The Definitive Guide to Yii 1.1

Qiang Xue and Xiang Wei Zhuo

CONTENTS

Contents   iii

License of Yii

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

CHAPTER 1

Getting Started

1.1   The Definitive Guide to Yii

This tutorial is released under the Terms of Yii Documentation.

All Rights Reserved.

2008-2010 copy; Yii Software LLC.

1.2   New Features

This page summarizes the main new features introduced in each Yii release.

1.2.1   Version 1.1.4

- Added support for automatic action parameter binding

1.2.2   Version 1.1.3

- Added support to configure widget default values in application configuration

1.2.3   Version 1.1.2

- Added a Web-based code generation tool called Gii

1.2.4   Version 1.1.1

- Added CActiveForm which simplifies writing form-related code and supports seamless and consistent validation on both client and server sides.

- Refactored the code generated by the yiic tool. In particular, the skeleton application is now generated with multiple layouts; the operation menu is reorganized for

CRUD pages; added search and filtering feature to the admin page generated by crud command; used CActiveForm to render a form.

- Added support to allow defining global yiic commands

1.2.5   Version 1.1.0

- Added support for writing unit and functional tests

- Added support for using widget skins

- Added an extensible form builder

- Improved the way of declaring safe model attributes. See Securing Attribute Assignments.

- Changed the default eager loading algorithm for relational active record queries so that all tables are joined in one single SQL statement.

- Changed the default table alias to be the name of active record relations.

- Added support for using table prefix.

- Added a whole set of new extensions known as the Zii library.

- The alias name for the primary table in an AR query is fixed to be 't'

1.2.6   Version 1.0.11

- Added support for parsing and creating URLs with parameterized hostnames

    - Parameterizing Hostnames

1.2.7   Version 1.0.10

- Enhanced support for using CPhpMessageSource to manage module messages

    - Message Translation

- Added support for attaching anonymous functions as event handlers

    - Component Event

1.2.8 Version 1.0.8

- Added support for retrieving multiple cached values at one time

  - Data Caching

- Introduced a new default root path alias `ext` which points to the directory containing all third-party extensions.

  - Using Extensions

1.2.9 Version 1.0.7

- Added support for displaying call stack information in trace messages

  - Logging Context Information

- Added `index` option to AR relations so that related objects can be indexed using the values of a specific column

  - Relational Query Options

1.2.10 Version 1.0.6

- Added support for using named scope with `update` and `delete` methods:

  - Named Scopes

- Added support for using named scope in the `with` option of relational rules:

  - Relational Query with Named Scopes

- Added support for profiling SQL executions

  - Profiling SQL Executions

- Added support for logging additional context information

  - Logging Context Information

- Added support for customizing a single URL rule by setting its urlFormat and caseSensitive options:

  - User-friendly URLs

- Added support for using a controller action to display application errors:

  - Handling Errors Using an Action

1.2.11   Version 1.0.5

- Enhanced active record by supporting named scopes. See:

    - Named Scopes
    - Default Named Scope
    - Relational Query with Named Scopes

- Enhanced active record by supporting lazy loading with dynamic query options. See:

    - Dynamic Relational Query Options

- Enhanced CUrlManager to support parameterizing the route part in URL rules. See:

    - Parameterizing Routes in URL Rules

1.3   Upgrading from Version 1.0 to 1.1

1.3.1   Changes Related with Model Scenarios

- Removed CModel::safeAttributes(). Safe attributes are now defined to be those that are being validated by some rules as defined in CModel::rules() for the particular scenario.

- Changed CModel::validate(), CModel::beforeValidate() and CModel::afterValidate(). CModel::setAttributes(), CModel::getSafeAttributeNames() The 'scenario' parameter is removed. You should get and set the model scenario via CModel::scenario.

- Changed CModel::getValidators() and removed CModel::getValidatorsForAttribute(). CModel::getValidators() now only returns validators applicable to the scenario as specified by the model's scenario property.

- Changed CModel::isAttributeRequired() and CModel::getValidatorsForAttribute(). The scenario parameter is removed. The model's scenario property will be used, instead.

- Removed CHtml::scenario. CHtml will use the model's scenario property instead.

1.3.2   Changes Related with Eager Loading for Relational Active Record

- By default, a single JOIN statement will be generated and executed for all relations involved in the eager loading. If the primary table has its `LIMIT` or `OFFSET` query option set, it will be queried alone first, followed by another SQL statement that

brings back all its related objects. Previously in version 1.0.x, the default behavior is that there will be `N+1` SQL statements if an eager loading involves `N` `HAS_MANY` or `MANY_MANY` relations.

1.3.3 Changes Related with Table Alias in Relational Active Record

- The default alias for a relational table is now the same as the corresponding relation name. Previously in version 1.0.x, by default Yii would automatically generate a table alias for each relational table, and we had to use the prefix `??.` to refer to this automatically generated alias.

- The alias name for the primary table in an AR query is fixed to be `t`. Previsouly in version 1.0.x, it was the same as the table name. This will cause existing AR query code to break if they explicity specify column prefixes using the table name. The solution is to replace these prefixes with 't.'.

1.3.4 Changes Related with Tabular Input

- For attribute names, using `Field[$i]` is not valid anymore, they should look like `[$i]Field` in order to support array-typed fields (e.g. `[$i]Field[$index]`).

1.3.5 Other Changes

- The signature of the CActiveRecord constructor is changed. The first parameter (list of attributes) is removed.

1.4 What is Yii

Yii is a high-performance component-based PHP framework for developing large-scale Web applications rapidly. It enables maximum reusability in Web programming and can significantly accelerate your Web application development process. The name Yii (pronounced as `Yee` or `[ji:]`) stands for *easy*, *efficient* and *extensible*.

1.4.1 Requirements

To run a Yii-powered Web application, you need a Web server supporting PHP 5.1.0 or higher.

For developers who want to use Yii, understanding object-oriented programming (OOP) is very helpful, because Yii is a pure OOP framework.

1.4.2   What is Yii Best for?

Yii is a generic Web programming framework that can be used for developing virtually all sorts of Web applications. Because it is light-weighted and equipped with sophisticated caching solutions, it is especially suitable for developing high-traffic applications, such as portals, forums, content management systems (CMS), e-commerce systems, etc.

1.4.3   How is Yii Compared with Other Frameworks?

Like most PHP frameworks, Yii is an MVC framework.

Yii excels over other PHP frameworks in that it is efficient, feature-rich and clearly-documented. Yii is carefully designed from the beginning to fit for serious Web application development. It is neither a byproduct of some project nor a conglomerate of third-party work. It is the result of the authors' rich experience of Web application development and the investigation and reflection of the most popular Web programming frameworks and applications.

1.5   Installation

Installation of Yii mainly involves the following two steps:

1. Download Yii Framework from yiiframework.com.

2. Unpack the Yii release file to a Web-accessible directory.

Tip: Yii does not need to be installed under a Web-accessible directory. A Yii application has one entry script which is usually the only file that needs to be exposed to Web users. Other PHP scripts, including those from Yii, should be protected from Web access since they may be exploited for hacking.

1.5.1   Requirements

After installing Yii, you may want to verify that your server satisfies all the requirements of using Yii. You can do so by accessing the requirement checker script at the following URL in a Web browser:

```
http://hostname/path/to/yii/requirements/index.php
```

The minimum requirement by Yii is that your Web server supports PHP 5.1.0 or above. Yii has been tested with Apache HTTP server on Windows and Linux operating systems. It may also run on other Web servers and platforms, provided PHP 5 is supported.

1.6   Creating First Yii Application

To get an initial experience with Yii, we describe in this section how to create our first Yii application. We will use `yiic` (command line tool) to create a new Yii application and `Gii` (powerful web based code generator) to automate code creation for certain tasks. For convenience, we assume that `YiiRoot` is the directory where Yii is installed, and `WebRoot` is the document root of our Web server.

Run `yiic` on the command line as follows:

```
% YiiRoot/framework/yiic webapp WebRoot/testdrive
```

> **Note:** When running `yiic` on Mac OS, Linux or Unix, you may need to change the permission of the `yiic` file so that it is executable. Alternatively, you may run the tool as follows,
>
> ```
> % cd WebRoot
> % php YiiRoot/framework/yiic.php webapp testdrive
> ```

This will create a skeleton Yii application under the directory `WebRoot/testdrive`. The application has a directory structure that is needed by most Yii applications.

Without writing a single line of code, we can test drive our first Yii application by accessing the following URL in a Web browser:

```
http://hostname/testdrive/index.php
```

As we can see, the application has four pages: the homepage, the about page, the contact page and the login page. The contact page displays a contact form that users can fill in to submit their inquiries to the webmaster, and the login page allows users to be authenticated before accessing privileged contents. See the following screenshots for more details.

The following diagram shows the directory structure of our application. Please see Conventions for detailed explanation about this structure.
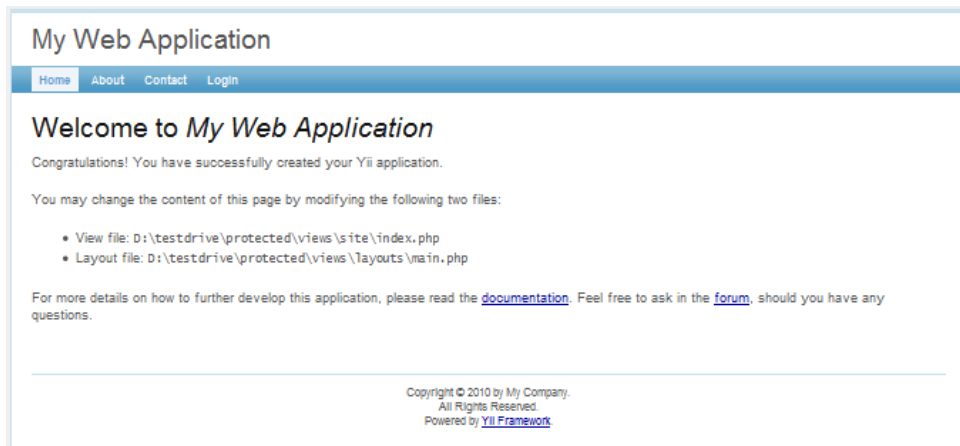
Figure 1.1: Home page

Figure 1.2: Contact page

Figure 1.3: Contact page with input errors

Figure 1.4: Contact page with success

Figure 1.5: Login page

```
testdrive/
   index.php              Web application entry script file
   index-test.php         entry script file for the functional tests
   assets/                containing published resource files
   css/                   containing CSS files
   images/                containing image files
   themes/                containing application themes
   protected/             containing protected application files
      yiic                yiic command line script for Unix/Linux
      yiic.bat            yiic command line script for Windows
      yiic.php            yiic command line PHP script
      commands/           containing customized 'yiic' commands
         shell/           containing customized 'yiic shell' commands
      components/         containing reusable user components
         Controller.php   the base class for all controller classes
         UserIdentity.php the 'UserIdentity' class used for authentication
      config/             containing configuration files
         console.php      the console application configuration
         main.php         the Web application configuration
         test.php         the configuration for the functional tests
      controllers/        containing controller class files
         SiteController.php the default controller class
      data/               containing the sample database
         schema.mysql.sql the DB schema for the sample MySQL database
         schema.sqlite.sql the DB schema for the sample SQLite database
         testdrive.db     the sample SQLite database file
      extensions/         containing third-party extensions
      messages/           containing translated messages
```

```
models/                containing model class files
   LoginForm.php       the form model for 'login' action
   ContactForm.php     the form model for 'contact' action
runtime/               containing temporarily generated files
tests/                 containing test scripts
views/                 containing controller view and layout files
   layouts/            containing layout view files
      main.php         the base layout shared by all pages
      column1.php      the layout for pages using a single column
      column2.php      the layout for pages using two columns
   site/               containing view files for the 'site' controller
      pages/           containing "static" pages
         about.php     the view for the "about" page
      contact.php      the view for 'contact' action
      error.php        the view for 'error' action (displaying external errors)
      index.php        the view for 'index' action
      login.php        the view for 'login' action
```

1.6.1   Connecting to Database

Most Web applications are backed by databases. Our test-drive application is not an exception. To use a database, we need to tell the application how to connect to it. This is done in the application configuration file `WebRoot/testdrive/protected/config/main.php`, highlighted as follows,

```php
return array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'connectionString'=>'sqlite:protected/data/testdrive.db',
        ),
    ),
    ......
);
```

The above code instructs Yii that the application should connect to the SQLite database `WebRoot/testdrive/protected/data/testdrive.db` when needed. Note that the SQLite database is already included in the skeleton application that we just generated. The database contains only a single table named `tbl_user`:

```sql
CREATE TABLE tbl_user (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    username VARCHAR(128) NOT NULL,
    password VARCHAR(128) NOT NULL,
```

```
    email VARCHAR(128) NOT NULL
);
```

If you want to try a MySQL database instead, you may use the included MySQL schema file `WebRoot/testdrive/protected/data/schema.mysql.sql` to create the database.

> **Note:** To use Yii's database feature, we need to enable PHP PDO extension and the driver-specific PDO extension. For the test-drive application, we need to turn on both the `php_pdo` and `php_pdo_sqlite` extensions.

1.6.2 Implementing CRUD Operations

Now is the fun part. We would like to implement the CRUD (create, read, update and delete) operations for the `tbl_user` table we just created. This is also commonly needed in practical applications. Instead of taking the trouble to write the actual code, we will use `Gii` – a powerful Web-based code generator.

> **Info:** Gii has been available since version 1.1.2. Before that, we can use the aforementioned `yiic` tool to accomplish the same goal. For more details, please refer to Implementing CRUD Operations with yiic shell.

Configuring Gii

In order to use Gii, we first need to edit the file `WebRoot/testdrive/protected/config/main.php`, which is known as the application configuration file:
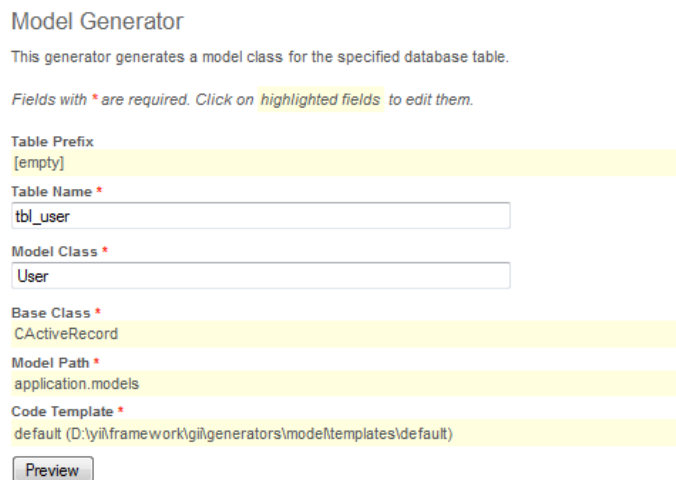
```
return array(
    ......
    'import'=>array(
        'application.models.*',
        'application.components.*',
    ),

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

Then, visit the URL `http://hostname/testdrive/index.php?r=gii`. We will be prompted for a password, which should be the one that we just entered in the above application configuration.

Generating User Model

After login, click on the link `Model Generator`. This will bring us to the following model generation page,

Figure 1.6: Model Generator

In the `Table Name` field, enter `tbl_user`. In the `Model Class` field, enter `User`. Then press the `Preview` button. This will show us the new code file to be generated. Now press the `Generate` button. A new file named `User.php` will be generated under `protected/models`. As we will describe later in this guide, this `User` model class allows us to talk to the underlying database `tbl_user` table in an object-oriented fashion.

Generating CRUD Code

After creating the model class file, we will generate the code that implements the CRUD operations about the user data. We choose the `Crud Generator` in Gii, shown as follows,

In the `Model Class` field, enter `User`. In the `Controller ID` field, enter `user` (in lower case). Now press the `Preview` button followed by the `Generate` button. We are done with the CRUD code generation.

Figure 1.7: CRUD Generator

Accessing CRUD Pages

Let's enjoy our work by browsing the following URL:

`http://hostname/testdrive/index.php?r=user`

This will display a list of user entries in the `tbl_user` table.

Click the `Create User` button on the page. We will be brought to the login page if we have not logged in before. After logged in, we see an input form that allows us to add a new user entry. Complete the form and click on the `Create` button. If there is any input error, a nice error prompt will show up which prevents us from saving the input. Back to the user list page, we should see the newly added user appearing in the list.

Repeat the above steps to add more users. Notice that user list page will automatically paginate the user entries if there are too many to be displayed in one page.

If we login as an administrator using `admin/admin`, we can view the user admin page with the following URL:

`http://hostname/testdrive/index.php?r=user/admin`

This will show us the user entries in a nice tabular format. We can click on the table header cells to sort the corresponding columns. We can click on the buttons on each row of data to view, update or delete the corresponding row of data. We can browse different pages. We can also filter and search to look for the data we are interested in.

Figure 1.8: User admin page

Figure 1.9: Create new user page

All these nice features come without requiring us to write a single line of code!

CHAPTER 2

Fundamentals

2.1 Model-View-Controller (MVC)

Yii implements the model-view-controller (MVC) design pattern which is widely adopted in Web programming. MVC aims to separate business logic from user interface considerations so that developers can more easily change each part without affecting the other. In MVC, the model represents the information (the data) and the business rules; the view contains elements of the user interface such as text, form inputs; and the controller manages the communication between the model and the view.

Besides MVC, Yii also introduces a front-controller, called application, which represents the execution context of request processing. Application resolves the user request and dispatches it to an appropriate controller for further handling.

The following diagram shows the static structure of a Yii application:
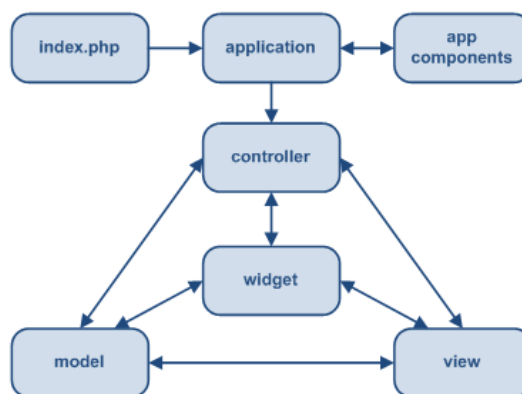
Figure 2.1: Static structure of Yii application

2.1.1   A Typical Workflow

The following diagram shows a typical workflow of a Yii application when it is handling a user request:
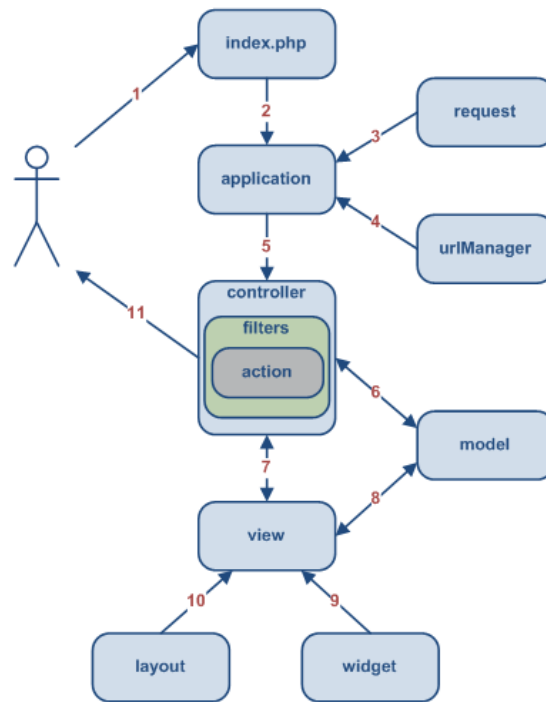
Figure 2.2: A typical workflow of Yii application

1. A user makes a request with the URL `http://www.example.com/index.php?r=post/show&id=1` and the Web server handles the request by executing the bootstrap script `index.php`.

2. The bootstrap script creates an application instance and runs it.

3. The application obtains the detailed user request information from an application component named `request`.

4. The application determines the requested controller and action with the help of an application component named `urlManager`. For this example, the controller is `post` which refers to the `PostController` class; and the action is `show` whose actual meaning is determined by the controller.

5. The application creates an instance of the requested controller to further handle the user request. The controller determines that the action `show` refers to a method

named `actionShow` in the controller class. It then creates and executes filters (e.g. access control, benchmarking) associated with this action. The action is executed if it is allowed by the filters.

6. The action reads a `Post` model whose ID is 1 from the database.

7. The action renders a view named `show` with the `Post` model.

8. The view reads and displays the attributes of the `Post` model.

9. The view executes some widgets.

10. The view rendering result is embedded in a layout.

11. The action completes the view rendering and displays the result to the user.

2.2   Entry Script

Entry script is the bootstrap PHP script that handles user requests initially. It is the only PHP script that end users can directly request to execute.

In most cases, entry script of a Yii application contains the code that is as simple as follows,

```
// remove the following line when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);
// include Yii bootstrap file
require_once('path/to/yii/framework/yii.php');
// create application instance and run
$configFile='path/to/config/file.php';
Yii::createWebApplication($configFile)->run();
```

The script first includes the Yii framework bootstrap file `yii.php`. It then creates a Web application instance with the specified configuration and runs it.

2.2.1   Debug Mode

A Yii application can run in either debug or production mode according to the constant value `YII_DEBUG`. By default, this constant value is defined as `false`, meaning production mode. To run in debug mode, define this constant as `true` before including the `yii.php` file. Running application in debug mode is less efficient because it keeps many internal logs. On the other hand, debug mode is also more helpful during development stage because it provides richer debugging information when error occurs.

2.3   Application

Application represents the execution context of request processing. Its main task is to resolve the user request and dispatch it to an appropriate controller for further processing. It also serves as the central place for keeping application-level configurations. For this reason, application is also called `front-controller`.

Application is created as a singleton by the entry script. The application singleton can be accessed at any place via Yii::app().

2.3.1   Application Configuration

By default, application is an instance of CWebApplication. To customize it, we normally provide a configuration file (or array) to initialize its property values when the application instance is being created. An alternative way of customizing application is to extend CWebApplication.

The configuration is an array of key-value pairs. Each key represents the name of a property of the application instance, and each value the corresponding property's initial value. For example, the following configuration configures the name and defaultController properties of the application.

```php
array(
    'name'=>'Yii Framework',
    'defaultController'=>'site',
)
```

We usually store the configuration in a separate PHP script (e.g. `protected/config/main.php`). Inside the script, we return the configuration array as follows,

```php
return array(...);
```

To apply the configuration, we pass the configuration file name as a parameter to the application's constructor, or to Yii::createWebApplication() like the following, which is usually done in the entry script:

```php
$app=Yii::createWebApplication($configFile);
```

> **Tip:** If the application configuration is very complex, we can split it into several files, each returning a portion of the configuration array. Then, in the main configuration file, we call PHP `include()` to include the rest configuration files and merge them into a complete configuration array.

2.3.2 Application Base Directory

Application base directory refers to the root directory that contains all security-sensitive PHP scripts and data. By default, it is a subdirectory named `protected` that is located under the directory containing the entry script. It can be customized via setting the basePath property in the application configuration.

Contents under the application base directory should be protected from being accessed by Web users. With Apache HTTP server, this can be done easily by placing a `.htaccess` file under the base directory. The content of the `.htaccess` file is as follows,

```
deny from all
```

2.3.3 Application Component

Functionalities of application can be easily customized and enriched with its flexible component architecture. Application manages a set of application components, each implementing specific features. For example, application resolves a user request with the help of CUrlManager and CHttpRequest components.

By configuring the components property of application, we can customize the class and property values of any application component used in an application. For example, we can configure CMemCache component so that it can use multiple memcache servers for caching,

```php
array(
    ......
    'components'=>array(
        ......
        'cache'=>array(
            'class'=>'CMemCache',
            'servers'=>array(
                array('host'=>'server1', 'port'=>11211, 'weight'=>60),
                array('host'=>'server2', 'port'=>11211, 'weight'=>40),
            ),
        ),
    ),
)
```

In the above, we add the `cache` element to the `components` array. The `cache` element states that the class of the component is `CMemCache` and its `servers` property should be initialized as such.

To access an application component, use `Yii::app()->ComponentID`, where `ComponentID` refers to the ID of the component (e.g. `Yii::app()->cache`).

An application component may be disabled by setting `enabled` to be false in its configuration. Null is returned when we access a disabled component.

> **Tip:** By default, application components are created on demand. This means an application component may not be created at all if it is not accessed during a user request. As a result, the overall performance may not be degraded even if an application is configured with many components. Some application components (e.g. CLogRouter) may need to be created no matter they are accessed or not. To do so, list their IDs in the preload property of the application.

2.3.4 Core Application Components

Yii predefines a set of core application components to provide features common among Web applications. For example, the request component is used to resolve user requests and provide information such as URL, cookies. By configuring the properties of these core components, we can change the default behaviors of Yii in nearly every aspect.

Below we list the core components that are pre-declared by CWebApplication.

- assetManager: CAssetManager - manages the publishing of private asset files.

- authManager: CAuthManager - manages role-based access control (RBAC).

- cache: CCache - provides data caching functionality. Note, you must specify the actual class (e.g. CMemCache, CDbCache). Otherwise, null will be returned when you access this component.

- clientScript: CClientScript - manages client scripts (javascripts and CSS).

- coreMessages: CPhpMessageSource - provides translated core messages used by Yii framework.

- db: CDbConnection - provides the database connection. Note, you must configure its connectionString property in order to use this component.

- errorHandler: CErrorHandler - handles uncaught PHP errors and exceptions.

- format: CFormatter - formats data values for display purpose. This has been available since version 1.1.0.

- messages: CPhpMessageSource - provides translated messaged used by Yii application.

- request: CHttpRequest - provides information related with user requests.

- securityManager: CSecurityManager - provides security-related services, such as hashing, encryption.

- session: CHttpSession - provides session-related functionalities.

- statePersister: CStatePersister - provides global state persistence method.

- urlManager: CUrlManager - provides URL parsing and creation functionality.

- user: CWebUser - represents the identity information of the current user.

- themeManager: CThemeManager - manages themes.

2.3.5 Application Lifecycles

When handling a user request, an application will undergo the following lifecycles:

1. Pre-initializes the application with CApplication::preinit();

2. Set up class autoloader and error handling;

3. Register core application components;

4. Load application configuration;

5. Initialize the application with CApplication::init()

   - Register application behaviors;
   - Load static application components;

6. Raise onBeginRequest event;

7. Process the user request:

   - Resolve the user request;
   - Create controller;
   - Run controller;

8. Raise onEndRequest event;

2.4   Controller

A `controller` is an instance of CController or its child class. It is created by application
when the user requests for it. When a controller runs, it performs the requested action
which usually brings in the needed models and renders an appropriate view. An `action`,
at its simplest form, is just a controller class method whose name starts with `action`.

A controller has a default action. When the user request does not specify which action to
execute, the default action will be executed. By default, the default action is named as
`index`. It can be changed by setting CController::defaultAction.

Below is the minimal code needed by a controller class. Since this controller does not
define any action, requesting for it would throw an exception.

```
class SiteController extends CController
{
}
```

2.4.1   Route

Controllers and actions are identified by IDs. Controller ID is in the format of `path/`
`to/xyz` which corresponds to the controller class file `protected/controllers/path/to/`
`XyzController.php`, where the token `xyz` should be replaced by actual names (e.g. `post` cor-
responds to `protected/controllers/PostController.php`). Action ID is the action method
name without the `action` prefix. For example, if a controller class contains a method named
`actionEdit`, the ID of the corresponding action would be `edit`.

> **Note:** Before version 1.0.3, the controller ID format was `path.to.xyz` instead of
> `path/to/xyz`.

Users request for a particular controller and action in terms of route. A route is formed
by concatenating a controller ID and an action ID separated by a slash. For example, the
route `post/edit` refers to `PostController` and its `edit` action. And by default, the URL
`http://hostname/index.php?r=post/edit` would request for this controller and action.

> **Note:** By default, routes are case-sensitive. Since version 1.0.1, it is possible to
> make routes case-insensitive by setting CUrlManager::caseSensitive to be false in
> the application configuration. When in case-insensitive mode, make sure you follow
> the convention that directories containing controller class files are in lower case, and
> both controller map and action map are using keys in lower case.

Since version 1.0.3, an application can contain modules. The route for a controller action inside a module is in the format of `moduleID/controllerID/actionID`. For more details, see the section about modules.

2.4.2 Controller Instantiation

A controller instance is created when CWebApplication handles an incoming request. Given the ID of the controller, the application will use the following rules to determine what the controller class is and where the class file is located.

- If CWebApplication::catchAllRequest is specified, a controller will be created based on this property, and the user-specified controller ID will be ignored. This is mainly used to put the application under maintenance mode and display a static notice page.

- If the ID is found in CWebApplication::controllerMap, the corresponding controller configuration will be used to create the controller instance.

- If the ID is in the format of `'path/to/xyz'`, the controller class name is assumed to be `XyzController` and the corresponding class file is `protected/controllers/path/to/XyzController.php`. For example, a controller ID `admin/user` would be resolved as the controller class `UserController` and the class file `protected/controllers/admin/UserController.php`. If the class file does not exist, a 404 CHttpException will be raised.

In case when modules are used (available since version 1.0.3), the above process is slighly different. In particular, the application will check if the ID refers to a controller inside a module, and if so, the module instance will be created first followed by the controller instance.

2.4.3 Action

As aforementioned, an action can be defined as a method whose name starts with the word `action`. A more advanced way is to define an action class and ask the controller to instantiate it when requested. This allows actions to be reused and thus introduces more reusability.

To define a new action class, do the following:

```
class UpdateAction extends CAction
{
```

```php
    public function run()
    {
        // place the action logic here
    }
}
```

In order for the controller to be aware of this action, we override the actions() method of
our controller class:

```php
class PostController extends CController
{
    public function actions()
    {
        return array(
            'edit'=>'application.controllers.post.UpdateAction',
        );
    }
}
```

In the above, we use the path alias `application.controllers.post.UpdateAction` to specify
that the action class file is `protected/controllers/post/UpdateAction.php`.

Writing class-based actions, we can organize an application in a modular fashion. For
example, the following directory structure may be used to organize the code for controllers:

```
protected/
    controllers/
        PostController.php
        UserController.php
        post/
            CreateAction.php
            ReadAction.php
            UpdateAction.php
        user/
            CreateAction.php
            ListAction.php
            ProfileAction.php
            UpdateAction.php
```

Action Parameter Binding

Since version 1.1.4, Yii has added support for automatic action parameter binding. That is,
a controller action method can define named parameters whose value will be automatically
populated from `$_GET` by Yii.

In order to elaborate this feature, let's assume we need to write a `create` action for `PostController`. The action requires two parameters:

- `category`: an integer indicating the category ID under which the new post will be created;

- `language`: a string indicating the language code that the new post will be in.

We may end up with the following boring code for the purpose of retrieving the needed parameter values from `$_GET`:

```php
class PostController extends CController
{
    public function actionCreate()
    {
        if(isset($_GET['category']))
            $category=(int)$_GET['category'];
        else
            throw new CHttpException(404,'invalid request');

        if(isset($_GET['language']))
            $language=$_GET['language'];
        else
            $language='en';

        // ... fun code starts here ...
    }
}
```

Now using the action parameter feature, we can achieve our task more pleasantly:

```php
class PostController extends CController
{
    public function actionCreate($category, $language='en')
    {
        $category=(int)$category;

        // ... fun code starts here ...
    }
}
```

Notice that we add two parameters to the action method `actionCreate`. The name of these parameters must be exactly the same as the ones we expect from `$_GET`. The `$language`

parameter takes a default value `en` in case when the user does not provide such a parameter in his request. Because `$category` does not have a default value, if the user does not provide `category` parameter in `$_GET`, a CHttpException (error code 400) will be thrown automatically.

2.4.4 Filter

Filter is a piece of code that is configured to be executed before and/or after a controller action executes. For example, an access control filter may be executed to ensure that the user is authenticated before executing the requested action; a performance filter may be used to measure the time spent in the action execution.

An action can have multiple filters. The filters are executed in the order that they appear in the filter list. A filter can prevent the execution of the action and the rest of the unexecuted filters.

A filter can be defined as a controller class method. The method name must begin with `filter`. For example, the existence of the `filterAccessControl` method defines a filter named `accessControl`. The filter method must be of the signature:

```
public function filterAccessControl($filterChain)
{
    // call $filterChain->run() to continue filtering and action execution
}
```

where `$filterChain` is an instance of CFilterChain which represents the filter list associated with the requested action. Inside the filter method, we can call `$filterChain->run()` to continue filtering and action execution.

A filter can also be an instance of CFilter or its child class. The following code defines a new filter class:

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

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

```
        }
}
```

To apply filters to actions, we need to override the `CController::filters()` method. The method should return an array of filter configurations. For example,

```
class PostController extends CController
{
    ......
    public function filters()
    {
        return array(
            'postOnly + edit, create',
            array(
                'application.filters.PerformanceFilter - edit, create',
                'unit'=>'second',
            ),
        );
    }
}
```

The above code specifies two filters: `postOnly` and `PerformanceFilter`. The `postOnly` filter is method-based (the corresponding filter method is defined in CController already); while the `PerformanceFilter` filter is object-based. The path alias `application.filters.PerformanceFilter` specifies that the filter class file is `protected/filters/PerformanceFilter`. We use an array to configure `PerformanceFilter` so that it may be used to initialize the property values of the filter object. Here the `unit` property of `PerformanceFilter` will be initialized as `'second'`.

Using the plus and the minus operators, we can specify which actions the filter should and should not be applied to. In the above, the `postOnly` should be applied to the `edit` and `create` actions, while `PerformanceFilter` should be applied to all actions EXCEPT `edit` and `create`. If neither plus nor minus appears in the filter configuration, the filter will be applied to all actions.

2.5 Model

A model is an instance of CModel or its child class. Models are used to keep data and their relevant business rules.

A model represents a single data object. It could be a row in a database table or a form of user inputs. Each field of the data object is represented as an attribute of the model. The attribute has a label and can be validated against a set of rules.

Yii implements two kinds of models: form model and active record. They both extend from the same base class CModel.

A form model is an instance of CFormModel. Form model is used to keep data collected from user inputs. Such data are often collected, used and then discarded. For example, on a login page, we can use a form model to represent the username and password information that are provided by an end user. For more details, please refer to Working with Form

Active Record (AR) is a design pattern used to abstract database access in an object-oriented fashion. Each AR object is an instance of CActiveRecord or its child class, representing a single row in a database table. The fields in the row are represented as properties of the AR object. Details about AR can be found in Active Record.

2.6   View

A view is a PHP script consisting of mainly elements of user interface. It can contain PHP statements, but it is recommended that these statements should not alter data models and should remain relatively simple. For the spirit of separation of logic and presentation, large chunk of logic should be placed in controller or model instead of view.

A view has a name which is used to identify the view script file when rendering. The name of a view is the same as the name of its view script file. For example, view `edit` refers to a view script file named as `edit.php`. To render a view, call CController::render() with the name of the view. The method will look for the corresponding view file under the directory `protected/views/ControllerID`.

Inside the view script, we can access the controller instance using `$this`. We can thus `pull` in any property of the controller by evaluating `$this->propertyName` in the view.

We can also use the following `push` approach to pass data to the view:

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

In the above, the render() method will extract the second array parameter into variables. As a result, in the view script we can access local variables `$var1` and `$var2`.

2.6.1 Layout

Layout is a special view that is used to decorate views. It usually contains portions of user interface that are common among several views. For example, a layout may contain header and footer portions and embed the content view in between,

```
......header here......
<?php echo $content; ?>
......footer here......
```

where `$content` stores the rendering result of the content view.

Layout is implicitly applied when calling render(). By default, the view script `protected/views/layouts/main.php` is used as the layout. This can be customized by changing either CWebApplication::layout or CController::layout. To render a view without applying any layout, call renderPartial() instead.

2.6.2 Widget

A widget is an instance of CWidget or its child class. It is a component mainly for presentational purpose. Widgets are usually embedded in a view script to generate some complex yet self-contained user interface. For example, a calendar widget can be used to render a complex calendar user interface. Widgets enable better reusability in user interface.

To use a widget, do as follows in a view script:

```
<?php $this->beginWidget('path.to.WidgetClass'); ?>
...body content that may be captured by the widget...
<?php $this->endWidget(); ?>
```

or

```
<?php $this->widget('path.to.WidgetClass'); ?>
```

The latter is used when the widget does not need any body content.

Widgets can be configured to customize its behaviors. This is done by setting their initial property values when calling CBaseController::beginWidget or CBaseController::widget. For example, when using CMaskedTextField widget, we would like to specify the mask

being used. We can do so by passing an array of those property initial values as follows, where the array keys are property names and array values the initial values of the corresponding widget properties:

```php
<?php
$this->widget('CMaskedTextField',array(
    'mask'=>'99/99/9999'
));
?>
```

To define a new widget, extend CWidget and override its init() and run() methods:

```php
class MyWidget extends CWidget
{
    public function init()
    {
        // this method is called by CController::beginWidget()
    }

    public function run()
    {
        // this method is called by CController::endWidget()
    }
}
```

Like a controller, a widget can also have its own view. By default, widget view files are located under the `views` subdirectory of the directory containing the widget class file. These views can be rendered by calling CWidget::render(), similar to that in controller. The only difference is that no layout will be applied to a widget view. Also, `$this` in the view refers to the widget instance instead of the controller instance.

2.6.3   System View

System views refer to the views used by Yii to display error and logging information. For example, when a user requests for a non-existing controller or action, Yii will throw an exception explaining the error. Yii displays the exception using a specific system view.

The naming of system views follows some rules. Names like `errorXXX` refer to views for displaying CHttpException with error code `XXX`. For example, if CHttpException is raised with error code 404, the `error404` view will be displayed.

Yii provides a set of default system views located under `framework/views`. They can be customized by creating the same-named view files under `protected/views/system`.

2.7   Component

Yii applications are built upon components which are objects written to a specification.
A component is an instance of CComponent or its derived class. Using a component
mainly involves accessing its properties and raising/handling its events. The base class
CComponent specifies how to define properties and events.

2.7.1   Component Property

A component property is like an object's public member variable. We can read its value
or assign a value to it. For example,

```
$width=$component->textWidth;      // get the textWidth property
$component->enableCaching=true;    // set the enableCaching property
```

To define a component property, we can simply declare a public member variable in the
component class. A more flexible way, however, is by defining getter and setter methods
like the following:

```
public function getTextWidth()
{
    return $this->_textWidth;
}

public function setTextWidth($value)
{
    $this->_textWidth=$value;
}
```

The above code defines a writable property named `textWidth` (the name is case-insensitive).
When reading the property, `getTextWidth()` is invoked and its returned value becomes the
property value; Similarly, when writing the property, `setTextWidth()` is invoked. If the
setter method is not defined, the property would be read-only and writing it would throw
an exception. Using getter and setter methods to define a property has the benefit that
additional logic (e.g. performing validation, raising events) can be executed when reading
and writing the property.

> **Note:** There is a slight difference between a property defined via getter/setter
> methods and a class member variable. The name of the former is case-insensitive
> while the latter is case-sensitive.

2.7.2   Component Event

Component events are special properties that take methods (called `event handlers`) as
their values. Attaching (assigning) a method to an event will cause the method to be
invoked automatically at the places where the event is raised. Therefore, the behavior of
a component can be modified in a way that may not be foreseen during the development
of the component.

A component event is defined by defining a method whose name starts with `on`. Like
property names defined via getter/setter methods, event names are case-insensitive. The
following code defines an `onClicked` event:

```php
public function onClicked($event)
{
    $this->raiseEvent('onClicked', $event);
}
```

where `$event` is an instance of CEvent or its child class representing the event parameter.

We can attach a method to this event as follows:

```php
$component->onClicked=$callback;
```

where `$callback` refers to a valid PHP callback. It can be a global function or a class
method. If the latter, the callback must be given as an array: `array($object,'methodName')`.

The signature of an event handler must be as follows:

```php
function methodName($event)
{
    ......
}
```

where `$event` is the parameter describing the event (it originates from the `raiseEvent()`
call). The `$event` parameter is an instance of CEvent or its derived class. At the minimum,
it contains the information about who raises the event.

Starting from version 1.0.10, an event handler can also be an anonymous function which
is supported by PHP 5.3 or above. For example,

```php
$component->onClicked=function($event) {
    ......
}
```

If we call `onClicked()` now, the `onClicked` event will be raised (inside `onClicked()`), and the attached event handler will be invoked automatically.

An event can be attached with multiple handlers. When the event is raised, the handlers will be invoked in the order that they are attached to the event. If a handler decides to prevent the rest handlers from being invoked, it can set $event-¿handled to be true.

2.7.3 Component Behavior

Starting from version 1.0.2, a component has added support for mixin and can be attached with one or several behaviors. A *behavior* is an object whose methods can be 'inherited' by its attached component through the means of collecting functionality instead of specialization (i.e., normal class inheritance). A component can be attached with several behaviors and thus achieve 'multiple inheritance'.

Behavior classes must implement the IBehavior interface. Most behaviors can extend from the CBehavior base class. If a behavior needs to be attached to a model, it may also extend from CModelBehavior or CActiveRecordBehavior which implements additional features specifc for models.

To use a behavior, it must be attached to a component first by calling the behavior's attach() method. Then we can call a behavior method via the component:

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

An attached behavior can be accessed like a normal property of the component. For example, if a behavior named `tree` is attached to a component, we can obtain the reference to this behavior object using:

```php
$behavior=$component->tree;
// equivalent to the following:
// $behavior=$component->asa('tree');
```

A behavior can be temporarily disabled so that its methods are not available via the component. For example,

```php
$component->disableBehavior($name);
// the following statement will throw an exception
```

```
$component->test();
$component->enableBehavior($name);
// it works now
$component->test();
```

It is possible that two behaviors attached to the same component have methods of the same name. In this case, the method of the first attached behavior will take precedence.

When used together with events, behaviors are even more powerful. A behavior, when being attached to a component, can attach some of its methods to some events of the component. By doing so, the behavior gets a chance to observe or change the normal execution flow of the component.

Starting from version 1.1.0, a behavior's properties can also be accessed via the component it is attached to. The properties include both the public member variables and the properties defined via getters and/or setters of the behavior. For example, if a behavior has a property named xyz and the behavior is attached to a component $a. Then we can use the expression $a->xyz to access the behavior's property.

2.8 Module

> **Note:** Support for module has been available since version 1.0.3.

A module is a self-contained software unit that consists of models, views, controllers and other supporting components. In many aspects, a module resembles to an application. The main difference is that a module cannot be deployed alone and it must reside inside of an application. Users can access the controllers in a module like they do with normal application controllers.

Modules are useful in several scenarios. For a large-scale application, we may divide it into several modules, each being developed and maintained separately. Some commonly used features, such as user management, comment management, may be developed in terms of modules so that they can be reused easily in future projects.

2.8.1 Creating Module

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

```
forum/
```

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

A module must have a module class that extends from CWebModule. The class name is determined using the expression `ucfirst($id).'Module'`, where `$id` refers to the module ID (or the module directory name). The module class serves as the central place for storing information shared among the module code. For example, we can use CWebModule::params to store module parameters, and use CWebModule::components to share application components at the module level.

> **Tip:** We can use the module generator in Gii to create the basic skeleton of a new module.

2.8.2 Using Module

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

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

A module can also be configured with initial property values. The usage is very similar to configuring application components. For example, the `forum` module may have a property named `postPerPage` in its module class which can be configured in the application configuration as follows:

```
return array(
```

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

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

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

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

> **Tip:** If a controller is in a sub-directory of `controllers`, we can still use the above route format. For example, assuming `PostController` is under `forum/controllers/admin`, we can refer to the `create` action using `forum/admin/post/create`.

2.8.3   Nested Module

Modules can be nested in unlimited levels. That is, a module can contain another module which can contain yet another module. We call the former *parent module* while the latter *child module*. Child modules must be declared in the modules property of their parent module, like we declare modules in the application configuration shown as above.

To access a controller action in a child module, we should use the route `parentModuleID/childModuleID/controllerID/actionID`.

2.9 Path Alias and Namespace

Yii uses path aliases extensively. A path alias is associated with a directory or file path. It is specified in dot syntax, similar to that of widely adopted namespace format:

```
RootAlias.path.to.target
```

where `RootAlias` is the alias of some existing directory. By calling YiiBase::setPathOfAlias(), we can define new path aliases.For convenience, Yii predefines the following root aliases:

- `system`: refers to the Yii framework directory;

- `zii`: refers to the Zii library directory;

- `application`: refers to the application's base directory;

- `webroot`: refers to the directory containing the entry script file. This alias has been available since version 1.0.3.

- `ext`: refers to the directory containing all third-party extensions. This alias has been available since version 1.0.8.

Additionally, if the application uses modules, a root alias is also predefined for each module ID and refers to the base path of the corresponding module. This feature has been available since version 1.0.3.

By using YiiBase::getPathOfAlias(), an alias can be translated to its corresponding path. For example, `system.web.CController` would be translated as `yii/framework/web/CController`.

Using aliases, it is very convenient to import the definition of a class. For example, if we want to include the definition of the CController class, we can call the following:

```
Yii::import('system.web.CController');
```

The import method differs from `include` and `require` in that it is more efficient. The class definition being imported is actually not included until it is referenced for the first time. Importing the same namespace multiple times is also much faster than `include_once` and `require_once`.

> **Tip:** When referring to a class defined by the Yii framework, we do not need to import or include it. All core Yii classes are pre-imported.

We can also use the following syntax to import a whole directory so that the class files under the directory can be automatically included when needed.

```
Yii::import('system.web.*');
```

Besides import, aliases are also used in many other places to refer to classes. For example, an alias can be passed to Yii::createComponent() to create an instance of the corresponding class, even if the class file was not included previously.

Do not confuse path alias with namespace. A namespace refers to a logical grouping of some class names so that they can be differentiated from other class names even if their names are the same, while path alias is used to refer to a class file or directory. Path alias does not conflict with namespace.

> **Tip:** Because PHP prior to 5.3.0 does not support namespace intrinsically, you cannot create instances of two classes who have the same name but with different definitions. For this reason, all Yii framework classes are prefixed with a letter 'C' (meaning 'class') so that they can be differentiated from user-defined classes. It is recommended that the prefix 'C' be reserved for Yii framework use only, and user-defined classes be prefixed with other letters.

2.10 Conventions

Yii favors conventions over configurations. Follow the conventions and one can create sophisticated Yii applications without writing and managing complex configurations. Of course, Yii can still be customized in nearly every aspect with configurations when needed.

Below we describe conventions that are recommended for Yii programming. For convenience, we assume that WebRoot is the directory that a Yii application is installed at.

2.10.1 URL

By default, Yii recognizes URLs with the following format:

```
http://hostname/index.php?r=ControllerID/ActionID
```

The r GET variable refers to the route that can be resolved by Yii into controller and action. If ActionID is omitted, the controller will take the default action (defined via CController::defaultAction); and if ControllerID is also omitted (or the r variable is absent), the application will use the default controller (defined via CWebApplication::defaultController).

With the help of CUrlManager, it is possible to create and recognize more SEO-friendly URLs, such as `http://hostname/ControllerID/ActionID.html`. This feature is covered in detail in URL Management.

2.10.2 Code

Yii recommends naming variables, functions and class types in camel case which capitalizes the first letter of each word in the name and joins them without spaces. Variable and function names should have their first word all in lower-case, in order to differentiate from class names (e.g. `$basePath`, `runController()`, `LinkPager`). For private class member variables, it is recommended to prefix their names with an underscore character (e.g. `$_actionList`).

Because namespace is not supported prior to PHP 5.3.0, it is recommended that classes be named in some unique way to avoid name conflict with third-party classes. For this reason, all Yii framework classes are prefixed with letter "C".

A special rule for controller class names is that they must be appended with the word `Controller`. The controller ID is then defined as the class name with first letter in lower case and the word `Controller` truncated. For example, the `PageController` class will have the ID `page`. This rule makes the application more secure. It also makes the URLs related with controllers a bit cleaner (e.g. `/index.php?r=page/index` instead of `/index.php?r=PageController/index`).

2.10.3 Configuration

A configuration is an array of key-value pairs. Each key represents the name of a property of the object to be configured, and each value the corresponding property's initial value. For example, `array('name'=>'My application', 'basePath'=>'./protected')` initializes the `name` and `basePath` properties to their corresponding array values.

Any writable properties of an object can be configured. If not configured, the properties will take their default values. When configuring a property, it is worthwhile to read the corresponding documentation so that the initial value can be given properly.

2.10.4 File

Conventions for naming and using files depend on their types.

Class files should be named after the public class they contain. For example, the CController class is in the `CController.php` file. A public class is a class that may be used by any other classes. Each class file should contain at most one public class. Private classes

(classes that are only used by a single public class) may reside in the same file with the public class.

View files should be named after the view name. For example, the `index` view is in the `index.php` file. A view file is a PHP script file that contains HTML and PHP code mainly for presentational purpose.

Configuration files can be named arbitrarily. A configuration file is a PHP script whose sole purpose is to return an associative array representing the configuration.

2.10.5   Directory

Yii assumes a default set of directories used for various purposes. Each of them can be customized if needed.

- `WebRoot/protected`: this is the application base directory holding all security-sensitive PHP scripts and data files. Yii has a default alias named `application` associated with this path. This directory and everything under should be protected from being accessed by Web users. It can be customized via CWebApplication::basePath.

- `WebRoot/protected/runtime`: this directory holds private temporary files generated during runtime of the application. This directory must be writable by Web server process. It can be customized via CApplication::runtimePath.

- `WebRoot/protected/extensions`: this directory holds all third-party extensions. It can be customized via CApplication::extensionPath.

- `WebRoot/protected/modules`: this directory holds all application modules, each represented as a subdirectory.

- `WebRoot/protected/controllers`: this directory holds all controller class files. It can be customized via CWebApplication::controllerPath.

- `WebRoot/protected/views`: this directory holds all view files, including controller views, layout views and system views. It can be customized via CWebApplication::viewPath.

- `WebRoot/protected/views/ControllerID`: this directory holds view files for a single controller class. Here `ControllerID` stands for the ID of the controller. It can be customized via CController::viewPath.

- `WebRoot/protected/views/layouts`: this directory holds all layout view files. It can be customized via CWebApplication::layoutPath.

- `WebRoot/protected/views/system`: this directory holds all system view files. System views are templates used in displaying exceptions and errors. It can be customized via CWebApplication::systemViewPath.

- `WebRoot/assets`: this directory holds published asset files. An asset file is a private file that may be published to become accessible to Web users. This directory must be writable by Web server process. It can be customized via CAssetManager::basePath.

- `WebRoot/themes`: this directory holds various themes that can be applied to the application. Each subdirectory represents a single theme whose name is the subdirectory name. It can be customized via CThemeManager::basePath.

2.10.6 Database

Most Web applications are backed by some database. For best practice, we propose the following naming conventions for database tables and columns. Note that they are not required by Yii.

- Both database tables and columns are named in lower case.

- Words in a name should be separated using underscores (e.g. `product_order`).

- For table names, you may use either singular or plural names, but not both. For simplicity, we recommend using singular names.

- Table names may be prefixed with a common token such as `tbl_`. This is especially useful when the tables of an application coexist in the same database with the tables of another application. The two sets of tables can be readily separate by using different table name prefixes.

2.11 Development Workflow

Having described the fundamental concepts of Yii, we show the common workflow for developing a web application using Yii. The workflow assumes that we have done the requirement analysis as well as the necessary design analysis for the application.

1. Create the skeleton directory structure. The `yiic` tool described in Creating First Yii Application can be used to speed up this step.

2. Configure the application. This is done by modifying the application configuration file. This step may also require writing some application components (e.g. the user component).

3. Create a model class for each type of data to be managed. The `Gii` tool described in

   and in can be used to automatically generate the active record class for each interested database table.

4. Create a controller class for each type of user requests. How to classify user requests depends on the actual requirement. In general, if a model class needs to be accessed by users, it should have a corresponding controller class. The `Gii` tool can automate this step, too.

5. Implement actions and their corresponding views. This is where the real work needs to be done.

6. Configure necessary action filters in controller classes.

7. Create themes if the theming feature is required.

8. Create translated messages if internationalization is required.

9. Spot data and views that can be cached and apply appropriate caching techniques.

10. Final tune up and deployment.

For each of the above steps, test cases may need to be created and performed.

CHAPTER 3

Working with Forms

3.1 Working with Form

Collecting user data via HTML forms is one of the major tasks in Web application development. Besides designing forms, developers need to populate the form with existing data or default values, validate user input, display appropriate error messages for invalid input, and save the input to persistent storage. Yii greatly simplifies this workflow with its MVC architecture.

The following steps are typically needed when dealing with forms in Yii:

1. Create a model class representing the data fields to be collected;

2. Create a controller action with code that responds to form submission.

3. Create a form in the view script file associated with the controller action.

In the next subsections, we describe each of these steps in detail.

3.2 Creating Model

Before writing the HTML code needed by a form, we should decide what kind of data we are expecting from end users and what rules these data should comply with. A model class can be used to record these information. A model, as defined in the Model subsection, is the central place for keeping user inputs and validating them.

Depending on how we make use of the user input, we can create two types of model. If the user input is collected, used and then discarded, we would create a form model; if the user input is collected and saved into database, we would use an active record instead. Both types of model share the same base class CModel which defines the common interface needed by form.

> **Note:** We are mainly using form models in the examples of this section. However, the same can also be applied to active record models.

3.2.1 Defining Model Class

Below we create a `LoginForm` model class used to collect user input on a login page. Because the login information is only used to authenticate the user and does not need to be saved, we create `LoginForm` as a form model.

```
class LoginForm extends CFormModel
{
    public $username;
    public $password;
    public $rememberMe=false;
}
```

Three attributes are declared in `LoginForm`: $username, $password and $rememberMe. They are used to keep the user-entered username and password, and the option whether the user wants to remember his login. Because $rememberMe has a default value `false`, the corresponding option when initially displayed in the login form will be unchecked.

> **Info:** Instead of calling these member variables properties, we use the name *attributes* to differentiate them from normal properties. An attribute is a property that is mainly used to store data coming from user input or database.

3.2.2 Declaring Validation Rules

Once a user submits his inputs and the model gets populated, we need to make sure the inputs are valid before using them. This is done by performing validation of the inputs against a set of rules. We specify the validation rules in the `rules()` method which should return an array of rule configurations.

```
class LoginForm extends CFormModel
{
    public $username;
    public $password;
    public $rememberMe=false;

    private $_identity;
```

```php
    public function rules()
    {
        return array(
            array('username, password', 'required'),
            array('rememberMe', 'boolean'),
            array('password', 'authenticate'),
        );
    }

    public function authenticate($attribute,$params)
    {
        $this->_identity=new UserIdentity($this->username,$this->password);
        if(!$this->_identity->authenticate())
            $this->addError('password','Incorrect username or password.');
    }
}
```

The above code specifies that `username` and `password` are both required, `password` should be authenticated, and `rememberMe` should be a boolean.

Each rule returned by `rules()` must be of the following format:

```php
array('AttributeList', 'Validator', 'on'=>'ScenarioList', ...additional options)
```

where `AttributeList` is a string of comma-separated attribute names which need to be validated according to the rule; `Validator` specifies what kind of validation should be performed; the `on` parameter is optional which specifies a list of scenarios where the rule should be applied; and additional options are name-value pairs which are used to initialize the corresponding validator's property values.

There are three ways to specify `Validator` in a validation rule. First, `Validator` can be the name of a method in the model class, like `authenticate` in the above example. The validator method must be of the following signature:

```php
/**
 * @param string the name of the attribute to be validated
 * @param array options specified in the validation rule
 */
public function ValidatorName($attribute,$params) { ... }
```

Second, `Validator` can be the name of a validator class. When the rule is applied, an instance of the validator class will be created to perform the actual validation. The additional options in the rule are used to initialize the instance's attribute values. A validator class must extend from CValidator.

Third, `Validator` can be a predefined alias to a validator class. In the above example, the name `required` is the alias to CRequiredValidator which ensures the attribute value being validated is not empty. Below is the complete list of predefined validator aliases:

- `boolean`: alias of CBooleanValidator, ensuring the attribute has a value that is either CBooleanValidator::trueValue or CBooleanValidator::falseValue.

- `captcha`: alias of CCaptchaValidator, ensuring the attribute is equal to the verification code displayed in a CAPTCHA.

- `compare`: alias of CCompareValidator, ensuring the attribute is equal to another attribute or constant.

- `email`: alias of CEmailValidator, ensuring the attribute is a valid email address.

- `default`: alias of CDefaultValueValidator, assigning a default value to the specified attributes.

- `exist`: alias of CExistValidator, ensuring the attribute value can be found in the specified table column.

- `file`: alias of CFileValidator, ensuring the attribute contains the name of an uploaded file.

- `filter`: alias of CFilterValidator, transforming the attribute with a filter.

- `in`: alias of CRangeValidator, ensuring the data is among a pre-specified list of values.

- `length`: alias of CStringValidator, ensuring the length of the data is within certain range.

- `match`: alias of CRegularExpressionValidator, ensuring the data matches a regular expression.

- `numerical`: alias of CNumberValidator, ensuring the data is a valid number.

- `required`: alias of CRequiredValidator, ensuring the attribute is not empty.

- `type`: alias of CTypeValidator, ensuring the attribute is of specific data type.

- `unique`: alias of CUniqueValidator, ensuring the data is unique in a database table column.

- `url`: alias of CUrlValidator, ensuring the data is a valid URL.

Below we list some examples of using the predefined validators:

```
// username is required
array('username', 'required'),
// username must be between 3 and 12 characters
array('username', 'length', 'min'=>3, 'max'=>12),
// when in register scenario, password must match password2
array('password', 'compare', 'compareAttribute'=>'password2', 'on'=>'register'),
// when in login scenario, password must be authenticated
array('password', 'authenticate', 'on'=>'login'),
```

3.2.3 Securing Attribute Assignments

After a model instance is created, we often need to populate its attributes with the data submitted by end-users. This can be done conveniently using the following massive assignment:

```
$model=new LoginForm;
if(isset($_POST['LoginForm']))
    $model->attributes=$_POST['LoginForm'];
```

The last statement is called *massive assignment* which assigns every entry in `$_POST['LoginForm']` to the corresponding model attribute. It is equivalent to the following assignments:

```
foreach($_POST['LoginForm'] as $name=>$value)
{
    if($name is a safe attribute)
        $model->$name=$value;
}
```

It is crucial to determine which attributes are safe. For example, if we expose the primary key of a table to be safe, then an attacker could get a chance to modify the primary key of the given record and thus tamper the data he is not authorized to.

The policy of deciding which attributes are safe is different in version 1.0 and 1.1. In the following, we will describe them separately.

Safe Attributes in 1.1

In version 1.1, an attribute is considered safe if it appears in a validation rule that is applicable in the given scenario. For example,

```
array('username, password', 'required', 'on'=>'login, register'),
array('email', 'required', 'on'=>'register'),
```

In the above, the `username` and `password` attributes are required in `login` scenario, while the `username`, `password` and `email` attributes are required in `register` scenario. As a result, if we perform a massive assign when in `login` scenario, only `username` and `password` will be massively assigned since they are the only attributes appearing in the validation rules for `login`. On the other hand, if the scenario is `register`, the three attributes can all be massively assigned.

```
// in login scenario
$model=new User('login');
if(isset($_POST['User']))
    $model->attributes=$_POST['User'];

// in register scenario
$model=new User('register');
if(isset($_POST['User']))
    $model->attributes=$_POST['User'];
```

So why do we use such a policy to determine if an attribute is safe or not? The rationale behind is that if an attribute already has one or several validation rules to check its validity, what else should we worry about it?

It is important to remember that validation rules are used to check user input data rather than the data that we generate in the code (e.g. timestamp, auto-generated primary key). Therefore, DO NOT add validation rules for those attributes which do not expect inputs from end-users.

Sometimes, we want to declare an attribute to be safe, even though we do not really have any specific rule for it. An example is an article's content attribute which can take any user input. We can use the special `safe` rule to achieve this goal:

```
array('content', 'safe')
```

For completeness, there is also an `unsafe` rule which is used to explicitly declare an attribute to be unsafe:

```
array('permission', 'unsafe')
```

The `unsafe` rule is rarely used, and it is an exception to our previous definition of safe attributes.

Safe Attributes in 1.0

In version 1.0, the task of deciding whether a data entry is safe or not is based on the return value of a method named `safeAttributes` and the specified scenario. By default, the method returns all public member variables as safe attributes for CFormModel, while it returns all table columns except the primary key as safe attributes for CActiveRecord. We may override this method to limit safe attributes according to scenarios. For example, a user model may contain many attributes, but in `login` scenario we only need to use `username` and `password` attributes. We can specify this limit as follows:

```php
public function safeAttributes()
{
    return array(
        parent::safeAttributes(),
        'login' => 'username, password',
    );
}
```

More accurately, the return value of the `safeAttributes` method should be of the following structure:

```php
array(
    // these attributes can be massively assigned in any scenario
    // that is not explicitly specified below
    'attr1, attr2, ...',
      *
    // these attributes can be massively assigned only in scenario 1
    'scenario1' => 'attr2, attr3, ...',
      *
    // these attributes can be massively assigned only in scenario 2
    'scenario2' => 'attr1, attr3, ...',
)
```

If the model is not scenario-sensitive (i.e., it is only used in one scenario, or all scenarios share the same set of safe attributes), the return value can be simplified as a single string:

```php
'attr1, attr2, ...'
```

For data entries that are not safe, we need to assign them to the corresponding attributes using individual assign statements, like the following:

```php
$model->permission='admin';
$model->id=1;
```

3.2.4 Triggering Validation

Once a model is populated with user-submitted data, we can call CModel::validate() to trigger the data validation process. The method returns a value indicating whether the validation is successful or not. For CActiveRecord models, validation may also be automatically triggered when we call its CActiveRecord::save() method.

We can set a scenario with the scenario property and therewith indicate which set of validation rules should be applied.

Validation is performed in a scenario basis. The scenario property specifies which scenario the model is being used in and which set of validation rules should be used. For example, in the `login` scenario, we only want to validate the `username` and `password` inputs of a user model; while in the `register` scenario, we need to validate more inputs, such as `email`, `address`, etc. The following example shows how to perform validation in the `register` scenario:

```
// creates a User model in register scenario. It is equivalent to:
// $model=new User;
// $model->scenario='register';
$model=new User('register');

// populates the input values into the model
$model->attributes=$_POST['User'];

// performs the validation
if($model->validate())  // if the inputs are valid
    ...
else
    ...
```

The applicable scenarios that a rule is associated can be specified via the `on` option in the rule. If the `on` option is not set, it means the rule will be used for all scenarios. For example,

```
public function rules()
{
    return array(
        array('username, password', 'required'),
        array('password_repeat', 'required', 'on'=>'register'),
        array('password', 'compare', 'on'=>'register'),
    );
}
```

The first rule will be applied in all scenarios, while the next two rules will only be applied in the `register` scenario.

3.2.5   Retrieving Validation Errors

Once validation is done, any possible errors will be stored in the model object. We can retrieve the error messages by calling CModel::getErrors() and CModel::getError(). The difference between the two methods is that the first method will return *all* errors for the specified model attribute while the second method will only return the *first* error.

3.2.6   Attribute Labels

When designing a form, we often need to display a label for each input field. The label tells a user what kind of information he is expected to enter into the field. Although we can hardcode a label in a view, it would offer more flexibility and convenience if we specify it in the corresponding model.

By default, CModel will simply return the name of an attribute as its label. This can be customized by overriding the attributeLabels() method. As we will see in the following subsections, specifying labels in the model allows us to create a form more quickly and powerful.

3.3   Creating Action

Once we have a model, we can start to write logic that is needed to manipulate the model. We place this logic inside a controller action. For the login form example, the following code is needed:

```php
public function actionLogin()
{
    $model=new LoginForm;
    if(isset($_POST['LoginForm']))
    {
        // collects user input data
        $model->attributes=$_POST['LoginForm'];
        // validates user input and redirect to previous page if validated
        if($model->validate())
            $this->redirect(Yii::app()->user->returnUrl);
    }
    // displays the login form
    $this->render('login',array('model'=>$model));
}
```

In the above, we first create a `LoginForm` model instance; if the request is a POST request (meaning the login form is submitted), we populate `$model` with the submitted data `$_POST['LoginForm']`; we then validate the input and if successful, redirect the user browser to the page that previously needed authentication. If the validation fails, or if the action is initially accessed, we render the `login` view whose content is to be described in the next subsection.

> **Tip:** In the `login` action, we use `Yii::app()->user->returnUrl` to get the URL of the page that previously needed authentication. The component `Yii::app()->user` is of type CWebUser (or its child class) which represents user session information (e.g. username, status). For more details, see Authentication and Authorization.

Let's pay special attention to the following PHP statement that appears in the `login` action:

```
$model->attributes=$_POST['LoginForm'];
```

As we described in Securing Attribute Assignments, this line of code populates the model with the user submitted data. The `attributes` property is defined by CModel which expects an array of name-value pairs and assigns each value to the corresponding model attribute. So if `$_POST['LoginForm']` gives us such an array, the above code would be equivalent to the following lengthy one (assuming every needed attribute is present in the array):

```
$model->username=$_POST['LoginForm']['username'];
$model->password=$_POST['LoginForm']['password'];
$model->rememberMe=$_POST['LoginForm']['rememberMe'];
```

> **Note:** In order to let `$_POST['LoginForm']` to give us an array instead of a string, we stick to a convention when naming input fields in the view. In particular, for an input field corresponding to attribute `a` of model class `C`, we name it as `C[a]`. For example, we would use `LoginForm[username]` to name the input field corresponding to the `username` attribute.

The remaining task now is to create the `login` view which should contain an HTML form with the needed input fields.

3.4   Creating Form

Writing the `login` view is straightforward. We start with a `form` tag whose action attribute should be the URL of the `login` action described previously. We then insert labels and input fields for the attributes declared in the `LoginForm` class. At the end we insert a submit button which can be clicked by users to submit the form. All these can be done in pure HTML code.

Yii provides a few helper classes to facilitate view composition. For example, to create a text input field, we can call CHtml::textField(); to create a drop-down list, call CHtml::dropDownList().

Info: One may wonder what is the benefit of using helpers if they require similar amount of code when compared with plain HTML code. The answer is that the helpers can provide more than just HTML code. For example, the following code would generate a text input field which can trigger form submission if its value is changed by users.

```
CHtml::textField($name,$value,array('submit'=>''));
```

It would otherwise require writing clumsy JavaScript everywhere.

In the following, we use CHtml to create the login form. We assume that the variable `$model` represents `LoginForm` instance.

```
<div class="form">
<?php echo CHtml::beginForm(); ?>

    <?php echo CHtml::errorSummary($model); ?>

    <div class="row">
        <?php echo CHtml::activeLabel($model,'username'); ?>
        <?php echo CHtml::activeTextField($model,'username') ?>
    </div>

    <div class="row">
        <?php echo CHtml::activeLabel($model,'password'); ?>
        <?php echo CHtml::activePasswordField($model,'password') ?>
    </div>

    <div class="row rememberMe">
        <?php echo CHtml::activeCheckBox($model,'rememberMe'); ?>
        <?php echo CHtml::activeLabel($model,'rememberMe'); ?>
    </div>
```

```
    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>

<?php echo CHtml::endForm(); ?>
</div><!-- form -->
```

The above code generates a more dynamic form. For example, CHtml::activeLabel() generates a label associated with the specified model attribute. If the attribute has an input error, the label's CSS class will be changed to `error`, which changes the appearance of the label with appropriate CSS styles. Similarly, CHtml::activeTextField() generates a text input field for the specified model attribute and changes its CSS class upon any input error.

If we use the CSS style file `form.css` provided by the `yiic` script, the generated form would be like the following:
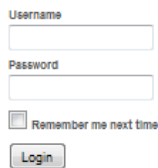
Figure 3.1: The login page
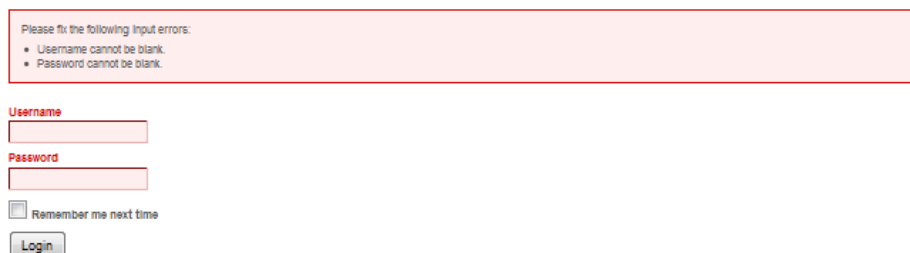
Figure 3.2: The login with error page

Starting from version 1.1.1, a new widget called CActiveForm is provided to facilitate form creation. The widget is capable of supporting seamless and consistent validation on both client and server sides. Using CActiveForm, the above view code can be rewritten as:

```
<div class="form">
<?php $form=$this->beginWidget('CActiveForm'); ?>
```

```php
    <?php echo $form->errorSummary($model); ?>

    <div class="row">
        <?php echo $form->label($model,'username'); ?>
        <?php echo $form->textField($model,'username') ?>
    </div>

    <div class="row">
        <?php echo $form->label($model,'password'); ?>
        <?php echo $form->passwordField($model,'password') ?>
    </div>

    <div class="row rememberMe">
        <?php echo $form->checkBox($model,'rememberMe'); ?>
        <?php echo $form->label($model,'rememberMe'); ?>
    </div>

    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>

<?php $this->endWidget(); ?>
</div><!-- form -->
```

3.5 Collecting Tabular Input

Sometimes we want to collect user input in a batch mode. That is, the user can enter the information for multiple model instances and submit them all at once. We call this *tabular input* because the input fields are often presented in an HTML table.

To work with tabular input, we first need to create or populate an array of model instances, depending on whether we are inserting or updating the data. We then retrieve the user input data from the $_POST variable and assign it to each model. A slight difference from single model input is that we retrieve the input data using $_POST['ModelClass'][$i] instead of $_POST['ModelClass'].

```php
public function actionBatchUpdate()
{
    // retrieve items to be updated in a batch mode
    // assuming each item is of model class 'Item'
    $items=$this->getItemsToUpdate();
    if(isset($_POST['Item']))
    {
        $valid=true;
        foreach($items as $i=>$item)
```

```
    {
        if(isset($_POST['Item'][$i]))
            $item->attributes=$_POST['Item'][$i];
        $valid=$valid && $item->validate();
    }
    if($valid)  // all items are valid
        // ...do something here
    }
    // displays the view to collect tabular input
    $this->render('batchUpdate',array('items'=>$items));
}
```

Having the action ready, we need to work on the `batchUpdate` view to display the input fields in an HTML table.

```
<div class="form">
<?php echo CHtml::beginForm(); ?>
<table>
<tr><th>Name</th><th>Price</th><th>Count</th><th>Description</th></tr>
<?php foreach($items as $i=>$item): ?>
<tr>
<td><?php echo CHtml::activeTextField($item,"[$i]name"); ?></td>
<td><?php echo CHtml::activeTextField($item,"[$i]price"); ?></td>
<td><?php echo CHtml::activeTextField($item,"[$i]count"); ?></td>
<td><?php echo CHtml::activeTextArea($item,"[$i]description"); ?></td>
</tr>
<?php endforeach; ?>
</table>

<?php echo CHtml::submitButton('Save'); ?>
<?php echo CHtml::endForm(); ?>
</div><!-- form -->
```

Note in the above that we use `"[$i]name"` instead of `"name"` as the second parameter when calling CHtml::activeTextField.

If there are any validation errors, the corresponding input fields will be highlighted automatically, just like the single model input we described earlier on.

3.6   Using Form Builder

When creating HTML forms, we often find that we are writing a lot of repetitive view code which is difficult to be reused in a different project. For example, for every input field, we need to associate it with a text label and display possible validation errors. To

improve the reusability of these code, we can use the form builder feature which has been available since Yii version 1.1.0.

3.6.1 Basic Concepts

The Yii form builder uses a CForm object to represent the specifications needed to describe an HTML form, including which data models are associated with the form, what kind of input fields there are in the form, and how to render the whole form. Developers mainly need to create and configure this CForm object, and then call its rendering method to display the form.

Form input specifications are organized in terms of a form element hierarchy. At the root of the hierarchy, it is the CForm object. The root form object maintains its children in two collections: CForm::buttons and CForm::elements. The former contains the button elements (such as submit buttons, reset buttons), while the latter contains the input elements, static text and sub-forms. A sub-form is a CForm object contained in the CForm::elements collection of another form. It can have its own data model, CForm::buttons and CForm::elements collections.

When users submit a form, the data entered into the input fields of the whole form hierarchy are submitted, including those input fields that belong to the sub-forms. CForm provides convenient methods that can automatically assign the input data to the corresponding model attributes and perform data validation.

3.6.2 Creating a Simple Form

In the following, we show how to use the form builder to create a login form.

First, we write the login action code:

```php
public function actionLogin()
{
    $model = new LoginForm;
    $form = new CForm('application.views.site.loginForm', $model);
    if($form->submitted('login') && $form->validate())
        $this->redirect(array('site/index'));
    else
        $this->render('login', array('form'=>$form));
}
```

In the above code, we create a CForm object using the specifications pointed to by the path alias `application.views.site.loginForm` (to be explained shortly). The CForm object is associated with the `LoginForm` model as described in Creating Model.

As the code reads, if the form is submitted and all inputs are validated without any error, we would redirect the user browser to the `site/index` page. Otherwise, we render the `login` view with the form.

The path alias `application.views.site.loginForm` actually refers to the PHP file `protected/views/site/loginForm.php`. The file should return a PHP array representing the configuration needed by CForm, as shown in the following:

```php
return array(
    'title'=>'Please provide your login credential',

    'elements'=>array(
        'username'=>array(
            'type'=>'text',
            'maxlength'=>32,
        ),
        'password'=>array(
            'type'=>'password',
            'maxlength'=>32,
        ),
        'rememberMe'=>array(
            'type'=>'checkbox',
        )
    ),

    'buttons'=>array(
        'login'=>array(
            'type'=>'submit',
            'label'=>'Login',
        ),
    ),
);
```

The configuration is an associative array consisting of name-value pairs that are used to initialize the corresponding properties of CForm. The most important properties to configure, as we aforementioned, are CForm::elements and CForm::buttons. Each of them takes an array specifying a list of form elements. We will give more details on how to configure form elements in the next sub-section.

Finally, we write the `login` view script, which can be as simple as follows,

```php
<h1>Login</h1>

<div class="form">
<?php echo $form; ?>
</div>
```

> **Tip:** The above code `echo $form;` is equivalent to `echo $form->render();`. This is because CForm implements `__toString` magic method which calls `render()` and returns its result as the string representation of the form object.

3.6.3 Specifying Form Elements

Using the form builder, the majority of our effort is shifted from writing view script code to specifying the form elements. In this sub-section, we describe how to specify the CForm::elements property. We are not going to describe CForm::buttons because its configuration is nearly the same as CForm::elements.

The CForm::elements property accepts an array as its value. Each array element specifies a single form element which can be an input element, a static text string or a sub-form.

Specifying Input Element

An input element mainly consists of a label, an input field, a hint text and an error display. It must be associated with a model attribute. The specification for an input element is represented as a CFormInputElement instance. The following code in the CForm::elements array specifies a single input element:

```
'username'=>array(
    'type'=>'text',
    'maxlength'=>32,
),
```

It states that the model attribute is named as `username`, and the input field type is `text` whose `maxlength` attribute is 32.

Any writable property of CFormInputElement can be configured like above. For example, we may specify the hint option in order to display a hint text, or we may specify the items option if the input field is a list box, a drop-down list, a check-box list or a radio-button list. If an option name is not a property of CFormInputElement, it will be treated the attribute of the corresponding HTML input element. For example, because `maxlength` in the above is not a property of CFormInputElement, it will be rendered as the `maxlength` attribute of the HTML text input field.

The type option deserves additional attention. It specifies the type of the input field to be rendered. For example, the `text` type means a normal text input field should be rendered;

the `password` type means a password input field should be rendered. CFormInputElement recognizes the following built-in types:

- text

- hidden

- password

- textarea

- file

- radio

- checkbox

- listbox

- dropdownlist

- checkboxlist

- radiolist

Among the above built-in types, we would like to describe a bit more about the usage of those "list" types, which include `dropdownlist`, `checkboxlist` and `radiolist`. These types require setting the items property of the corresponding input element. One can do so like the following:

```php
'gender'=>array(
    'type'=>'dropdownlist',
    'items'=>User::model()->getGenderOptions(),
    'prompt'=>'Please select:',
),

...

class User extends CActiveRecord
{
    public function getGenderOptions()
    {
        return array(
            0 => 'Male',
            1 => 'Female',
        );
    }
}
```

The above code will generate a drop-down list selector with prompt text "please select:". The selector options include "Male" and "Female", which are returned by the `getGenderOptions` method in the `User` model class.

Besides these built-in types, the type option can also take a widget class name or the path alias to it. The widget class must extend from CInputWidget or CJuiInputWidget. When rendering the input element, an instance of the specified widget class will be created and rendered. The widget will be configured using the specification as given for the input element.

Specifying Static Text

In many cases, a form may contain some decorational HTML code besides the input fields. For example, a horizontal line may be needed to separate different portions of the form; an image may be needed at certain places to enhance the visual appearance of the form. We may specify these HTML code as static text in the CForm::elements collection. To do so, we simply specify a static text string as an array element in the appropriate position in CForm::elements. For example,

```php
return array(
    'elements'=>array(
        ......
        'password'=>array(
            'type'=>'password',
            'maxlength'=>32,
        ),

        '<hr />',

        'rememberMe'=>array(
            'type'=>'checkbox',
        )
    ),
    ......
);
```

In the above, we insert a horizontal line between the `password` input and the `rememberMe` input.

Static text is best used when the text content and their position are irregular. If each input element in a form needs to be decorated similarly, we should customize the form rendering approach, as to be explained shortly in this section.

Specifying Sub-form

Sub-forms are used to divide a lengthy form into several logically connected portions. For example, we may divide user registration form into two sub-forms: login information and profile information. Each sub-form may or may not be associated with a data model. In the user registration form example, if we store user login information and profile information in two separate database tables (and thus two data models), then each sub-form would be associated with a corresponding data model. If we store everything in a single database table, then neither sub-form has a data model because they share the same model with the root form.

A sub-form is also represented as a CForm object. In order to specify a sub-form, we should configure the CForm::elements property with an element whose type is `form`:

```
return array(
    'elements'=>array(
        ......
        'user'=>array(
            'type'=>'form',
            'title'=>'Login Credential',
            'elements'=>array(
                'username'=>array(
                    'type'=>'text',
                ),
                'password'=>array(
                    'type'=>'password',
                ),
                'email'=>array(
                    'type'=>'text',
                ),
            ),
        ),

        'profile'=>array(
            'type'=>'form',
            ......
        ),
        ......
    ),
    ......
);
```

Like configuring a root form, we mainly need to specify the CForm::elements property for a sub-form. If a sub-form needs to be associated with a data model, we can configure its CForm::model property as well.

Sometimes, we may want to represent a form using a class other than the default CForm. For example, as will show shortly in this section, we may extend CForm to customize the form rendering logic. By specifying the input element type to be `form`, a sub-form will automatically be represented as an object whose class is the same as its parent form. If we specify the input element type to be something like `XyzForm` (a string terminated with `Form`), then the sub-form will be represented as a `XyzForm` object.

3.6.4 Accessing Form Elements

Accessing form elements is as simple as accessing array elements. The CForm::elements property returns a CFormElementCollection object, which extends from CMap and allows accessing its elements like a normal array. For example, in order to access the `username` element in the login form example, we can use the following code:

```
$username = $form->elements['username'];
```

And to access the `email` element in the user registration form example, we can use

```
$email = $form->elements['user']->elements['email'];
```

Because CForm implements array access for its CForm::elements property, the above code can be further simplified as:

```
$username = $form['username'];
$email = $form['user']['email'];
```

3.6.5 Creating a Nested Form

We already described sub-forms. We call a form with sub-forms a nested form. In this section, we use the user registration form as an example to show how to create a nested form associated with multiple data models. We assume the user credential information is stored as a `User` model, while the user profile information is stored as a `Profile` model.

We first create the `register` action as follows:

```
public function actionRegister()
{
    $form = new CForm('application.views.user.registerForm');
    $form['user']->model = new User;
    $form['profile']->model = new Profile;
```

```php
if($form->submitted('register') && $form->validate())
{
    $user = $form['user']->model;
    $profile = $form['profile']->model;
    if($user->save(false))
    {
        $profile->userID = $user->id;
        $profile->save(false);
        $this->redirect(array('site/index'));
    }
}

$this->render('register', array('form'=>$form));
}
```

In the above, we create the form using the configuration specified by `application.views.user.registerForm`. After the form is submitted and validated successfully, we attempt to save the user and profile models. We retrieve the user and profile models by accessing the `model` property of the corresponding sub-form objects. Because the input validation is already done, we call `$user->save(false)` to skip the validation. We do this similarly for the profile model.

Next, we write the form configuration file `protected/views/user/registerForm.php`:

```php
return array(
    'elements'=>array(
        'user'=>array(
            'type'=>'form',
            'title'=>'Login information',
            'elements'=>array(
                'username'=>array(
                    'type'=>'text',
                ),
                'password'=>array(
                    'type'=>'password',
                ),
                'email'=>array(
                    'type'=>'text',
                )
            ),
        ),

        'profile'=>array(
            'type'=>'form',
            'title'=>'Profile information',
            'elements'=>array(
                'firstName'=>array(
```

```
                'type'=>'text',
            ),
            'lastName'=>array(
                'type'=>'text',
            ),
        ),
    ),
),

'buttons'=>array(
    'register'=>array(
        'type'=>'submit',
        'label'=>'Register',
    ),
),
);
```

In the above, when specifying each sub-form, we also specify its CForm::title property. The default form rendering logic will enclose each sub-form in a field-set which uses this property as its title.

Finally, we write the simple `register` view script:

```
<h1>Register</h1>

<div class="form">
<?php echo $form; ?>
</div>
```

3.6.6 Customizing Form Display

The main benefit of using form builder is the separation of logic (form configuration stored in a separate file) and presentation (CForm::render method). As a result, we can customize the form display by either overriding CForm::render or providing a partial view to render the form. Both approaches can keep the form configuration intact and can be reused easily.

When overriding CForm::render, one mainly needs to traverse through the CForm::elements and CForm::buttons collections and call the CFormElement::render method of each form element. For example,

```
class MyForm extends CForm
{
    public function render()
```

```
    {
        $output = $this->renderBegin();

        foreach($this->getElements() as $element)
            $output .= $element->render();

        $output .= $this->renderEnd();

        return $output;
    }
}
```

We may also write a view script _form to render a form:

```
<?php
echo $form->renderBegin();

foreach($form->getElements() as $element)
    echo $element->render();

echo $form->renderEnd();
```

To use this view script, we can simply call:

```
<div class="form">
$this->renderPartial('_form', array('form'=>$form));
</div>
```

If a generic form rendering does not work for a particular form (for example, the form needs some irregular decorations for certain elements), we can do like the following in a view script:

```
some complex UI elements here

<?php echo $form['username']; ?>

some complex UI elements here

<?php echo $form['password']; ?>

some complex UI elements here
```

In the last approach, the form builder seems not to bring us much benefit, as we still need to write similar amount of form code. It is still beneficial, however, that the form is

specified using a separate configuration file as it helps developers to better focus on the logic.

CHAPTER 4

Working with Databases

4.1 Working with Database

Yii provides powerful support for database programming. Built on top of the PHP Data Objects (PDO) extension, Yii Data Access Objects (DAO) enables accessing to different database management systems (DBMS) in a single uniform interface. Applications developed using Yii DAO can be easily switched to use a different DBMS without the need to modify the data accessing code. Yii Active Record (AR), implemented as a widely adopted Object-Relational Mapping (ORM) approach, further simplifies database programming. Representing a table in terms of a class and a row an instance, Yii AR eliminates the repetitive task of writing those SQL statements that mainly deal with CRUD (create, read, update and delete) operations.

Although Yii DAO and AR can handle nearly all database-related tasks, you can still use your own database libraries in your Yii application. As a matter of fact, Yii framework is carefully designed to be used together with other third-party libraries.

4.2 Data Access Objects (DAO)

Data Access Objects (DAO) provides a generic API to access data stored in different database management systems (DBMS). As a result, the underlying DBMS can be changed to a different one without requiring change of the code which uses DAO to access the data.

Yii DAO is built on top of PHP Data Objects (PDO) which is an extension providing unified data access to many popular DBMS, such as MySQL, PostgreSQL. Therefore, to use Yii DAO, the PDO extension and the specific PDO database driver (e.g. `PDO_MYSQL`) have to be installed.

Yii DAO mainly consists of the following four classes:

- CDbConnection: represents a connection to a database.

- CDbCommand: represents an SQL statement to execute against a database.

- CDbDataReader: represents a forward-only stream of rows from a query result set.

- CDbTransaction: represents a DB transaction.

In the following, we introduce the usage of Yii DAO in different scenarios.

4.2.1   Establishing Database Connection

To establish a database connection, create a CDbConnection instance and activate it. A data source name (DSN) is needed to specify the information required to connect to the database. A username and password may also be needed to establish the connection. An exception will be raised in case an error occurs during establishing the connection (e.g. bad DSN or invalid username/password).

```
$connection=new CDbConnection($dsn,$username,$password);
// establish connection. You may try...catch possible exceptions
$connection->active=true;
......
$connection->active=false;  // close connection
```

The format of DSN depends on the PDO database driver in use. In general, a DSN consists of the PDO driver name, followed by a colon, followed by the driver-specific connection syntax. See PDO documentation for complete information. Below is a list of commonly used DSN formats:

- SQLite: `sqlite:/path/to/dbfile`

- MySQL: `mysql:host=localhost;dbname=testdb`

- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=testdb`

- SQL Server: `mssql:host=localhost;dbname=testdb`

- Oracle: `oci:dbname=//localhost:1521/testdb`

Because CDbConnection extends from CApplicationComponent, we can also use it as an application component. To do so, configure in a `db` (or other name) application component in the application configuration as follows,

```
array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'mysql:host=localhost;dbname=testdb',
            'username'=>'root',
            'password'=>'password',
            'emulatePrepare'=>true,  // needed by some MySQL installations
        ),
    ),
)
```

We can then access the DB connection via `Yii::app()->db` which is already activated automatically, unless we explictly configure CDbConnection::autoConnect to be false. Using this approach, the single DB connection can be shared in multiple places in our code.

4.2.2 Executing SQL Statements

Once a database connection is established, SQL statements can be executed using CDbCommand. One creates a CDbCommand instance by calling CDbConnection::createCommand() with the specified SQL statement:

```
$connection=Yii::app()->db;   // assuming you have configured a "db" connection
// If not, you may explicitly create a connection:
// $connection=new CDbConnection($dsn,$username,$password);
$command=$connection->createCommand($sql);
// if needed, the SQL statement may be updated as follows:
// $command->text=$newSQL;
```

A SQL statement is executed via CDbCommand in one of the following two ways:

- execute(): performs a non-query SQL statement, such as `INSERT`, `UPDATE` and `DELETE`. If successful, it returns the number of rows that are affected by the execution.

- query(): performs an SQL statement that returns rows of data, such as `SELECT`. If successful, it returns a CDbDataReader instance from which one can traverse the resulting rows of data. For convenience, a set of `queryXXX()` methods are also implemented which directly return the query results.

An exception will be raised if an error occurs during the execution of SQL statements.

```php
$rowCount=$command->execute();    // execute the non-query SQL
$dataReader=$command->query();    // execute a query SQL
$rows=$command->queryAll();       // query and return all rows of result
$row=$command->queryRow();        // query and return the first row of result
$column=$command->queryColumn();  // query and return the first column of result
$value=$command->queryScalar();   // query and return the first field in the first row
```

4.2.3 Fetching Query Results

After CDbCommand::query() generates the CDbDataReader instance, one can retrieve rows of resulting data by calling CDbDataReader::read() repeatedly. One can also use CDbDataReader in PHP's `foreach` language construct to retrieve row by row.

```php
$dataReader=$command->query();
// calling read() repeatedly until it returns false
while(($row=$dataReader->read())!==false) { ... }
// using foreach to traverse through every row of data
foreach($dataReader as $row) { ... }
// retrieving all rows at once in a single array
$rows=$dataReader->readAll();
```

> **Note:** Unlike query(), all `queryXXX()` methods return data directly. For example, queryRow() returns an array representing the first row of the querying result.

4.2.4 Using Transactions

When an application executes a few queries, each reading and/or writing information in the database, it is important to be sure that the database is not left with only some of the queries carried out. A transaction, represented as a CDbTransaction instance in Yii, may be initiated in this case:

- Begin the transaction.

- Execute queries one by one. Any updates to the database are not visible to the outside world.

- Commit the transaction. Updates become visible if the transaction is successful.

- If one of the queries fails, the entire transaction is rolled back.

The above workflow can be implemented using the following code:

```php
$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //.... other SQL executions
    $transaction->commit();
}
catch(Exception $e) // an exception is raised if a query fails
{
    $transaction->rollBack();
}
```

4.2.5 Binding Parameters

To avoid SQL injection attacks and to improve performance of executing repeatedly used SQL statements, one can "prepare" an SQL statement with optional parameter placeholders that are to be replaced with the actual parameters during the parameter binding process.

The parameter placeholders can be either named (represented as unique tokens) or unnamed (represented as question marks). Call CDbCommand::bindParam() or CDbCommand::bindValue() to replace these placeholders with the actual parameters. The parameters do not need to be quoted: the underlying database driver does it for you. Parameter binding must be done before the SQL statement is executed.

```php
// an SQL with two placeholders ":username" and ":email"
$sql="INSERT INTO tbl_user (username, email) VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// replace the placeholder ":username" with the actual username value
$command->bindParam(":username",$username,PDO::PARAM_STR);
// replace the placeholder ":email" with the actual email value
$command->bindParam(":email",$email,PDO::PARAM_STR);
$command->execute();
// insert another row with a new set of parameters
$command->bindParam(":username",$username2,PDO::PARAM_STR);
$command->bindParam(":email",$email2,PDO::PARAM_STR);
$command->execute();
```

The methods bindParam() and bindValue() are very similar. The only difference is that the former binds a parameter with a PHP variable reference while the latter with a value. For parameters that represent large blocks of data memory, the former is preferred for performance consideration.

For more details about binding parameters, see the relevant PHP documentation.

4.2.6   Binding Columns

When fetching query results, one can also bind columns with PHP variables so that they are automatically populated with the latest data each time a row is fetched.

```php
$sql="SELECT username, email FROM tbl_user";
$dataReader=$connection->createCommand($sql)->query();
// bind the 1st column (username) with the $username variable
$dataReader->bindColumn(1,$username);
// bind the 2nd column (email) with the $email variable
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // $username and $email contain the username and email in the current row
}
```

4.2.7   Using Table Prefix

Starting from version 1.1.0, Yii provides integrated support for using table prefix. Table prefix means a string that is prepended to the names of the tables in the currently connected database. It is mostly used in a shared hosting environment where multiple applications share a single database and use different table prefixes to differentiate from each other. For example, one application could use `tbl_` as prefix while the other `yii_`.

To use table prefix, configure the CDbConnection::tablePrefix property to be the desired table prefix. Then, in SQL statements use {{TableName}} to refer to table names, where `TableName` means the table name without prefix. For example, if the database contains a table named `tbl_user` where `tbl_` is configured as the table prefix, then we can use the following code to query about users:

```php
$sql='SELECT * FROM {{user}}';
$users=$connection->createCommand($sql)->queryAll();
```

4.3   Active Record

Although Yii DAO can handle virtually any database-related task, chances are that we would spend 90% of our time in writing some SQL statements which perform the common CRUD (create, read, update and delete) operations. It is also difficult to maintain our code when they are mixed with SQL statements. To solve these problems, we can use Active Record.

Active Record (AR) is a popular Object-Relational Mapping (ORM) technique. Each AR class represents a database table (or view) whose attributes are represented as the AR

class properties, and an AR instance represents a row in that table. Common CRUD operations are implemented as AR methods. As a result, we can access our data in a more object-oriented way. For example, we can use the following code to insert a new row to the `tbl_post` table:

```php
$post=new Post;
$post->title='sample post';
$post->content='post body content';
$post->save();
```

In the following we describe how to set up AR and use it to perform CRUD operations. We will show how to use AR to deal with database relationships in the next section. For simplicity, we use the following database table for our examples in this section. Note that if you are using MySQL database, you should replace AUTOINCREMENT with AUTO_INCREMENT in the following SQL.

```sql
CREATE TABLE tbl_post (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    title VARCHAR(128) NOT NULL,
    content TEXT NOT NULL,
    create_time INTEGER NOT NULL
);
```

> **Note:** AR is not meant to solve all database-related tasks. It is best used for modeling database tables in PHP constructs and performing queries that do not involve complex SQLs. Yii DAO should be used for those complex scenarios.

4.3.1 Establishing DB Connection

AR relies on a DB connection to perform DB-related operations. By default, it assumes that the `db` application component gives the needed CDbConnection instance which serves as the DB connection. The following application configuration shows an example:

```php
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:path/to/dbfile',
            // turn on schema caching to improve performance
            // 'schemaCachingDuration'=>3600,
        ),
    ),
);
```

> **Tip:** Because Active Record relies on the metadata about tables to determine the column information, it takes time to read the metadata and analyze it. If the schema of your database is less likely to be changed, you should turn on schema caching by configuring the CDbConnection::schemaCachingDuration property to be a value greater than 0.

Support for AR is limited by DBMS. Currently, only the following DBMS are supported:

- MySQL 4.1 or later

- PostgreSQL 7.3 or later

- SQLite 2 and 3

- Microsoft SQL Server 2000 or later

- Oracle

> **Note:** The support for Microsoft SQL Server has been available since version 1.0.4; And the support for Oracle has been available since version 1.0.5.

If you want to use an application component other than db, or if you want to work with multiple databases using AR, you should override CActiveRecord::getDbConnection(). The CActiveRecord class is the base class for all AR classes.

> **Tip:** There are two ways to work with multiple databases in AR. If the schemas of the databases are different, you may create different base AR classes with different implementation of getDbConnection(). Otherwise, dynamically changing the static variable CActiveRecord::db is a better idea.

4.3.2 Defining AR Class

To access a database table, we first need to define an AR class by extending CActiveRecord. Each AR class represents a single database table, and an AR instance represents a row in that table. The following example shows the minimal code needed for the AR class representing the tbl_post table.

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

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

> **Tip:** Because AR classes are often referenced in many places, we can import the whole directory containing the AR class, instead of including them one by one. For example, if all our AR class files are under `protected/models`, we can configure the application as follows:
>
> ```php
> return array(
>   'import'=>array(
>       'application.models.*',
>   ),
> );
> ```

By default, the name of the AR class is the same as the database table name. Override the tableName() method if they are different. The model() method is declared as such for every AR class (to be explained shortly).

> **Info:** To use the table prefix feature introduced in version 1.1.0, the tableName() method for an AR class may be overridden as follows,
>
> ```php
> public function tableName()
> {
>     return '{{post}}';
> }
> ```
>
> That is, instead of returning the fully qualified table name, we return the table name without the prefix and enclose it in double curly brackets.

Column values of a table row can be accessed as properties of the corresponding AR instance. For example, the following code sets the `title` column (attribute):

```php
$post=new Post;
$post->title='a sample post';
```

Although we never explicitly declare the `title` property in the `Post` class, we can still access it in the above code. This is because `title` is a column in the `tbl_post` table, and CActiveRecord makes it accessible as a property with the help of the PHP `__get()` magic method. An exception will be thrown if we attempt to access a non-existing column in the same way.

Info: In this guide, we use lower case for all table names and column names. This is because different DBMS handle case-sensitivity differently. For example, PostgreSQL treats column names as case-insensitive by default, and we must quote a column in a query condition if the column contains mixed-case letters. Using lower case would help eliminate this problem.

AR relies on well defined primary keys of tables. If a table does not have a primary key, it is required that the corresponding AR class specify which column(s) should be the primary key by overriding the `primaryKey()` method as follows,

```php
public function primaryKey()
{
    return 'id';
    // For composite primary key, return an array like the following
    // return array('pk1', 'pk2');
}
```

4.3.3 Creating Record

To insert a new row into a database table, we create a new instance of the corresponding AR class, set its properties associated with the table columns, and call the save() method to finish the insertion.

```php
$post=new Post;
$post->title='sample post';
$post->content='content for the sample post';
$post->create_time=time();
$post->save();
```

If the table's primary key is auto-incremental, after the insertion the AR instance will contain an updated primary key. In the above example, the `id` property will reflect the primary key value of the newly inserted post, even though we never change it explicitly.

If a column is defined with some static default value (e.g. a string, a number) in the table schema, the corresponding property in the AR instance will automatically has such

a value after the instance is created. One way to change this default value is by explicitly declaring the property in the AR class:

```php
class Post extends CActiveRecord
{
    public $title='please enter a title';
    ......
}

$post=new Post;
echo $post->title;  // this would display: please enter a title
```

Starting from version 1.0.2, an attribute can be assigned a value of CDbExpression type before the record is saved (either insertion or updating) to the database. For example, in order to save a timestamp returned by the MySQL NOW() function, we can use the following code:

```php
$post=new Post;
$post->create_time=new CDbExpression('NOW()');
// $post->create_time='NOW()'; will not work because
// 'NOW()' will be treated as a string
$post->save();
```

> **Tip:** While AR allows us to perform database operations without writing cumbersom SQL statements, we often want to know what SQL statements are executed by AR underneath. This can be achieved by turning on the logging feature of Yii. For example, we can turn on CWebLogRoute in the application configuration, and we will see the executed SQL statements being displayed at the end of each Web page. Since version 1.0.5, we can set CDbConnection::enableParamLogging to be true in the application configuration so that the parameter values bound to the SQL statements are also logged.

4.3.4 Reading Record

To read data in a database table, we call one of the `find` methods as follows.

```php
// find the first row satisfying the specified condition
$post=Post::model()->find($condition,$params);
// find the row with the specified primary key
$post=Post::model()->findByPk($postID,$condition,$params);
// find the row with the specified attribute values
```

```
$post=Post::model()->findByAttributes($attributes,$condition,$params);
// find the first row using the specified SQL statement
$post=Post::model()->findBySql($sql,$params);
```

In the above, we call the `find` method with `Post::model()`. Remember that the static method `model()` is required for every AR class. The method returns an AR instance that is used to access class-level methods (something similar to static class methods) in an object context.

If the `find` method finds a row satisfying the query conditions, it will return a `Post` instance whose properties contain the corresponding column values of the table row. We can then read the loaded values like we do with normal object properties, for example, `echo $post->title;`.

The `find` method will return null if nothing can be found in the database with the given query condition.

When calling `find`, we use `$condition` and `$params` to specify query conditions. Here `$condition` can be string representing the `WHERE` clause in a SQL statement, and `$params` is an array of parameters whose values should be bound to the placeholders in `$condition`. For example,

```
// find the row with postID=10
$post=Post::model()->find('postID=:postID', array(':postID'=>10));
```

> **Note:** In the above, we may need to escape the reference to the `postID` column for certain DBMS. For example, if we are using PostgreSQL, we would have to write the condition as `"postID"=:postID`, because PostgreSQL by default will treat column names as case-insensitive.

We can also use `$condition` to specify more complex query conditions. Instead of a string, we let `$condition` be a CDbCriteria instance, which allows us to specify conditions other than the `WHERE` clause. For example,

```
$criteria=new CDbCriteria;
$criteria->select='title';  // only select the 'title' column
$criteria->condition='postID=:postID';
$criteria->params=array(':postID'=>10);
$post=Post::model()->find($criteria); // $params is not needed
```

Note, when using CDbCriteria as query condition, the `$params` parameter is no longer needed since it can be specified in CDbCriteria, as shown above.

An alternative way to CDbCriteria is passing an array to the `find` method. The array keys and values correspond to the criteria's property name and value, respectively. The above example can be rewritten as follows,

```php
$post=Post::model()->find(array(
    'select'=>'title',
    'condition'=>'postID=:postID',
    'params'=>array(':postID'=>10),
));
```

> **Info:** When a query condition is about matching some columns with the specified values, we can use findByAttributes(). We let the `$attributes` parameters be an array of the values indexed by the column names. In some frameworks, this task can be achieved by calling methods like `findByNameAndTitle`. Although this approach looks attractive, it often causes confusion, conflict and issues like case-sensitivity of column names.

When multiple rows of data matching the specified query condition, we can bring them in all together using the following `findAll` methods, each of which has its counterpart `find` method, as we already described.

```php
// find all rows satisfying the specified condition
$posts=Post::model()->findAll($condition,$params);
// find all rows with the specified primary keys
$posts=Post::model()->findAllByPk($postIDs,$condition,$params);
// find all rows with the specified attribute values
$posts=Post::model()->findAllByAttributes($attributes,$condition,$params);
// find all rows using the specified SQL statement
$posts=Post::model()->findAllBySql($sql,$params);
```

If nothing matches the query condition, `findAll` would return an empty array. This is different from `find` who would return null if nothing is found.

Besides the `find` and `findAll` methods described above, the following methods are also provided for convenience:

```php
// get the number of rows satisfying the specified condition
$n=Post::model()->count($condition,$params);
```

```
// get the number of rows using the specified SQL statement
$n=Post::model()->countBySql($sql,$params);
// check if there is at least a row satisfying the specified condition
$exists=Post::model()->exists($condition,$params);
```

4.3.5   Updating Record

After an AR instance is populated with column values, we can change them and save them
back to the database table.

```
$post=Post::model()->findByPk(10);
$post->title='new post title';
$post->save(); // save the change to database
```

As we can see, we use the same save() method to perform insertion and updating opera-
tions. If an AR instance is created using the new operator, calling save() would insert a
new row into the database table; if the AR instance is the result of some find or findAll
method call, calling save() would update the existing row in the table. In fact, we can use
CActiveRecord::isNewRecord to tell if an AR instance is new or not.

It is also possible to update one or several rows in a database table without loading them
first. AR provides the following convenient class-level methods for this purpose:

```
// update the rows matching the specified condition
Post::model()->updateAll($attributes,$condition,$params);
// update the rows matching the specified condition and primary key(s)
Post::model()->updateByPk($pk,$attributes,$condition,$params);
// update counter columns in the rows satisfying the specified conditions
Post::model()->updateCounters($counters,$condition,$params);
```

In the above, $attributes is an array of column values indexed by column names; $counters
is an array of incremental values indexed by column names; and $condition and $params
are as described in the previous subsection.

4.3.6   Deleting Record

We can also delete a row of data if an AR instance has been populated with this row.

```
$post=Post::model()->findByPk(10); // assuming there is a post whose ID is 10
$post->delete(); // delete the row from the database table
```

Note, after deletion, the AR instance remains unchanged, but the corresponding row in the database table is already gone.

The following class-level methods are provided to delete rows without the need of loading them first:

```
// delete the rows matching the specified condition
Post::model()->deleteAll($condition,$params);
// delete the rows matching the specified condition and primary key(s)
Post::model()->deleteByPk($pk,$condition,$params);
```

4.3.7 Data Validation

When inserting or updating a row, we often need to check if the column values comply to certain rules. This is especially important if the column values are provided by end users. In general, we should never trust anything coming from the client side.

AR performs data validation automatically when save() is being invoked. The validation is based on the rules specified by in the rules() method of the AR class. For more details about how to specify validation rules, refer to the Declaring Validation Rules section. Below is the typical workflow needed by saving a record:

```
if($post->save())
{
    // data is valid and is successfully inserted/updated
}
else
{
    // data is invalid. call getErrors() to retrieve error messages
}
```

When the data for inserting or updating is submitted by end users in an HTML form, we need to assign them to the corresponding AR properties. We can do so like the following:

```
$post->title=$_POST['title'];
$post->content=$_POST['content'];
$post->save();
```

If there are many columns, we would see a long list of such assignments. This can be alleviated by making use of the attributes property as shown below. More details can be found in the Securing Attribute Assignments section and the Creating Action section.

```
// assume $_POST['Post'] is an array of column values indexed by column names
$post->attributes=$_POST['Post'];
$post->save();
```

4.3.8 Comparing Records

Like table rows, AR instances are uniquely identified by their primary key values. Therefore, to compare two AR instances, we merely need to compare their primary key values, assuming they belong to the same AR class. A simpler way is to call CActiveRecord::equals(), however.

> **Info:** Unlike AR implementation in other frameworks, Yii supports composite primary keys in its AR. A composite primary key consists of two or more columns. Correspondingly, the primary key value is represented as an array in Yii. The primaryKey property gives the primary key value of an AR instance.

4.3.9 Customization

CActiveRecord provides a few placeholder methods that can be overridden in child classes to customize its workflow.

- beforeValidate and afterValidate: these are invoked before and after validation is performed.

- beforeSave and afterSave: these are invoked before and after saving an AR instance.

- beforeDelete and afterDelete: these are invoked before and after an AR instance is deleted.

- afterConstruct: this is invoked for every AR instance created using the `new` operator.

- beforeFind: this is invoked before an AR finder is used to perform a query (e.g. `find()`, `findAll()`). This has been available since version 1.0.9.

- afterFind: this is invoked after every AR instance created as a result of query.

4.3.10 Using Transaction with AR

Every AR instance contains a property named dbConnection which is a CDbConnection instance. We thus can use the transaction feature provided by Yii DAO if it is desired when working with AR:

```
$model=Post::model();
$transaction=$model->dbConnection->beginTransaction();
try
{
    // find and save are two steps which may be intervened by another request
    // we therefore use a transaction to ensure consistency and integrity
    $post=$model->findByPk(10);
    $post->title='new post title';
    $post->save();
    $transaction->commit();
}
catch(Exception $e)
{
    $transaction->rollBack();
}
```

4.3.11 Named Scopes

> **Note:** The support for named scopes has been available since version 1.0.5. The
> original idea of named scopes came from Ruby on Rails.

A *named scope* represents a *named* query criteria that can be combined with other named
scopes and applied to an active record query.

Named scopes are mainly declared in the CActiveRecord::scopes() method as name-criteria
pairs. The following code declares two named scopes, `published` and `recently`, in the `Post`
model class:

```
class Post extends CActiveRecord
{
    ......
    public function scopes()
    {
        return array(
            'published'=>array(
                'condition'=>'status=1',
            ),
            'recently'=>array(
                'order'=>'create_time DESC',
                'limit'=>5,
            ),
        );
    }
}
```

Each named scope is declared as an array which can be used to initialize a CDbCriteria instance. For example, the `recently` named scope specifies that the `order` property to be `create_time DESC` and the `limit` property to be 5, which translates to a query criteria that should bring back the most recent 5 posts.

Named scopes are mostly used as modifiers to the `find` method calls. Several named scopes may be chained together and result in a more restrictive query result set. For example, to find the recently published posts, we can use the following code:

```
$posts=Post::model()->published()->recently()->findAll();
```

In general, named scopes must appear to the left of a `find` method call. Each of them provides a query criteria, which is combined with other criterias, including the one passed to the `find` method call. The net effect is like adding a list of filters to a query.

Starting from version 1.0.6, named scopes can also be used with `update` and `delete` methods. For example, the following code would delete all recently published posts:

```
Post::model()->published()->recently()->delete();
```

> **Note:** Named scopes can only be used with class-level methods. That is, the method must be called using `ClassName::model()`.

Parameterized Named Scopes

Named scopes can be parameterized. For example, we may want to customize the number of posts specified by the `recently` named scope. To do so, instead of declaring the named scope in the CActiveRecord::scopes method, we need to define a new method whose name is the same as the scope name:

```php
public function recently($limit=5)
{
    $this->getDbCriteria()->mergeWith(array(
        'order'=>'create_time DESC',
        'limit'=>$limit,
    ));
    return $this;
}
```

Then, we can use the following statement to retrieve the 3 recently published posts:

```
$posts=Post::model()->published()->recently(3)->findAll();
```

If we do not supply the parameter 3 in the above, we would retrieve the 5 recently published posts by default.

Default Named Scope

A model class can have a default named scope that would be applied for all queries (including relational ones) about the model. For example, a website supporting multiple languages may only want to display contents that are in the language the current user specifies. Because there may be many queries about the site contents, we can define a default named scope to solve this problem. To do so, we override the CActiveRecord::defaultScope method as follows,

```
class Content extends CActiveRecord
{
    public function defaultScope()
    {
        return array(
            'condition'=>"language='".Yii::app()->language."'",
        );
    }
}
```

Now, if the following method call will automatically use the query criteria as defined above:

```
$contents=Content::model()->findAll();
```

Note that default named scope only applies to SELECT queries. It is ignored for INSERT, UPDATE and DELETE queries.

4.4   Relational Active Record

We have already seen how to use Active Record (AR) to select data from a single database table. In this section, we describe how to use AR to join several related database tables and bring back the joint data set.

In order to use relational AR, it is recommended that primary-foreign key constraints are declared for tables that need to be joined. The constraints will help to keep the consistency and integrity of the relational data.

For simplicity, we will use the database schema shown in the following entity-relationship (ER) diagram to illustrate examples in this section.
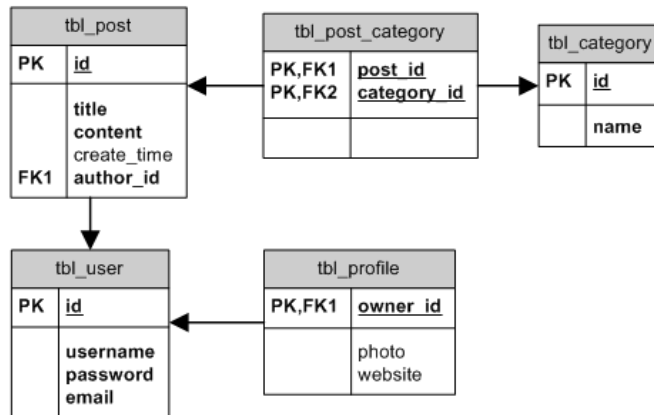
Figure 4.1: ER Diagram

Info: Support for foreign key constraints varies in different DBMS. SQLite lt; 3.6.19 does not support foreign key constraints, but you can still declare the constraints when creating tables.

4.4.1   Declaring Relationship

Before we use AR to perform relational query, we need to let AR know how one AR class is related with another.

Relationship between two AR classes is directly associated with the relationship between the database tables represented by the AR classes. From database point of view, a relationship between two tables A and B has three types: one-to-many (e.g. `tbl_user` and `tbl_post`), one-to-one (e.g. `tbl_user` and `tbl_profile`) and many-to-many (e.g. `tbl_category` and `tbl_post`). In AR, there are four types of relationships:

- `BELONGS_TO`: if the relationship between table A and B is one-to-many, then B belongs to A (e.g. `Post` belongs to `User`);

- `HAS_MANY`: if the relationship between table A and B is one-to-many, then A has many B (e.g. `User` has many `Post`);

- **HAS_ONE**: this is special case of **HAS_MANY** where A has at most one B (e.g. **User** has at most one **Profile**);

- **MANY_MANY**: this corresponds to the many-to-many relationship in database. An associative table is needed to break a many-to-many relationship into one-to-many relationships, as most DBMS do not support many-to-many relationship directly. In our example database schema, the **tbl_post_category** serves for this purpose. In AR terminology, we can explain **MANY_MANY** as the combination of **BELONGS_TO** and **HAS_MANY**. For example, **Post** belongs to many **Category** and **Category** has many **Post**.

Declaring relationship in AR involves overriding the relations() method of CActiveRecord. The method returns an array of relationship configurations. Each array element represents a single relationship with the following format:

```
'VarName'=>array('RelationType', 'ClassName', 'ForeignKey', ...additional options)
```

where **VarName** is the name of the relationship; **RelationType** specifies the type of the relationship, which can be one of the four constants: **self::BELONGS_TO**, **self::HAS_ONE**, **self::HAS_MANY** and **self::MANY_MANY**; **ClassName** is the name of the AR class related to this AR class; and **ForeignKey** specifies the foreign key(s) involved in the relationship. Additional options can be specified at the end for each relationship (to be described later).

The following code shows how we declare the relationships for the **User** and **Post** classes.

```php
class Post extends CActiveRecord
{
    ......

    public function relations()
    {
        return array(
            'author'=>array(self::BELONGS_TO, 'User', 'author_id'),
            'categories'=>array(self::MANY_MANY, 'Category',
                'tbl_post_category(post_id, category_id)'),
        );
    }
}

class User extends CActiveRecord
{
    ......

    public function relations()
```

```
{
    return array(
        'posts'=>array(self::HAS_MANY, 'Post', 'author_id'),
        'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
    );
}
}
```

> **Info:** A foreign key may be composite, consisting of two or more columns. In this case, we should concatenate the names of the foreign key columns and separate them with space or comma. For `MANY_MANY` relationship type, the associative table name must also be specified in the foreign key. For example, the `categories` relationship in `Post` is specified with the foreign key `tbl_post_category(post_id, category_id)`.

The declaration of relationships in an AR class implicitly adds a property to the class for each relationship. After a relational query is performed, the corresponding property will be populated with the related AR instance(s). For example, if `$author` represents a `User` AR instance, we can use `$author->posts` to access its related `Post` instances.

4.4.2   Performing Relational Query

The simplest way of performing relational query is by reading a relational property of an AR instance. If the property is not accessed previously, a relational query will be initiated, which joins the two related tables and filters with the primary key of the current AR instance. The query result will be saved to the property as instance(s) of the related AR class. This is known as the *lazy loading* approach, i.e., the relational query is performed only when the related objects are initially accessed. The example below shows how to use this approach:

```
// retrieve the post whose ID is 10
$post=Post::model()->findByPk(10);
// retrieve the post's author: a relational query will be performed here
$author=$post->author;
```

> **Info:** If there is no related instance for a relationship, the corresponding property could be either null or an empty array. For `BELONGS_TO` and `HAS_ONE` relationships, the result is null; for `HAS_MANY` and `MANY_MANY`, it is an empty array. Note that the `HAS_MANY` and `MANY_MANY` relationships return arrays of objects, you will need to loop through the results before trying to access any properties. Otherwise, you may receive "Trying to get property of non-object" errors.

The lazy loading approach is very convenient to use, but it is not efficient in some scenarios. For example, if we want to access the author information for `N` posts, using the lazy approach would involve executing `N` join queries. We should resort to the so-called *eager loading* approach under this circumstance.

The eager loading approach retrieves the related AR instances together with the main AR instance(s). This is accomplished by using the with() method together with one of the find or findAll methods in AR. For example,

```
$posts=Post::model()->with('author')->findAll();
```

The above code will return an array of `Post` instances. Unlike the lazy approach, the `author` property in each `Post` instance is already populated with the related `User` instance before we access the property. Instead of executing a join query for each post, the eager loading approach brings back all posts together with their authors in a single join query!

We can specify multiple relationship names in the with() method and the eager loading approach will bring them back all in one shot. For example, the following code will bring back posts together with their authors and categories:

```
$posts=Post::model()->with('author','categories')->findAll();
```

We can also do nested eager loading. Instead of a list of relationship names, we pass in a hierarchical representation of relationship names to the with() method, like the following,

```
$posts=Post::model()->with(
    'author.profile',
    'author.posts',
    'categories')->findAll();
```

The above example will bring back all posts together with their author and categories. It will also bring back each author's profile and posts.

Starting from version 1.1.0, eager loading may also be executed by specifying the CDbCriteria::with property, like the following:

```
$criteria=new CDbCriteria;
$criteria->with=array(
    'author.profile',
    'author.posts',
    'categories',
);
$posts=Post::model()->findAll($criteria);
```

or

```php
$posts=Post::model()->findAll(array(
    'with'=>array(
        'author.profile',
        'author.posts',
        'categories',
    )
);
```

4.4.3 Relational Query Options

We mentioned that additional options can be specified in relationship declaration. These options, specified as name-value pairs, are used to customize the relational query. They are summarized as below.

- `select`: a list of columns to be selected for the related AR class. It defaults to '*', meaning all columns. Column names referenced in this option should be disambiguated.

- `condition`: the `WHERE` clause. It defaults to empty. Column names referenced in this option should be disambiguated.

- `params`: the parameters to be bound to the generated SQL statement. This should be given as an array of name-value pairs. This option has been available since version 1.0.3.

- `on`: the `ON` clause. The condition specified here will be appended to the joining condition using the `AND` operator. Column names referenced in this option should be disambiguated. This option does not apply to `MANY_MANY` relations. This option has been available since version 1.0.2.

- `order`: the `ORDER BY` clause. It defaults to empty. Column names referenced in this option should be disambiguated.

- `with`: a list of child related objects that should be loaded together with this object. Be aware that using this option inappropriately may form an infinite relation loop.

- `joinType`: type of join for this relationship. It defaults to `LEFT OUTER JOIN`.

- `alias`: the alias for the table associated with this relationship. This option has been available since version 1.0.1. It defaults to null, meaning the table alias is the same as the relation name.

- `together`: whether the table associated with this relationship should be forced to join together with the primary table and other tables. This option is only meaningful for `HAS_MANY` and `MANY_MANY` relations. If this option is set false, the table associated with the `HAS_MANY` or `MANY_MANY` relation will be joined with the primary table in a separate SQL query, which may improve the overall query performance since less duplicated data is returned. If this option is set true, the associated table will always be joined with the primary table in a single SQL query, even if the primary table is paginated. If this option is not set, the associated table will be joined with the primary table in a single SQL query only when the primary table is not paginated. For more details, see the section "Relational Query Performance". This option has been available since version 1.0.3.

- `join`: the extra `JOIN` clause. It defaults to empty. This option has been available since version 1.1.3.

- `group`: the `GROUP BY` clause. It defaults to empty. Column names referenced in this option should be disambiguated.

- `having`: the `HAVING` clause. It defaults to empty. Column names referenced in this option should be disambiguated. Note: option has been available since version 1.0.1.

- `index`: the name of the column whose values should be used as keys of the array that stores related objects. Without setting this option, an related object array would use zero-based integer index. This option can only be set for `HAS_MANY` and `MANY_MANY` relations. This option has been available since version 1.0.7.

In addition, the following options are available for certain relationships during lazy loading:

- `limit`: limit of the rows to be selected. This option does NOT apply to `BELONGS_TO` relation.

- `offset`: offset of the rows to be selected. This option does NOT apply to `BELONGS_TO` relation.

Below we modify the `posts` relationship declaration in the `User` by including some of the above options:

```
class User extends CActiveRecord
{
    public function relations()
    {
        return array(
```

```
        'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                       'order'=>'posts.create_time DESC',
                       'with'=>'categories'),
        'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
    );
  }
}
```

Now if we access `$author->posts`, we would obtain the author's posts sorted according to their creation time in descending order. Each post instance also has its categories loaded.

4.4.4 Disambiguating Column Names

When a column name appears in two or more tables being joined together, it needs to be disambiguated. This is done by prefixing the column name with its table's alias name.

In relational AR query, the alias name for the primary table is fixed as `t`, while the alias name for a relational table is the same as the corresponding relation name by default. For example, in the following statement, the alias name for `Post` and `Comment` is `t` and `comments`, respectively:

```
$posts=Post::model()->with('comments')->findAll();
```

Now assume both `Post` and `Comment` have a column called `create_time` indicating the creation time of a post or comment, and we would like to fetch posts together with their comments by ordering first the posts' creation time and then the comments' creation time. We need to disambiguate the `create_time` column like the following:

```
$posts=Post::model()->with('comments')->findAll(array(
    'order'=>'t.create_time, comments.create_time'
));
```

> **Note:** the behavior of column disambiguation has been changed since version 1.1.0. Previously in version 1.0.x, by default Yii would automatically generate a table alias for each relational table, and we had to use the prefix `??.` to refer to this automatically generated alias. Also, in version 1.0.x, the alias name of the primary table is the table name itself.

4.4.5 Dynamic Relational Query Options

Starting from version 1.0.2, we can use dynamic relational query options in both with() and the `with` option. The dynamic options will overwrite existing options as specified in

the relations() method. For example, with the above `User` model, if we want to use eager loading approach to bring back posts belonging to an author in *ascending order* (the `order` option in the relation specification is descending order), we can do the following:

```
User::model()->with(array(
    'posts'=>array('order'=>'posts.create_time ASC'),
    'profile',
))->findAll();
```

Starting from version 1.0.5, dynamic query options can also be used when using the lazy loading approach to perform relational query. To do so, we should call a method whose name is the same as the relation name and pass the dynamic query options as the method parameter. For example, the following code returns a user's posts whose `status` is 1:

```
$user=User::model()->findByPk(1);
$posts=$user->posts(array('condition'=>'status=1'));
```

4.4.6 Relational Query Performance

As we described above, the eager loading approach is mainly used in the scenario when we need to access many related objects. It generates a big complex SQL statement by joining all needed tables. A big SQL statement is preferrable in many cases since it simplifies filtering based on a column in a related table. It may not be efficient in some cases, however.

Consider an example where we need to find the latest posts together with their comments. Assuming each post has 10 comments, using a single big SQL statement, we will bring back a lot of redundant post data since each post will be repeated for every comment it has. Now let's try another approach: we first query for the latest posts, and then query for their comments. In this new approach, we need to execute two SQL statements. The benefit is that there is no redundancy in the query results.

So which approach is more efficient? There is no absolute answer. Executing a single big SQL statement may be more efficient because it causes less overhead in DBMS for yparsing and executing the SQL statements. On the other hand, using the single SQL statement, we end up with more redundant data and thus need more time to read and process them.

For this reason, Yii provides the `together` query option so that we choose between the two approaches as needed. By default, Yii adopts the first approach, i.e., generating a single SQL statement to perform eager loading. We can set the `together` option to be false in

the relation declarations so that some of tables are joined in separate SQL statements. For example, in order to use the second approach to query for the latest posts with their comments, we can declare the `comments` relation in `Post` class as follows,

```
public function relations()
{
    return array(
        'comments' => array(self::HAS_MANY, 'Comment', 'post_id', 'together'=>false),
    );
}
```

We can also dynamically set this option when we perform the eager loading:

```
$posts = Post::model()->with(array('comments'=>array('together'=>false)))->findAll();
```

> **Note:** In version 1.0.x, the default behavior is that Yii will generate and execute N+1 SQL statements if there are N `HAS_MANY` or `MANY_MANY` relations. Each `HAS_MANY` or `MANY_MANY` relation has its own SQL statement. By calling the `together()` method after `with()`, we can enforce only a single SQL statement is generated and executed. For example,
>
> ```
> $posts=Post::model()->with(
>   'author.profile',
>   'author.posts',
>   'categories')->together()->findAll();
> ```

4.4.7 Statistical Query

> **Note:** Statistical query has been supported since version 1.0.4.

Besides the relational query described above, Yii also supports the so-called statistical query (or aggregational query). It refers to retrieving the aggregational information about the related objects, such as the number of comments for each post, the average rating for each product, etc. Statistical query can only be performed for objects related in `HAS_MANY` (e.g. a post has many comments) or `MANY_MANY` (e.g. a post belongs to many categories and a category has many posts).

Performing statistical query is very similar to performing relation query as we described before. We first need to declare the statistical query in the relations() method of CActiveRecord like we do with relational query.

```
class Post extends CActiveRecord
{
    public function relations()
    {
        return array(
            'commentCount'=>array(self::STAT, 'Comment', 'post_id'),
            'categoryCount'=>array(self::STAT, 'Category', 'post_category(post_id, category_
id)'),
        );
    }
}
```

In the above, we declare two statistical queries: `commentCount` calculates the number of comments belonging to a post, and `categoryCount` calculates the number of categories that a post belongs to. Note that the relationship between `Post` and `Comment` is `HAS_MANY`, while the relationship between `Post` and `Category` is `MANY_MANY` (with the joining table `post_category`). As we can see, the declaration is very similar to those relations we described in earlier subsections. The only difference is that the relation type is `STAT` here.

With the above declaration, we can retrieve the number of comments for a post using the expression `$post->commentCount`. When we access this property for the first time, a SQL statement will be executed implicitly to retrieve the corresponding result. As we already know, this is the so-called *lazy loading* approach. We can also use the *eager loading* approach if we need to determine the comment count for multiple posts:

```
$posts=Post::model()->with('commentCount', 'categoryCount')->findAll();
```

The above statement will execute three SQLs to bring back all posts together with their comment counts and category counts. Using the lazy loading approach, we would end up with `2*N+1` SQL queries if there are `N` posts.

By default, a statistical query will calculate the `COUNT` expression (and thus the comment count and category count in the above example). We can customize it by specifying additional options when we declare it in relations(). The available options are summarized as below.

- `select`: the statistical expression. Defaults to `COUNT(*)`, meaning the count of child objects.

- `defaultValue`: the value to be assigned to those records that do not receive a statistical query result. For example, if a post does not have any comments, its `commentCount` would receive this value. The default value for this option is 0.

- condition: the `WHERE` clause. It defaults to empty.

- params: the parameters to be bound to the generated SQL statement. This should be given as an array of name-value pairs.

- order: the `ORDER BY` clause. It defaults to empty.

- group: the `GROUP BY` clause. It defaults to empty.

- having: the `HAVING` clause. It defaults to empty.

4.4.8 Relational Query with Named Scopes

> **Note:** The support for named scopes has been available since version 1.0.5.

Relational query can also be performed in combination with named scopes. It comes in two forms. In the first form, named scopes are applied to the main model. In the second form, named scopes are applied to the related models.

The following code shows how to apply named scopes to the main model.

```
$posts=Post::model()->published()->recently()->with('comments')->findAll();
```

This is very similar to non-relational queries. The only difference is that we have the `with()` call after the named-scope chain. This query would bring back recently published posts together with their comments.

And the following code shows how to apply named scopes to the related models.

```
$posts=Post::model()->with('comments:recently:approved')->findAll();
```

The above query will bring back all posts together with their approved comments. Note that `comments` refers to the relation name, while `recently` and `approved` refer to two named scopes declared in the `Comment` model class. The relation name and the named scopes should be separated by colons.

Named scopes can also be specified in the `with` option of the relational rules declared in CActiveRecord::relations(). In the following example, if we access `$user->posts`, it would bring back all *approved* comments of the posts.

```php
class User extends CActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                'with'=>'comments:approved'),
        );
    }
}
```

> **Note:** Named scopes applied to related models must be specified in CActiveRecord::scopes. As a result, they cannot be parameterized.

CHAPTER 5

Caching

5.1 Caching

Caching is a cheap and effective way to improve the performance of a Web application. By storing relatively static data in cache and serving it from cache when requested, we save the time needed to generate the data.

Using cache in Yii mainly involves configuring and accessing a cache application component. The following application configuration specifies a cache component that uses memcache with two cache servers.

```
array(
    ......
    'components'=>array(
        ......
        'cache'=>array(
            'class'=>'system.caching.CMemCache',
            'servers'=>array(
                array('host'=>'server1', 'port'=>11211, 'weight'=>60),
                array('host'=>'server2', 'port'=>11211, 'weight'=>40),
            ),
        ),
    ),
);
```

When the application is running, the cache component can be accessed via `Yii::app()->cache`.

Yii provides various cache components that can store cached data in different media. For example, the CMemCache component encapsulates the PHP memcache extension and uses memory as the medium of cache storage; the CApcCache component encapsulates the PHP APC extension; and the CDbCache component stores cached data in database. The following is a summary of the available cache components:

- CMemCache: uses PHP memcache extension.

- **CApcCache**: uses PHP APC extension.

- **CXCache**: uses PHP XCache extension. Note, this has been available since version 1.0.1.

- **CEAcceleratorCache**: uses PHP EAccelerator extension.

- **CDbCache**: uses a database table to store cached data. By default, it will create and use a SQLite3 database under the runtime directory. You can explicitly specify a database for it to use by setting its connectionID property.

- **CZendDataCache**: uses Zend Data Cache as the underlying caching medium. Note, this has been available since version 1.0.4.

- **CFileCache**: uses files to store cached data. This is particular suitable to cache large chunk of data (such as pages). Note that this has been available since version 1.0.6.

- **CDummyCache**: presents dummy cache that does no caching at all. The purpose of this component is to simplify the code that needs to check the availability of cache. For example, during development or if the server doesn't have actual cache support, we can use this cache component. When an actual cache support is enabled, we can switch to use the corresponding cache component. In both cases, we can use the same code `Yii::app()->cache->get($key)` to attempt retrieving a piece of data without worrying that `Yii::app()->cache` might be `null`. This component has been available since version 1.0.5.

> **Tip:** Because all these cache components extend from the same base class CCache, one can switch to use a different type of cache without modifying the code that uses cache.

Caching can be used at different levels. At the lowest level, we use cache to store a single piece of data, such as a variable, and we call this *data caching*. At the next level, we store in cache a page fragment which is generated by a portion of a view script. And at the highest level, we store a whole page in cache and serve it from cache as needed.

In the next few subsections, we elaborate how to use cache at these levels.

> **Note:** By definition, cache is a volatile storage medium. It does not ensure the existence of the cached data even if it does not expire. Therefore, do not use cache as a persistent storage (e.g. do not use cache to store session data).

5.2   Data Caching

Data caching is about storing some PHP variable in cache and retrieving it later from cache. For this purpose, the cache component base class CCache provides two methods that are used most of the time: set() and get().

To store a variable $value in cache, we choose a unique ID and call set() to store it:

```php
Yii::app()->cache->set($id, $value);
```

The cached data will remain in the cache forever unless it is removed because of some caching policy (e.g. caching space is full and the oldest data are removed). To change this behavior, we can also supply an expiration parameter when calling set() so that the data will be removed from the cache after a certain period of time:

```php
// keep the value in cache for at most 30 seconds
Yii::app()->cache->set($id, $value, 30);
```

Later when we need to access this variable (in either the same or a different Web request), we call get() with the ID to retrieve it from cache. If the value returned is false, it means the value is not available in cache and we should regenerate it.

```php
$value=Yii::app()->cache->get($id);
if($value===false)
{
    // regenerate $value because it is not found in cache
    // and save it in cache for later use:
    // Yii::app()->cache->set($id,$value);
}
```

When choosing the ID for a variable to be cached, make sure the ID is unique among all other variables that may be cached in the application. It is NOT required that the ID is unique across applications because the cache component is intelligent enough to differentiate IDs for different applications.

Some cache storages, such as MemCache, APC, support retrieving multiple cached values in a batch mode, which may reduce the overhead involved in retrieving cached data. Starting from version 1.0.8, a new method named mget() is provided to exploit this feature. In case the underlying cache storage does not support this feature, mget() will still simulate it.

To remove a cached value from cache, call delete(); and to remove everything from cache, call flush(). Be very careful when calling flush() because it also removes cached data that are from other applications.

Tip: Because CCache implements `ArrayAccess`, a cache component can be used liked an array. The followings are some examples:

```
$cache=Yii::app()->cache;
$cache['var1']=$value1;  // equivalent to: $cache->set('var1',$value1);
$value2=$cache['var2'];  // equivalent to: $value2=$cache->get('var2');
```

5.2.1   Cache Dependency

Besides expiration setting, cached data may also be invalidated according to some dependency changes. For example, if we are caching the content of some file and the file is changed, we should invalidate the cached copy and read the latest content from the file instead of the cache.

We represent a dependency as an instance of CCacheDependency or its child class. We pass the dependency instance along with the data to be cached when calling set().

```
// the value will expire in 30 seconds
// it may also be invalidated earlier if the dependent file is changed
Yii::app()->cache->set($id, $value, 30, new CFileCacheDependency('FileName'));
```

Now if we retrieve `$value` from cache by calling get(), the dependency will be evaluated and if it is changed, we will get a false value, indicating the data needs to be regenerated.

Below is a summary of the available cache dependencies:

- CFileCacheDependency: the dependency is changed if the file's last modification time is changed.

- CDirectoryCacheDependency: the dependency is changed if any of the files under the directory and its subdirectories is changed.

- CDbCacheDependency: the dependency is changed if the query result of the specified SQL statement is changed.

- CGlobalStateCacheDependency: the dependency is changed if the value of the specified global state is changed. A global state is a variable that is persistent across multiple requests and multiple sessions in an application. It is defined via CApplication::setGlobalState().

- CChainedCacheDependency: the dependency is changed if any of the dependencies on the chain is changed.

- CExpressionDependency: the dependency is changed if the result of the specified PHP expression is changed. This class has been available since version 1.0.4.

5.3   Fragment Caching

Fragment caching refers to caching a fragment of a page. For example, if a page displays a summary of yearly sale in a table, we can store this table in cache to eliminate the time needed to generate it for each request.

To use fragment caching, we call CController::beginCache() and CController::endCache() in a controller's view script. The two methods mark the beginning and the end of the page content that should be cached, respectively. Like data caching, we need an ID to identify the fragment being cached.

```php
...other HTML content...
<?php if($this->beginCache($id)) { ?>
...content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

In the above, if beginCache() returns false, the cached content will be automatically inserted at the place; otherwise, the content inside the if-statement will be executed and be cached when endCache() is invoked.

5.3.1   Caching Options

When calling beginCache(), we can supply an array as the second parameter consisting of caching options to customize the fragment caching. As a matter of fact, the beginCache() and endCache() methods are a convenient wrapper of the COutputCache widget. Therefore, the caching options can be initial values for any properties of COutputCache.

Duration

Perhaps the most commonly option is duration which specifies how long the content can remain valid in cache. It is similar to the expiration parameter of CCache::set(). The following code caches the content fragment for at most one hour:

```
...other HTML content...
```

```php
<?php if($this->beginCache($id, array('duration'=>3600))) { ?>
...content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

If we do not set the duration, it would default to 60, meaning the cached content will be invalidated after 60 seconds.

Dependency

Like data caching, content fragment being cached can also have dependencies. For example, the content of a post being displayed depends on whether or not the post is modified.

To specify a dependency, we set the dependency option, which can be either an object implementing ICacheDependency or a configuration array that can be used to generate the dependency object. The following code specifies the fragment content depends on the change of `lastModified` column value:

```php
...other HTML content...
<?php if($this->beginCache($id, array('dependency'=>array(
        'class'=>'system.caching.dependencies.CDbCacheDependency',
        'sql'=>'SELECT MAX(lastModified) FROM Post')))) { ?>
...content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

Variation

Content being cached may be variated according to some parameters. For example, the personal profile may look differently to different users. To cache the profile content, we would like the cached copy to be variated according to user IDs. This essentially means that we should use different IDs when calling beginCache().

Instead of asking developers to variate the IDs according to some scheme, COutputCache is built-in with such a feature. Below is a summary.

- varyByRoute: by setting this option to true, the cached content will be variated according to route. Therefore, each combination of the requested controller and action will have a separate cached content.

- varyBySession: by setting this option to true, we can make the cached content to be variated according to session IDs. Therefore, each user session may see different content and they are all served from cache.

- varyByParam: by setting this option to an array of names, we can make the cached content to be variated according to the values of the specified GET parameters. For example, if a page displays the content of a post according to the `id` GET parameter, we can specify varyByParam to be `array('id')` so that we can cache the content for each post. Without such variation, we would only be able to cache a single post.

- varyByExpression: by setting this option to a PHP expression, we can make the cached content to be variated according to the result of this PHP expression. This option has been available since version 1.0.4.

Request Types

Sometimes we want the fragment caching to be enabled only for certain types of request. For example, for a page displaying a form, we only want to cache the form when it is initially requested (via GET request). Any subsequent display (via POST request) of the form should not be cached because the form may contain user input. To do so, we can specify the requestTypes option:

```php
...other HTML content...
<?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>
...content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

5.3.2 Nested Caching

Fragment caching can be nested. That is, a cached fragment is enclosed within a bigger fragment that is also cached. For example, the comments are cached in an inner fragment cache, and they are cached together with the post content in an outer fragment cache.

```php
...other HTML content...
<?php if($this->beginCache($id1)) { ?>
...outer content to be cached...
    <?php if($this->beginCache($id2)) { ?>
    ...inner content to be cached...
    <?php $this->endCache(); } ?>
...outer content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

Different caching options can be set to the nested caches. For example, the inner cache and the outer cache in the above example can be set with different duration values. When the data cached in the outer cache is invalidated, the inner cache may still provide valid inner fragment. However, it is not true vice versa. If the outer cache contains valid data, it will always provide the cached copy, even though the content in the inner cache already expires.

5.4   Page Caching

Page caching refers to caching the content of a whole page. Page caching can occur at different places. For example, by choosing an appropriate page header, the client browser may cache the page being viewed for a limited time. The Web application itself can also store the page content in cache. In this subsection, we focus on this latter approach.

Page caching can be considered as a special case of fragment caching. Because the content of a page is often generated by applying a layout to a view, it will not work if we simply call beginCache() and endCache() in the layout. The reason is because the layout is applied within the CController::render() method AFTER the content view is evaluated.

To cache a whole page, we should skip the execution of the action generating the page content. We can use COutputCache as an action filter to accomplish this task. The following code shows how we configure the cache filter:

```php
public function filters()
{
    return array(
        array(
            'COutputCache',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}
```

The above filter configuration would make the filter to be applied to all actions in the controller. We may limit it to one or a few actions only by using the plus operator. More details can be found in filter.

> **Tip:** We can use COutputCache as a filter because it extends from CFilterWidget, which means it is both a widget and a filter. In fact, the way a widget works is very similar to a filter: a widget (filter) begins before any enclosed content (action) is evaluated, and the widget (filter) ends after the enclosed content (action) is evaluated.

5.5   Dynamic Content

When using fragment caching or page caching, we often encounter the situation where the whole portion of the output is relatively static except at one or several places. For example, a help page may display static help information with the name of the user currently logged in displayed at the top.

To solve this issue, we can variate the cache content according to the username, but this would be a big waste of our precious cache space since most content are the same except the username. We can also divide the page into several fragments and cache them individually, but this complicates our view and makes our code very complex. A better approach is to use the *dynamic content* feature provided by CController.

A dynamic content means a fragment of output that should not be cached even if it is enclosed within a fragment cache. To make the content dynamic all the time, it has to be generated every time even when the enclosing content is being served from cache. For this reason, we require that dynamic content be generated by some method or function.

We call CController::renderDynamic() to insert dynamic content at the desired place.

```php
...other HTML content...
<?php if($this->beginCache($id)) { ?>
...fragment content to be cached...
    <?php $this->renderDynamic($callback); ?>
...fragment content to be cached...
<?php $this->endCache(); } ?>
...other HTML content...
```

In the above, `$callback` refers to a valid PHP callback. It can be a string referring to the name of a method in the current controller class or a global function. It can also be an array referring to a class method. Any additional parameters to renderDynamic() will be passed to the callback. The callback should return the dynamic content instead of displaying it.

CHAPTER 6

Extending Yii

6.1 Overview

Extending Yii is a common activity during development. For example, when you write a new controller, you extend Yii by inheriting its CController class; when you write a new widget, you are extending CWidget or an existing widget class. If the extended code is designed to be reused by third-party developers, we call it an *extension*.

An extension usually serves for a single purpose. In Yii's terms, it can be classified as follows,

- application component

- behavior

- widget

- controller

- action

- filter

- console command

- validator: a validator is a component class extending CValidator.

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

- module: a module is a self-contained software unit that consists of models, views, controllers and other supporting components. In many aspects, a module resembles to an application. The main difference is that a module is inside an application. For example, we could have a module that provides user management functionalities.

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

6.2   Using Extensions

Using an extension usually involves the following three steps:

1. Download the extension from Yii's extension repository.

2. Unpack the extension under the `extensions/xyz` subdirectory of application base directory, where `xyz` is the name of the extension.

3. Import, configure and use the extension.

Each extension has a name that uniquely identifies it among all extensions. Given an extension named as `xyz`, we can always use the path alias `ext.xyz` to locate its base directory which contains all files of `xyz`.

> **Note:** The root path alias `ext` has been available since version 1.0.8. Previously, we would need to use `application.extensions` to refer to the directory containing all extensions. In the following description, we assume `ext` is defined. You will need to replace it with `application.extensions` if you are using version 1.0.7 or lower.

Different extensions have different requirements about importing, configuration and usage. In the following, we summarize common usage scenarios about extensions, according to their categorization described in the overview.

6.2.1   Zii Extensions

Before we start describing the usage of third-party extensions, we would like to introduce the Zii extension library, which is a set of extensions developed by the Yii developer team and included in every release since Yii version 1.1.0. The Zii library is hosted as a Google project called zii.

When using a Zii extension, one must refer to the corresponding class using a path alias in the form of `zii.path.to.ClassName`. Here the root alias `zii` is predefined by Yii. It refers to the root directory of the Zii library. For example, to use CGridView, we would use the following code in a view script when referring to the extension:

```
$this->widget('zii.widgets.grid.CGridView', array(
    'dataProvider'=>$dataProvider,
));
```

6.2.2 Application Component

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

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

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

6.2.3 Behavior

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

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

More often, a behavior is attached to a component using a configurative way instead of calling the `attachBehavior` method. For example, to attach a behavior to an application component, we could use the following application configuration:

```
return array(
    'components'=>array(
```

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

The above code attaches the `xyz` behavior to the `db` application component. We can do so because CApplicationComponent defines a property named `behaviors`. By setting this property with a list of behavior configurations, the component will attach the corresponding behaviors when it is being initialized.

For CController, CFormModel and CActiveRecord classes which usually need to be extended, attaching behaviors can be done by overriding their `behaviors()` method. The classes will automatically attach any behaviors declared in this method during initialization. For example,

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

6.2.4   Widget

Widgets are mainly used in views. Given a widget class `XyzClass` belonging to the `xyz` extension, we can use it in a view as follows,

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

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

...body content of the widget...

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

6.2.5 Action

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

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

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

6.2.6 Filter

Filters are also used by a controller. Their mainly pre- and post-process the user request when it is handled by an action. Given a filter class XyzClass belonging to the xyz extension, we can use it by overriding the CController::filters method in our controller class:

```php
class TestController extends CController
{
    public function filters()
    {
        return array(
```

```
        array(
            'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // other filters
    );
    }
}
```

In the above, we can use plus and minus operators in the first array element to apply the filter to limited actions only. For more details, please refer to the documentation of CController.

6.2.7 Controller

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

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

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

6.2.8 Validator

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

```
class MyModel extends CActiveRecord // or CFormModel
{
    public function rules()
    {
```

```
        return array(
            array(
                'attr1, attr2',
                'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // other validation rules
        );
    }
}
```

6.2.9   Console Command

A console command extension usually enhances the `yiic` tool with an additional command. Given a console command `XyzClass` belonging to the `xyz` extension, we can use it by configuring the configuration for the console application:

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

Then, we can use the `yiic` tool is equipped with an additional command `xyz`.

> **Note:** A console application usually uses a configuration file that is different from the one used by a Web application. If an application is created using `yiic webapp` command, then the configuration file for the console application `protected/yiic` is `protected/config/console.php`, while the configuration file for the Web application is `protected/config/main.php`.

6.2.10   Module

Please refer to the section about modules on how to use a module.

6.2.11   Generic Component

To use a generic component, we first need to include its class file by using

```
Yii::import('ext.xyz.XyzClass');
```

Then, we can either create an instance of the class, configure its properties, and call its methods. We may also extend it to create new child classes.

6.3   Creating Extensions

Because an extension is meant to be used by third-party developers, it takes some additional efforts to create it. The followings are some general guidelines:

- An extension should be self-contained. That is, its external dependency should be minimal. It would be a headache for its users if an extension requires installation of additional packages, classes or resource files.

- Files belonging to an extension should be organized under the same directory whose name is the extension name

- Classes in an extension should be prefixed with some letter(s) to avoid naming conflict with classes in other extensions.

- An extension should come with detailed installation and API documentation. This would reduce the time and effort needed by other developers when they use the extension.

- An extension should be using an appropriate license. If you want to make your extension to be used by both open-source and closed-source projects, you may consider using licenses such as BSD, MIT, etc., but not GPL as it requires its derived code to be open-source as well.

In the following, we describe how to create a new extension, according to its categorization as described in overview. These descriptions also apply when you are creating a component mainly used in your own projects.

6.3.1   Application Component

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

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

6.3.2 Behavior

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

When developing behaviors for CModel and CActiveRecord, one can also extend CModelBehavior and CActiveRecordBehavior, respectively. These base classes offer additional features that are specifically made for CModel and CActiveRecord. For example, the CActiveRecordBehavior class implements a set of methods to respond to the life cycle events raised in an ActiveRecord object. A child class can thus override these methods to put in customized code which will participate in the AR life cycles.

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

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

6.3.3 Widget

A widget should extend from CWidget or its child classes.

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

configure the property when using the widget.

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

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

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

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

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

```
}
```

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

6.3.4   Action

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

6.3.5   Filter

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

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

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

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

6.3.6   Controller

A controller distributed as an extension should extend from CExtController, instead of CController. The main reason is because CController assumes the controller view files are located under `application.views.ControllerID`, while CExtController assumes the view files are located under the `views` directory which is a subdirectory of the directory containing the controller class file. Therefore, it is easier to redistribute the controller since its view files are staying together with the controller class file.

6.3.7   Validator

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

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

6.3.8   Console Command

A console command should extend from CConsoleCommand and implement its CConsoleCommand::run method. Optionally, we can override CConsoleCommand::getHelp to provide
some nice help information about the command.

```
class MyCommand extends CConsoleCommand
{
    public function run($args)
    {
        // $args gives an array of the command-line arguments for this command
    }

    public function getHelp()
    {
        return 'Usage: how to use this command';
    }
}
```

6.3.9   Module

Please refer to the section about modules on how to create a module.

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

6.3.10 Generic Component

Developing a generic component extension is like writing a class. Again, the component should also be self-contained so that it can be easily used by other developers.

6.4 Using 3rd-Party Libraries

Yii is carefully designed so that third-party libraries can be easily integrated to further extend Yii's functionalities. When using third-party libraries in a project, developers often encounter issues about class naming and file inclusion. Because all Yii classes are prefixed with letter `C`, it is less likely class naming issue would occur; and because Yii relies on SPL autoload to perform class file inclusion, it can play nicely with other libraries if they use the same autoloading feature or PHP include path to include class files.

Below we use an example to illustrate how to use the Zend_Search_Lucene component from the Zend framework in a Yii application.

First, we extract the Zend framework release file to a directory under `protected/vendors`, assuming `protected` is the application base directory. Verify that the file `protected/vendors/Zend/Search/Lucene.php` exists.

Second, at the beginning of a controller class file, insert the following lines:

```php
Yii::import('application.vendors.*');
require_once('Zend/Search/Lucene.php');
```

The above code includes the class file `Lucene.php`. Because we are using a relative path, we need to change the PHP include path so that the file can be located correctly. This is done by calling `Yii::import` before `require_once`.

Once the above set up is ready, we can use the `Lucene` class in a controller action, like the following:

```php
$lucene=new Zend_Search_Lucene($pathOfIndex);
$hits=$lucene->find(strtolower($keyword));
```

Chapter 7

Testing

7.1 Overview

> **Note:** The testing support described in this chapter requires Yii version 1.1 or
> higher. This does not mean, however, that you cannot test applications developed
> using Yii 1.0.x. There are many great testing frameworks available to help you
> accomplish this task, such as PHPUnit, SimpleTest.

Testing is an indispensable process of software development. Whether we are aware of
it or not, we conduct testing all the time when we are developing a Web application.
For example, when we write a class in PHP, we may use some `echo` or `die` statement to
show that we implement a method correctly; when we implement a Web page containing
a complex HTML form, we may try entering some test data to ensure the page interacts
with us as expected. More advanced developers would write some code to automate this
testing process so that each time when we need to test something, we just need to call
up the code and let the computer to perform testing for us. This is known as *automated
testing*, which is the main topic of this chapter.

The testing support provided by Yii includes *unit testing* and *functional testing*.

A unit test verifies that a single unit of code is working as expected. In object-oriented
programming, the most basic code unit is a class. A unit test thus mainly needs to verify
that each of the class interface methods works properly. That is, given different input
parameters, the test verifies the method returns expected results. Unit tests are usually
developed by people who write the classes being tested.

A functional test verifies that a feature (e.g. post management in a blog system) is working
as expected. Compared with a unit test, a functional test sits at a higher level because a
feature being tested often involves multiple classes. Functional tests are usually developed
by people who know very well the system requirements (they could be either developers
or quality engineers).

7.2 Test-Driven Development

Below we show the development cycles in the so-called test-driven development (TDD):

1. Create a new test that covers a feature to be implemented. The test is expected to fail at its first execution because the feature has yet to be implemented.

2. Run all tests and make sure the new test fails.

3. Write code to make the new test pass.

4. Run all tests and make sure they all pass.

5. Refactor the code that is newly written and make sure the tests still pass.

Repeat step 1 to 5 to push forward the functionality implementation.

7.3 Test Environment Setup

The testing supported provided by Yii requires PHPUnit 3.4+ and Selenium Remote Control 1.0+. Please refer to their documentation on how to install PHPUnit and Selenium Remote Control.

When we use the `yiic webapp` console command to create a new Yii application, it will generate the following files and directories for us to write and perform new tests:

```
testdrive/
   protected/              containing protected application files
      tests/               containing tests for the application
         fixtures/         containing database fixtures
         functional/       containing functional tests
         unit/             containing unit tests
         report/           containing coverage reports
         bootstrap.php     the script executed at the very beginning
         phpunit.xml       the PHPUnit configuration file
         WebTestCase.php   the base class for Web-based functional tests
```

As shown in the above, our test code will be mainly put into three directories: `fixtures`, `functional` and `unit`, and the directory `report` will be used to store the generated code coverage reports.

To execute tests (whether unit tests or functional tests), we can execute the following commands in a console window:

```
% cd testdrive/protected/tests
% phpunit functional/PostTest.php    // executes an individual test
% phpunit --verbose functional       // executes all tests under 'functional'
% phpunit --coverage-html ./report unit
```

In the above, the last command will execute all tests under the `unit` directory and generate a code-coverage report under the `report` directory. Note that xdebug extension must be installed and enabled in order to generate code-coverage reports.

7.4   Test Bootstrap Script

Let's take a look what may be in the `bootstrap.php` file. This file is so special because it is like the entry script and is the starting point when we execute a set of tests.

```php
$yiit='path/to/yii/framework/yiit.php';
$config=dirname(__FILE__).'/../config/test.php';
require_once($yiit);
require_once(dirname(__FILE__).'/WebTestCase.php');
Yii::createWebApplication($config);
```

In the above, we first include the `yiit.php` file from the Yii framework, which initializes some global constants and includes necessary test base classes. We then create a Web application instance using the `test.php` configuration file. If we check `test.php`, we shall find that it inherits from the `main.php` configuration file and adds a `fixture` application component whose class is CDbFixtureManager. We will describe fixtures in detail in the next section.

```php
return CMap::mergeArray(
    require(dirname(__FILE__).'/main.php'),
    array(
        'components'=>array(
            'fixture'=>array(
                'class'=>'system.test.CDbFixtureManager',
            ),
            /* uncomment the following to provide test database connection
            'db'=>array(
                'connectionString'=>'DSN for test database',
            ),
            */
        ),
    )
);
```

When we run tests that involve database, we should provide a test database so that the test execution does not interfere with normal development or production activities. To do so, we just need to uncomment the `db` configuration in the above and fill in the `connectionString` property with the DSN (data source name) to the test database.

With such a bootstrap script, when we run unit tests, we will have an application instance that is nearly the same as the one that serves for Web requests. The main difference is that it has the fixture manager and is using the test database.

7.5 Defining Fixtures

Automated tests need to be executed many times. To ensure the testing process is repeatable, we would like to run the tests in some known state called *fixture*. For example, to test the post creation feature in a blog application, each time when we run the tests, the tables storing relevant data about posts (e.g. the `Post` table, the `Comment` table) should be restored to some fixed state. The PHPUnit documentation has described well about generic fixture setup. In this section, we mainly describe how to set up database fixtures, as we just described in the example.

Setting up database fixtures is perhaps one of the most time-consuming parts in testing database-backed Web applications. Yii introduces the CDbFixtureManager application component to alleviate this problem. It basically does the following things when running a set of tests:

- Before all tests run, it resets all tables relevant to the tests to some known state.

- Before a single test method runs, it resets the specified tables to some known state.

- During the execution of a test method, it provides access to the rows of the data that contribute to the fixture.

To use CDbFixtureManager, we configure it in the application configuration as follows,

```
return array(
    'components'=>array(
        'fixture'=>array(
            'class'=>'system.test.CDbFixtureManager',
        ),
    ),
);
```

We then provide the fixture data under the directory `protected/tests/fixtures`. This

directory may be customized to be a different one by configuring the CDbFixtureManager::basePath property in the application configuration. The fixture data is organized as a collection of PHP files called fixture files. Each fixture file returns an array representing the initial rows of data for a particular table. The file name is the same as the table name. The following is an example of the fixture data for the `Post` table stored in a file named `Post.php`:

```php
<?php
return array(
    'sample1'=>array(
        'title'=>'test post 1',
        'content'=>'test post content 1',
        'createTime'=>1230952187,
        'authorId'=>1,
    ),
    'sample2'=>array(
        'title'=>'test post 2',
        'content'=>'test post content 2',
        'createTime'=>1230952287,
        'authorId'=>1,
    ),
);
```

As we can see, two rows of data are returned in the above. Each row is represented as an associative array whose keys are column names and whose values are the corresponding column values. In addition, each row is indexed by a string (e.g. `sample1`, `sample2`) which is called *row alias*. Later when we write test scripts, we can conveniently refer to a row by its alias. We will describe this in detail in the next section.

You may notice that we do not specify the `id` column values in the above fixture. This is because the `id` column is defined to be an auto-incremental primary key whose value will be filled up when we insert new rows.

When CDbFixtureManager is referenced for the first time, it will go through every fixture file and use it to reset the corresponding table. It resets a table by truncating the table, resetting the sequence value for the table's auto-incremental primary key, and then inserting the rows of data from the fixture file into the table.

Sometimes, we may not want to reset every table which has a fixture file before we run a set of tests, because resetting too many fixture files could take very long time. In this case, we can write a PHP script to do the initialization work in a customized way. The script should be saved in a file named `init.php` under the same directory that contains other fixture files. When CDbFixtureManager detects the existence of this script, it will execute this script instead of resetting every table.

It is also possible that we do not like the default way of resetting a table, i.e., truncating it and inserting it with the fixture data. If this is the case, we can write an initialization script for the specific fixture file. The script must be named as the table name suffixed with `.init.php`. For example, the initialization script for the `Post` table would be `Post.init.php`. When CDbFixtureManager sees this script, it will execute this script instead of using the default way to reset the table.

> **Tip:** Having too many fixture files could increase the test time dramatically. For this reason, you should only provide fixture files for those tables whose content may change during the test. Tables that serve as look-ups do not change and thus do not need fixture files.

In the next two sections, we will describe how to make use of the fixtures managed by CDbFixtureManager in unit tests and functional tests.

7.6 Unit Testing

Because the Yii testing framework is built on top of PHPUnit, it is recommended that you go through the PHPUnit documentation first to get the basic understanding on how to write a unit test. We summarize in the following the basic principles of writing a unit test in Yii:

- A unit test is written in terms of a class `XyzTest` which extends from CTestCase or CDbTestCase, where `Xyz` stands for the class being tested. For example, to test the `Post` class, we would name the corresponding unit test as `PostTest` by convention. The base class CTestCase is meant for generic unit tests, while CDbTestCase is suitable for testing active record model classes. Because `PHPUnit_Framework_TestCase` is the ancestor class for both classes, we can use all methods inherited from this class.

- The unit test class is saved in a PHP file named as `XyzTest.php`. By convention, the unit test file may be stored under the directory `protected/tests/unit`.

- The test class mainly contains a set of test methods named as `testAbc`, where `Abc` is often the name of the class method to be tested.

- A test method usually contains a sequence of assertion statements (e.g. `assertTrue`, `assertEquals`) which serve as checkpoints on validating the behavior of the target class.

In the following, we mainly describe how to write unit tests for active record model classes. We will extend our test classes from CDbTestCase because it provides the database fixture

support that we introduced in the previous section.

Assume we want to test the `Comment` model class in the blog demo. We start by creating a class named `CommentTest` and saving it as `protected/tests/unit/CommentTest.php`:

```php
class CommentTest extends CDbTestCase
{
    public $fixtures=array(
        'posts'=>'Post',
        'comments'=>'Comment',
    );

    ......
}
```

In this class, we specify the `fixtures` member variable to be an array that specifies which fixtures will be used by this test. The array represents a mapping from fixture names to model class names or fixture table names (e.g. from fixture name `posts` to model class `Post`). Note that when mapping to fixture table names, we should prefix the table name with a colon (e.g. `:Post`) to differentiate it from model class name. And when using model class names, the corresponding tables will be considered as fixture tables. As we described earlier, fixture tables will be reset to some known state each time when a test method is executed.

Fixture names allow us to access the fixture data in test methods in a convenient way. The following code shows its typical usage:

```php
// return all rows in the 'Comment' fixture table
$comments = $this->comments;
// return the row whose alias is 'sample1' in the 'Post' fixture table
$post = $this->posts['sample1'];
// return the AR instance representing the 'sample1' fixture data row
$post = $this->posts('sample1');
```

> **Note:** If a fixture is declared using its table name (e.g. `'posts'=>':Post'`), then the third usage in the above is not valid because we have no information about which model class the table is associated with.

Next, we write the `testApprove` method to test the `approve` method in the `Comment` model class. The code is very straightforward: we first insert a comment that is pending status; we then verify this comment is in pending status by retrieving it from database; and finally we call the `approve` method and verify the status is changed as expected.

```php
public function testApprove()
{
    // insert a comment in pending status
    $comment=new Comment;
    $comment->setAttributes(array(
        'content'=>'comment 1',
        'status'=>Comment::STATUS_PENDING,
        'createTime'=>time(),
        'author'=>'me',
        'email'=>'me@example.com',
        'postId'=>$this->posts['sample1']['id'],
    ),false);
    $this->assertTrue($comment->save(false));

    // verify the comment is in pending status
    $comment=Comment::model()->findByPk($comment->id);
    $this->assertTrue($comment instanceof Comment);
    $this->assertEquals(Comment::STATUS_PENDING,$comment->status);

    // call approve() and verify the comment is in approved status
    $comment->approve();
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
    $comment=Comment::model()->findByPk($comment->id);
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
}
```

7.7   Functional Testing

Before reading this section, it is recommended that you read the Selenium documentation and the PHPUnit documentation first. We summarize in the following the basic principles of writing a functional test in Yii:

- Like unit test, a functional test is written in terms of a class `XyzTest` which extends from CWebTestCase, where `Xyz` stands for the class being tested. Because `PHPUnit_Extensions_SeleniumTestCase` is the ancestor class for CWebTestCase, we can use all methods inherited from this class.

- The functional test class is saved in a PHP file named as `XyzTest.php`. By convention, the functional test file may be stored under the directory `protected/tests/functional`.

- The test class mainly contains a set of test methods named as `testAbc`, where `Abc` is often the name of a feature to be tested. For example, to test the user login feature, we can have a test method named as `testLogin`.

- A test method usually contains a sequence of statements that would issue commands to Selenium RC to interact with the Web application being tested. It also contains assertion statements to verify that the Web application responds as expected.

Before we describe how to write a functional test, let's take a look at the `WebTestCase.php` file generated by the `yiic webapp` command. This file defines `WebTestCase` that may serve as the base class for all functional test classes.

```php
define('TEST_BASE_URL','http://localhost/yii/demos/blog/index-test.php/');

class WebTestCase extends CWebTestCase
{
    /**
     * Sets up before each test method runs.
     * This mainly sets the base URL for the test application.
     */
    protected function setUp()
    {
        parent::setUp();
        $this->setBrowserUrl(TEST_BASE_URL);
    }

    ......
}
```

The class `WebTestCase` mainly sets the base URL of the pages to be tested. Later in test methods, we can use relative URLs to specify which pages to be tested.

We should also pay attention that in the base test URL, we use `index-test.php` as the entry script instead of `index.php`. The only difference between `index-test.php` and `index.php` is that the former uses `test.php` as the application configuration file while the latter `main.php`.

We now describe how to test the feature about showing a post in the blog demo. We first write the test class as follows, noting that the test class extends from the base class we just described:

```php
class PostTest extends WebTestCase
{
    public $fixtures=array(
        'posts'=>'Post',
    );

    public function testShow()
```

```
    {
        $this->open('post/1');
        // verify the sample post title exists
        $this->assertTextPresent($this->posts['sample1']['title']);
        // verify comment form exists
        $this->assertTextPresent('Leave a Comment');
    }

    ......
}
```

Like writing a unit test class, we declare the fixtures to be used by this test. Here we indicate that the `Post` fixture should be used. In the `testShow` test method, we first instruct Selenium RC to open the URL `post/1`. Note that this is a relative URL, and the complete URL is formed by appending it to the base URL we set in the base class (i.e. `http://localhost/yii/demos/blog/index-test.php/post/1`). We then verify that we can find the title of the `sample1` post can be found in the current Web page. And we also verify that the page contains the text `Leave a Comment`.

> **Tip:** Before running functional tests, the Selenium-RC server must be started. This can be done by executing the command `java -jar selenium-server.jar` under your Selenium server installation directory.

CHAPTER 8

Special Topics

8.1 Automatic Code Generation

Starting from version 1.1.2, Yii is equipped with a Web-based code generation tool called *Gii*. It supercedes the previous `yiic shell` generation tool which runs on command line. In this section, we will describe how to use Gii and how to extend Gii to increase our development productivity.

8.1.1 Using Gii

Gii is implemented in terms of a module and must be used within an existing Yii application. To use Gii, we first modify the application configuration as follows:

```
return array(
    ......
    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',
            'password'=>'pick up a password here',
            // 'ipFilters'=>array(...a list of IPs...),
            // 'newFileMode'=>0666,
            // 'newDirMode'=>0777,
        ),
    ),
);
```

In the above, we declare a module named `gii` whose class is GiiModule. We also specify a password for the module which we will be prompted for when accessing Gii.

By default for security reasons, Gii is configured to be accessible only on localhost. If we want to make it accessible on other trustable computers, we can configure the GiiModule::ipFilters property as shown in the above code.

Because Gii may generate and save new code files in the existing application, we need to make sure that the Web server process has the proper permission to do so. The above GiiModule::newFileMode and GiiModule::newDirMode properties control how the new files and directories should be generated.

> **Note:** Gii is mainly provided as a development tool. Therefore, it should only be installed on a development machine. Because it can generate new PHP script files in the application, we should pay sufficient attention to its security measures (e.g. password, IP filters).

We can now access Gii via the URL `http://hostname/path/to/index.php?r=gii`. Here we assume `http://hostname/path/to/index.php` is the URL for accessing the existing Yii application.

If the existing Yii application uses `path`-format URLs (see URL management), we can access Gii via the URL `http://hostnamepath/to/index.php/gii`. We may need to add the following URL rules to the front of the existing URL rules:

```
'components'=>array(
    ......
    'urlManager'=>array(
        'urlFormat'=>'path',
        'rules'=>array(
            'gii'=>'gii',
            'gii/<controller:\w+>'=>'gii/<controller>',
            'gii/<controller:\w+>/<action:\w+>'=>'gii/<controller>/<action>',
            ...existing rules...
        ),
    ),
)
```

Gii comes with a few default code generators. Each code generator is responsible for generating a specific type of code. For example, the controller generator generates a controller class together with a few action view scripts; the model generator generates an ActiveRecord class for the specified database table.

The basic workflow of using a generator is as follows:

1. Enter the generator page;

2. Fill in the fields that specify the code generation parameters. For example, to use the module generator to create a new module, you need to specify the module ID;

3. Press the `Preview` button to preview the code to be generated. You will see a table showing a list of code files to be generated. You can click on any of them to preview the code;

4. Press the `Generate` button to generate the code files;

5. Review the code generation log.

8.1.2 Extending Gii

While the default code generators coming with Gii can generate very powerful code, we often want to customize them or create new ones to fit for our taste and needs. For example, we may want the generated code to be in our own favorite coding styles, or we may want to make the code to support multiple languages. All these can be done easily with Gii.

Gii can be extended in two ways: customizing the code templates of the existing code generators, and writing new code generators.

Structure of a Code Generator

A code generator is stored under a directory whose name is treated as the generator name. The directory usually consists of the following content:

```
model/                     the model generator root folder
   ModelCode.php           the code model used to generate code
   ModelGenerator.php      the code generation controller
   views/                  containing view scripts for the generator
      index.php            the default view script
   templates/              containing code template sets
      default/             the 'default' code template set
         model.php         the code template for generating model class code
```

Generator Search Path

Gii looks for available generators in a set of directories specified by the GiiModule::generatorPaths property. When customization is needed, we can configure this property in the application configuration as follows,

```php
return array(
    'modules'=>array(
```

```
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'generatorPaths'=>array(
            'application.gii',   // a path alias
        ),
    ),
),
);
```

The above configuration instructs Gii to look for generators under the directory aliased as `application.gii`, in addition to the default location `system.gii.generators`.

It is possible to have two generators with the same name but under different search paths. In this case, the generator under the path specified earlier in GiiModule::generatorPaths will take precedence.

Customizing Code Templates

This is the easiest and the most common way of extending Gii. We use an example to explain how to customize code templates. Assume we want to customize the code generated by the model generator.

We first create a directory named `protected/gii/model/templates/compact`. Here `model` means that we are going to *override* the default model generator. And `templates/compact` means we will add a new code template set named `compact`.

We then modify our application configuration to add `application.gii` to GiiModule::generatorPaths, as shown in the previous sub-section.

Now open the model code generator page. Click on the `Code Template` field. We should see a dropdown list which contains our newly created template directory `compact`. However, if we choose this template to generate the code, we will see an error. This is because we have yet to put any actual code template file in this new `compact` template set.

Copy the file `framework/gii/generators/model/templates/default/model.php` to `protected/gii/model/templates/compact`. If we try generating again with the `compact` template, we should succeed. However, the code generated is no different from the one generated by the `default` template set.

It is time for us to do the real customization work. Open the file `protected/gii/model/templates/compact/model.php` to edit it. Remember that this file will be used like a view script, which means it can contain PHP expressions and statements. Let's modify the

template so that the `attributeLabels()` method in the generated code uses `Yii::t()` to translate the attribute labels:

```php
public function attributeLabels()
{
    return array(
<?php foreach($labels as $name=>$label): ?>
            <?php echo "'$name' => Yii::t('application', '$label'),\n"; ?>
<?php endforeach; ?>
    );
}
```

In each code template, we can access some predefined variables, such as `$labels` in the above example. These variables are provided by the corresponding code generator. Different code generators may provide different set of variables in their code templates. Please read the description in the default code templates carefully.

Creating New Generators

In this sub-section, we show how to create a new generator that can generate a new widget class.

We first create a directory named `protected/gii/widget`. Under this directory, we will create the following files:

- `WidgetGenerator.php`: contains the `WidgetGenerator` controller class. This is the entry point of the widget generator.

- `WidgetCode.php`: contains the `WidgetCode` model class. This class has the main logic for code generation.

- `views/index.php`: the view script showing the code generator input form.

- `templates/default/widget.php`: the default code template for generating a widget class file.

¡h4 id="creating-x-3086x"¿Creating `WidgetGenerator.php`¡/h4¿

The `WidgetGenerator.php` file is extremely simple. It only contains the following code:

```php
class WidgetGenerator extends CCodeGenerator
```

```
{
    public $codeModel='application.gii.widget.WidgetCode';
}
```

In the above code, we specify that the generator will use the model class whose path alias is `application.gii.widget.WidgetCode`. The `WidgetGenerator` class extends from CCode-Generator which implements a lot of functionalities, including the controller actions needed to coordinate the code generation process.

¡h4 id="creating-x-3089x"¿Creating `WidgetCode.php`¡/h4¿

The `WidgetCode.php` file contains the `WidgetCode` model class that has the main logic for generating a widget class based on the user input. In this example, we assume that the only input we want from the user is the widget class name. Our `WidgetCode` looks like the following:

```php
class WidgetCode extends CCodeModel
{
    public $className;

    public function rules()
    {
        return array_merge(parent::rules(), array(
            array('className', 'required'),
            array('className', 'match', 'pattern'=>'/^\w+$/'),
        ));
    }

    public function attributeLabels()
    {
        return array_merge(parent::attributeLabels(), array(
            'className'=>'Widget Class Name',
        ));
    }

    public function prepare()
    {
        $path=Yii::getPathOfAlias('application.components.' . $this->className) . '.php';
        $code=$this->render($this->templatepath.'/widget.php');

        $this->files[]=new CCodeFile($path, $code);
    }
}
```

The `WidgetCode` class extends from CCodeModel. Like a normal model class, in this class we can declare `rules()` and `attributeLabels()` to validate user inputs and provide at-

tribute labels, respectively. Note that because the base class CCodeModel already defines some rules and attribute labels, we should merge them with our new rules and labels here.

The `prepare()` method prepares the code to be generated. Its main task is to prepare a list of CCodeFile objects, each of which represent a code file being generated. In our example, we only need to create one CCodeFile object that represents the widget class file being generated. The new widget class will be generated under the `protected/components` directory. We call CCodeFile::render method to generate the actual code. This method includes the code template as a PHP script and returns the echoed content as the generated code.

¡h4 id="creating-x-3092x"¿Creating `views/index.php`¡/h4¿

Having the controller (`WidgetGenerator`) and the model (`WidgetCode`), it is time for us to create the view `views/index.php`.

```
<h1>Widget Generator</h1>

<?php $form=$this->beginWidget('CCodeForm', array('model'=>$model)); ?>

    <div class="row">
        <?php echo $form->labelEx($model,'className'); ?>
        <?php echo $form->textField($model,'className',array('size'=>65)); ?>
        <div class="tooltip">
            Widget class name must only contain word characters.
        </div>
        <?php echo $form->error($model,'className'); ?>
    </div>

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

In the above, we mainly display a form using the CCodeForm widget. In this form, we display the field to collect the input for the `className` attribute in `WidgetCode`.

When creating the form, we can exploit two nice features provided by the CCodeForm widget. One is about input tooltips. The other is about sticky inputs.

If you have tried any default code generator, you will notice that when setting focus in one input field, a nice tooltip will show up next to the field. This can easily achieved here by writing next to the input field a `div` whose CSS class is `tooltip`.

For some input fields, we may want to remember their last valid values so that the user can save the trouble of re-entering them each time they use the generator to generate code. An example is the input field collecting the controller base class name default controller

generator. These sticky fields are initially displayed as highlighted static text. If we click on them, they will turn into input fields to take user inputs.

In order to declare an input field to be sticky, we need to do two things.

First, we need to declare a `sticky` validation rule for the corresponding model attribute. For example, the default controller generator has the following rule to declare that `baseClass` and `actions` attributes are sticky:

```php
public function rules()
{
    return array_merge(parent::rules(), array(
        ......
        array('baseClass, actions', 'sticky'),
    ));
}
```

Second, we need to add a CSS class named `sticky` to the container `div` of the input field in the view, like the following:

```html
<div class="row sticky">
    ...input field here...
</div>
```

¡h4 id="creating-x-3095x"¿Creating `templates/default/widget.php`¡/h4¿

Finally, we create the code template `templates/default/widget.php`. As we described earlier, this is used like a view script that can contain PHP expressions and statements. In a code template, we can always access the `$this` variable which refers to the code model object. In our example, `$this` refers to the `WidgetModel` object. We can thus get the user-entered widget class name via `$this->className`.

```php
<?php echo '<?php'; ?>

class <?php echo $this->className; ?> extends CWidget
{
    public function run()
    {

    }
}
```

This concludes the creation of a new code generator. We can access this code generator immediately via the URL `http://hostname/path/to/index.php?r=gii/widget`.

8.2   URL Management

Complete URL management for a Web application involves two aspects. First, when a user request comes in terms of a URL, the application needs to parse it into understandable parameters. Second, the application needs to provide a way of creating URLs so that the created URLs can be understood by the application. For a Yii application, these are accomplished with the help of CUrlManager.

8.2.1   Creating URLs

Although URLs can be hardcoded in controller views, it is often more flexible to create them dynamically:

```
$url=$this->createUrl($route,$params);
```

where $this refers to the controller instance; $route specifies the route of the request; and $params is a list of GET parameters to be appended to the URL.

By default, URLs created by createUrl is in the so-called get format. For example, given $route='post/read' and $params=array('id'=>100), we would obtain the following URL:

```
/index.php?r=post/read&id=100
```

where parameters appear in the query string as a list of Name=Value concatenated with the ampersand characters, and the r parameter specifies the request route. This URL format is not very user-friendly because it requires several non-word characters.

We could make the above URL look cleaner and more self-explanatory by using the so-called path format which eliminates the query string and puts the GET parameters into the path info part of URL:

```
/index.php/post/read/id/100
```

To change the URL format, we should configure the urlManager application component so that createUrl can automatically switch to the new format and the application can properly understand the new URLs:

```
array(
    ......
    'components'=>array(
```

```
        ......
        'urlManager'=>array(
            'urlFormat'=>'path',
        ),
    ),
);
```

Note that we do not need to specify the class of the urlManager component because it is pre-declared as CUrlManager in CWebApplication.

> **Tip:** The URL generated by the createUrl method is a relative one. In order to get an absolute URL, we can prefix it with `Yii::app()->request->hostInfo`, or call createAbsoluteUrl.

8.2.2   User-friendly URLs

When `path` is used as the URL format, we can specify some URL rules to make our URLs even more user-friendly. For example, we can generate a URL as short as `/post/100`, instead of the lengthy `/index.php/post/read/id/100`. URL rules are used by CUrlManager for both URL creation and parsing purposes.

To specify URL rules, we need to configure the rules property of the urlManager application component:

```
array(
    ......
    'components'=>array(
        ......
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
                'pattern1'=>'route1',
                'pattern2'=>'route2',
                'pattern3'=>'route3',
            ),
        ),
    ),
);
```

The rules are specified as an array of pattern-route pairs, each corresponding to a single rule. The pattern of a rule is a string used to match the path info part of URLs. And the route of a rule should refer to a valid controller route.

Besides the above pattern-route format, a rule may also be specified with customized options, like the following:

```
'pattern1'=>array('route1', 'urlSuffix'=>'.xml', 'caseSensitive'=>false)
```

In the above, the array contains a list of customized options. As of version 1.1.0, the following options are available:

- urlSuffix: the URL suffix used specifically for this rule. Defaults to null, meaning using the value of CUrlManager::urlSuffix.

- caseSensitive: whether this rule is case sensitive. Defaults to null, meaning using the value of CUrlManager::caseSensitive.

- defaultParams: the default GET parameters (name=¿value) that this rule provides. When this rule is used to parse the incoming request, the values declared in this property will be injected into $_GET.matchValue : whethertheGET parametervaluesshouldmatchthecorrespondingsub patternsintherulewhencreatingaURL.Defaultstonull, meaningusingthevalueof CUrlManager ::matchValue.If thispropertyisf alse, itmeansarulewillbeusedforcreatingaURLif itsrouteandparame patterns.Notethatsettingthispropertytotruewilldegradeperformance.$

Using Named Parameters

A rule can be associated with a few GET parameters. These GET parameters appear in the rule's pattern as special tokens in the following format:

- `<ParamName:ParamPattern>`

where `ParamName` specifies the name of a GET parameter, and the optional `ParamPattern` specifies the regular expression that should be used to match the value of the GET parameter. In case when `ParamPattern` is omitted, it means the parameter should match any characters except the slash `/`. When creating a URL, these parameter tokens will be replaced with the corresponding parameter values; when parsing a URL, the corresponding GET parameters will be populated with the parsed results.

Let's use some examples to explain how URL rules work. We assume that our rule set consists of three rules:

```
array(
```

```
    'posts'=>'post/list',
    'post/<id:\d+>'=>'post/read',
    'post/<year:\d{4}>/<title>'=>'post/read',
)
```

- Calling `$this->createUrl('post/list')` generates `/index.php/posts`. The first rule is applied.

- Calling `$this->createUrl('post/read',array('id'=>100))` generates `/index.php/post/100`. The second rule is applied.

- Calling `$this->createUrl('post/read',array('year'=>2008,'title'=>'a sample post'))` generates `/index.php/post/2008/a%20sample%20post`. The third rule is applied.

- Calling `$this->createUrl('post/read')` generates `/index.php/post/read`. None of the rules is applied.

In summary, when using createUrl to generate a URL, the route and the GET parameters passed to the method are used to decide which URL rule to be applied. If every parameter associated with a rule can be found in the GET parameters passed to createUrl, and if the route of the rule also matches the route parameter, the rule will be used to generate the URL.

If the GET parameters passed to createUrl are more than those required by a rule, the additional parameters will appear in the query string. For example, if we call `$this->createUrl('post/read',array('id'=>100,'year'=>2008))`, we would obtain `/index.php/post/100?year=2008`. In order to make these additional parameters appear in the path info part, we should append `/*` to the rule. Therefore, with the rule `post/<id:\d+>/*`, we can obtain the URL as `/index.php/post/100/year/2008`.

As we mentioned, the other purpose of URL rules is to parse the requesting URLs. Naturally, this is an inverse process of URL creation. For example, when a user requests for `/index.php/post/100`, the second rule in the above example will apply, which resolves in the route `post/read` and the GET parameter `array('id'=>100)` (accessible via `$_GET`).

> **Note:** Using URL rules will degrade application performance. This is because when parsing the request URL, CUrlManager will attempt to match it with each rule until one can be applied. The more the number of rules, the more the performance impact. Therefore, a high-traffic Web application should minimize its use of URL rules.

Parameterizing Routes

Starting from version 1.0.5, we may reference named parameters in the route part of a rule. This allows a rule to be applied to multiple routes based on matching criteria. It may also help reduce the number of rules needed for an application, and thus improve the overall performance.

We use the following example rules to illustrate how to parameterize routes with named parameters:

```
array(
    '<_c:(post|comment)>/<id:\d+>/<_a:(create|update|delete)>' => '<_c>/<_a>',
    '<_c:(post|comment)>/<id:\d+>' => '<_c>/read',
    '<_c:(post|comment)>s' => '<_c>/list',
)
```

In the above, we use two named parameters in the route part of the rules: `_c` and `_a`. The former matches a controller ID to be either `post` or `comment`, while the latter matches an action ID to be `create`, `update` or `delete`. You may name the parameters differently as long as they do not conflict with GET parameters that may appear in URLs.

Using the aboving rules, the URL `/index.php/post/123/create` would be parsed as the route `post/create` with GET parameter `id=123`. And given the route `comment/list` and GET parameter `page=2`, we can create a URL `/index.php/comments?page=2`.

Parameterizing Hostnames

Starting from version 1.0.11, it is also possible to include hostname into the rules for parsing and creating URLs. One may extract part of the hostname to be a GET parameter. For example, the URL `http://admin.example.com/en/profile` may be parsed into GET parameters `user=admin` and `lang=en`. On the other hand, rules with hostname may also be used to create URLs with paratermized hostnames.

In order to use parameterized hostnames, simply declare URL rules with host info, e.g.:

```
array(
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
)
```

The above example says that the first segment in the hostname should be treated as `user` parameter while the first segment in the path info should be `lang` parameter. The rule corresponds to the `user/profile` route.

Note that CUrlManager::showScriptName will not take effect when a URL is being created using a rule with parameterized hostname.

Also note that the rule with parameterized hostname should NOT contain the sub-folder if the application is under a sub-folder of the Web root. For example, if the application is under `http://www.example.com/sandbox/blog`, then we should still use the same URL rule as described above without the sub-folder `sandbox/blog`.

Hiding `index.php`

There is one more thing that we can do to further clean our URLs, i.e., hiding the entry script `index.php` in the URL. This requires us to configure the Web server as well as the urlManager application component.

We first need to configure the Web server so that a URL without the entry script can still be handled by the entry script. For Apache HTTP server, this can be done by turning on the URL rewriting engine and specifying some rewriting rules. We can create the file `/wwwroot/blog/.htaccess` with the following content. Note that the same content can also be put in the Apache configuration file within the `Directory` element for `/wwwroot/blog`.

```
Options +FollowSymLinks
IndexIgnore */*
RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

We then configure the showScriptName property of the urlManager component to be `false`.

Now if we call `$this->createUrl('post/read',array('id'=>100))`, we would obtain the URL `/post/100`. More importantly, this URL can be properly recognized by our Web application.

Faking URL Suffix

We may also add some suffix to our URLs. For example, we can have `/post/100.html` instead of `/post/100`. This makes it look more like a URL to a static Web page. To do so, simply configure the urlManager component by setting its urlSuffix property to the suffix you like.

8.3 Authentication and Authorization

Authentication and authorization are required for a Web page that should be limited to certain users. Authentication is about verifying whether someone is who he claims he is. It usually involves a username and a password, but may include any other methods of demonstrating identity, such as a smart card, fingerprints, etc. Authorization is finding out if the person, once identified (authenticated), is permitted to manipulate specific resources. This is usually determined by finding out if that person is of a particular role that has access to the resources.

Yii has a built-in authentication/authorization (auth) framework which is easy to use and can be customized for special needs.

The central piece in the Yii auth framework is a pre-declared *user application component* which is an object implementing the IWebUser interface. The user component represents the persistent identity information for the current user. We can access it at any place using `Yii::app()->user`.

Using the user component, we can check if a user is logged in or not via CWebUser::isGuest; we can login and logout a user; we can check if the user can perform specific operations by calling CWebUser::checkAccess; and we can also obtain the unique identifier and other persistent identity information about the user.

8.3.1 Defining Identity Class

In order to authenticate a user, we define an identity class which contains the actual authentication logic. The identity class should implement the IUserIdentity interface. Different classes may be implemented for different authentication approaches (e.g. OpenID, LDAP). A good start is by extending CUserIdentity which is a base class for the authentication approach based on username and password.

The main work in defining an identity class is the implementation of the IUserIdentity::authenticate method. An identity class may also declare additional identity information that needs to be persistent during the user session.

In the following example, we validate the given username and password against the user table in a database using Active Record. We also override the `getId` method to return the `_id` variable which is set during authentication (the default implementation would return the username as the ID). During authentication, we store the retrieved `title` information in a state with the same name by calling CBaseUserIdentity::setState.

```
class UserIdentity extends CUserIdentity
{
    private $_id;
    public function authenticate()
    {
        $record=User::model()->findByAttributes(array('username'=>$this->username));
        if($record===null)
            $this->errorCode=self::ERROR_USERNAME_INVALID;
        else if($record->password!==md5($this->password))
            $this->errorCode=self::ERROR_PASSWORD_INVALID;
        else
        {
            $this->_id=$record->id;
            $this->setState('title', $record->title);
            $this->errorCode=self::ERROR_NONE;
        }
        return !$this->errorCode;
    }

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

Information stored in a state (by calling CBaseUserIdentity::setState) will be passed to CWebUser which stores them in a persistent storage, such as session. These information can be accessed like properties of CWebUser. For example, we can obtain the `title` information of the current user by `Yii::app()->user->title` (This has been available since version 1.0.3. In prior versions, we should use `Yii::app()->user->getState('title')`, instead.)

> **Info:** By default, CWebUser uses session as persistent storage for user identity information. If cookie-based login is enabled (by setting CWebUser::allowAutoLogin to be true), the user identity information may also be saved in cookie. Make sure you do not declare sensitive information (e.g. password) to be persistent.

8.3.2   Login and Logout

Using the identity class and the user component, we can implement login and logout actions easily.

```
// Login a user with the provided username and password.
$identity=new UserIdentity($username,$password);
if($identity->authenticate())
    Yii::app()->user->login($identity);
else
    echo $identity->errorMessage;
......
// Logout the current user
Yii::app()->user->logout();
```

By default, a user will be logged out after a certain period of inactivity, depending on the session configuration. To change this behavior, we can set the allowAutoLogin property of the user component to be true and pass a duration parameter to the CWebUser::login method. The user will then remain logged in for the specified duration, even if he closes his browser window. Note that this feature requires the user's browser to accept cookies.

```
// Keep the user logged in for 7 days.
// Make sure allowAutoLogin is set true for the user component.
Yii::app()->user->login($identity,3600*24*7);
```

8.3.3   Access Control Filter

Access control filter is a preliminary authorization scheme that checks if the current user can perform the requested controller action. The authorization is based on user's name, client IP address and request types. It is provided as a filter named as "accessControl".

> **Tip:** Access control filter is sufficient for simple scenarios. For complex access control, you may use role-based access (RBAC) which is to be covered shortly.

To control the access to actions in a controller, we install the access control filter by overriding CController::filters (see Filter for more details about installing filters).

```
class PostController extends CController
{
    ......
    public function filters()
```

```
    {
        return array(
            'accessControl',
        );
    }
}
```

In the above, we specify that the access control filter should be applied to every action of `PostController`. The detailed authorization rules used by the filter are specified by overriding CController::accessRules in the controller class.

```
class PostController extends CController
{
    ......
    public function accessRules()
    {
        return array(
            array('deny',
                'actions'=>array('create', 'edit'),
                'users'=>array('?'),
            ),
            array('allow',
                'actions'=>array('delete'),
                'roles'=>array('admin'),
            ),
            array('deny',
                'actions'=>array('delete'),
                'users'=>array('*'),
            ),
        );
    }
}
```

The above code specifies three rules, each represented as an array. The first element of the array is either `'allow'` or `'deny'` and the rest name-value pairs specify the pattern parameters of the rule. These rules read: the `create` and `edit` actions cannot be executed by anonymous users; the `delete` action can be executed by users with `admin` role; and the `delete` action cannot be executed by anyone.

The access rules are evaluated one by one in the order they are specified. The first rule that matches the current pattern (e.g. username, roles, client IP, address) determines the authorization result. If this rule is an `allow` rule, the action can be executed; if it is a `deny` rule, the action cannot be executed; if none of the rules matches the context, the action can still be executed.

> **Tip:** To ensure an action does not get executed under certain contexts, it is beneficial to always specify a matching-all `deny` rule at the end of rule set, like the following:
>
> ```
> return array(
>     // ... other rules...
>     // the following rule denies 'delete' action for all contexts
>     array('deny',
>         'actions'=>array('delete'),
>     ),
> );
> ```
>
> The reason for this rule is because if none of the rules matches a context, an action will be executed.

An access rule can match the following context parameters:

- actions: specifies which actions this rule matches. This should be an array of action IDs. The comparison is case-insensitive.

- controllers: specifies which controllers this rule matches. This should be an array of controller IDs. The comparison is case-insensitive. This option has been available since version 1.0.4.

- users: specifies which users this rule matches. The current user's name is used for matching. The comparison is case-insensitive. Three special characters can be used here:

    - `*`: any user, including both anonymous and authenticated users.
    - `?`: anonymous users.
    - `@`: authenticated users.

- roles: specifies which roles that this rule matches. This makes use of the role-based access control feature to be described in the next subsection. In particular, the rule is applied if CWebUser::checkAccess returns true for one of the roles. Note, you should mainly use roles in an `allow` rule because by definition, a role represents a permission to do something. Also note, although we use the term `roles` here, its value can actually be any auth item, including roles, tasks and operations.

- ips: specifies which client IP addresses this rule matches.

- verbs: specifies which request types (e.g. `GET`, `POST`) this rule matches. The comparison is case-insensitive.

- expression: specifies a PHP expression whose value indicates whether this rule matches. In the expression, you can use variable $user which refers to Yii::app()->user. This option has been available since version 1.0.3.

Handling Authorization Result

When authorization fails, i.e., the user is not allowed to perform the specified action, one of the following two scenarios may happen:

- If the user is not logged in and if the loginUrl property of the user component is configured to be the URL of the login page, the browser will be redirected to that page. Note that by default, loginUrl points to the site/login page.

- Otherwise an HTTP exception will be displayed with error code 403.

When configuring the loginUrl property, one can provide a relative or absolute URL. One can also provide an array which will be used to generate a URL by calling CWebApplication::createUrl. The first array element should specify the route to the login controller action, and the rest name-value pairs are GET parameters. For example,

```php
array(
    ......
    'components'=>array(
        'user'=>array(
            // this is actually the default value
            'loginUrl'=>array('site/login'),
        ),
    ),
)
```

If the browser is redirected to the login page and the login is successful, we may want to redirect the browser back to the page that caused the authorization failure. How do we know the URL for that page? We can get this information from the returnUrl property of the user component. We can thus do the following to perform the redirection:

```php
Yii::app()->request->redirect(Yii::app()->user->returnUrl);
```

8.3.4 Role-Based Access Control

Role-Based Access Control (RBAC) provides a simple yet powerful centralized access control. Please refer to the Wiki article for more details about comparing RBAC with other more traditional access control schemes.

Yii implements a hierarchical RBAC scheme via its authManager application component. In the following ,we first introduce the main concepts used in this scheme; we then describe how to define authorization data; at the end we show how to make use of the authorization data to perform access checking.

Overview

A fundamental concept in Yii's RBAC is *authorization item*. An authorization item is a permission to do something (e.g. creating new blog posts, managing users). According to its granularity and targeted audience, authorization items can be classified as *operations*, *tasks* and *roles*. A role consists of tasks, a task consists of operations, and an operation is a permission that is atomic. For example, we can have a system with `administrator` role which consists of `post management` task and `user management` task. The `user management` task may consist of `create user`, `update user` and `delete user` operations. For more flexibility, Yii also allows a role to consist of other roles or operations, a task to consist of other tasks, and an operation to consist of other operations.

An authorization item is uniquely identified by its name.

An authorization item may be associated with a *business rule*. A business rule is a piece of PHP code that will be executed when performing access checking with respect to the item. Only when the execution returns true, will the user be considered to have the permission represented by the item. For example, when defining an operation `updatePost`, we would like to add a business rule that checks if the user ID is the same as the post's author ID so that only the author himself can have the permission to update a post.

Using authorization items, we can build up an *authorization hierarchy*. An item `A` is a parent of another item `B` in the hierarchy if `A` consists of `B` (or say `A` inherits the permission(s) represented by `B`). An item can have multiple child items, and it can also have multipe parent items. Therefore, an authorization hierarchy is a partial-order graph rather than a tree. In this hierarchy, role items sit on top levels, operation items on bottom levels, while task items in between.

Once we have an authorization hierarchy, we can assign roles in this hierarchy to application users. A user, once assigned with a role, will have the permissions represented by the role. For example, if we assign the `administrator` role to a user, he will have the administrator permissions which include `post management` and `user management` (and the corresponding operations such as `create user`).

Now the fun part starts. In a controller action, we want to check if the current user can delete the specified post. Using the RBAC hierarchy and assignment, this can be done

easily as follows:

```
if(Yii::app()->user->checkAccess('deletePost'))
{
    // delete the post
}
```

Configuring Authorization Manager

Before we set off to define an authorization hierarchy and perform access checking, we need to configure the authManager application component. Yii provides two types of authorization managers: CPhpAuthManager and CDbAuthManager. The former uses a PHP script file to store authorization data, while the latter stores authorization data in database. When we configure the authManager application component, we need to specify which component class to use and what are the initial property values for the component. For example,

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'sqlite:path/to/file.db',
        ),
        'authManager'=>array(
            'class'=>'CDbAuthManager',
            'connectionID'=>'db',
        ),
    ),
);
```

We can then access the authManager application component using Yii::app()->authManager.

Defining Authorization Hierarchy

Defining authorization hierarchy involves three steps: defining authorization items, establishing relationships between authorization items, and assigning roles to application users. The authManager application component provides a whole set of APIs to accomplish these tasks.

To define an authorization item, call one of the following methods, depending on the type of the item:

- CAuthManager::createRole

- CAuthManager::createTask

- CAuthManager::createOperation

Once we have a set of authorization items, we can call the following methods to establish relationships between authorization items:

- CAuthManager::addItemChild

- CAuthManager::removeItemChild

- CAuthItem::addChild

- CAuthItem::removeChild

And finally, we call the following methods to assign role items to individual users:

- CAuthManager::assign

- CAuthManager::revoke

Below we show an example about building an authorization hierarchy with the provided APIs:

```php
$auth=Yii::app()->authManager;

$auth->createOperation('createPost','create a post');
$auth->createOperation('readPost','read a post');
$auth->createOperation('updatePost','update a post');
$auth->createOperation('deletePost','delete a post');

$bizRule='return Yii::app()->user->id==$params["post"]->authID;';
$task=$auth->createTask('updateOwnPost','update a post by author himself',$bizRule);
$task->addChild('updatePost');

$role=$auth->createRole('reader');
$role->addChild('readPost');

$role=$auth->createRole('author');
$role->addChild('reader');
$role->addChild('createPost');
$role->addChild('updateOwnPost');
```

```
$role=$auth->createRole('editor');
$role->addChild('reader');
$role->addChild('updatePost');

$role=$auth->createRole('admin');
$role->addChild('editor');
$role->addChild('author');
$role->addChild('deletePost');

$auth->assign('reader','readerA');
$auth->assign('author','authorB');
$auth->assign('editor','editorC');
$auth->assign('admin','adminD');
```

> **Info:** While the above example looks long and tedious, it is mainly for demonstrative purpose. Developers usually need to develop some user interfaces so that end users can use to establish an authorization hierarchy more intuitively.

Using Business Rules

When we are defining the authorization hierarchy, we can associate a role, a task or an operation with a so-called *business rule*. We may also associate a business rule when we assign a role to a user. A business rule is a piece of PHP code that is executed when we perform access checking. The returning value of the code is used to determine if the role or assignment applies to the current user. In the example above, we associated a business rule with the `updateOwnPost` task. In the business rule we simply check if the current user ID is the same as the specified post's author ID. The post information in the `$params` array is supplied by developers when performing access checking.

Access Checking

To perform access checking, we first need to know the name of the authorization item. For example, to check if the current user can create a post, we would check if he has the permission represented by the `createPost` operation. We then call CWebUser::checkAccess to perform the access checking:

```
if(Yii::app()->user->checkAccess('createPost'))
{
    // create post
}
```

If the authorization rule is associated with a business rule which requires additional parameters, we can pass them as well. For example, to check if a user can update a post, we would do

```php
$params=array('post'=>$post);
if(Yii::app()->user->checkAccess('updateOwnPost',$params))
{
    // update post
}
```

Using Default Roles

> **Note:** The default role feature has been available since version 1.0.3

Many Web applications need some very special roles that would be assigned to every or most of the system users. For example, we may want to assign some privileges to all authenticated users. It poses a lot of maintenance trouble if we explicitly specify and store these role assignments. We can exploit *default roles* to solve this problem.

A default role is a role that is implicitly assigned to every user, including both authenticated and guest. We do not need to explicitly assign it to a user. When CWebUser::checkAccess is invoked, default roles will be checked first as if they are assigned to the user.

Default roles must be declared in the CAuthManager::defaultRoles property. For example, the following configuration declares two roles to be default roles: `authenticated` and `guest`.

```php
return array(
    'components'=>array(
        'authManager'=>array(
            'class'=>'CDbAuthManager',
            'defaultRoles'=>array('authenticated', 'guest'),
        ),
    ),
);
```

Because a default role is assigned to every user, it usually needs to be associated with a business rule that determines whether the role really applies to the user. For example, the following code defines two roles, `authenticated` and `guest`, which effectively apply to authenticated users and guest users, respectively.

```php
$bizRule='return !Yii::app()->user->isGuest;';
$auth->createRole('authenticated', 'authenticated user', $bizRule);

$bizRule='return Yii::app()->user->isGuest;';
$auth->createRole('guest', 'guest user', $bizRule);
```

8.4   Theming

Theming is a systematic way of customizing the outlook of pages in a Web application. By applying a new theme, the overall appearance of a Web application can be changed instantly and dramatically.

In Yii, each theme is represented as a directory consisting of view files, layout files, and relevant resource files such as images, CSS files, JavaScript files, etc. The name of a theme is its directory name. All themes reside under the same directory `WebRoot/themes`. At any time, only one theme can be active.

> **Tip:** The default theme root directory `WebRoot/themes` can be configured to be a different one. Simply configure the basePath and the baseUrl properties of the themeManager application component to be the desired ones.

To activate a theme, set the theme property of the Web application to be the name of the desired theme. This can be done either in the application configuration or during runtime in controller actions.

> **Note:** Theme name is case-sensitive. If you attempt to activate a theme that does not exist, `Yii::app()->theme` will return `null`.

Contents under a theme directory should be organized in the same way as those under the application base path. For example, all view files must be located under `views`, layout view files under `views/layouts`, and system view files under `views/system`. For example, if we want to replace the `create` view of `PostController` with a view in the `classic` theme, we should save the new view file as `WebRoot/themes/classic/views/post/create.php`.

For views belonging to controllers in a module, the corresponding themed view files should also be placed under the `views` directory. For example, if the aforementioned `PostController` is in a module named `forum`, we should save the `create` view file as `WebRoot/themes/classic/views/forum/post/create.php`. If the `forum` module is nested in another module named `support`, then the view file should be `WebRoot/themes/classic/views/support/forum/post/create.php`.

> **Note:** Because the `views` directory may contain security-sensitive data, it should be configured to prevent from being accessed by Web users.

When we call render or renderPartial to display a view, the corresponding view file as well as the layout file will be looked for in the currently active theme. And if found, those files will be rendered. Otherwise, it falls back to the default location as specified by viewPath and layoutPath.

> **Tip:** Inside a theme view, we often need to link other theme resource files. For example, we may want to show an image file under the theme's `images` directory. Using the baseUrl property of the currently active theme, we can generate the URL for the image as follows,
>
> ```
> Yii::app()->theme->baseUrl . '/images/FileName.gif'
> ```

Below is an example of directory organization for an application with two themes `basic` and `fancy`.

```
WebRoot/
    assets
    protected/
        .htaccess
        components/
        controllers/
        models/
        views/
            layouts/
                main.php
            site/
                index.php
    themes/
        basic/
            views/
                .htaccess
                layouts/
                    main.php
                site/
                    index.php
        fancy/
            views/
                .htaccess
                layouts/
                    main.php
```

```
        site/
            index.php
```

In the application configuration, if we configure

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

then the `basic` theme will be in effect, which means the application's layout will use the one under the directory `themes/basic/views/layouts`, and the site's index view will use the one under `themes/basic/views/site`. In case a view file is not found in the theme, it will fall back to the one under the `protected/views` directory.

8.4.1 Customizing Widgets Globally

> **Note:** this feature has been available since version 1.1.3.

When using a widget provided by third party or Yii, we often need to customize it for specific needs. For example, we may want to change the value of CLinkPager::maxButtonCount from 10 (default) to 5. We can accomplish this by passing the initial property values when calling CBaseController::widget to create a widget. However, it becomes troublesome to do so if we have to repeat the same customization in every place we use CLinkPager.

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>5,
    'cssFile'=>false,
));
```

Using the global widget customization feature, we only need to specify these initial values in a single place, i.e., the application configuration. This makes the customization of widgets more manageable.

To use the global widget customization feature, we need to configure the widgetFactory as follows:

```
return array(
    'components'=>array(
```

```
    'widgetFactory'=>array(
        'widgets'=>array(
            'CLinkPager'=>array(
                'maxButtonCount'=>5,
                'cssFile'=>false,
            ),
            'CJuiDatePicker'=>array(
                'language'=>'ru',
            ),
        ),
    ),
);
```

In the above, we specify the global widget customization for both CLinkPager and CJui-DatePicker widgets by configuring the CWidgetFactory::widgets property. Note that the global customization for each widget is represented as a key-value pair in the array, where the key refers to the wiget class name while the value specifies the initial property value array.

Now, whenever we create a CLinkPager widget in a view, the above property values will be assigned to the widget, and we only need to write the following code in the view to create the widget:

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
));
```

We can still override the initial property values when necessary. For example, if in some view we want to set `maxButtonCount` to be 2, we can do the following:

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>2,
));
```

8.4.2   Skin

> **Note:** The skin feature has been available since version 1.1.0.

While using a theme we can quickly change the outlook of views, we can use skins to systematically customize the outlook of the widgets used in the views.

A skin is an array of name-value pairs that can be used to initialize the properties of a widget. A skin belongs to a widget class, and a widget class can have multiple skins identified by their names. For example, we can have a skin for the CLinkPager widget and the skin is named as `classic`.

In order to use the skin feature, we first need to modify the application configuration by configuring the CWidgetFactory::enableSkin property to be true for the `widgetFactory` application component:

```
return array(
    'components'=>array(
        'widgetFactory'=>array(
            'enableSkin'=>true,
        ),
    ),
);
```

Please note that in versions prior to 1.1.3, you need to use the following configuration to enable widget skinning:

```
return array(
    'components'=>array(
        'widgetFactory'=>array(
            'class'=>'CWidgetFactory',
        ),
    ),
);
```

We then create the needed skins. Skins belonging to the same widget class are stored in a single PHP script file whose name is the widget class name. All these skin files are stored under `protected/views/skins`, by default. If you want to change this to be a different directory, you may configure the `skinPath` property of the `widgetFactory` component. As an example, we may create under `protected/views/skins` a file named `CLinkPager.php` whose content is as follows,

```
<?php
return array(
    'default'=>array(
        'nextPageLabel'=>' ',
        'prevPageLabel'=>'',
    ),
    'classic'=>array(
        'header'=>'',
```

```
        'maxButtonCount'=>5,
    ),
);
```

In the above, we create two skins for the CLinkPager widget: `default` and `classic`. The former is the skin that will be applied to any CLinkPager widget that we do not explicitly specify its `skin` property, while the latter is the skin to be applied to a CLinkPager widget whose `skin` property is specified as `classic`. For example, in the following view code, the first pager will use the `default` skin while the second the `classic` skin:

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

<?php $this->widget('CLinkPager', array('skin'=>'classic')); ?>
```

If we create a widget with a set of initial property values, they will take precedence and be merged with any applicable skin. For example, the following view code will create a pager whose initial values will be `array('header'=>'', 'maxButtonCount'=>6, 'cssFile'=>false)`, which is the result of merging the initial property values specified in the view and the `classic` skin.

```php
<?php $this->widget('CLinkPager', array(
    'skin'=>'classic',
    'maxButtonCount'=>6,
    'cssFile'=>false,
)); ?>
```

Note that the skin feature does NOT require using themes. However, when a theme is active, Yii will also look for skins under the `skins` directory of the theme's view directory (e.g. `WebRoot/themes/classic/views/skins`). In case a skin with the same name exists in both the theme and the main application view directories, the theme skin will take precedence.

If a widget is using a skin that does not exist, Yii will still create the widget as usual without any error.

> **Info:** Using skin may degrade the performance because Yii needs to look for the skin file the first time a widget is being created.

Skin is very similar to the global widget customization feature. The main differences are as follows.

- Skin is more related with the customization of presentational property values;

- A widget can have multiple skins;

- Skin is themeable;

- Using skin is more expensive than using global widget customization.

8.5   Logging

Yii provides a flexible and extensible logging feature. Messages logged can be classified according to log levels and message categories. Using level and category filters, selected messages can be further routed to different destinations, such as files, emails, browser windows, etc.

8.5.1   Message Logging

Messages can be logged by calling either Yii::log or Yii::trace. The difference between these two methods is that the latter logs a message only when the application is in debug mode.

```
Yii::log($message, $level, $category);
Yii::trace($message, $category);
```

When logging a message, we need to specify its category and level. Category is a string in the format of xxx.yyy.zzz which resembles to the path alias. For example, if a message is logged in CController, we may use the category system.web.CController. Message level should be one of the following values:

- trace: this is the level used by Yii::trace. It is for tracing the execution flow of the application during development.

- info: this is for logging general information.

- profile: this is for performance profile which is to be described shortly.

- warning: this is for warning messages.

- error: this is for fatal error messages.

8.5.2   Message Routing

Messages logged using Yii::log or Yii::trace are kept in memory. We usually need to display them in browser windows, or save them in some persistent storage such as files, emails. This is called *message routing*, i.e., sending messages to different destinations.

In Yii, message routing is managed by a CLogRouter application component. It manages a set of the so-called *log routes*. Each log route represents a single log destination. Messages sent along a log route can be filtered according to their levels and categories.

To use message routing, we need to install and preload a CLogRouter application component. We also need to configure its routes property with the log routes that we want. The following shows an example of the needed application configuration:

```
array(
    ......
    'preload'=>array('log'),
    'components'=>array(
        ......
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'trace, info',
                    'categories'=>'system.*',
                ),
                array(
                    'class'=>'CEmailLogRoute',
                    'levels'=>'error, warning',
                    'emails'=>'admin@example.com',
                ),
            ),
        ),
    ),
)
```

In the above example, we have two log routes. The first route is CFileLogRoute which saves messages in a file under the application runtime directory. Only messages whose level is `trace` or `info` and whose category starts with `system.` are saved. The second route is CEmailLogRoute which sends messages to the specified email addresses. Only messages whose level is `error` or `warning` are sent.

The following log routes are available in Yii:

- CDbLogRoute: saves messages in a database table.

- CEmailLogRoute: sends messages to specified email addresses.

- CFileLogRoute: saves messages in a file under the application runtime directory.

- CWebLogRoute: displays messages at the end of the current Web page.

- CProfileLogRoute: displays profiling messages at the end of the current Web page.

> **Info:** Message routing occurs at the end of the current request cycle when the onEndRequest event is raised. To explicitly terminate the processing of the current request, call CApplication::end() instead of `die()` or `exit()`, because CApplication::end() will raise the onEndRequest event so that the messages can be properly logged.

Message Filtering

As we mentioned, messages can be filtered according to their levels and categories before they are sent long a log route. This is done by setting the levels and categories properties of the corresponding log route. Multiple levels or categories should be concatenated by commas.

Because message categories are in the format of xxx.yyy.zzz, we may treat them as a category hierarchy. In particular, we say xxx is the parent of xxx.yyy which is the parent of xxx.yyy.zzz. We can then use xxx.* to represent category xxx and all its child and grandchild categories.

Logging Context Information

Starting from version 1.0.6, we can specify to log additional context information, such as PHP predefined variables (e.g. $_GET, $_SERVER), session ID, user name, etc. This is accomplished by specifying the CLogRoute::filter property of a log route to be a suitable log filter.

The framework comes with the convenient CLogFilter that may be used as the needed log filter in most cases. By default, CLogFilter will log a message with variables like $_GET, $_SERVER which often contains valuable system context information. CLogFilter can also be configured to prefix each logged message with session ID, username, etc., which may greatly simplifying the global search when we are checking the numerous logged messages.

The following configuration shows how to enable logging context information. Note that each log route may have its own log filter. And by default, a log route does not have a log filter.

```
array(
    ......
    'preload'=>array('log'),
    'components'=>array(
        ......
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error',
                    'filter'=>'CLogFilter',
                ),
                ...other log routes...
            ),
        ),
    ),
)
```

Starting from version 1.0.7, Yii supports logging call stack information in the messages that are logged by calling `Yii::trace`. This feature is disabled by default because it lowers performance. To use this feature, simply define a constant named `YII_TRACE_LEVEL` at the beginning of the entry script (before including `yii.php`) to be an integer greater than 0. Yii will then append to every trace message with the file name and line number of the call stacks belonging to application code. The number `YII_TRACE_LEVEL` determines how many layers of each call stack should be recorded. This information is particularly useful during development stage as it can help us identify the places that trigger the trace messages.

8.5.3 Performance Profiling

Performance profiling is a special type of message logging. Performance profiling can be used to measure the time needed for the specified code blocks and find out what the performance bottleneck is.

To use performance profiling, we need to identify which code blocks need to be profiled. We mark the beginning and the end of each code block by inserting the following methods:

```
Yii::beginProfile('blockID');
...code block being profiled...
Yii::endProfile('blockID');
```

where `blockID` is an ID that uniquely identifies the code block.

Note, code blocks need to be nested properly. That is, a code block cannot intersect with another. It must be either at a parallel level or be completely enclosed by the other code block.

To show profiling result, we need to install a CLogRouter application component with a CProfileLogRoute log route. This is the same as we do with normal message routing. The CProfileLogRoute route will display the performance results at the end of the current page.

Profiling SQL Executions

Profiling is especially useful when working with database since SQL executions are often the main performance bottleneck of an application. While we can manually insert `beginProfile` and `endProfile` statements at appropriate places to measure the time spent in each SQL execution, starting from version 1.0.6, Yii provides a more systematic approach to solve this problem.

By setting CDbConnection::enableProfiling to be true in the application configuration, every SQL statement being executed will be profiled. The results can be readily displayed using the aforementioned CProfileLogRoute, which can show us how much time is spent in executing what SQL statement. We can also call CDbConnection::getStats() to retrieve the total number SQL statements executed and their total execution time.

8.6   Error Handling

Yii provides a complete error handling framework based on the PHP 5 exception mechanism. When the application is created to handle an incoming user request, it registers its handleError method to handle PHP warnings and notices; and it registers its handleException method to handle uncaught PHP exceptions. Consequently, if a PHP warning/notice or an uncaught exception occurs during the application execution, one of the error handlers will take over the control and start the necessary error handling procedure.

> **Tip:** The registration of error handlers is done in the application's constructor by calling PHP functions set_exception_handler and set_error_handler. If you do not want Yii to handle the errors and exceptions, you may define constant YII_ENABLE_ERROR_HANDLER and YII_ENABLE_EXCEPTION_HANDLER to be false in the entry script.

By default, errorHandler (or exceptionHandler) will raise an onError event (or onException event). If the error (or exception) is not handled by any event handler, it will call for help from the errorHandler application component.

8.6.1 Raising Exceptions

Raising exceptions in Yii is not different from raising a normal PHP exception. One uses the following syntax to raise an exception when needed:

```
throw new ExceptionClass('ExceptionMessage');
```

Yii defines two exception classes: CException and CHttpException. The former is a generic exception class, while the latter represents an exception that should be displayed to end users. The latter also carries a statusCode property representing an HTTP status code. The class of an exception determines how it should be displayed, as we will explain next.

> **Tip:** Raising a CHttpException exception is a simple way of reporting errors caused by user misoperation. For example, if the user provides an invalid post ID in the URL, we can simply do the following to show a 404 error (page not found):

```
// if post ID is invalid
throw new CHttpException(404,'The specified post cannot be found.');
```

8.6.2 Displaying Errors

When an error is forwarded to the CErrorHandler application component, it chooses an appropriate view to display the error. If the error is meant to be displayed to end users, such as a CHttpException, it will use a view named `errorXXX`, where `XXX` stands for the HTTP status code (e.g. 400, 404, 500). If the error is an internal one and should only be displayed to developers, it will use a view named `exception`. In the latter case, complete call stack as well as the error line information will be displayed.

> **Info:** When the application runs in production mode, all errors including those internal ones will be displayed using view `errorXXX`. This is because the call stack of an error may contain sensitive information. In this case, developers should rely on the error logs to determine what is the real cause of an error.

CErrorHandler searches for the view file corresponding to a view in the following order:

1. `WebRoot/themes/ThemeName/views/system`: this is the `system` view directory under the currently active theme.

2. `WebRoot/protected/views/system`: this is the default `system` view directory for an application.

3. `yii/framework/views`: this is the standard system view directory provided by the Yii framework.

Therefore, if we want to customize the error display, we can simply create error view files under the system view directory of our application or theme. Each view file is a normal PHP script consisting of mainly HTML code. For more details, please refer to the default view files under the framework's `view` directory.

Handling Errors Using an Action

Starting from version 1.0.6, Yii allows using a controller action to handle the error display work. To do so, we should configure the error handler in the application configuration as follows:

```php
return array(
    ......
    'components'=>array(
        'errorHandler'=>array(
            'errorAction'=>'site/error',
        ),
    ),
);
```

In the above, we configure the CErrorHandler::errorAction property to be the route `site/error` which refers to the `error` action in `SiteController`. We may use a different route if needed.

We can write the `error` action like the following:

```php
public function actionError()
{
    if($error=Yii::app()->errorHandler->error)
        $this->render('error', $error);
}
```

In the action, we first retrieve the detailed error information from CErrorHandler::error. If it is not empty, we render the `error` view together with the error information. The error information returned from CErrorHandler::error is an array with the following fields:

- `code`: the HTTP status code (e.g. 403, 500);

- `type`: the error type (e.g. CHttpException, `PHP Error`);

- `message`: the error message;

- `file`: the name of the PHP script file where the error occurs;

- `line`: the line number of the code where the error occurs;

- `trace`: the call stack of the error;

- `source`: the context source code where the error occurs.

> **Tip:** The reason we check if CErrorHandler::error is empty or not is because the `error` action may be directly requested by an end user, in which case there is no error. Since we are passing the `$error` array to the view, it will be automatically expanded to individual variables. As a result, in the view we can access directly the variables such as `$code`, `$type`.

8.6.3   Message Logging

A message of level `error` will always be logged when an error occurs. If the error is caused by a PHP warning or notice, the message will be logged with category `php`; if the error is caused by an uncaught exception, the category would be `exception.ExceptionClassName` (for CHttpException its statusCode will also be appended to the category). One can thus exploit the logging feature to monitor errors happened during application execution.

8.7   Web Service

Web service is a software system designed to support interoperable machine-to-machine interaction over a network. In the context of Web applications, it usually refers to a set of APIs that can be accessed over the Internet and executed on a remote system hosting the requested service. For example, a Flex-based client may invoke a function implemented on the server side running a PHP-based Web application. Web service relies on SOAP as its foundation layer of the communication protocol stack.

Yii provides CWebService and CWebServiceAction to simplify the work of implementing Web service in a Web application. The APIs are grouped into classes, called *service providers*. Yii will generate for each class a WSDL specification which describes what APIs are available and how they should be invoked by client. When an API is invoked by a client, Yii will instantiate the corresponding service provider and call the requested API to fulfill the request.

> **Note:** CWebService relies on the PHP SOAP extension. Make sure you have enabled it before trying the examples displayed in this section.

8.7.1   Defining Service Provider

As we mentioned above, a service provider is a class defining the methods that can be remotely invoked. Yii relies on doc comment and class reflection to identify which methods can be remotely invoked and what are their parameters and return value.

Let's start with a simple stock quoting service. This service allows a client to request for the quote of the specified stock. We define the service provider as follows. Note that we define the provider class StockController by extending CController. This is not required. We will explain why we do so shortly.

```php
class StockController extends CController
{
    /**
     * @param string the symbol of the stock
     * @return float the stock price
     * @soap
     */
    public function getPrice($symbol)
    {
        $prices=array('IBM'=>100, 'GOOGLE'=>350);
        return isset($prices[$symbol])?$prices[$symbol]:0;
        //...return stock price for $symbol
    }
}
```

In the above, we declare the method getPrice to be a Web service API by marking it with the tag @soap in its doc comment. We rely on doc comment to specify the data type of the input parameters and return value. Additional APIs can be declared in the similar way.

8.7.2 Declaring Web Service Action

Having defined the service provider, we need to make it available to clients. In particular, we want to create a controller action to expose the service. This can be done easily by declaring a CWebServiceAction action in a controller class. For our example, we will just put it in `StockController`.

```php
class StockController extends CController
{
    public function actions()
    {
        return array(
            'quote'=>array(
                'class'=>'CWebServiceAction',
            ),
        );
    }

    /**
     * @param string the symbol of the stock
     * @return float the stock price
     * @soap
     */
    public function getPrice($symbol)
    {
        //...return stock price for $symbol
    }
}
```

That is all we need to create a Web service! If we try to access the action by URL `http://hostname/path/to/index.php?r=stock/quote`, we will see a lot of XML content which is actually the WSDL for the Web service we defined.

> **Tip:** By default, CWebServiceAction assumes the current controller is the service provider. That is why we define the `getPrice` method inside the `StockController` class.

8.7.3 Consuming Web Service

To complete the example, let's create a client to consume the Web service we just created. The example client is written in PHP, but it could be in other languages, such as `Java`, `C#`, `Flex`, etc.

```php
$client=new SoapClient('http://hostname/path/to/index.php?r=stock/quote');
```

```
echo $client->getPrice('GOOGLE');
```

Run the above script in either Web or console mode, and we shall see 350 which is the price for GOOGLE.

8.7.4   Data Types

When declaring class methods and properties to be remotely accessible, we need to specify the data types of the input and output parameters. The following primitive data types can be used:

- str/string: maps to xsd:string;

- int/integer: maps to xsd:int;

- float/double: maps to xsd:float;

- bool/boolean: maps to xsd:boolean;

- date: maps to xsd:date;

- time: maps to xsd:time;

- datetime: maps to xsd:dateTime;

- array: maps to xsd:string;

- object: maps to xsd:struct;

- mixed: maps to xsd:anyType.

If a type is not any of the above primitive types, it is considered as a composite type consisting of properties. A composite type is represented in terms of a class, and its properties are the class' public member variables marked with @soap in their doc comments.

We can also use array type by appending [] to the end of a primitive or composite type. This would specify an array of the specified type.

Below is an example defining the getPosts Web API which returns an array of Post objects.

```
class PostController extends CController
{
    /**
     * @return Post[] a list of posts
```

```php
     * @soap
     */
    public function getPosts()
    {
        return Post::model()->findAll();
    }
}

class Post extends CActiveRecord
{
    /**
     * @var integer post ID
     * @soap
     */
    public $id;
    /**
     * @var string post title
     * @soap
     */
    public $title;

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

8.7.5 Class Mapping

In order to receive parameters of composite type from client, an application needs to declare the mapping from WSDL types to the corresponding PHP classes. This is done by configuring the classMap property of CWebServiceAction.

```php
class PostController extends CController
{
    public function actions()
    {
        return array(
            'service'=>array(
                'class'=>'CWebServiceAction',
                'classMap'=>array(
                    'Post'=>'Post',  // or simply 'Post'
                ),
            ),
        );
    }
    ......
}
```

8.7.6 Intercepting Remote Method Invocation

By implementing the IWebServiceProvider interface, a sevice provider can intercept remote method invocations. In IWebServiceProvider::beforeWebMethod, the provider may retrieve the current CWebService instance and obtain the the name of the method currently being requested via CWebService::methodName. It can return false if the remote method should not be invoked for some reason (e.g. unauthorized access).

8.8 Internationalization

Internationalization (I18N) refers to the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. For Web applications, this is of particular importance because the potential users may be from worldwide.

Yii provides support for I18N in several aspects.

- It provides the locale data for each possible language and variant.

- It provides message and file translation service.

- It provides locale-dependent date and time formatting.

- It provides locale-dependent number formatting.

In the following subsections, we will elaborate each of the above aspects.

8.8.1 Locale and Language

Locale is a set of parameters that defines the user's language, country and any special variant preferences that the user wants to see in their user interface. It is usually identified by an ID consisting of a language ID and a region ID. For example, the ID `en_US` stands for the locale of English and United States. For consistency, all locale IDs in Yii are canonicalized to the format of `LanguageID` or `LanguageID_RegionID` in lower case (e.g. `en`, `en_us`).

Locale data is represented as a CLocale instance. It provides locale-dependent information, including currency symbols, number symbols, currency formats, number formats, date and time formats, and date-related names. Since the language information is already implied in the locale ID, it is not provided by CLocale. For the same reason, we often interchangeably using the term locale and language.

Given a locale ID, one can get the corresponding CLocale instance by CLocale::getInstance($localeID) or CApplication::getLocale($localeID).

> **Info:** Yii comes with locale data for nearly every language and region. The data is obtained from Common Locale Data Repository (CLDR). For each locale, only a subset of the CLDR data is provided as the original data contains a lot of rarely used information. Starting from version 1.1.0, users can also supply their own customized locale data. To do so, configure the CApplication::localeDataPath property with the directory that contains the customized locale data. Please refer to the locale data files under `framework/i18n/data` in order to create customized locale data files.

For a Yii application, we differentiate its target language from source language. The target language is the language (locale) of the users that the application is targeted at, while the source language refers to the language (locale) that the application source files are written in. Internationalization occurs only when the two languages are different.

One can configure target language in the application configuration, or change it dynamically before any internationalization occurs.

> **Tip:** Sometimes, we may want to set the target language as the language preferred by a user (specified in user's browser preference). To do so, we can retrieve the user preferred language ID using CHttpRequest::preferredLanguage.

8.8.2 Translation

The most needed I18N feature is perhaps translation, including message translation and view translation. The former translates a text message to the desired language, while the latter translates a whole file to the desired language.

A translation request consists of the object to be translated, the source language that the object is in, and the target language that the object needs to be translated to. In Yii, the source language is default to the application source language while the target language is default to the application language. If the source and target languages are the same, translation will not occur.

Message Translation

Message translation is done by calling Yii::t(). The method translates the given message from source language to target language.

When translating a message, its category has to be specified since a message may be translated differently under different categories (contexts). The category yii is reserved for messages used by the Yii framework core code.

Messages can contain parameter placeholders which will be replaced with the actual parameter values when calling Yii::t(). For example, the following message translation request would replace the {alias} placeholder in the original message with the actual alias value.

```
Yii::t('app', 'Path alias "{alias}" is redefined.',
    array('{alias}'=>$alias))
```

> **Note:** Messages to be translated must be constant strings. They should not contain variables that would change message content (e.g. "Invalid {$message} content."). Use parameter placeholders if a message needs to vary according to some parameters.

Translated messages are stored in a repository called *message source*. A message source is represented as an instance of CMessageSource or its child class. When Yii::t() is invoked, it will look for the message in the message source and return its translated version if it is found.

Yii comes with the following types of message sources. You may also extend CMessageSource to create your own message source type.

- CPhpMessageSource: the message translations are stored as key-value pairs in a PHP array. The original message is the key and the translated message is the value. Each array represents the translations for a particular category of messages and is stored in a separate PHP script file whose name is the category name. The PHP translation files for the same language are stored under the same directory named as the locale ID. And all these directories are located under the directory specified by basePath.

- CGettextMessageSource: the message translations are stored as GNU Gettext files.

- CDbMessageSource: the message translations are stored in database tables. For more details, see the API documentation for CDbMessageSource.

A message source is loaded as an application component. Yii pre-declares an application component named messages to store messages that are used in user application. By default,

the type of this message source is CPhpMessageSource and the base path for storing the PHP translation files is `protected/messages`.

In summary, in order to use message translation, the following steps are needed:

1. Call Yii::t() at appropriate places;

2. Create PHP translation files as `protected/messages/LocaleID/CategoryName.php`. Each file simply returns an array of message translations. Note, this assumes you are using the default CPhpMessageSource to store the translated messages.

3. Configure CApplication::sourceLanguage and CApplication::language.

> **Tip:** The `yiic` tool in Yii can be used to manage message translations when CPhpMessageSource is used as the message source. Its `message` command can automatically extract messages to be translated from selected source files and merge them with existing translations if necessary.

Starting from version 1.0.10, when using CPhpMessageSource to manage message source, messages for an extension class (e.g. a widget, a module) can be specially managed and used. In particular, if a message belongs to an extension whose class name is `Xyz`, then the message category can be specified in the format of `Xyz.categoryName`. The corresponding message file will be assumed to be `BasePath/messages/LanguageID/categoryName.php`, where `BasePath` refers to the directory that contains the extension class file. And when using `Yii::t()` to translate an extension message, the following format should be used, instead:

```
Yii::t('Xyz.categoryName', 'message to be translated')
```

Since version 1.0.2, Yii has added the support for choice format. Choice format refers to choosing a translated according to a given number value. For example, in English the word 'book' may either take a singular form or a plural form depending on the number of books, while in other languages, the word may not have different form (such as Chinese) or may have more complex plural form rules (such as Russian). Choice format solves this problem in a simple yet effective way.

To use choice format, a translated message must consist of a sequence of expression-message pairs separated by `|`, as shown below:

```
'expr1#message1|expr2#message2|expr3#message3'
```

where `exprN` refers to a valid PHP expression which evaluates to a boolean value indicating whether the corresponding message should be returned. Only the message corresponding to the first expression that evaluates to true will be returned. An expression can contain a special variable named `n` (note, it is not `$n`) which will take the number value passed as the first message parameter. For example, assuming a translated message is:

```
'n==1#one book|n>1#many books'
```

and we are passing a number value 2 in the message parameter array when calling Yii::t(), we would obtain `many books` as the final translated message.

As a shortcut notation, if an expression is a number, it will be treated as `n==Number`. Therefore, the above translated message can be also be written as:

```
'1#one book|n>1#many books'
```

File Translation

File translation is accomplished by calling CApplication::findLocalizedFile(). Given the path of a file to be translated, the method will look for a file with the same name under the `LocaleID` subdirectory. If found, the file path will be returned; otherwise, the original file path will be returned.

File translation is mainly used when rendering a view. When calling one of the render methods in a controller or widget, the view files will be translated automatically. For example, if the target language is `zh_cn` while the source language is `en_us`, rendering a view named `edit` would resulting in searching for the view file `protected/views/ControllerID/zh_cn/edit.php`. If the file is found, this translated version will be used for rendering; otherwise, the file `protected/views/ControllerID/edit.php` will be rendered instead.

File translation may also be used for other purposes, for example, displaying a translated image or loading a locale-dependent data file.

8.8.3   Date and Time Formatting

Date and time are often in different formats in different countries or regions. The task of date and time formatting is thus to generate a date or time string that fits for the specified locale. Yii provides CDateFormatter for this purpose.

Each CDateFormatter instance is associated with a target locale. To get the formatter associated with the target locale of the whole application, we can simply access the

dateFormatter property of the application.

The CDateFormatter class mainly provides two methods to format a UNIX timestamp.

- format: this method formats the given UNIX timestamp into a string according to a customized pattern (e.g. `$dateFormatter->format('yyyy-MM-dd',$timestamp)`).

- formatDateTime: this method formats the given UNIX timestamp into a string according to a pattern predefined in the target locale data (e.g. `short` format of date, `long` format of time).

8.8.4   Number Formatting

Like data and time, numbers may also be formatted differently in different countries or regions. Number formatting includes decimal formatting, currency formatting and percentage formatting. Yii provides CNumberFormatter for these tasks.

To get the number formatter associated with the target locale of the whole application, we can access the numberFormatter property of the application.

The following methods are provided by CNumberFormatter to format an integer or double value.

- format: this method formats the given number into a string according to a customized pattern (e.g. `$numberFormatter->format('#,##0.00',$number)`).

- formatDecimal: this method formats the given number using the decimal pattern predefined in the target locale data.

- formatCurrency: this method formats the given number and currency code using the currency pattern predefined in the target locale data.

- formatPercentage: this method formats the given number using the percentage pattern predefined in the target locale data.

8.9   Using Alternative Template Syntax

Yii allows developers to use their own favorite template syntax (e.g. Prado, Smarty) to write controller or widget views. This is achieved by writing and installing a viewRenderer application component. The view renderer intercepts the invocations of CBaseController::renderFile, compiles the view file with customized template syntax, and renders the compiling results.

> **Info:** It is recommended to use customized template syntax only when writing views that are less likely to be reused. Otherwise, people who are reusing the views would be forced to use the same customized template syntax in their applications.

In the following, we introduce how to use CPradoViewRenderer, a view renderer that allows developers to use the template syntax similar to that in Prado framework. For people who want to develop their own view renderers, CPradoViewRenderer is a good reference.

8.9.1 Using `CPradoViewRenderer`

To use CPradoViewRenderer, we just need to configure the application as follows:

```php
return array(
    'components'=>array(
        ......,
        'viewRenderer'=>array(
            'class'=>'CPradoViewRenderer',
        ),
    ),
);
```

By default, CPradoViewRenderer will compile source view files and save the resulting PHP files under the runtime directory. Only when the source view files are changed, will the PHP files be re-generated. Therefore, using CPradoViewRenderer incurs very little performance degradation.

> **Tip:** While CPradoViewRenderer mainly introduces some new template tags to make writing views easier and faster, you can still write PHP code as usual in the source views.

In the following, we introduce the template tags that are supported by CPradoViewRenderer.

Short PHP Tags

Short PHP tags are shortcuts to writing PHP expressions and statements in a view. The expression tag `<%= expression %>` is translated into `<?php echo expression ?>`; while the statement tag `<% statement %>` to `<?php statement ?>`. For example,

```
<%= CHtml::textField($name,'value'); %>
<% foreach($models as $model): %>
```

is translated into

```
<?php echo CHtml::textField($name,'value'); ?>
<?php foreach($models as $model): ?>
```

Component Tags

Component tags are used to insert a widget in a view. It uses the following syntax:

```
<com:WidgetClass property1=value1 property2=value2 ...>
    // body content for the widget
</com:WidgetClass>

// a widget without body content
<com:WidgetClass property1=value1 property2=value2 .../>
```

where `WidgetClass` specifies the widget class name or class path alias, and property initial values can be either quoted strings or PHP expressions enclosed within a pair of curly brackets. For example,

```
<com:CCaptcha captchaAction="captcha" showRefreshButton={false} />
```

would be translated as

```
<?php $this->widget('CCaptcha', array(
    'captchaAction'=>'captcha',
    'showRefreshButton'=>false)); ?>
```

> **Note:** The value for `showRefreshButton` is specified as {false} instead of "false" because the latter means a string instead of a boolean.

Cache Tags

Cache tags are shortcuts to using fragment caching. Its syntax is as follows,

```
<cache:fragmentID property1=value1 property2=value2 ...>
    // content being cached
</cache:fragmentID >
```

where `fragmentID` should be an identifier that uniquely identifies the content being cached, and the property-value pairs are used to configure the fragment cache. For example,

```
<cache:profile duration={3600}>
    // user profile information here
</cache:profile >
```

would be translated as

```
<?php if($this->cache('profile', array('duration'=>3600))): ?>
    // user profile information here
<?php $this->endCache(); endif; ?>
```

Clip Tags

Like cache tags, clip tags are shortcuts to calling CBaseController::beginClip and CBaseController::endClip in a view. The syntax is as follows,

```
<clip:clipID>
    // content for this clip
</clip:clipID >
```

where `clipID` is an identifier that uniquely identifies the clip content. The clip tags will be translated as

```
<?php $this->beginClip('clipID'); ?>
    // content for this clip
<?php $this->endClip(); ?>
```

Comment Tags

Comment tags are used to write view comments that should only be visible to developers. Comment tags will be stripped off when the view is displayed to end users. The syntax for comment tags is as follows,

```
<!---
view comments that will be stripped off
--->
```

8.9.2   Mixing Template Formats

Starting from version 1.1.2, it is possible to mix the usage of some alternative template syntax with the normal PHP syntax. To do so, the CViewRenderer::fileExtension property of the installed view renderer must be configured with a value other than `.php`. For example, if the property is set as `.tpl`, then any view file ending with `.tpl` will be rendered using the installed view renderer, while all other view files ending with `.php` will be treated as normal PHP view script.

8.10   Console Applications

Console applications are mainly used by a Web application to perform offline work, such as code generation, search index compiling, email sending, etc. Yii provides a framework for writing console applications in an object-oriented and systematic way.

Yii represents each console task in terms of a command, and a console application instance is used to dispatch a command line request to an appropriate command. The application instance is created in an entry script. To execute a console task, we simply run the corresponding command on the command line as follows,

```
php entryScript.php CommandName Param0 Param1 ...
```

where `CommandName` refers to the command name which is case-insensitive, and `Param0`, `Param1` and so on are parameters to be passed to the command instance.

The entry script for a console application is usually written like the following, similar to that in a Web application,

```
defined('YII_DEBUG') or define('YII_DEBUG',true);
// include Yii bootstrap file
require_once('path/to/yii/framework/yii.php');
// create application instance and run
$configFile='path/to/config/file.php';
Yii::createConsoleApplication($configFile)->run();
```

We then create command classes which should extend from CConsoleCommand. Each command class should be named as its command name appended with `Command`. For

example, to define an `email` command, we should write an `EmailCommand` class. All command class files should be placed under the `commands` subdirectory of the application base directory.

> **Tip:** By configuring CConsoleApplication::commandMap, one can also have command classes in different naming conventions and located in different directories.

Writing a command class mainly involves implementing the CConsoleCommand::run method. Command line parameters are passed as an array to this method. Below is an example:

```
class EmailCommand extends CConsoleCommand
{
    public function run($args)
    {
        $receiver=$args[0];
        // send email to $receiver
    }
}
```

At any time in a command, we can access the console application instance via `Yii::app()`. Like a Web application instance, console application can also be configured. For example, we can configure a `db` application component to access the database. The configuration is usually specified as a PHP file and passed to the constructor of the console application class (or createConsoleApplication in the entry script).

8.10.1   Using the `yiic` Tool

We have used the `yiic` tool to create our first application. The `yiic` tool is in fact implemented as a console application whose entry script file is `framework/yiic.php`. Using `yiic`, we can accomplish tasks such as creating a Web application skeleton, generating a controller class or model class, generating code needed by CRUD operations, extracting messages to be translated, etc.

We can enhance `yiic` by adding our own customized commands. To do so, we should start with a skeleton application created using `yiic webapp` command, as described in Creating First Yii Application. The `yiic webapp` command will generate two files under the `protected` directory: `yiic` and `yiic.bat`. They are the *local* version of the `yiic` tool created specifically for the Web application.

We can then create our own commands under the `protected/commands` directory. Running the local `yiic` tool, we will see our own commands appearing together with the standard

ones. We can also create our own commands to be used in the `yiic shell` tool. To do so, just drop our command class files under the `protected/commands/shell` directory.

Starting from version 1.1.1, we can also create global commands that can be shared by all Yii applications on the same machine. To do so, define an environment variable named `YII_CONSOLE_COMMANDS` which should point to an existing directory. We then put our global command class files under this directory, and we will see these commands become available wherever we use the `yiic` tool.

8.11 Security

8.11.1 Cross-site Scripting Prevention

Cross-site scripting (also known as XSS) occurs when a web application gathers malicious data from a user. Often attackers will inject JavaScript, VBScript, ActiveX, HTML, or Flash into a vulnerable application to fool other application users and gather data from them. For example, a poorly design forum system may display user input in forum posts without any checking. An attacker can then inject a piece of malicious JavaScript code into a post so that when other users read this post, the JavaScript runs unexpectedly on their computers.

One of the most important measures to prevent XSS attacks is to check user input before displaying them. One can do HTML-encoding with the user input to achieve this goal. However, in some situations, HTML-encoding may not be preferable because it disables all HTML tags.

Yii incorporates the work of HTMLPurifier and provides developers with a useful component called CHtmlPurifier that encapsulates HTMLPurifier. This component is capable of removing all malicious code with a thoroughly audited, secure yet permissive whitelist and making sure the filtered content is standard-compliant.

The CHtmlPurifier component can be used as either a widget or a filter. When used as a widget, CHtmlPurifier will purify contents displayed in its body in a view. For example,

```php
<?php $this->beginWidget('CHtmlPurifier'); ?>
...display user-entered content here...
<?php $this->endWidget(); ?>
```

8.11.2 Cross-site Request Forgery Prevention

Cross-Site Request Forgery (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. For example, a malicious

web site has a page that contains an image tag whose `src` points to a banking site: `http://bank.example/withdraw?transfer=10000&to=someone`. If a user who has a login cookie for the banking site happens to visit this malicous page, the action of transferring 10000 dollars to someone will be executed. Contrary to cross-site, which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has for a particular user.

To prevent CSRF attacks, it is important to abide to the rule that `GET` requests should only be allowed to retrieve data rather than modify any data on the server. And for `POST` requests, they should include some random value which can be recognized by the server to ensure the form is submitted from and the result is sent back to the same origin.

Yii implements a CSRF prevention scheme to help defeat `POST`-based attacks. It is based on storing a random value in a cookie and comparing this value with the value submitted via the `POST` request.

By default, the CSRF prevention is disabled. To enable it, configure the CHttpRequest application component in the application configuration as follows,

```
return array(
    'components'=>array(
        'request'=>array(
            'enableCsrfValidation'=>true,
        ),
    ),
);
```

And to display a form, call CHtml::form instead of writing the HTML form tag directly. The CHtml::form method will embed the necessary random value in a hidden field so that it can be submitted for CSRF validation.

8.11.3   Cookie Attack Prevention

Protecting cookies from being attacked is of extreme importance, as session IDs are commonly stored in cookies. If one gets hold of a session ID, he essentially owns all relevant session information.

There are several countermeasures to prevent cookies from being attacked.

- An application can use SSL to create a secure communication channel and only pass the authentication cookie over an HTTPS connection. Attackers are thus unable to decipher the contents in the transferred cookies.

- Expire sessions appropriately, including all cookies and session tokens, to reduce the likelihood of being attacked.

- Prevent cross-site scripting which causes arbitrary code to run in a user's browser and expose his cookies.

- Validate cookie data and detect if they are altered.

Yii implements a cookie validation scheme that prevents cookies from being modified. In particular, it does HMAC check for the cookie values if cookie validation is enable.

Cookie validation is disabled by default. To enable it, configure the CHttpRequest application component in the application configuration as follows,

```php
return array(
    'components'=>array(
        'request'=>array(
            'enableCookieValidation'=>true,
        ),
    ),
);
```

To make use of the cookie validation scheme provided by Yii, we also need to access cookies through the cookies collection, instead of directly through $_COOKIES:

```php
// retrieve the cookie with the specified name
$cookie=Yii::app()->request->cookies[$name];
$value=$cookie->value;
......
// send a cookie
$cookie=new CHttpCookie($name,$value);
Yii::app()->request->cookies[$name]=$cookie;
```

8.12   Performance Tuning

Performance of Web applications is affected by many factors. Database access, file system operations, network bandwidth are all potential affecting factors. Yii has tried in every aspect to reduce the performance impact caused by the framework. But still, there are many places in the user application that can be improved to boost performance.

8.12.1   Enabling APC Extension

Enabling the PHP APC extension is perhaps the easiest way to improve the overall performance of an application. The extension caches and optimizes PHP intermediate code

and avoids the time spent in parsing PHP scripts for every incoming request.

8.12.2 Disabling Debug Mode

Disabling debug mode is another easy way to improve performance. A Yii application runs in debug mode if the constant `YII_DEBUG` is defined as true. Debug mode is useful during development stage, but it would impact performance because some components cause extra burden in debug mode. For example, the message logger may record additional debug information for every message being logged.

8.12.3 Using `yiilite.php`

When the PHP APC extension is enabled, we can replace `yii.php` with a different Yii bootstrap file named `yiilite.php` to further boost the performance of a Yii-powered application.

The file `yiilite.php` comes with every Yii release. It is the result of merging some commonly used Yii class files. Both comments and trace statements are stripped from the merged file. Therefore, using `yiilite.php` would reduce the number of files being included and avoid execution of trace statements.

Note, using `yiilite.php` without APC may actually reduce performance, because `yiilite.php` contains some classes that are not necessarily used in every request and would take extra parsing time. It is also observed that using `yiilite.php` is slower with some server configurations, even when APC is turned on. The best way to judge whether to use `yiilite.php` or not is to run a benchmark using the included `hello world` demo.

8.12.4 Using Caching Techniques

As described in the Caching section, Yii provides several caching solutions that may improve the performance of a Web application significantly. If the generation of some data takes long time, we can use the data caching approach to reduce the data generation frequency; If a portion of page remains relatively static, we can use the fragment caching approach to reduce its rendering frequency; If a whole page remains relative static, we can use the page caching approach to save the rendering cost for the whole page.

If the application is using Active Record, we should turn on the schema caching to save the time of parsing database schema. This can be done by configuring the CDbConnection::schemaCachingDuration property to be a value greater than 0.

Besides these application-level caching techniques, we can also use server-level caching solutions to boost the application performance. As a matter of fact, the APC caching we

described earlier belongs to this category. There are other server techniques, such as Zend Optimizer, eAccelerator, Squid, to name a few.

8.12.5 Database Optimization

Fetching data from database is often the main performance bottleneck in a Web application. Although using caching may alleviate the performance hit, it does not fully solve the problem. When the database contains enormous data and the cached data is invalid, fetching the latest data could be prohibitively expensive without proper database and query design.

Design index wisely in a database. Indexing can make `SELECT` queries much faster, but it may slow down `INSERT`, `UPDATE` or `DELETE` queries.

For complex queries, it is recommended to create a database view for it instead of issuing the queries inside the PHP code and asking DBMS to parse them repetitively.

Do not overuse Active Record. Although Active Record is good at modelling data in an OOP fashion, it actually degrades performance due to the fact that it needs to create one or several objects to represent each row of query result. For data intensive applications, using DAO or database APIs at lower level could be a better choice.

Last but not least, use `LIMIT` in your `SELECT` queries. This avoids fetching overwhelming data from database and exhausting the memory allocated to PHP.

8.12.6 Minimizing Script Files

Complex pages often need to include many external JavaScript and CSS files. Because each file would cause one extra round trip to the server and back, we should minimize the number of script files by merging them into fewer ones. We should also consider reducing the size of each script file to reduce the network transmission time. There are many tools around to help on these two aspects.

For a page generated by Yii, chances are that some script files are rendered by components that we do not want to modify (e.g. Yii core components, third-party components). In order to minimizing these script files, we need two steps.

> **Note:** The `scriptMap` feature described in the following has been available since version 1.0.3.

First, we declare the scripts to be minimized by configuring the scriptMap property of the

clientScript application component. This can be done either in the application configuration or in code. For example,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>'/js/all.js',
    'jquery.ajaxqueue.js'=>'/js/all.js',
    'jquery.metadata.js'=>'/js/all.js',
    ......
);
```

What the above code does is that it maps those JavaScript files to the URL /js/all.js. If any of these JavaScript files need to be included by some components, Yii will include the URL (once) instead of the individual script files.

Second, we need to use some tools to merge (and perhaps compress) the JavaScript files into a single one and save it as js/all.js.

The same trick also applies to CSS files.

We can also improve page loading speed with the help of Google AJAX Libraries API. For example, we can include jquery.js from Google servers instead of our own server. To do so, we first configure the scriptMap as follows,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>false,
    'jquery.ajaxqueue.js'=>false,
    'jquery.metadata.js'=>false,
    ......
);
```

By mapping these script files to false, we prevent Yii from generating the code to include these files. Instead, we write the following code in our pages to explicitly include the script files from Google,

```
<head>
<?php echo CGoogleApi::init(); ?>

<?php echo CHtml::script(
    CGoogleApi::load('jquery','1.3.2') . "\n" .
    CGoogleApi::load('jquery.ajaxqueue.js') . "\n" .
    CGoogleApi::load('jquery.metadata.js')
); ?>
```

```
......
</head>
```

8.13   Code Generation using Command Line Tools (depre-cated)

> **Note:** The code generators in `yiic shell` have been deprecated since version 1.1.2.
> Please use the more powerful and extensible Web-based code generators available
> in Gii, instead.

Open a command line window, and execute the commands listed as follows,

```
% cd WebRoot/testdrive
% protected/yiic shell
Yii Interactive Tool v1.1
Please type 'help' for help. Type 'exit' to quit.
>> model User tbl_user
   generate models/User.php
   generate fixtures/tbl_user.php
   generate unit/UserTest.php

The following model classes are successfully generated:
    User

If you have a 'db' database connection, you can test these models now with:
    $model=User::model()->find();
    print_r($model);

>> crud User
   generate UserController.php
   generate UserTest.php
   mkdir D:/testdrive/protected/views/user
   generate create.php
   generate update.php
   generate index.php
   generate view.php
   generate admin.php
   generate _form.php
   generate _view.php

Crud 'user' has been successfully created. You may access it via:
http://hostname/path/to/index.php?r=user
```

In the above, we use the `yiic shell` command to interact with our skeleton application.
At the prompt, we execute two sub-commands: `model User tbl_user` and `crud User`. The

former generates a model class named `User` for the `tbl_user` table, while the latter analyzes the `User` model and generates the code implementing the corresponding CRUD operations.

> **Note:** You may encounter errors like "...could not find driver", even though the requirement checker shows you have already enabled PDO and the corresponding PDO driver. If this happens, you may try to run the `yiic` tool as follows,
>
> `% php -c path/to/php.ini protected/yiic.php shell`
>
> where `path/to/php.ini` represents the correct PHP ini file.

Let's enjoy our work by browsing the following URL:

`http://hostname/testdrive/index.php?r=user`

This will display a list of user entries in the `tbl_user` table.

Click the `Create User` button on the page. We will be brought to the login page if we have not logged in before. After logged in, we see an input form that allows us to add a new user entry. Complete the form and click on the `Create` button. If there is any input error, a nice error prompt will show up which prevents us from saving the input. Back to the user list page, we should see the newly added user appearing in the list.

Repeat the above steps to add more users. Notice that user list page will automatically paginate the user entries if there are too many to be displayed in one page.

If we login as an administrator using `admin/admin`, we can view the user admin page with the following URL:

`http://hostname/testdrive/index.php?r=user/admin`

This will show us the user entries in a nice tabular format. We can click on the table header cells to sort the corresponding columns. We can click on the buttons on each row of data to view, update or delete the corresponding row of data. We can browse different pages. We can also filter and search to look for the data we are interested in.

All these nice features come without requiring us to write a single line of code!

Figure 8.1: User admin page

Figure 8.2: Create new user page