

The Definitive Guide to Yii 1.0

Qiang Xue and Xiang Wei Zhuo

Copyright 2008. All Rights Reserved.

CONTENTS

Contents	i
License	ix
1 Getting Started	1
1.1 What is Yii	1
1.1.1 Requirements	1
1.1.2 What is Yii Best for?	1
1.1.3 How is Yii Compared with Other Frameworks?	1
1.2 Installation	2
1.2.1 Requirements	2
1.3 Creating First Yii Application	2
1.3.1 Connecting to Database	5
1.3.2 Implementing CRUD Operations	6
2 Fundamentals	11
2.1 Model-View-Controller (MVC)	11
2.2 Entry Script	12
2.2.1 Debug Mode	12
2.3 Application	12
2.3.1 Application Configuration	12
2.3.2 Application Base Directory	13

2.3.3	Application Component	14
2.3.4	Core Application Components	14
2.3.5	Application Lifecycles	16
2.4	Controller	16
2.4.1	Route	17
2.4.2	Controller Instantiation	17
2.4.3	Action	17
2.4.4	Filter	19
2.5	Model	20
2.6	View	21
2.6.1	Layout	21
2.6.2	Widget	22
2.6.3	System View	23
2.7	Component	23
2.7.1	Component Property	23
2.7.2	Component Event	24
2.8	Path Alias and Namespace	25
2.9	Conventions	26
2.9.1	URL	26
2.9.2	Code	27
2.9.3	Configuration	27
2.9.4	File	27
2.9.5	Directory	28
2.10	Development Workflow	29

3	Working with Forms	31
3.1	Working with Form	31
3.2	Creating Model	31
3.2.1	Defining Model Class	32
3.2.2	Declaring Validation Rules	32
3.2.3	Triggering Validation	35
3.2.4	Retrieving Validation Errors	35
3.2.5	Attribute Labels	35
3.3	Creating Action	35
3.4	Creating Form	37
3.5	Collecting Tabular Input	38
4	Working with Databases	41
4.1	Working with Database	41
4.2	Data Access Objects (DAO)	41
4.2.1	Establishing Database Connection	42
4.2.2	Executing SQL Statements	43
4.2.3	Fetching Query Results	44
4.2.4	Using Transactions	44
4.2.5	Binding Parameters	45
4.2.6	Binding Columns	45
4.3	Active Record	46
4.3.1	Establishing DB Connection	47
4.3.2	Defining AR Class	48
4.3.3	Creating Record	49

4.3.4	Reading Record	50
4.3.5	Updating Record	52
4.3.6	Deleting Record	52
4.3.7	Data Validation	53
4.3.8	Comparing Records	54
4.3.9	Customization	54
4.3.10	Using Transaction with AR	55
4.4	Relational Active Record	55
4.4.1	Declaring Relationship	56
4.4.2	Performing Relational Query	58
4.4.3	Relational Query Options	60
5	Caching	63
5.1	Caching	63
5.2	Data Caching	64
5.2.1	Cache Dependency	65
5.3	Fragment Caching	66
5.3.1	Caching Options	67
5.3.2	Nested Caching	68
5.4	Page Caching	69
5.5	Dynamic Content	70
6	Extending Yii	73
6.1	Overview	73
6.2	Using Extensions	74

6.2.1	Application Component	74
6.2.2	Widget	75
6.2.3	Action	75
6.2.4	Filter	75
6.2.5	Controller	76
6.2.6	Validator	76
6.2.7	Console Command	77
6.2.8	Module	78
6.2.9	Generic Component	78
6.3	Creating Extensions	78
6.3.1	Application Component	79
6.3.2	Widget	79
6.3.3	Action	80
6.3.4	Filter	80
6.3.5	Controller	81
6.3.6	Validator	81
6.3.7	Console Command	81
6.3.8	Module	82
6.3.9	Generic Component	82
6.4	Using 3rd-Party Libraries	82
7	Special Topics	85
7.1	URL Management	85
7.1.1	Creating URLs	85
7.1.2	User-friendly URLs	86

7.2	Authentication and Authorization	89
7.2.1	Defining Identity Class	90
7.2.2	Login and Logout	91
7.2.3	Access Control Filter	91
7.2.4	Role-Based Access Control	94
7.3	Theme	98
7.4	Logging	99
7.4.1	Message Logging	100
7.4.2	Message Routing	100
7.4.3	Performance Profiling	102
7.5	Error Handling	102
7.5.1	Raising Exceptions	103
7.5.2	Displaying Errors	103
7.5.3	Message Logging	104
7.6	Web Service	104
7.6.1	Defining Service Provider	105
7.6.2	Declaring Web Service Action	105
7.6.3	Consuming Web Service	106
7.6.4	Data Types	107
7.6.5	Class Mapping	108
7.6.6	Intercepting Remote Method Invocation	108
7.7	Internationalization	109
7.7.1	Locale and Language	109
7.7.2	Translation	110

7.7.3	Date and Time Formatting	112
7.7.4	Number Formatting	113
7.8	Using Alternative Template Syntax	113
7.8.1	Using CPradoViewRenderer	114
7.9	Console Applications	116
7.9.1	Using the yiic Tool	118
7.10	Security	118
7.10.1	Cross-site Scripting Prevention	118
7.10.2	Cross-site Request Forgery Prevention	119
7.10.3	Cookie Attack Prevention	120
7.11	Performance Tuning	121
7.11.1	Enabling APC Extension	121
7.11.2	Disabling Debug Mode	121
7.11.3	Using yiilite.php	121
7.11.4	Using Caching Techniques	122
7.11.5	Database Optimization	122

LICENSE OF YII

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

Copyright ©2008 by Yii Software LLC. All rights reserved.

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 1

Getting Started

1.1 What is Yii

Yii is a high-performance component-based PHP framework for developing large-scale Web applications. It enables maximum reusability in Web programming and can significantly accelerate the development process. The name Yii (pronounced as /i:/) stands for *easy*, *efficient* and *extensible*.

1.1.1 Requirements

To run an 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.

1.1.2 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.

1.1.3 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.

1.2 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. An 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.

1.2.1 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/framework/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.

1.3 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 the powerful `yiic` tool which can be used to automate code creation for several purposes. We assume that `YiiRoot` is the directory where Yii is installed.

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. You may also use `php YiiRoot/framework/yiic.php` to replace `yiic`.

This will create a skeleton Yii application under the directory `WebRoot/testdrive`, where `WebRoot` stands for the document root of your Web server. The application is a good starting point for adding more features as it has all the needed directories and files.

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 two pages: the homepage and the login page. The homepage contains some nice information about the application as well as the user login status, while the login page shows us how form input and validation are done. 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.

<code>testdrive/</code>	
<code>index.php</code>	Web application entry script file
<code>assets/</code>	containing published resource files
<code>css/</code>	containing CSS files
<code>images/</code>	containing image files
<code>themes/</code>	containing application themes
<code>protected/</code>	containing protected application files
<code>yiic</code>	yiic command line script
<code>yiic.bat</code>	yiic command line script for Windows
<code>commands/</code>	containing customized 'yiic' commands
<code>shell/</code>	containing customized 'yiic shell' commands
<code>components/</code>	containing reusable user components
<code>MainMenu.php</code>	the 'MainMenu' widget class
<code>Identity.php</code>	the 'Identity' class used for authentication
<code>views/</code>	containing view files for widgets
<code>mainMenu.php</code>	the view file for 'MainMenu' widget
<code>config/</code>	containing configuration files
<code>console.php</code>	the console application configuration
<code>main.php</code>	the Web application configuration
<code>controllers/</code>	containing controller class files
<code>SiteController.php</code>	the default controller class
<code>extensions/</code>	containing third-party extensions

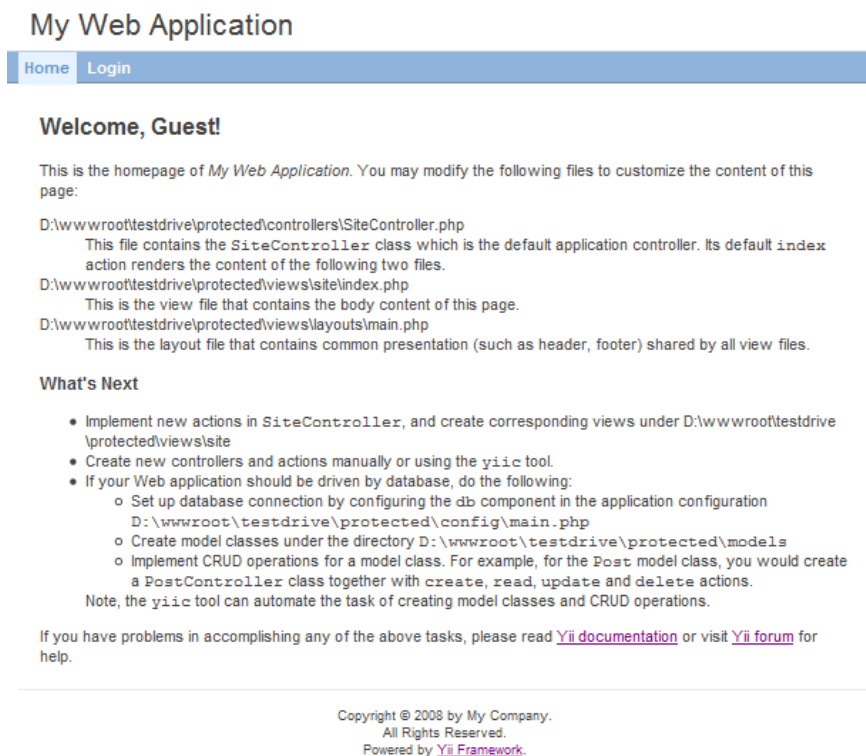


Figure 1.1: Home page

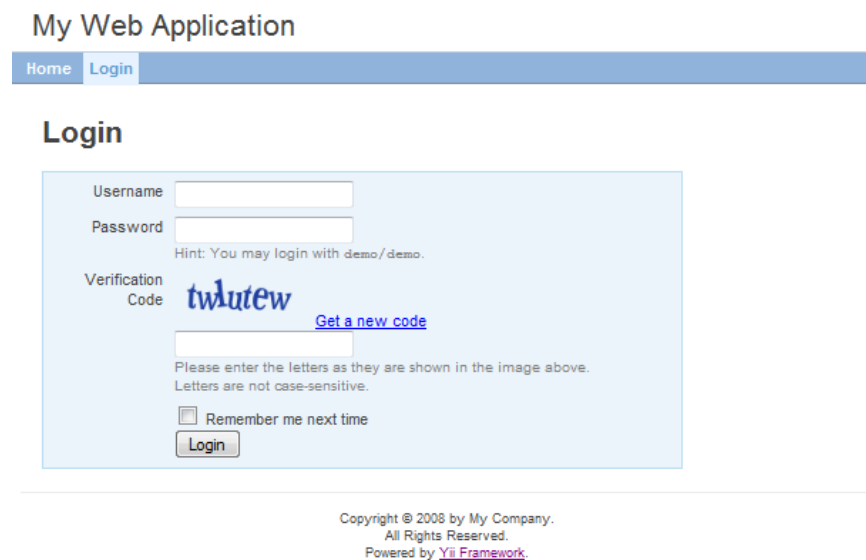
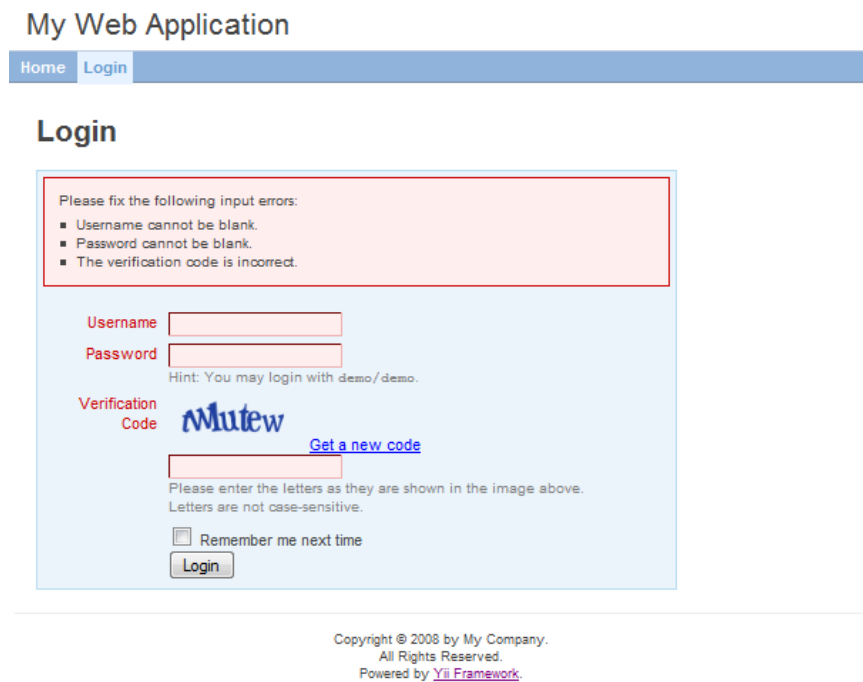


Figure 1.2: Login page



My Web Application

Home Login

Login


Please fix the following input errors:

- Username cannot be blank.
- Password cannot be blank.
- The verification code is incorrect.

Username

Password

Hint: You may login with demo/demo.

Verification Code  [Get a new code](#)

Please enter the letters as they are shown in the image above.
Letters are not case-sensitive.

☐ Remember me next time

Login

Copyright © 2008 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Figure 1.3: Login page with error

messages/	containing translated messages
models/	containing model class files
LoginForm.php	the form model for 'login' action
runtime/	containing temporarily generated files
views/	containing controller view and layout files
layouts/	containing layout view files
main.php	the default layout for all views
site/	containing view files for the 'site' controller
index.php	the view for 'index' action
login.php	the view for 'login' action
system/	containing system view files

1.3.1 Connecting to Database

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

```
return array(
    .....
```

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

In the above, we add a db entry to `components`, which instructs the application to connect to the SQLite database `WebRoot/testdrive/protected/data/source.db` when needed.

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 would need the `php_pdo` and `php_pdo_sqlite` extensions to be turned on.

To this end, we need to prepare a SQLite database so that the above configuration can be effective. Using some SQLite admin tool, we can create a database with the following schema:

```
CREATE TABLE User (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    username VARCHAR(128) NOT NULL,
    password VARCHAR(128) NOT NULL,
    email VARCHAR(128) NOT NULL
);
```

For simplicity, we only create a single `User` table in our database. The SQLite database file is saved as `WebRoot/testdrive/protected/data/source.db`. Note that both the file and the containing directory must be made writable by the Web server process, as required by SQLite.

1.3.2 Implementing CRUD Operations

Now is the fun part. We would like to implement the CRUD (create, read, update and delete) operations for the `User` table we just created. This is also commonly needed in practical applications.

Again, we use the `yiic` tool to help us generate the needed code. This process is also known as *scaffolding*.

```
% cd WebRoot/testdrive
% YiiRoot/framework/yiic shell
Yii Interactive Tool v1.0
Please type 'help' for help. Type 'exit' to quit.
>> model User
    generate User.php
```

The 'User' class has been successfully created in the following file:
D:\wwwroot\testdrive\protected\models\User.php

If you have a 'db' database connection, you can test it now with:

```
$model=User::model()->find();
print_r($model);
```

```
>> crud User
    generate UserController.php
    generate create.php
        mkdir D:\wwwroot\testdrive\protected\views\user
    generate update.php
    generate list.php
    generate show.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 `shell` command of `yiic` to interact with our newly created application. At the prompt, we can enter a valid PHP statement or expression to evaluate and print it. We can also perform tasks such as `model` and `crud` as shown above. The `model` command automatically generates a `User` model class based on the schema of the `User` table. And the `crud` command generates the controller class and views that implement the CRUD operations for the `User` model.

Note: If you make any change to your code or configuration, please re-enter `yiic shell` so that the new code or configuration can take effect. Also make sure you are using the right PHP CLI to execute `yiic`. Otherwise you may encounter errors like "...could not find driver", even though you are sure you have already enabled PDO and the corresponding database driver. The reason for such kind of errors is often because the wrong PHP CLI is using an unexpected `php.ini`.

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 `User` table. Since our table is empty, nothing will appear at the moment.

Click the `New User` link on the page. We will be brought to the login page if we have not logged in before. After logged in, we are shown with 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 be shown which prevents us from saving the input. Back to the user list, 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 a nice table of user entries. We can click on the table header cells to sort the corresponding columns. And like the user list page, the admin page also performs pagination when there are too many user entries to be displayed in one page.

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

My Web Application

[Home](#)
[Logout](#)

User List

[\[New User\]](#)

Id	Username	Password	Email	Actions
11	test11	pass11	test11@example.com	Update Delete
12	test12	pass12	test12@example.com	Update Delete
13	test13	pass13	test13@example.com	Update Delete
14	test14	pass14	test14@example.com	Update Delete
15	test15	pass15	test15@example.com	Update Delete
16	test16	pass16	test16@example.com	Update Delete
17	test17	pass17	test17@example.com	Update Delete
18	test18	pass18	test18@example.com	Update Delete
19	test19	pass19	test19@example.com	Update Delete
20	test21	pass21	test21@example.com	Update Delete

Go to page: [<<](#) [Prev](#) [1](#) [2](#) [3](#) [Next](#) [>>](#)

Copyright © 2008 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Figure 1.4: User admin page

My Web Application

[Home](#) [Logout](#)

New User

[\[User List\]](#)

Please fix the following input errors:

- Password cannot be blank.
- Email cannot be blank.

Username

Password

Email

Copyright © 2008 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Figure 1.5: Create new user page

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 typical Yii application:

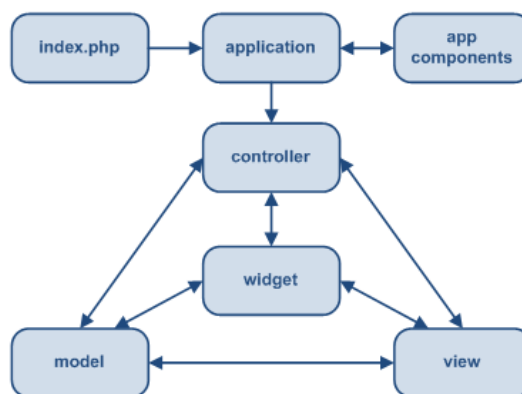


Figure 2.1: Static structure of Yii application

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 an 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

An 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.

Application is implemented as singleton. It is created in the entry script, as we have already seen. The application singleton can be accessed anywhere via `Yii::app()`. For this reason, application may be considered as the common part shared by all controllers.

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 configure its property initial values when the

application instance is being created. We can also extend `CWebApplication`, but it is less convenient.

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.

```
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,

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

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

```
$app=Yii::createWebApplication($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 a master 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 of which implements features in some aspect. For example, a [CMemCache](#) component may be plugged into the application so that caching via memcache is available anywhere in the code. As a matter of fact, application resolves the user request with the help of [CUrlManager](#) and [CHttpRequest](#) components.

Properties of application components can also be configured. This is done by configuring the [components](#) property of application. For example, we can configure [CMemCache](#) component so that it can use multiple memcache servers for caching,

```
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`).

Tip: Application components are created on demand. Therefore, performance is not affected even if we have many components plugged in. In case a component needs to be created beforehand, we can add its ID to 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`.

- `db`: `CDbConnection` - provides the database connection. Note, you must configure its `connectionString` property.
- `errorHandler`: `CErrorHandler` - handles uncaught PHP errors and exceptions.
- `securityManager`: `CSecurityManager` - provides security-related services, such as hashing, encryption.
- `statePersister`: `CStatePersister` - provides global state persistence method.
- `coreMessages`: `CPhpMessageSource` - provides translated core messages used by Yii framework.
- `messages`: `CPhpMessageSource` - provides translated messages used by Yii application.
- `urlManager`: `CUrlManager` - provides URL parsing and creation functionality.
- `request`: `CHttpRequest` - provides information related with user requests.
- `session`: `CHttpSession` - provides the session-related functionalities.
- `assetManager`: `CAssetManager` - manages the publishing of private asset files.
- `user`: `CWebUser` - represents the user session information.
- `themeManager`: `CThemeManager` - manages themes.
- `authManager`: `CAuthManager` - manages role-based access control (RBAC).
- `clientScript`: `CClientScript` - manages client scripts (javascripts and CSS).

Besides the above components, a component named `cache` is also declared but without specifying its class. The `cache` component is used by Yii for data caching purpose. When it is enabled by specifying an actual class (e.g. `CMemCache`, `CDbCache`), Yii will automatically cache some data that would otherwise take long time to generate.

2.3.5 Application Lifecycles

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

1. load application configuration;
2. initialize application with `CApplication::init()`
 - set up class autoloader and error handling;
 - load static application components;
3. raise `onBeginRequest` event;
4. process the user request:
 - resolve the user request;
 - create controller;
 - run controller;
5. 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 indicate 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:

```
class PostController extends CController
{
}
```

Since the above controller does not define any action, accessing it would throw an exception.

2.4.1 Route

Controllers and actions are identified by IDs. Controller ID is in the format of `path.to.xxx` which corresponds to the controller class file `protected/controllers/path/to/XxxController.php`, where the token `xxx` should be replaced by actual names (e.g. `post` corresponds 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`.

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.

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 defined, the user-specified controller ID will be ignored. Instead, the controller will be created based on the configuration specified by `CWebApplication::catchAllRequest`. 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.
- Otherwise, the controller class name is assumed to be `ucfirst(ControllerID).'Controller'`. For example, a 'post' controller ID corresponds to the class name `PostController`.
 - The application first searches the `controllers` directory under every activated extension for the controller class file.
 - If not found, it searches the default `protected/controllers` directory.
 - If the class file is found, it will be included and used to create the controller instance. Otherwise, a 404 `CHttpException` will be raised.

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. By doing so, we allow action classes to be reused and thus introduce more reusability.

To define a new action class, do the following:

```
class UpdateAction extends CAction
{
    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:

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

In the above, we use 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
```

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 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 done in controller 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 `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 also pass in additional local variables as follows,

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

The [render\(\)](#) method will extract the second array parameter into variables. Therefore, in the view script we can access 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 while embeds the content view in between,

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

where `$content` represents 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 `CWebApplication::layout` and `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'); ?>
```

if the widget does not need any body content.

Sometimes, we may want to configure a widget when using it. 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:

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

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

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

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

A widget can also own views, like a controller does. 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.

2.6.3 System View

System views are views used by Yii to display error 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 framework and applications are built upon components which are objects written to a specification. In Yii, a component is an instance of `CComponent` or its derived class. Using an Yii component mainly involves configuring 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) { $this->_width=$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.

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

```
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 now attach a method to this event:

```
$component->onClicked=$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:

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

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

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 by repeating the assignments. 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 true.

2.8 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. Yii predefines two aliases: `system` and `application`. The former is the alias of the Yii framework directory, while the latter is the alias of the application's `protected` directory. We can call `YiiBase::setPathOfAlias()` to define new path aliases (including root aliases).

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.

2.9 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 an Yii application is installed at.

2.9.1 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](#).

2.9.2 Code

Yii recommends naming variables, functions and class types in camel case which capitalizes 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`).

2.9.3 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.

2.9.4 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.

2.9.5 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/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::getViewPath](#).
- `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 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. Again, `yiic` 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 `yiic` 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;

    public function rules()
    {
```

```

        return array(
            array('username', 'password', 'required'),
            array('password', 'authenticate'),
        );
    }

    public function authenticate($attribute,$params)
    {
        if(!$this->hasErrors()) // we only want to authenticate when no input errors
        {
            $identity=new UserIdentity($this->username,$this->password);
            if($identity->authenticate())
            {
                $duration=$this->rememberMe ? 3600*24*30 : 0; // 30 days
                Yii::app()->user->login($identity,$duration);
            }
            else
                $this->addError('password','Incorrect password.');
```

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

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

```
array('AttributeList', 'Validator', ...additional options)
```

where AttributeList is a string of comma-separated attribute names which need to be validated with the rule; Validator specifies what kind of validation should be performed; and additional options are used to configure the validation.

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:

```
/**
 * @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](#).

Note: When specifying rules for an active record model, we can use a special option named `on`. The option can be either `'insert'` or `'update'` so that the rule is applied only when inserting or updating the record, respectively. If not set, the rule would be applied in both cases when `save()` is called.

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:

- `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.
- `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:

```
// ensure password attribute is the same as password2 attribute
array('password', 'compare', 'compareAttribute'=>'password2')
// ensure username's length is between 3 and 12
array('username', 'length', 'min'=>3, 'max'=>12)
// ensure age is a non-negative integer
array('age', 'numeric', 'integerOnly'=>true, 'min'=>0)
```

3.2.3 Triggering Validation

To trigger validation, we can call `CModel::validate()`. The method returns a value indicating whether the validation is successful without any error. For active record model, when we call `save()` to save the model to database, validation will be automatically performed.

3.2.4 Retrieving Validation Errors

We can use `CModel::hasErrors()` to check if there is any validation error, and if yes, we can use `CModel::getErrors()` to obtain the error messages. Both methods can be used for all attributes or an individual attribute.

3.2.5 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:

```
public function actionLogin()
{
    $form=new LoginForm;
```

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

In the above, we first create a `LoginForm` instance; if the request is a POST request (meaning the login form is submitted), we populate `$form` with the submitted data `$_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 `$this->user->returnUrl` to get the URL of the page that previously needed authentication. This is because `CController` has a property named `user` which is an `CWebUser` (or its child class) instance representing 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:

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

As we mentioned, this line of code simply populate 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 correspondingly named 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):

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


Note: In order to let `$_POST['LoginForm']` to give us an array instead of a string, we adhere 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 `$user` represents `LoginForm` instance.

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

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

<div class="simple">
<?php echo CHtml::activeLabel($user,'username'); ?>
```

```

<?php echo CHtml::activeTextField($user,'username') ?>
</div>

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

<div class="action">
<?php echo CHtml::activeCheckBox($user,'rememberMe'); ?>
Remember me next time<br/>
<?php echo CHtml::submitButton('Login'); ?>
</div>

</form>
</div><!-- yiiForm -->

```

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. For more details about using `CHtml`, see [Using Helpers](#).

If we use the CSS style file `form.css` provided by the `yii` script, the generated form would be like the following:

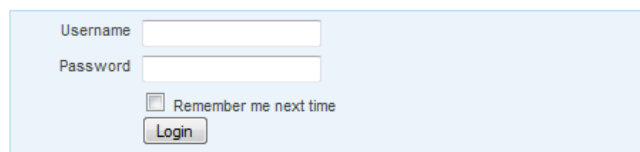


Figure 3.1: The login page

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

Figure 3.2: The login with error page

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']`.

```
public function actionBatchUpdate()
{
    // retrieve items to be updated in a batch mode
    // assuming each item is of model class 'Item'
    $items=$this->getItemsToUpdate();
    if(Yii::app()->request->isPostRequest && 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="yiiForm">
<?php echo CHtml::form(); ?>
<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($item,"name[$i]"); ?></td>
<td><?php echo CHtml::activeTextField($item,"price[$i]"); ?></td>
<td><?php echo CHtml::activeTextField($item,"count[$i]"); ?></td>
<td><?php echo CHtml::activeTextArea($item,"description[$i]"); ?></td>
</tr>
<?php endforeach; ?>
</table>

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

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

If there is anything validation error, the corresponding input fields will be highlighted automatically, just like the single model input we described earlier on.

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`

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',
    ),
),
)

```

We can then access the DB connection via `Yii::app()->db` which is already activated automatically, unless we explicitly 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:

```

$command=$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.

```

$rowCount=$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.

```
$dataReader=$command->query($sql);
// calling read() repeatedly until it returns false
while(($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:

```
$transaction=$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 will be raised
{
    $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.

```

// an SQL with two placeholders ":username" and ":email"
$sql="INSERT INTO users(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 block 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.

```
$sql="SELECT username, email FROM users";
$dataReader=$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
}
```

4.3 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 Post table:

```
$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.

```
CREATE TABLE Post (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    title VARCHAR(128) NOT NULL,
    content TEXT NOT NULL,
    createTime 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:

```
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

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 `Post` table.

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

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:

```
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).

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

```
$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 `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: For better readability, we suggest naming database tables and columns in camel case. In particular, table names are formed by capitalizing each word in the name and joining them without spaces; column names are similar to table names except their first letter is in lower case. For example, we use `Post` to name the table storing posts; and we use `createTime` to name the table's primary key column. This makes tables look more like class types and columns like variables.

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.

```
$post=new Post;
$post->title='sample post';
$post->content='content for the sample post';
$post->createTime=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:

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

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

4.3.4 Reading Record

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

```
// 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));
```

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,

```
$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.

```
// 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:

```
// 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 operations. 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 `CActive::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 the `rules()` which should be overridden in AR classes. For more details about how to specifying validation rules, refer to [http://php.net/manual/en/active-record-validation.php](#) CHtml::link('Data Validation',array('form.validation')); `?i` in the "Working with Form" section. Below we show an example of typical workflow with validation:

```
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. Again, more details can be found in the "Working with Form" section.

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

By default, all columns except primary key(s) can be massively assigned like the above. This can be changed by overriding either `protectedAttributes()` or `safeAttributes()` in the AR class. For example, if we do not want the `content` property to be massively assigned, we can write:

```
public function protectedAttributes()
{
    return array('content');
}
```

Protected columns can only be assigned individually.

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.
- `afterFind`: this is invoked for 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();
}
```

4.4 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 required that primary-foreign key relationships are well defined between tables that need to be joined. AR relies on the metadata about these relationships to determine how to join the tables. For simplicity, we will use the database schema shown in the following entity-relationship (ER) diagram to illustrate examples in this section.

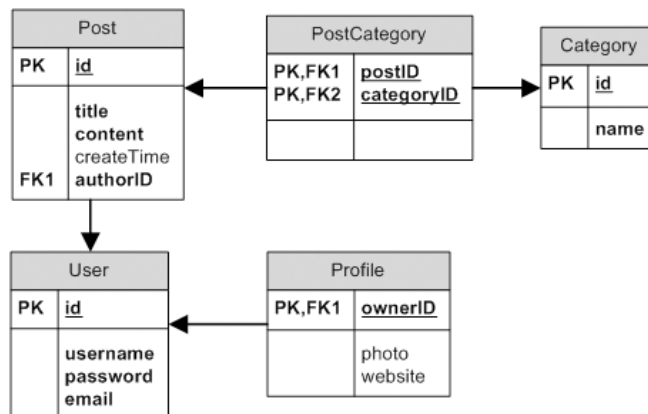


Figure 4.1: ER Diagram

Info: Support for foreign key constraints varies in different DBMS.

SQLite does not support foreign key constraints, but you can still declare the constraints when creating the tables. AR can exploit these declarations to correctly support relational queries.

MySQL supports foreign key constraints with InnoDB engine, but not with MyISAM. It is thus recommended that you use InnoDB for your MySQL database. When using MyISAM, you may exploit the following trick so that you can perform relational queries using AR:

```

CREATE TABLE Foo
(
  id INTEGER NOT NULL PRIMARY KEY
);
CREATE TABLE bar
(
  id INTEGER NOT NULL PRIMARY KEY,
  fooID INTEGER
  COMMENT 'CONSTRAINT FOREIGN KEY (fooID) REFERENCES Foo(id)'
);
  
```

In the above, we use `COMMENT` keyword to describe the foreign key constraint which can be read by AR to recognize the described relationship.

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. `User` and `Post`), one-to-one (e.g. `User` and `Profile`) and many-to-many (e.g. `Category` and `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 `PostCategory` 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.

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

```

        'author'=>array(self::BELONGS_TO, 'User', 'authorID'),
        'categories'=>array(self::MANY_MANY, 'Category', 'PostCategory(postID, categoryID)'),
    );
}
}

class User extends CActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'authorID'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'ownerID'),
        );
    }
}

```

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 `PostCategory(postID, categoryID)`.

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()->queryByPk(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.

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(array(
    'author'=>array(
        'profile',
        '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.

Info: The AR implementation in Yii is very efficient. When eager loading a hierarchy of related objects involving `N HAS_MANY` or `MANY_MANY` relationships, it will take `N+1` SQL queries to obtain the needed results. This means it needs to execute 3 SQL queries in the last example because of the `posts` and `categories` properties. Other frameworks take a more radical approach by using only one SQL query. At first look, this approach seems more efficient because fewer queries are being parsed and executed by DBMS. It is in fact impractical in reality for two reasons. First, there are many repetitive data columns in the result which takes extra time to transmit and process. Second, the number of rows in the result set grows exponentially with the number of tables involved, which makes it simply unmanageable as more relationships are involved.

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 should be disambiguated using `aliasToken` if they appear in an expression (e.g. `COUNT(??name) AS nameCount`).
- **condition:** the `WHERE` clause. It defaults to empty. Note, column references need to be disambiguated using `aliasToken` (e.g. `??id=10`).
- **order:** the `ORDER BY` clause. It defaults to empty. Note, column references need to be disambiguated using `aliasToken` (e.g. `??age DESC`).
- **with:** a list of child related objects that should be loaded together with this object. Note, this is only honored by lazy loading, not eager loading.
- **joinType:** type of join for this relationship. It defaults to `LEFT OUTER JOIN`.
- **aliasToken:** the column prefix for column reference disambiguation. It defaults to `'??.'`.

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

- **group:** the `GROUP BY` clause. It defaults to empty. Note, column references need to be disambiguated using `aliasToken` (e.g. `??age`). This option only applies to `HAS_MANY` and `MANY_MANY` relationships.

- **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 ActiveRecord
{
  public function relations()
  {
    return array(
      'posts'=>array(self::HAS_MANY, 'Post', 'authorID',
        'order'=>'??.createTime DESC',
        'with'=>'categories'),
      'profile'=>array(self::HAS_ONE, 'Profile', 'ownerID'),
    );
  }
}
```

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.

Info: 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 name. For example, `id` becomes `Team.id`. In AR relational queries, however, we do not have this freedom because the SQL statements are automatically generated by AR which systematically gives each table an alias. Therefore, in order to avoid column name conflict, we use a placeholder to indicate the existence of a column which needs to be disambiguated. AR will replace the placeholder with a suitable table alias and properly disambiguate the column.

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 medium. 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](#).
- **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.

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 in 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:

```
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:

```
// 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.

```
$value=Yii::app()->cache->get($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.

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.

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.

```
...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 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:

```
...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 varied 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 varied according to user IDs. This essentially means that we should use different IDs when calling `beginCache()`.

Instead of asking developers to vary 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 varied 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 varied 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 varied 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.

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:

```
...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.

```
...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:

```
public function filters()
{
    return array(
        array(
            'system.web.widgets.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.

```
...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
- 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 package of several class files and the corresponding asset files. A module often achieves higher level and more advanced functionalities than a single component. 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 `application.extensions.xyz` to locate its base directory which contains all files of `xyz`.

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 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'=>'application.extensions.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.

6.2.2 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,

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

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

...body content of the widget...

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

6.2.3 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:

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

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

6.2.4 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:

```
class TestController extends CController
{
    public function filters()
    {
        return array(
            array(
                'application.extensions.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`.

6.2.5 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'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // other controllers
    ),
);
```

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

6.2.6 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',
                'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // other validation rules
        );
    }
}
```

6.2.7 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'=>'application.extensions.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`.

6.2.8 Module

A module typically consists of more than one class files and is often a mix of the above extension types. Therefore, you should follow the corresponding instructions to use a module.

6.2.9 Generic Component

To use a generic [component](#), we first need to include its class file by using

```
Yii::import('application.extensions.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 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.

6.3.3 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`.

6.3.4 Filter

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

```
class MyFilter extends CFilter
{
    protected function preFilter($filterChain)
```

```

{
    // logic being applied before the action is executed
    return true; // false if the action should not be executed
}

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

```

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

6.3.5 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.

6.3.6 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);
    }
}

```

6.3.7 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';
    }
}
```

6.3.8 Module

The 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.

6.3.9 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 an 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:

```
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:

```
$lucene=new Zend_Search_Lucene($pathOfIndex);  
$hits=$lucene->find(strtolower($keyword));
```


CHAPTER 7

Special Topics

7.1 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 an Yii application, these are accomplished with the help of [CUrlManager](#).

7.1.1 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()->hostInfo`, or call `createAbsolutePath`.

7.1.2 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 must be a valid regular expression without delimiters and modifiers. It is used to match the path info part of URLs. And the [route](#) should refer to a valid controller route.

A rule can be associated with a few GET parameters. These GET parameters appear in the rule's pattern as special tokens of 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. 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. Therefore, a high-traffic Web application should minimize its use of URL rules.

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. Both can be in a `.htaccess` file that is placed under the directory containing the entry script. The following is an example:

```
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.

7.2 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.

7.2.1 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 define the `title` property to be persistent using `CBaseUserIdentity::getState` and `CBaseUserIdentity::setState` methods. As a result, during the whole user session, we can obtain the user title information via `Yii::app()->user->getState('title')`.

```
class UserIdentity extends CUserIdentity
{
    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
        {
            $this->title=$record->title;
            $this->errorCode=self::ERROR_NONE;
        }
        return !$this->errorCode;
    }

    public function getTitle()
    {
        return $this->getState('title');
    }

    public function setTitle($value)
    {
        $this->setState('title',$value);
    }
}
```

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.

7.2.2 Login and Logout

Using the identity class and the user component 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 once he closes his browser window. 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);
```

7.2.3 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',  
        'action'=>'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.
- **users:** specifies which users this rule matches. The current user's `name` is used for matching. 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 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.
- **ips:** specifies which client IP addresses this rule matches.
- **verbs:** specifies which request types (e.g. `GET`, `POST`) this rule matches.

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.
- Otherwise an HTTP exception will be displayed with error code 401.

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,

```
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:

```
Yii::app()->request->redirect(Yii::app()->user->returnUrl);
```

7.2.4 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 multiple 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:

```
$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');
```

Note that we associate 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.

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.

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

```
$params=array('post'=>$post);  
if(Yii::app()->user->checkAccess('updateOwnPost',$params))  
{  
    // update post  
}
```

7.3 Theme

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.

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`.

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`. Because the `views` directory may contain security-sensitive data, it should be configured to prevent from being accessed by Web users.

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'
```

7.4 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.

7.4.1 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.

7.4.2 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',
                    'email'=>'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.

7.4.3 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.

7.5 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 `exception`) 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.

7.5.1 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.');
```

7.5.2 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.

7.5.3 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.

7.6 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.

7.6.1 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.

```
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.

7.6.2 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`.

```
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.

7.6.3 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.

```
$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.

7.6.4 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
     * @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;
}

```

7.6.5 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`.

```

class PostController extends CController
{
    public function actions()
    {
        return array(
            'service'=>array(
                'class'=>'CWebServiceAction',
                'classMap'=>array(
                    'Post'=>'Post', // or simply 'Post'
                ),
            ),
        );
    }
    .....
}

```

7.6.6 Intercepting Remote Method Invocation

By implementing the `IWebServiceProvider` interface, a service provider can intercept remote method invocations. In `IWebServiceProvider::beforeWebMethod`, the provider may retrieve the current `CWebService` instance and obtain 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).

7.7 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.

7.7.1 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_RegionID` in lower case (e.g. `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 much rarely used information.

For an 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](#).

7.7.2 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('yii', '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.
3. Configure `CApplication::sourceLanguage` and `CApplication::language`.

Tip: The `yiic` tool in Yii can be used to manage message translations. Using its `message` command, we can extract messages to be translated from source files and merge them with existing translations.

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 result 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.

7.7.3 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).

7.7.4 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.

7.8 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.

7.8.1 Using CPradoViewRenderer

To use `CPradoViewRenderer`, we just need to configure the application as follows:

```
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,

```
<cache:clipID>
    // content for this clip
</cache: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
--->
```

7.9 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).

7.9.1 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. Run the local `yiic` tool, we will see that our own commands appearing together with the standard ones. We can also create our own commands to be used when `yiic shell` is used. To do so, just drop our command class files under the `protected/commands/shell` directory.

7.10 Security

7.10.1 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 $this->beginWidget('CHtmlPurifier'); ?>
...display user-entered content here...
<?php $this->endWidget(); ?>
```

7.10.2 Cross-site Request Forgery Prevention

Cross-Site Request Forgery (CSRF) attacks occur when a malicious web site causes a users 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 malicious 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.

7.10.3 Cookie Attack Prevention

Protecting cookies from being attacked is of extreme important, 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,

```
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 `$_COOKIE`:

```
// 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;
```

7.11 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.

7.11.1 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.

7.11.2 Disabling Debug Mode

Disabling debug mode is another easy way to improve performance. An 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.

7.11.3 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 an Yii-power 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.

7.11.4 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 , [eAccelerator](#), [Squid](#), to name a few.

7.11.5 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.