed in a new environment, we can allow users to choose a table prefix they like.

Tip: If you want to use MySQL instead of SQLite to store data, you may create a MySQL database named `blog` using the SQL statements in `/wwwroot/yii/demos/blog/protected/data/schema.mysql.sql`. Then, modify the application configuration as follows,

```
return array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'connectionString' => 'mysql:host=localhost;dbname=blog',
            'emulatePrepare' => true,
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
            'tablePrefix' => 'tbl_',
        ),
    ),
  ......
);
```

2.2 Scaffolding

Create, read, update and delete (CRUD) are the four basic operations of data objects in an application. Because the task of implementing the CRUD operations is so common when developing Web applications, Yii provides some code generation tools under the name of *Gii* that can automate this process (also known as *scaffolding*) for us.

Note: Gii has been available since version 1.1.2. Before that, you would have to use the yiic shell tool to achieve the same task.

In the following, we will describe how to use this tool to implement CRUD operations for posts and comments in our blog application.

2.2.1 Installing Gii

We first need to install Gii. Open the file `/wwwroot/blog/protected/config/main.php` and add the following code:

```
return array(
    ......
    'import'=>array(
```

```
        'application.models.*',
        'application.components.*',
    ),

    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',
            'password'=>'pick up a password here',
        ),
    ),
);
```

The above code installs the a module named `gii`, which enables us to access the Gii module by visiting the following URL in browser:

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

We will be prompted to enter a password. Enter the password that we set in `/wwwroot/blog/protected/config/main.php` previously, and we should see a page listing all available code generation tools.

> **Note:** The above code should be removed when running on the production machine. Code generation tools should only be used on development machines.

2.2.2 Creating Models

We first need to create a model class for each of our database tables. The model classes will allow us to access the database in an intuitive object-oriented fashion, as we will see later in this tutorial.

Click on the `Model Generator` link to start using the model generation tool.

On the `Model Generator` page, enter `tbl_user` (the user table name) in the `Table Name` field, `tbl_` in the `Table Prefix` field and then press the `Preview` button. A preview table will show up. We can click on the link in the table to preview the code to be generated. If everything is ok, we can press the `Generate` button to generate the code and save it into a file.

> **Info:** Because the code generator needs to save the generated code into files, it is required that the Web process have the permission to create and modify the corresponding files. For simplicity, we may give the Web process the write permission to the whole `/wwwroot/blog` directory. Note that this is only needed on development machines when using `Gii`.

Repeat the same procedure for the rest of the database tables, including `tbl_post`, `tbl_comment`, `tbl_tag` and `tbl_lookup`.

> **Tip:** We can also enter an asterisk character '*' *in the* `Table Name` *field. This will generate a model class for *every* database table in a single shot.

At this stage, we will have the following newly created files:

- `models/User.php` contains the `User` class that extends from CActiveRecord and can be used to access the `tbl_user` database table;
- `models/Post.php` contains the `Post` class that extends from CActiveRecord and can be used to access the `tbl_post` database table;
- `models/Tag.php` contains the `Tag` class that extends from CActiveRecord and can be used to access the `tbl_tag` database table;
- `models/Comment.php` contains the `Comment` class that extends from CActiveRecord and can be used to access the `tbl_comment` database table;
- `models/Lookup.php` contains the `Lookup` class that extends from CActiveRecord and can be used to access the `tbl_lookup` database table.

2.2.3 Implementing CRUD Operations

After the model classes are created, we can use the `Crud Generator` to generate the code implementing the CRUD operations for these models. We will do this for the `Post` and `Comment` models.

On the `Crud Generator` page, enter `Post` (the name of the post model class we just created) in the `Model Class` field, and then press the `Preview` button. We will see a lot more files will be generated. Press the `Generate` button to generate them.

Repeat the same procedure for the `Comment` model.

Let's take a look at the files generated by the CRUD generator. All the files are generated under `/wwwroot/blog/protected`. For convenience, we group them into controller files and view files:

- controller files:
  - `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/view.php` is the view file that displays the detailed information of a post;
  - `views/post/index.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 embedded in `views/post/create.php` and `views/post/update.php`. It displays the HTML form for collecting post information.
  - `views/post/_view.php` is the partial view file used by `views/post/index.php`. It displays the brief view of a single post.
  - `views/post/_search.php` is the partial view file used by `views/post/admin.php`. It displays a search form.
  - a similar set of view files are also generated for comment.

2.2.4 Testing

We can test the features implemented by the code we just generated 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 `author_id` and `create_time`, 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.

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 user requests the URL `http://www.example.com/blog/index.php?r=post`;
2. The entry script is executed by the Web server which creates and initializes an application instance to handle the request;
3. The application creates an instance of `PostController` and executes it;
4. The `PostController` instance executes the `index` action by calling its `actionIndex()` method. Note that `index` is the default action if the user does not specify an action to execute in the URL;
5. The `actionIndex()` method queries database to bring back the list of recent posts;
6. The `actionIndex()` method renders the `index` 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)
            $this->errorCode=self::ERROR_USERNAME_INVALID;
        else if(!$user->validatePassword($this->password))
            $this->errorCode=self::ERROR_PASSWORD_INVALID;
        else
        {
            $this->_id=$user->id;
            $this->username=$user->username;
            $this->errorCode=self::ERROR_NONE;
        }
        return $this->errorCode==self::ERROR_NONE;
    }

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

In the `authenticate()` method, we use the `User` class to look for a row in the `tbl_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 `gii` tool in the prior section. Because the `User` class extends from CActiveRecord, we can exploit the ActiveRecord feature to access the `tbl_user` table in an OOP fashion.

In order to check if the user has entered a valid password, we invoke the `validatePassword` method of the `User` class. We need to modify the file `/wwwroot/blog/protected/models/User.php` as follows. Note that instead of storing the plain password in the database, we store a hash of the password and a randomly generated salt. When validating the user-entered password, we should compare the hash results, instead. We use the PHP built-in function `crypt()` to hash the password and to validate it, for complete details see the Wiki article Use crypt() for password storage.

```
class User extends CActiveRecord
{
    ......
    public function validatePassword($password)
    {
        return crypt($password,$this->password)===$this->password;
    }

    public function hashPassword($password)
    {
        return crypt($password, $this->generateSalt());
    }
}
```

In the `UserIdentity` class, we also override the `getId()` method which returns the `id` value of the user found in the `tbl_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 `tbl_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 `tbl_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 `gii` 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 `Gii` tool mainly needs to be modified in two places:

- the `rules()` method: specifies the validation rules for the model attributes;
- the `relations()` method: specifies the related objects;

> **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 entered by users are correct before they are saved to the database. For example, the `status` attribute of `Post` should be an integer 1, 2 or 3. The `Gii` 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 requirement 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(1,2,3)),
        array('tags', 'match', 'pattern'=>'/^[\w\s,]+$/',
            'message'=>'Tags can only contain word characters.'),
```

```
        array('tags', 'normalizeTags'),

        array('title, status', 'safe', 'on'=>'search'),
    );
}
```

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 1 (draft), 2 (published) or 3 (archived); and the `tags` attribute should only contain word characters and commas. In addition, we use `normalizeTags` to normalize the user-entered tags so that the tags are unique and properly separated with commas. The last rule is used by the search feature, which we will describe later.

The validators such as `required`, `length`, `in` and `match` are all built-in validators provided by Yii. The `normalizeTags` validator is a method-based validator that we need to define in the `Post` class. For more information about how to specify validation rules, please refer to the Guide.

```
public function normalizeTags($attribute,$params)
{
    $this->tags=Tag::array2string(array_unique(Tag::string2array($this->tags)));
}
```

where `array2string` and `string2array` are new methods we need to define in the `Tag` model class:

```
public static function string2array($tags)
{
    return preg_split('/\s*,\s*/',trim($tags),-1,PREG_SPLIT_NO_EMPTY);
}

public static function array2string($tags)
{
    return implode(', ',$tags);
}
```

The rules declared in the `rules()` method are executed one by one when we call the validate() or save() method of the model instance.

> **Note:** It is very important to remember that attributes appearing in `rules()` must be those to be entered by end users. Other attributes, such as `id` and `create_time` in the `Post` model, which are set by our code or database, should not be in `rules()`. For more details, please refer to Securing Attribute Assignments.

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

3.1.2 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', 'author_id'),
        'comments' => array(self::HAS_MANY, 'Comment', 'post_id',
            'condition'=>'comments.status='.Comment::STATUS_APPROVED,
            'order'=>'comments.create_time DESC'),
        'commentCount' => array(self::STAT, 'Comment', 'post_id',
            'condition'=>'status='.Comment::STATUS_APPROVED),
    );
}
```

We also introduce in the `Comment` model class two constants that are used in the above method:

```
class Comment extends CActiveRecord
{
    const STATUS_PENDING=1;
    const STATUS_APPROVED=2;
    ......
}
```

The relations declared in `relations()` state that

- A post belongs to an author whose class is `User` and the relationship is established based on the `author_id` attribute value of the post;
- A post has many comments whose class is `Comment` and the relationship is established based on the `post_id` attribute value of the comments. These comments should be sorted according to their creation time and the comments must be approved.

- The `commentCount` relation is a bit special as it returns back an aggregation result which is about how many comments the post has.

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.

3.1.3 Adding `url` Property

A post is a content that is associated with a unique URL for viewing it. Instead of calling CWebApplication::createUrl everywhere in our code to get this URL, we may add a `url` property in the `Post` model so that the same piece of URL creation code can be reused. Later when we describe how beautify URLs, we will see adding this property will bring us great convenience.

To add the `url` property, we modify the `Post` class by adding a getter method like the following:

```
class Post extends CActiveRecord
{
    public function getUrl()
    {
        return Yii::app()->createUrl('post/view', array(
            'id'=>$this->id,
            'title'=>$this->title,
        ));
    }
}
```

Note that in addition to the post ID, we also add the post title as a GET parameter in the URL. This is mainly for search engine optimization (SEO) purpose, as we will describe in Beautifying URLs.

Because CComponent is the ultimate ancestor class of `Post`, adding the getter method `getUrl()` enables us to use the expression like `$post->url`. When we access `$post->url`,

the getter method will be executed and its result is returned as the expression value. For more details about such component features, 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 textual representation so that it is more intuitive when being displayed to end users. In a large system, the similar requirement is very common.

As a generic solution, we use the `tbl_lookup` table to store the mapping between integer values and textual representations that are needed by other data objects. We modify the `Lookup` model class as follows to more easily access the textual data in the table,

```
class Lookup extends CActiveRecord
{
......

    private static $_items=array();

    public static function items($type)
    {
        if(!isset(self::$_items[$type]))
            self::loadItems($type);
        return self::$_items[$type];
    }

    public static function item($type,$code)
    {
        if(!isset(self::$_items[$type]))
            self::loadItems($type);
        return isset(self::$_items[$type][$code]) ? self::$_items[$type][$code] : false;
    }

    private static function loadItems($type)
    {
        self::$_items[$type]=array();
        $models=self::model()->findAll(array(
            'condition'=>'type=:type',
            'params'=>array(':type'=>$type),
            'order'=>'position',
        ));
        foreach($models as $model)
            self::$_items[$type][$model->code]=$model->name;
    }
}
```

Our new code mainly provides two static methods: `Lookup::items()` and `Lookup::item()`. The former returns a list of strings belonging to the specified data type, while the latter returns a particular string for the given data type and data value.

Our blog database is pre-populated with two lookup types: `PostStatus` and `CommentStatus`. The former refers to the possible post statuses, while the latter the comment statuses.

In order to make our code easier to read, we also declare a set of constants to represent the status integer values. We should use these constants through our code when referring to the corresponding status values.

```
class Post extends CActiveRecord
{
    const STATUS_DRAFT=1;
    const STATUS_PUBLISHED=2;
    const STATUS_ARCHIVED=3;
    ......
}
```

Therefore, we can call `Lookup::items('PostStatus')` to get the list of possible post statuses (text strings indexed by the corresponding integer values), and call `Lookup::item('PostStatus', Post::STATUS_PUBLISHED)` to get the string representation of the published status.

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.

3.2.1 Customizing Access Control

The first thing we want to do is to customize the access control because the code generated by `gii` 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('index', 'view'),
            '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 `index` and `view` 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, `gii` 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`, `tags` and `status`. We use plain text fields to collect inputs for the first three 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 $form->dropDownList($model,'status',Lookup::items('PostStatus')); ?>
```

In the above, we call `Lookup::items('PostStatus')` to bring back the list of post statuses.

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

```
protected function beforeSave()
{
    if(parent::beforeSave())
```

```
    {
        if($this->isNewRecord)
        {
            $this->create_time=$this->update_time=time();
            $this->author_id=Yii::app()->user->id;
        }
        else
            $this->update_time=time();
        return true;
    }
    else
        return false;
}
```

When we save a post, we want to update the `tbl_tag` table to reflect the change of tag frequencies. We can do this work in the `afterSave()` method, which is automatically invoked by Yii after a post is successfully saved into the database.

```
protected function afterSave()
{
    parent::afterSave();
    Tag::model()->updateFrequency($this->_oldTags, $this->tags);
}

private $_oldTags;

protected function afterFind()
{
    parent::afterFind();
    $this->_oldTags=$this->tags;
}
```

In the implementation, because we want to detect if the user changes the tags in case he is updating an existing post, we need to know what the old tags are. For this reason, we also write the `afterFind()` method to keep the old tags in the variable `_oldTags`. The method `afterFind()` is invoked automatically by Yii when an AR record is populated with the data from database.

We are not going to give details of the `Tag::updateFrequency()` method here. Interested readers may refer to the file `/wwwroot/yii/demos/blog/protected/models/Tag.php`.

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 `index` operation while the latter the `view` operation. In this

section, we customize both operations to fulfill our initial requirements.

3.3.1 Customizing `view` Operation

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

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

```
public function actionView()
{
    $post=$this->loadModel();
    $this->render('view',array(
        'model'=>$post,
    ));
}

private $_model;

public function loadModel()
{
    if($this->_model===null)
    {
        if(isset($_GET['id']))
        {
            if(Yii::app()->user->isGuest)
                $condition='status='.Post::STATUS_PUBLISHED
                    .' OR status='.Post::STATUS_ARCHIVED;
            else
                $condition='';
            $this->_model=Post::model()->findByPk($_GET['id'], $condition);
        }
        if($this->_model===null)
            throw new CHttpException(404,'The requested page does not exist.');
    }
    return $this->_model;
}
```

Our change mainly lies in the `loadModel()` 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 or archived (when the user is a guest), we will throw a 404 HTTP error. Otherwise the post object is returned to `actionView()` which in turn passes the post object to the view script for further display.

> **Tip:** Yii captures HTTP exceptions (instances of CHttpException) and displays them in either predefined templates or customized error views. The skeleton application generated by `yiic` already contains a customized error view in `/wwwroot/blog/protected/views/site/error.php`. We can modify this file if we want to further customize the error display.

The change in the `view` script is mainly about ajdusting the formatting and styles of the post display. We will not go into details here. Interested readers may refer to `/wwwroot/blog/protected/views/post/view.php`.

3.3.2 Customizing `index` Operation

Like the `view` operation, we customize the `index` operation in two places: the `actionIndex()` method in `PostController` and the view file `/wwwroot/blog/protected/views/post/index.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 `actionIndex()` method in `PostController`:

```
public function actionIndex()
{
    $criteria=new CDbCriteria(array(
        'condition'=>'status='.Post::STATUS_PUBLISHED,
        'order'=>'update_time DESC',
        'with'=>'commentCount',
    ));
    if(isset($_GET['tag']))
        $criteria->addSearchCondition('tags',$_GET['tag']);

    $dataProvider=new CActiveDataProvider('Post', array(
        'pagination'=>array(
            'pageSize'=>5,
        ),
        'criteria'=>$criteria,
    ));

    $this->render('index',array(
        'dataProvider'=>$dataProvider,
    ));
}
```

In the above, we first create a query criteria for retrieving post list. The criteria states that only published posts should be returned and they should be sorted according to their

update time in descending order. Because when displaying a post in the list, we want to show how many comments the post has received, in the criteria we also specify to bring back `commentCount`, which if you remember, is a relation declared in `Post::relations()`.

In case when a user wants to see posts with a specific tag, we would add a search condition to the criteria to look for the specified tag.

Using the query criteria, we create a data provider, which mainly serves for three purposes. First, it does pagination of the data when too many results may be returned. Here we customize the pagination by setting the page size to be 5. Second, it does sorting according to the user request. And finally, it feeds the paginated and sorted data to widgets or view code for presentation.

After we finish with `actionIndex()`, we modify the `index` view as follows. Our change is mainly about adding the `h1` header when the user specifies to display posts with a tag.

```
<?php if(!empty($_GET['tag'])): ?>
<h1>Posts Tagged with <i><?php echo CHtml::encode($_GET['tag']); ?></i></h1>
<?php endif; ?>

<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
    'template'=>"{items}\n{pager}",
)); ?>
```

Note that in the above, we use CListView to display the post list. This widget requires a partial view to display the detail of each individual post. Here we specify the partial view to be `_view`, which means the file `/wwwroot/blog/protected/views/post/_view.php`. In this view script, we can acccess the post instance being displayed via a local variable named `$data`.

3.4 Managing Posts

Managing posts mainly refers to listing posts in an administrative view that allows us to see posts with all statuses, updating them and deleting them. They are accomplished by the `admin` operation and the `delete` operation, respectively. The code generated by `Gii` 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 posts with all statuses in a tabular view. The view supports sorting and pagination. The following is the `actionAdmin()` method in `PostController`:

```
public function actionAdmin()
{
    $model=new Post('search');
    if(isset($_GET['Post']))
        $model->attributes=$_GET['Post'];
    $this->render('admin',array(
        'model'=>$model,
    ));
}
```

The above code is generated by the `Gii` tool without any modification. It first creates a `Post` model under the `search` scenario. We will use this model to collect the search conditions that the user specifies. We then assign to the model the user-supplied data, if any. Finally, we render the `admin` view with the model.

Below is the code for the `admin` view:

```
<?php
$this->breadcrumbs=array(
    'Manage Posts',
);
?>
<h1>Manage Posts</h1>

<?php $this->widget('zii.widgets.grid.CGridView', array(
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'columns'=>array(
        array(
            'name'=>'title',
            'type'=>'raw',
            'value'=>'CHtml::link(CHtml::encode($data->title), $data->url)'
        ),
        array(
            'name'=>'status',
            'value'=>'Lookup::item("PostStatus",$data->status)',
            'filter'=>Lookup::items('PostStatus'),
        ),
        array(
            'name'=>'create_time',
            'type'=>'datetime',
```

```
            'filter'=>false,
        ),
        array(
            'class'=>'CButtonColumn',
        ),
    ),
)); ?>
```

We use CGridView to display the posts. It allows us to sort by a column and paginate through the posts if there are too many to be displayed in a single page. Our change is mainly about how to display each column. For example, for the `title` column, we specify that it should be displayed as a hyperlink that points to the detailed view of the post. The expression `$data->url` returns the value of the `url` property that we define in the `Post` class.

> **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

In the `admin` data grid, there is a delete button in each row. Clicking on the button should delete the corresponding post. Internally, this triggers the `delete` action implemented as follows:

```
public function actionDelete()
{
    if(Yii::app()->request->isPostRequest)
    {
        // we only allow deletion via POST request
        $this->loadModel()->delete();

        if(!isset($_GET['ajax']))
            $this->redirect(array('index'));
    }
    else
        throw new CHttpException(400,'Invalid request. Please do not repeat this request again.');
}
```

The above code is the one generated by the `Gii` tool without any change. We would like to explain a little bit more about the checking on `$_GET['ajax']`. The CGridView widget has a very nice feature that its sorting, pagination and deletion operations are all done in AJAX mode by default. That means, the whole page does not get reloaded if any of

the above operations is performed. However, it is also possible that the widget runs in non-AJAX mode (by setting its `ajaxUpdate` property to be false or disabling JavaScript on the client side). It is necessary for the `delete` action to differentiate these two scenarios: if the delete request is made via AJAX, we should not redirect the user browser; otherwise, we should.

Deleting a post should also cause the deletion of all comments for that post. In addition, we should also update the `tbl_tag` table regarding the tags for the deleted post. Both of these tasks can be achieved by writing an `afterDelete` method in the `Post` model class as follows,

```
protected function afterDelete()
{
    parent::afterDelete();
    Comment::model()->deleteAll('post_id='.$this->id);
    Tag::model()->updateFrequency($this->tags, '');
}
```

The above code is very straightforward: it first deletes all those comments whose `post_id` is the same as the ID of the deleted post; it then updates the `tbl_tag` table for the `tags` of the deleted post.

> **Tip:** We have to explicitly delete all comments for the deleted post here because SQLite does not really support foreign key constraints. In a DBMS that supports this constraint (such as MySQL, PostgreSQL), the foreign key constraint can be set up such that the DBMS automatically deletes the related comments if the post is deleted. In that case, we no longer this explicit deletion call in our code.

Chapter 4

Comment Management

4.1 Customizing Comment Model

For the `Comment` model, we mainly need to customize the `rules()` and `attributeLabels()` methods. The `attributeLabels()` method returns a mapping between attribute names and attribute labels. We do not need to touch `relations()` since the code generated by the `Gii` tool is good enough.

4.1.1 Customizing `rules()` Method

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

```
public function rules()
{
    return array(
        array('content, author, email', 'required'),
        array('author, email, url', 'length', 'max'=>128),
        array('email','email'),
        array('url','url'),
    );
}
```

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; and the `url` attribute must be a valid URL.

4.1.2 Customizing `attributeLabels()` Method

We then customize the `attributeLabels()` method to declare the label display for each model attribute. This method returns an array consisting of name-label pairs. When we call CHtml::activeLabel() to display an attribute label.

```
public function attributeLabels()
{
    return array(
        'id' => 'Id',
        'content' => 'Comment',
        'status' => 'Status',
        'create_time' => 'Create Time',
        'author' => 'Name',
        'email' => 'Email',
        'url' => 'Website',
        'post_id' => 'Post',
    );
}
```

> **Tip:** If the label for an attribute is not declared in `attributeLabels()`, an algorithm will be used to generate an appropriate label. For example, a label `Create Time` will be generated for attributes `create_time` or `createTime`.

4.1.3 Customizing Saving Process

Because we want to record the creation time of a comment, we override the `beforeSave()` method of `Comment` like we do for the `Post` model:

```
protected function beforeSave()
{
    if(parent::beforeSave())
    {
        if($this->isNewRecord)
            $this->create_time=time();
        return true;
    }
    else
        return false;
}
```

4.2 Creating and Displaying Comments

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

In order to enhance the user interactivity, we would like to prompt users the possible errors each time he finishes entering one field. This is known client-side input validation. We will show how this can be done in Yii seamlessly and extremely easy. Note that this requires Yii version 1.1.1 or later.

4.2.1 Displaying Comments

Instead of displaying and creating comments on individual pages, we use the post detail page (generated by the `view` action of `PostController`). 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 detail page, we modify the view script `/wwwroot/blog/protected/views/post/view.php` as follows,

```
...post view here...

<div id="comments">
    <?php if($model->commentCount>=1): ?>
        <h3>
            <?php echo $model->commentCount . 'comment(s)'; ?>
        </h3>

        <?php $this->renderPartial('_comments',array(
            'post'=>$model,
            'comments'=>$model->comments,
        )); ?>
    <?php endif; ?>
</div>
```

In the above, we call `renderPartial()` to render a partial view named `_comments` to display the list of comments belonging to the current post. Note that in the view we use the expression `$model->comments` to retrieve the comments for the post. This 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 proper comments. This feature is known as lazy relational query.

The partial view `_comments` is not very interesting. It mainly goes through every comment and displays the detail of it. Interested readers may refer to `/wwwroot/yii/demos/blog/protected/views/post/_comments.php`.

4.2.2 Creating Comments

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

```
public function actionView()
{
    $post=$this->loadModel();
```

```
    $comment=$this->newComment($post);

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

protected function newComment($post)
{
    $comment=new Comment;
    if(isset($_POST['Comment']))
    {
        $comment->attributes=$_POST['Comment'];
        if($post->addComment($comment))
        {
            if($comment->status==Comment::STATUS_PENDING)
                Yii::app()->user->setFlash('commentSubmitted','Thank you for your comment. Your comment will be pos
            $this->refresh();
        }
    }
    return $comment;
}
```

And then we modify the `Post` model class by adding the method `addComment()` as follows,

```
public function addComment($comment)
{
    if(Yii::app()->params['commentNeedApproval'])
        $comment->status=Comment::STATUS_PENDING;
    else
        $comment->status=Comment::STATUS_APPROVED;
    $comment->post_id=$this->id;
    return $comment->save();
}
```

In the above, we call the `newComment()` method before we render `view`. In the `newComment()` method, we generate a `Comment` instance and check if the comment form is submitted. If so, we try to add the comment for the post by calling `$post->addComment($comment)`. If it goes through, we refresh the post detail page, which will display the newly created comment unless approval is required. In the case where the comment first requires approval prior to display, we will show a flash message to indicate to the user that the comment will be displayed once approved. A flash message is usually a confirmation message displayed to end users. If the user clicks on the refresh button of his browser, the message will disappear.

We also need to modify `/wwwroot/blog/protected/views/post/view.php` furthermore,

```
......
<div id="comments">
    ......
    <h3>Leave a Comment</h3>

    <?php if(Yii::app()->user->hasFlash('commentSubmitted')): ?>
        <div class="flash-success">
            <?php echo Yii::app()->user->getFlash('commentSubmitted'); ?>
        </div>
    <?php else: ?>
        <?php $this->renderPartial('/comment/_form',array(
            'model'=>$comment,
        )); ?>
    <?php endif; ?>

</div><!-- comments -->
```

In the above code, we display the flash message if it is available. If not, we display the comment input form by rendering the partial view `/wwwroot/blog/protected/views/comment/_form.php`.

4.2.3 Ajax-based Validation

In order to improve user experience, we can use Ajax-based form field validation so that the user is provided with validation feedback as they fill out the form, before having to submit the entire form to the server. To support Ajax-based validation on the comment form, we need to make some minor changes to both the comment form view `/wwwroot/blog/protected/views/comment/_form.php` and the `newComment()` method.

In the `_form.php` file, we mainly need to set CActiveForm::enableAjaxValidation to be true when we create the CActiveForm widget:

```
<div class="form">

<?php $form=$this->beginWidget('CActiveForm', array(
    'id'=>'comment-form',
    'enableAjaxValidation'=>true,
)); ?>
......
<?php $this->endWidget(); ?>

</div><!-- form -->
```

And in the `newComment()` method, we insert a piece of code to respond to the AJAX validation requests. The code checks if there is a `POST` variable named `ajax`. If so, it displays the validation results by calling CActiveForm::validate.

```
protected function newComment($post)
{
    $comment=new Comment;

    if(isset($_POST['ajax']) && $_POST['ajax']==='comment-form')
    {
        echo CActiveForm::validate($comment);
        Yii::app()->end();
    }

    if(isset($_POST['Comment']))
    {
        $comment->attributes=$_POST['Comment'];
        if($post->addComment($comment))
        {
            if($comment->status==Comment::STATUS_PENDING)
                Yii::app()->user->setFlash('commentSubmitted','Thank you for your comment. Your comment will be pos
            $this->refresh();
        }
    }
    return $comment;
}
```

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 `Gii` for updating and deleting comments remains largely unchanged.

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->loadModel();
        $comment->approve();
        $this->redirect(array('index'));
    }
    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.

Of course, we also need to create the `approve()` method in the `Comment` model. It is as follows,

```
public function approve()
{
    $this->status=Comment::STATUS_APPROVED;
    $this->update(array('status'));
}
```

Here we are simply setting the status property of the comment to `approved` as defined by the status constants in the `Comment` class:

```
class Comment extends CActiveRecord
{
    ...

    const STATUS_PENDING=1;
    const STATUS_APPROVED=2;

    ..
}
```

and then calling the `update()` method to save this newly set property to the database.

We also modify the `actionIndex()` method of `CommentController` to show all comments. We would like to see comments pending approval show up first.

```
public function actionIndex()
```

```
{
    $dataProvider=new CActiveDataProvider('Comment', array(
        'criteria'=>array(
            'with'=>'post',
            'order'=>'t.status, t.create_time DESC',
        ),
    ));

    $this->render('index',array(
        'dataProvider'=>$dataProvider,
    ));
}
```

Notice that in the above code, because both `tbl_post` and `tbl_comment` have columns `status` and `create_time`, we need to disambiguate the corresponding column reference by prefixing them with table alias names. As described in the guide, the alias for the primary table in a relational query is always `t`. Therefore, we are prefixing `t` to the `status` and `create_time` columns in the above code to indicate we want these values taken from the primary table, `tbl_comment`.

Like the post index view, the `index` view for `CommentController` uses CListView to display the comment list which in turn uses the partial view `/wwwroot/blog/protected/views/comment/_view.php` to display the detail of each individual comment. We will not go into details here. Interested readers may refer to the corresponding file in the blog demo `/wwwroot/yii/demos/blog/protected/views/comment/_view.php`.

Chapter 5

Portlets

5.1 Creating User Menu Portlet

Based on the requirements analysis, we need three different portlets: the "user menu" portlet, the "tag cloud" portlet and the "recent comments" portlet. We will develop these portlets by extending the CPortlet widget provided by Yii.

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.

5.1.1 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:

```
Yii::import('zii.widgets.CPortlet');

class UserMenu extends CPortlet
{
    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 `CPortlet` class from the `zii` library. It overrides both the `init()` method and the `renderContent()` method of `CPortlet`. 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 have to explicitly include the `CPortlet` class by calling `Yii::import()` before we refer to it the first time. This is because `CPortlet` is part of the `zii` project – the official extension library for Yii. For performance consideration, classes in this project are not listed as core classes. Therefore, we have to import it before we use it the first time.

5.1.2 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('Create New Post',array('post/create')); ?></li>
    <li><?php echo CHtml::link('Manage Posts',array('post/admin')); ?></li>
    <li><?php echo CHtml::link('Approve Comments',array('comment/index'))
        . ' (' . Comment::model()->pendingCommentCount . ')'; ?></li>
    <li><?php echo CHtml::link('Logout',array('site/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.

5.1.3 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/column2.php` as follows:

```
......
<div id="sidebar">
    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>
</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 only call `widget()` when the `isGuest` property of the current user is false (meaning the user is authenticated).

5.1.4 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.

5.1.5 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.

5.2 Creating Tag Cloud Portlet

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

5.2.1 Creating `TagCloud` Class

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

```
Yii::import('zii.widgets.CPortlet');

class TagCloud extends CPortlet
{
    public $title='Tags';
    public $maxTags=20;

    protected function renderContent()
    {
        $tags=Tag::model()->findTagWeights($this->maxTags);

        foreach($tags as $tag=>$weight)
        {
            $link=CHtml::link(CHtml::encode($tag), array('post/index','tag'=>$tag));
            echo CHtml::tag('span', array(
                'class'=>'tag',
                'style'=>"font-size:{$weight}pt",
            ), $link)."\n";
        }
    }
}
```

Unlike the `UserMenu` portlet, the `TagCloud` portlet does not use a view. Instead, its presentation is done in the `renderContent()` method. This is because the presentation does not contain much HTML tags.

We display each tag as a hyperlink to the post index page with the corresponding tag parameter. The font size of each tag link is adjusted according to their relative weight among other tags. If a tag has higher frequency value than the other, it will have a bigger font size.

5.2.2 Using `TagCloud` Portlet

Usage of the `TagCloud` portlet is very simple. We modify the layout file `/wwwroot/blog/protected/views/layouts/column2.php` as follows,

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

    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>

    <?php $this->widget('TagCloud', array(
        'maxTags'=>Yii::app()->params['tagCloudCount'],
    )); ?>

</div>
```

.....

5.3 Creating Recent Comments Portlet

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

5.3.1 Creating `RecentComments` Class

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

```
Yii::import('zii.widgets.CPortlet');

class RecentComments extends CPortlet
{
    public $title='Recent Comments';
    public $maxComments=10;

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

    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)
    {
        return $this->with('post')->findAll(array(
            'condition'=>'t.status='.self::STATUS_APPROVED,
            'order'=>'t.create_time DESC',
            'limit'=>$limit,
        ));
    }
}
```

5.3.2 Creating `recentComments` View

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

5.3.3 Using `RecentComments` Portlet

We modify the layout file `/wwwroot/blog/protected/views/layouts/column2.php` to embed this last portlet,

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

    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>

    <?php $this->widget('TagCloud', array(
        'maxTags'=>Yii::app()->params['tagCloudCount'],
    )); ?>

    <?php $this->widget('RecentComments', array(
        'maxComments'=>Yii::app()->params['recentCommentCount'],
    )); ?>

</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&title=A+Test+Post
```

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

1. `/index.php/posts/yii`: leads to the page showing a list of posts with tag `yii`;
2. `/index.php/post/2/A+Test+Post`: leads to the page showing the detail of the post with ID 2 whose title is `A Test Post`;
3. `/index.php/post/update?id=1`: leads to the page that allows updating the post with ID 1.

Note that in the second URL format, we include the post title in the URL. This is mainly to make the URL SEO friendly. It is said that search engines may also respect the words found in a URL when it is being indexed.

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

```
return array(
    ......
    'components'=>array(
        ......
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
```

```
                'post/<id:\d+>/<title:.*?>'=>'post/view',
                'posts/<tag:.*?>'=>'post/index',
                '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
            ),
        ),
    ),
);
```

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 second rule says that if a URL `/index.php/posts/yii` is requested, the `urlManager` component should be responsible to dispatch the request to the route `post/index` and generate a `tag` GET parameter with the value `yii`. On the other hand, when creating a URL with the route `post/index` and parameter `tag`, the `urlManager` component will also use this rule to generate the desired URL `/index.php/posts/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 Final Tune-up and Deployment

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

6.3.1 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 `index` to be its default action, when we access the home page of the application, we will see the result generated by the `index` action of the post controller.

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.4 Future Enhancements

6.4.1 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',
    ......
);
```

6.4.2 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 `index` page of `PostController`, we can create a view file `/wwwroot/blog/protected/views/post/zh_cn/index.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.

6.4.3 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/column2.php`, we can enclose the tag cloud portlet with COutputCache:

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

    <?php $this->widget('TagCloud', array(
        'maxTags'=>Yii::app()->params['tagCloudCount'],
    )); ?>

<?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.

6.4.4 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.