



O Guia Definitivo do Yii 1.1

Qiang Xue e Xiang Wei Zhuo

Tradução para o Português:

Wanderson C. Bragança, luiz.uo e davi_alexandre

Adaptação para o formato de eBook:

Ivan Wilhelm

2008-2010 © Yii Software LLC.

| | |
|--------------------------------------|-----------|
| Primeiros Passos | 5 |
| Visão Geral | 5 |
| Novos Recursos | 5 |
| Atualizando a versão 1.0 para 1.1 | 7 |
| O que é Yii? | 8 |
| Instalação | 9 |
| Criando a primeira aplicação Yii | 9 |
| Conceitos Básicos | 21 |
| Modelo-Visão-Controle (MVC) | 21 |
| O Script de Entrada | 23 |
| Aplicação | 23 |
| Controle | 27 |
| Modelo | 32 |
| Visão | 32 |
| Componente | 35 |
| Module | 38 |
| Path Alias e Namespace | 40 |
| Convenções | 42 |
| Fluxo de trabalho do desenvolvimento | 44 |
| Melhores Práticas em MVC | 45 |
| Trabalhando com formulários | 48 |
| Visão Geral | 48 |
| Criando um Modelo | 48 |
| Criando uma Ação | 55 |
| Criando um Formulário | 57 |

| | |
|---------------------------------------|------------|
| Coletando Entradas Tabulares | 59 |
| Trabalhando com Banco de Dados | 62 |
| Visão Geral | 62 |
| Data Access Objects (DAO) | 62 |
| Gerador de Consultas | 67 |
| Active Record | 83 |
| Active Record Relacional | 96 |
| Migração de Bases de Dados | 106 |
| Caching | 112 |
| Visão Geral | 112 |
| Data Caching (Caching de Dados) | 113 |
| Caching de Fragmentos | 115 |
| Caching de Páginas | 118 |
| Conteúdo Dinâmico | 119 |
| Estendendo o Yii | 120 |
| Visão Geral | 120 |
| Usando Extensões | 120 |
| Criando Extensões | 126 |
| Utilizando Bibliotecas de Terceiros | 131 |
| Testes | 133 |
| Visão Geral | 133 |
| Definindo Fixtures | 135 |
| Testes Unitários | 138 |
| Testes Funcionais | 140 |

| | |
|----------------------------------|------------|
| Tópicos Especiais | 143 |
| Gerenciamento de URL | 143 |
| Autenticação e Autorização | 148 |
| Temas | 160 |
| Registros (Logs) | 167 |
| Erros | 171 |
| Web Service | 174 |
| Internacionalização (i18n) | 178 |
| Sintaxe Alternativa de Templates | 184 |
| Aplicativos de Console | 187 |
| Segurança | 191 |
| Ajustes no Desempenho | 194 |

Primeiros Passos

Visão Geral

Esse tutorial está disponível sob os termos da Documentação do Yii.

2008-2010 © Yii Software LLC. Todos os Direitos Reservados.

Tradução para o Português: Wanderson C. Bragança, luiz.uo e davi_alexandre

Adaptação para o formato de eBook: Ivan Wilhelm

Novos Recursos

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

Version 1.1.6

- Added query builder
- Added database migration
- Best MVC Practices
- Added support for using anonymous parameters and global options in console commands

Version 1.1.5

- Added support for console command actions and parameter binding
- Added support for autoloading namespaced classes
- Added support for theming widget views

Version 1.1.4

- Added support for automatic action parameter binding

Version 1.1.3

- Added support to configure widget default values in application configuration

Version 1.1.2

- Added a Web-based code generation tool called Gii

Version 1.1.1

- Added CActiveForm which simplifies writing form-related code and supports seamless and consistent validation on both client and server sides.
- Refactored the code generated by the yiic tool. In particular, the skeleton application is now generated with multiple layouts; the operation menu is reorganized for CRUD

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

- Added support to allow defining global yiic commands

Version 1.1.0

- Added support for writing unit and functional tests
- Added support for using widget skins
- Added an extensible form builder
- Improved the way of declaring safe model attributes. See Securing Attribute Assignments.
- Changed the default eager loading algorithm for relational active record queries so that all tables are joined in one single SQL statement.
- Changed the default table alias to be the name of active record relations.
- Added support for using table prefix.
- Added a whole set of new extensions known as the Zii library.
- The alias name for the primary table in an AR query is fixed to be 't'

Version 1.0.11

- Added support for parsing and creating URLs with parameterized hostnames
 - Parameterizing Hostnames

Version 1.0.10

- Enhanced support for using CPhpMessageSource to manage module messages
 - Message Translation
- Added support for attaching anonymous functions as event handlers
 - Component Event

Version 1.0.8

- Added support for retrieving multiple cached values at one time
 - Data Caching
- Introduced a new default root path alias ext which points to the directory containing all third-party extensions.
 - Using Extensions

Version 1.0.7

- Added support for displaying call stack information in trace messages
 - Logging Context Information
- Added index option to AR relations so that related objects can be indexed using the values of a specific column
 - Relational Query Options

Version 1.0.6

- Added support for using named scope with update and delete methods:
 - Named Scopes
- Added support for using named scope in the with option of relational rules:
 - Relational Query with Named Scopes

- Added support for profiling SQL executions
 - Profiling SQL Executions
- Added support for logging additional context information
 - Logging Context Information
- Added support for customizing a single URL rule by setting its `urlFormat` and `caseSensitive` options:
 - User-friendly URLs
- Added support for using a controller action to display application errors:
 - Handling Errors Using an Action

Version 1.0.5

- Enhanced active record by supporting named scopes. See:
 - Named Scopes
 - Default Named Scope
 - Relational Query with Named Scopes
- Enhanced active record by supporting lazy loading with dynamic query options. See:
 - Dynamic Relational Query Options
- Enhanced `CUrlManager` to support parameterizing the route part in URL rules. See:
 - Parameterizing Routes in URL Rules

Atualizando a versão 1.0 para 1.1

Changes Related with Model Scenarios

- Removed `CModel::safeAttributes()`. Safe attributes are now defined to be those that are being validated by some rules as defined in `CModel::rules()` for the particular scenario.
- Changed `CModel::validate()`, `CModel::beforeValidate()` and `CModel::afterValidate()`. `CModel::setAttributes()`, `CModel::getSafeAttributeName()` The 'scenario' parameter is removed. You should get and set the model scenario via `CModel::scenario`.
- Changed `CModel::getValidators()` and removed `CModel::getValidatorsForAttribute()`. `CModel::getValidators()` now only returns validators applicable to the scenario as specified by the model's scenario property.
- Changed `CModel::isAttributeRequired()` and `CModel::getValidatorsForAttribute()`. The scenario parameter is removed. The model's scenario property will be used, instead.
- Removed `CHtml::scenario`. `CHtml` will use the model's scenario property instead.

Changes Related with Eager Loading for Relational Active Record

- By default, a single JOIN statement will be generated and executed for all relations involved in the eager loading. If the primary table has its LIMIT or OFFSET query option set, it will be queried alone first, followed by another SQL statement that brings back all its related objects. Previously in version 1.0.x, the default behavior is that there will be N+1 SQL statements if an eager loading involves N HAS_MANY or MANY_MANY relations.

Changes Related with Table Alias in Relational Active Record

- The default alias for a relational table is now the same as the corresponding relation name. Previously in version 1.0.x, by default Yii would automatically generate a table

alias for each relational table, and we had to use the prefix `??`. to refer to this automatically generated alias.

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

Changes Related with Tabular Input

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

Other Changes

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

O que é Yii?

Yii é um framework de alta performance em PHP que utiliza componentes para o desenvolvimento de grandes aplicações Web. Permite máxima reutilização de códigos na programação Web e pode acelerar significativamente o processo de desenvolvimento. O nome Yii (pronunciado i) representa as palavras fácil (easy), eficiente (efficient) e extensível (extensible).

Requerimentos

Para executar uma aplicação Web que utilize o Yii, você precisará de um servidor Web com suporte a PHP 5.1.0 ou superior.

Para os desenvolvedores que desejam utilizar o Yii, é muito importante o conhecimento de programação orientada a objetos (POO), pois o Yii é um framework totalmente orientado a objetos.

Pra qual solução o Yii é melhor?

O Yii é um framework de programação Web genérico que pode ser usado para desenvolver praticamente todos os tipos de aplicações Web. Por ser um framework leve equipado com sofisticadas soluções em caching, é especialmente adequado para o desenvolvimento de aplicações com alto tráfego de dados, tais como portais, fóruns, sistemas de gerenciamento de conteúdo (CMS), sistemas de e-Commerce, etc.

Como é o Yii comparado com outros Frameworks?

Como a maioria dos frameworks PHP, O Yii é um framework MVC.

O Yii se sobressai dos outros frameworks PHP na medida em que é eficiente, rico em recursos e bem documentado. O Yii é cuidadosamente projetado para se ajustar a sérias aplicações Web desde seu início. Não é nem um subproduto de algum projeto, nem um conglomerado de trabalho de terceiros. É o resultado da rica experiência de seus autores

no desenvolvimento Web e da investigação e reflexão das aplicações e dos mais populares frameworks de programação Web.

Instalação

A instalação do Yii envolve principalmente, as duas etapas seguintes:

1. Fazer o Download do Yii Framework em yiiframework.com.
2. Descompactar o arquivo do Yii em um diretório acessível a Web.

Dica: O Yii não precisa ser instalado em um diretório acessível a Web. Uma aplicação Yii tem uma entrada de script que normalmente é o único arquivo que precisa ser exposto para os usuários da Web. Outros scripts de PHP, incluindo os do Yii, devem ter o acesso protegido, uma vez que podem ser explorados por hackers.

Requisitos

Depois de instalar o Yii, você pode verificar se o servidor satisfaz todos os requisitos de utilização do Yii. Você pode fazê-lo, acessando o script verificador de requisitos na seguinte URL em um navegador Web:

<http://nomedoservidor/caminho/do/yii/requirements/index.php>

O requisito mínimo para o Yii é que seu servidor Web tenha suporte ao PHP 5.1.0 ou acima. O Yii foi testado com [Apache HTTP Server](#) nos sistemas operacionais Windows e Linux. Também pode ser executado em outros servidores Web e plataformas desde que tenha o PHP 5.

Criando a primeira aplicação Yii

Para ter uma experiência inicial com o Yii, descrevemos nesta seção como criar nossa primeira aplicação em Yii. Iremos utilizar a poderosa ferramenta yiic que pode ser usada para automatizar a criação de código para várias finalidades. Assumiremos que YiiRoot é o diretório onde o Yii está instalado e WebRoot é o diretório raiz do servidor Web.

Execute o yiic pela linha de comando, como no exemplo a seguir:

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

Nota: Quando executamos o yiic no Mac OS, Linux ou Unix, devemos alterar a permissão do arquivo yiic para torna-lo executável.

Como forma alternativa, você pode executa-lo da seguinte maneira:

```
% cd WebRoot/testdrive
```

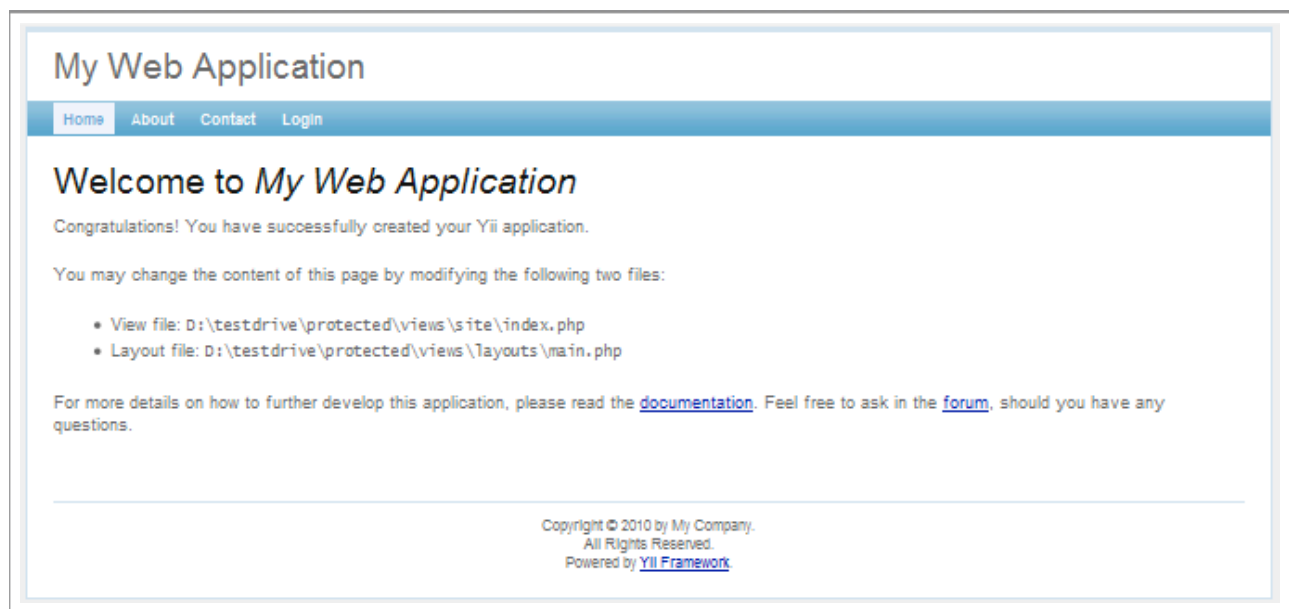
```
% php YiiRoot/framework/yiic.php webapp WebRoot/testdrive
```

Esse comando irá criar o esqueleto de uma aplicação Yii, no diretório WebRoot/testdrive. A aplicação tem um estrutura de diretórios que é a necessária para a maioria das aplicações feitas no Yii.

Sem ter escrito uma única linha de código, já podemos testar nossa primeira aplicação Yii, acessando a seguinte URL:

<http://nomedoservidor/testdrive/index.php>

Como podemos ver, a aplicação tem três páginas: a página inicial, a página de contato e a página de login. A página principal mostra algumas informações sobre a aplicação, como o login do usuário ativo, a página de contato exibe um formulário de contato que os usuários podem preencher e enviar suas mensagens, a página de login permite que os usuários se autenticem antes de acessar o conteúdo privilegiado. Veja as imagens a seguir para mais detalhes:



Página Principal

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

[Home](#) » [Contact](#)

Contact Us

If you have business inquiries or other questions, please fill out the following form to contact us. Thank you.

Fields with * are required.

Name *

Email *

Subject *

Body *

Verification Code

diquauj

[Get a new code](#)

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

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

Página de Contato

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

[Home](#) » [Contact](#)

Contact Us

If you have business inquiries or other questions, please fill out the following form to contact us. Thank you.

Fields with * are required.

Please fix the following input errors:

- Subject cannot be blank.
- Body cannot be blank.
- The verification code is incorrect.


Name *

Email *

Subject *

Body *

Verification Code

[Get a new code](#)

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

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

Página de Contato com erros de entrada

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

[Home](#) » [Contact](#)

Contact Us

Thank you for contacting us. We will respond to you as soon as possible.

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

Página de Contato com emissão bem sucedida

My Web Application

[Home](#)[About](#)[Contact](#)[Login](#)

[Home](#) » [Login](#)

Login

Please fill out the following form with your login credentials:

Fields with * are required.

Username *

Password *

Hint: You may login with demo/demo or admin/admin.

☐ Remember me next time

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Página de Login

A listagem seguinte mostra a estrutura de diretórios da nossa aplicação. Por favor, veja as [Convenções](#) para obter explicações detalhadas sobre essa estrutura.

testdrive/

| | |
|--------------------|--|
| index.php | Script de entrada da aplicação Web |
| assets/ | Contém arquivos de recurso publicados |
| css/ | Contém arquivos CSS |
| images/ | Contém arquivos de imagem |
| themes/ | Contém temas da aplicação |
| protected/ | Contém arquivos protegidos da aplicação |
| yiic | Script de linha de comando yiic |
| yiic.bat | Script de linha de comando yiic para o Windows |
| commands/ | Contém comandos 'yiic' customizados |
| shell/ | Contém comandos 'yiic shell' customizados |
| components/ | Contém componentes reutilizáveis do usuário |
| MainMenu.php | A classe widget 'MainMenu' (Menu Principal) |
| Identity.php | A classe 'Identity' usada nas autenticações |
| views/ | Contém arquivos de visão dos widgets |
| mainMenu.php | O arquivo de visão do widget 'MainMenu' |
| config/ | Contém arquivos de configurações |
| console.php | Configuração da aplicação console |
| main.php | Configuração da aplicação Web |
| controllers/ | Contém arquivos das classes de controle |
| SiteController.php | Classes de controle padrão |
| extensions/ | Contém extensões de terceiros |
| messages/ | Contém mensagens traduzidas |
| models/ | Contém arquivos das classes de modelo |
| LoginForm.php | Modelo do formulário para a ação 'login' |
| ContactForm.php | Modelo do formulário para a ação 'contact' |
| runtime/ | Contém arquivos gerados temporariamente |
| views/ | Contém arquivos de visão dos controles e layouts |
| layouts/ | Contém arquivos de visão do layout |
| main.php | O layout padrão para todas as visões |
| site/ | Contém arquivos de visão para o controle 'site' |
| contact.php | Visão para a ação 'contact' |
| index.php | Visão para a ação 'index' |
| login.php | Visão para a ação 'login' |
| system/ | Contém arquivos de visão do sistema |

Conectando ao Banco de Dados

A maioria das aplicações Web são auxiliadas com o uso de banco de dados. Nossa aplicação de test-drive não é uma exceção. Para usar banco de dados, primeiro precisamos dizer à aplicação como se conectar a ele. Isto é feito alterando o arquivo de configuração WebRoot/testdrive/protected/config/main.php, como mostrado abaixo:

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

Acima, nós adicionamos uma entrada para db ao array components, que instrui a aplicação para se conectar ao banco de dados SQLite WebRoot/testdrive/protected/data/source.db quando for preciso.

Nota: Para utilizar os recursos de banco de dados do Yii, precisamos ativar a extensão PDO do PHP e a extensão de driver PDO específico. Para a aplicação test-drive, as extensões php_pdo e php_pdo_sqlite deverão estar habilitadas.

Para este fim, precisamos de preparar uma base de dados SQLite, para que a configuração feita anteriormente seja eficaz. Usando alguma ferramenta de administração do SQLite, podemos criar um banco de dados com o seguinte esquema:

```
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
);
```

Nota: Se estiver utilizando um banco de dados MySQL, você deve substituir o AUTOINCREMENT, utilizado no código acima, por AUTO_INCREMENT.

Por simplicidade, criamos somente uma única tabela: User no nosso banco de dados. O arquivo do banco de dados SQLite foi salvo em WebRoot/testdrive/protected/data/

source.db. Observe que tanto o arquivo quanto o diretório devem ter permissão de leitura do servidor Web, como requerido pelo SQLite.

Implementando operações do tipo CRUD

Agora começa a parte divertida. Iremos implementar operações CRUD (create, read, update and delete) que realizará inserções, leituras, edições e deleções na tabela User que acabamos de criar. Este tipo de operação é frequentemente necessária em aplicações reais.

Em vez da dificuldade na escrita de um código real, iremos utilizar a poderosa ferramenta yiic para gerar automaticamente o código. Este processo é também conhecido como scaffolding. Abra a linha de comando e execute os comandos listados a seguir:


```
% cd WebRoot/testdrive
% protected/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
    mkdir D:/wwwroot/testdrive/protected/views/user
    generate create.php
    generate update.php
    generate list.php
    generate show.php
    generate admin.php
    generate _form.php

Crud 'user' has been successfully created. You may access it via:
http://hostname/path/to/index.php?r=user
```

Acima, utilizamos o comando yiic shell para interagir com nossa aplicação esqueleto. Na linha de comando, podemos digitar dois subcomandos: model User e crud User. O primeiro gera a classe modelo para a tabela User, enquanto que o segundo comando lê a classe modelo User e gera o código necessário para as operações do tipo CRUD.

Nota: Você poderá encontrar erros como "...could not find driver" ou "...driver não encontrado", mesmo que o verificador de requisitos mostre que você já tem o PDO ativado e o driver PDO correspondente ao Banco de Dados. Caso isso ocorra, você deve tentar rodar a ferramenta yiic do seguinte modo:

```
% php -c caminho/para/php.ini protected/yiic.php shell
```

onde caminho/para/php.ini representa o arquivo PHP.ini correto.

Podemos ver nossa primeira aplicação pela seguinte URL:

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

Essa página irá mostrar uma lista de entradas de usuários da tabela User. Se tabela estiver vazia, nada será exibido.

Clique no link New User da página. Caso não esteja autenticado seremos levados à página de login. Uma vez logado, será exibido um formulário de entrada que permite adicionar um novo usuário. Preencha o formulário e clique sobre o botão Create. Se houver qualquer erro de entrada, um erro será mostrado, o que nos impede de salvar os dados. Voltando à lista de usuários, iremos ver o recém adicionado usuário aparecendo na lista.

Repita as etapas acima para adicionar novos usuários. Repare que a tabela de usuários será automaticamente paginada, caso existam muitos usuários a serem exibidos em uma página.

Se logarmos como administrador utilizando o login/senha: admin/admin, veremos a página de administração de usuários pela seguinte URL:

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

Será mostrada uma tabela de usuários. Podemos clicar nas células do cabeçalho para ordenar as colunas correspondentes. E como na página de listagem dos usuários, a página de administração dos usuários também realiza a paginação quando existem muitos usuários a serem exibidos.

Todas essas incríveis funcionalidades foram criadas sem escrever uma única linha de código!

My Web Application

[Home](#) [About](#) [Contact](#) [Logout \(admin\)](#)

[Home](#) » [Users](#) » Manage

Manage Users

[Advanced Search](#)

Id

Username





















Email

Operations

[List User](#)

[Create User](#)

Displaying 1-10 of 21 result(s).

| Id | Username | Password | Email | |
|----------------------|----------------------|----------------------|----------------------|---|
| <input type="text"/> | <input type="text"/> | <input type="text"/> | <input type="text"/> | |
| 1 | test1 | pass1 | test1@example.com |   |
| 2 | test2 | pass2 | test2@example.com |   |
| 3 | test3 | pass3 | test3@example.com |   |
| 4 | test4 | pass4 | test4@example.com |   |
| 5 | test5 | pass5 | test5@example.com |   |
| 6 | test6 | pass6 | test6@example.com |   |
| 7 | test7 | pass7 | test7@example.com |   |
| 8 | test8 | pass8 | test8@example.com |   |
| 9 | test9 | pass9 | test9@example.com |   |
| 10 | test10 | pass10 | test10@example.com |   |

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

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Página de administração dos usuários

My Web Application

[Home](#) [About](#) [Contact](#) [Logout \(admin\)](#)

[Home](#) » [Users](#) » Create

Create User

Fields with * are required.

Please fix the following input errors:

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

Username *

test

Password *

Password cannot be blank.

Email *

Email cannot be blank.

Create

Operations

List User

Manage User

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

Página de criação de um novo usuário

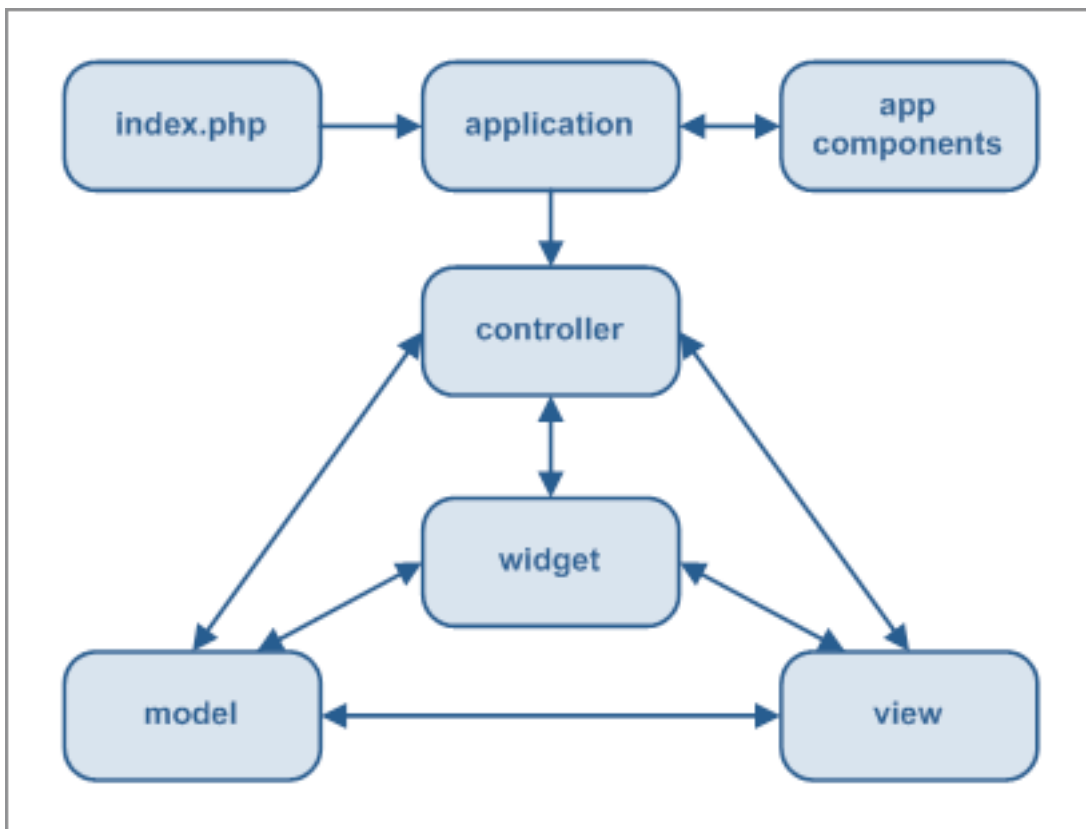
Conceitos Básicos

Modelo-Visão-Controle (MVC)

O Yii implementa o padrão de desenvolvimento modelo-visão-controle (MVC) que é amplamente adotado na programação Web. O MVC visa separar a lógica de negócio da interface com o usuário, assim os programadores podem mudar facilmente cada parte, sem afetar as outras. No padrão MVC, o modelo representa as informações (os dados) e as regras de negócio, a visão contém elemento de interface com o usuário, como textos, formulários, e o controle gerencia a comunicação entre o modelo e a visão.

Além MVC, o Yii também introduz um controle de frente, chamado aplicação (application), que representa o contexto de execução dos processos requisitados. A aplicação recebe a solicitação do usuário e a envia para um controlador adequado para ser processada.

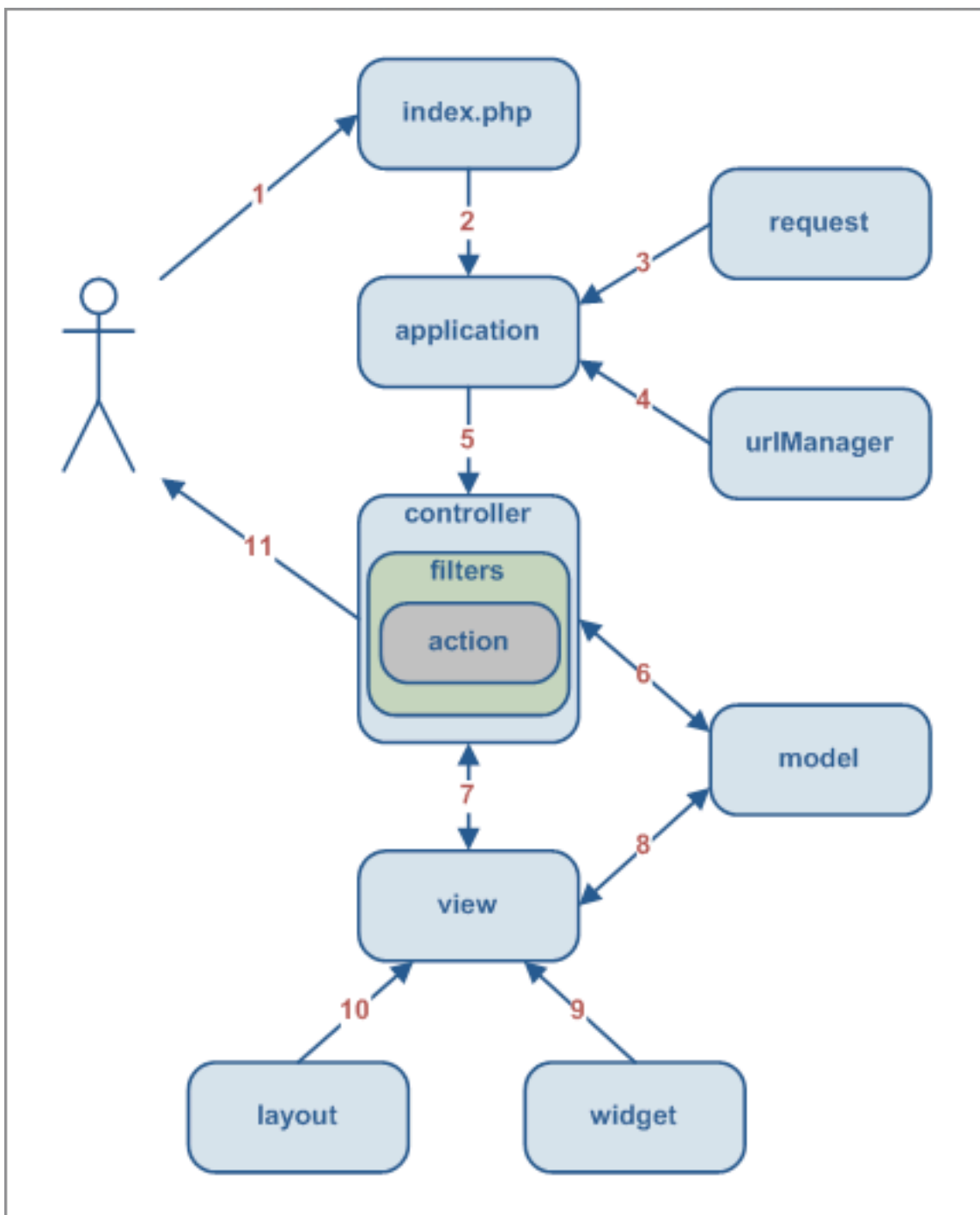
O diagrama seguinte mostra a estrutura estática de uma aplicação Yii:



Estrutura estática de uma aplicação Yii

Um típico fluxo de execução

O diagrama a seguir mostra um típico fluxo de execução de uma aplicação Yii quando esta está recebendo uma solicitação de um usuário.



1. O usuário faz uma solicitação com a URL <http://www.exemplo.com/index.php?r=post/show&id=1> e o servidor Web processa o pedido executando o script de bootstrap index.php.
2. O script de bootstrap cria uma instancia de [aplicação](#) (application) e a executa.
3. A aplicação obtém as informações detalhadas da solicitação de um [componente da aplicação](#) chamado request.
4. A aplicação determina o [controle](#) e a [ação](#) requerida com a ajuda do componente chamado urlManager. Para este exemplo, o controle é post que se refere à classe PostController e a ação é show cujo significado real é determinado no controle.
5. A aplicação cria uma instancia do controle solicitado para poder lidar com a solicitação do usuário. O controle determina que a ação show refere-se a um método chamado actionShow no controle da classe. Em seguida, cria e executa filtros (por exemplo, o

controle de acesso, benchmarking) associados a esta ação. A ação só é executada se permitida pelos filtros.

6. A ação lê um [modelo](#) Post cujo ID é 1 no Banco de Dados.
7. A ação processa a [visão](#) chamada show, com o Post.
8. A visão apresenta os atributos do modelo Post.
9. A visão executa alguns [widgets](#).
10. O resultado do processamento da visão é embutido em um [layout](#).
11. A ação conclui o processamento da visão e exibe o resultado ao usuário.

O Script de Entrada

O Script de Entrada, `index.php`, é um script bootstrap em PHP que processa solicitações de usuários inicialmente. É o único script PHP que os usuários finais podem executar diretamente.

Na maioria dos casos, o script de entrada de uma aplicação Yii contém um código simples, mostrado abaixo:

```
// 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();
```

O script inclui primeiramente o arquivo de bootstrap `yii.php` do framework Yii. Ele cria em seguida uma instância do aplicativo Web que especifica as configurações e o executa.

Modo de Debug

Uma aplicação Yii pode ser executado tanto em modo de debug (depuração) quanto em modo de produção de acordo com o valor da constante `YII_DEBUG`. Por padrão, esse valor constante é definido como `false`, o que significa modo de produção. Para executar em modo de debug, defina essa constante como `true` antes de incluir o arquivo `yii.php`. Executando a aplicação em modo de debug é importante durante a fase de desenvolvimento pois fornece ricas informações de depuração quando um erro ocorre.

Aplicação

A aplicação representa o contexto de execução do processamento de uma solicitação. Sua principal tarefa é a de receber uma solicitação do usuário e enviá-la para um controle adequado para o posterior processamento. Serve também como o lugar central para o processamento de configurações a nível da aplicação. Por esta razão, a aplicação é também chamada de controle de frente.

A aplicação é criada como um singleton pelo [script de entrada](#). O singleton da aplicação significa que esta pode ser acessada em qualquer lugar pelo `Yii::app()`.

Configuração da Aplicação

Por padrão, a aplicação é uma instancia de [CWebApplication](#). Para personalizá-la normalmente é utilizado um arquivo de configuração (ou um array) para inicializar os valores de suas propriedades quando a instância da aplicação é criada. Uma alternativa para a customização é estender o [CWebApplication](#).

A configuração é um conjunto de pares do tipo chave-valor. Cada chave representa o nome de uma propriedade da instancia da aplicação, e cada valor a propriedade inicial correspondente. Por exemplo, a seguinte configuração altera as propriedades [name](#) e [defaultController](#) da aplicação.

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

Costumamos salvar a configuração em um script PHP separado (ex.: protected/config/main.php). Dentro do script, retornamos o array de configuração do seguinte modo:

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

Para aplicar a configuração, passamos o nome do arquivo de configuração como um parâmetro ao construtor da aplicação, ou para o método [Yii::createWebApplication\(\)](#), como o seguinte exemplo que é feito normalmente no [script de entrada](#):

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

Dica: Se a configuração da aplicação é muito complexa, podemos dividi-la em vários arquivos, cada um retornando uma porção do array de configuração. Em seguida, no arquivo principal de configuração, chamamos a função `include()` do PHP para incluir o resto dos arquivos de configuração e combinamos em uma única array de configuração completa.

Diretório Base da Aplicação

O diretório base da aplicação é a pasta principal que contém todos os dados e scripts de PHP sensíveis à segurança. Por padrão, é um subdiretório chamado `protected` que está localizado sob o diretório que contém o script de entrada. Ele pode ser personalizado através da definição [basePath](#), uma propriedade da [configuração da aplicação](#).

O conteúdo dentro do diretório base da aplicação deve ter o acesso protegido de usuários da Web. Com o servidor [Apache HTTP Server](#), isto pode ser feito facilmente criando um arquivo `.htaccess` dentro do diretório base. O conteúdo do arquivo `.htaccess` deverá ser o seguinte:


```
deny from all
```

Componentes da Aplicação

As funcionalidades da aplicação podem ser facilmente customizadas e enriquecidas graças à arquitetura flexível de componentes. A aplicação gerencia um conjunto de componentes, que implementam recursos específicos. Por exemplo, a aplicação realiza um pedido de um usuário com a ajuda dos componentes [CUrlManager](#) e [CHttpRequest](#).

Ao configurar as propriedades dos [componentes](#) da aplicação, podemos personalizar a classe e os valores das propriedades de qualquer componente usado na aplicação. Por exemplo, podemos configurar o componente [CMemCache](#) para que ele possa utilizar múltiplos servidores de memcache para fazer o caching:

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

Acima, adicionamos o elemento cache ao array components. O elemento cache indica a classe do componente, que é CMemCache e sua propriedade servers deve ser inicializada como no exemplo.

Para acessar um componente da aplicação, use `Yii::app()->ComponentID`, onde ComponentID refere-se ao ID do componente (ex.: `Yii::app()->cache`).

Um componente da aplicação pode ser desativado setando a propriedade enabled para false na configuração. O valor Null é retornado quando um componente desativado é acessado.

Dica: Por padrão, os componentes da aplicação são criados sob demanda. Isto significa que um componente pode não ser completamente criado se não for acessado durante uma requisição de um usuário. Consequentemente, o desempenho global não será prejudicado, mesmo em uma aplicação com muitos componentes. Alguns componentes da aplicação (ex.: CLogRouter) necessitam serem criados, não importando se estão sendo acessados ou não. Para fazer isso, liste os IDs na propriedade preload da aplicação.

Principais Componentes da Aplicação

O Yii predefine um conjunto de componentes principais da aplicação para fornecer funcionalidades comuns em aplicações Web. Por exemplo, o componente [request](#) é usado para processar pedidos do usuário e prover informações como URL, cookies. Ao configurar as propriedades desses componentes principais, podemos mudar o padrão de comportamento do Yii em quase todos os aspectos.

Abaixo, listamos os principais componentes que são pré-declarados pelo [CWebApplication](#).

- [assetManager](#): [CAssetManager](#) - gerencia a criação dos ativos privados (assets).
- [authManager](#): [CAuthManager](#) - gerencia o controle de acesso baseado em regras (RBAC).
- [cache](#): [CCache](#) - fornece as funcionalidades do caching de dados. Note que você deve especificar a classe atual (ex.: [CMemCache](#), [CDBCache](#)). Caso contrário, será retornado Null ao acessar o componente.
- [clientScript](#): [CClientScript](#) - gerencia os scripts (javascripts e CSS) do cliente.
- [coreMessages](#): [CPhpMessageSource](#) - fornece as principais mensagens traduzidas usadas pelo framework Yii.
- [db](#): [CDBConnection](#) - fornece uma conexão ao banco de dados. Note que você deverá configurar a propriedade [connectionString](#) corretamente para utilizar esse componente.
- [errorHandler](#): [CErrorHandler](#) - processa erros e exceções do PHP.
- [messages](#): [CPhpMessageSource](#) - fornece mensagens traduzidas utilizadas pela aplicação Yii.
- [request](#): [CHttpRequest](#) - fornece informações relacionadas à solicitação do usuário.
- [securityManager](#): [CSecurityManager](#) - fornece serviços relacionados à segurança, como hashing e encriptação.
- [session](#): [CHttpSession](#) - fornece funcionalidades relacionadas à sessão.
- [statePersister](#): [CStatePersister](#) - fornece o estado global do método de persistência.
- [urlManager](#): [CUrlManager](#) - fornece funcionalidades de análise e criação de URLs.
- [user](#): [CWebUser](#) - representa a identidade e informações do usuário atual.
- [themeManager](#): [CThemeManager](#) - gerencia temas.

Ciclo de Vida de uma Aplicação

Quando processa uma solicitação de um usuário, a aplicação segue o ciclo de vida descrito a seguir:

1. Pré-inicia a aplicação com o método `CApplication::preinit()`;
2. Configura as o auto-carregamento de classes (autoloader) e o tratamento de erros;
3. Registra os principais componentes da aplicação;
4. Carrega as configurações da aplicação;
5. Inicia a aplicação com o `CApplication::init()`:
 1. Registra os comportamentos (behaviors) da aplicação;
 2. Carrega os componentes estáticos da aplicação;
6. Dispara o evento `onBeginRequest` (no início da requisição);
7. Processa a requisição do usuário:
 1. Resolve a requisição do usuário;
 2. Cria um controle;

3. Executa o controle;
8. Dispara o evento onEndRequest (ao fim da requisição);

Controle

Um controle é uma instância de [CController](#) ou uma de suas classes derivadas. Ele é criado pela aplicação durante a requisição do usuário. Quando um controle entra em execução, ele também executa a ação requisitada, que, geralmente, recupera os modelos necessários e exibe a visão apropriada. Uma ação, em sua forma mais simples, nada mais é do que um método na classe do controle, cujo nome começa com action.

Um controle tem uma ação padrão. Quando a requisição do usuário não especifica qual ação executar, a ação padrão será utilizada. Por padrão, essa ação é chamada index. Ela pode ser alterada através da propriedade [CController::defaultAction](#).

Abaixo temos o código mínimo necessário para uma classe de controle. Uma vez que esse controle não define nenhuma ação, qualquer requisição feita para ele irá disparar uma exceção.

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

Rota

Controles e ações são identificados por seus IDs. O ID de um controle é representado no formato caminho/para/xyz, que corresponde ao arquivo de classe do controle em protected/controllers/caminho/para/XYZController.php, onde o xyz deve ser substituído pelos nomes dos controles (por exemplo, post corresponde a protected/controllers/PostController.php).

O ID de uma ação é o nome de seu método sem o prefixo action. Por exemplo, se um controle contém o método chamado actionEdit, o ID da ação correspondente será edit.

Nota: Antes da versão 1.0.3, o formato do ID para os controles era caminho.para.xyz em vez de caminho/para/xyz.

Usuários fazem requisições para uma ação e um controle em particular, por meio de uma rota. Uma rota, é formada concatenando-se o ID de um controle e o ID de uma ação, separados por uma barra. Por exemplo, a rota post/edit refere-se ao controle PostController e sua ação edit. Por padrão, a URL `http://hostname/index.php?r=post/edit` irá requisitar esse controle e essa ação.

Nota: Por padrão, as rotas fazem diferença entre maiúsculas e minúsculas. Desde a versão 1.0.1, é possível eliminar esse comportamento alterando o valor da propriedade `CUrlManager::caseSensitive` para `false`, na configuração da aplicação. Configurada dessa forma, tenha certeza de que você irá seguir a convenção de manter os nomes dos diretórios onde estão os controles e as chaves dos vetores em `controller map` e `action map` em letras minúsculas.

A partir da versão 1.0.3, uma aplicação pode conter [módulos](#). A rota para a ação de um controle dentro de um módulo deve estar no formato `ID modulo/IDcontrole/IDacao`. Para mais detalhes, veja a [seção sobre módulos](#).

Instanciação do Controle

Uma instância do controle é criada no momento em que a [CWebApplication](#) recebe um requisição do usuário. Dado o ID do controle, a aplicação irá utilizar as seguintes regras para determinar qual classe deve ser utilizada e onde ela está localizada:

- Se a propriedade [CWebApplication::catchAllRequest](#) for especificada, o controle será criado com base nela, ignorando o ID do controle requisitado pelo usuário. Essa propriedade é utilizada principalmente para colocar a aplicação em modo de manutenção e exibir uma página estática de notificação.
- Se o ID for encontrado em [CWebApplication::controllerMap](#), a configuração do controle correspondente será utilizada para instanciar o controle.
- Se o ID estiver no formato `caminho/para/xyz`, será assumido que o nome da classe é `XyzController` e seu arquivo correspondente está em `protected/controllers/caminho/para/XyzController.php`. Por exemplo, o ID de controle `admin/user` será resolvido para a classe `UserController` e seu arquivo correspondente será `protected/controllers/admin/UserController.php`. Se o arquivo não existir, um erro 404 [CHttpException](#) será disparado.

Nos casos em que [módulos](#) são utilizados (a partir da versão 1.0.3), o processo acima é um pouco diferente. Nessa situação em particular, a aplicação irá verificar se o ID refere-se a um controle dentro de um módulo e, caso positivo, o módulo será instanciado, primeiro, seguido da instanciação do controle.

Ação

Como já mencionado, uma ação pode ser definida como um método cujo nome começa com a palavra `action`. De uma maneira mais avançada, podemos definir uma ação como uma classe, e pedir para que o controle a instancie quando requisitada. Dessa forma, as ações podem ser reutilizadas com facilidade.

Para definir uma ação como uma classe, faça como no exemplo a seguir:

```
class UpdateAction extends CAction
{
    public function run()
    {
        // coloque a lógica aqui
    }
}
```

Para que o controle tenha conhecimento dessa ação, devemos sobrescrever o método [actions\(\)](#) na classe do controle:

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

No exemplo acima, utilizamos o path alias `application.controllers.post.UpdateAction` para especificar que a classe de ação está em `protected/controllers/post/UpdateAction.php`.

Trabalhando com ações baseadas em classes, podemos organizar nossa aplicação de forma modular. Por exemplo, a estrutura de diretórios a seguir pode ser utilizada para organizar o código para os controles:

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

Filtro

Um filtro é uma porção de código, configurada para ser executada antes e/ou depois que uma ação de um controle. Por exemplo, o filtro de controle de acesso deve ser utilizado para assegurar que o usuário está autenticado, antes de executar a ação requisitada; um filtro de desempenho pode ser utilizado para medir o tempo gasto para a execução de uma ação.

Uma ação pode ter vários filtros. Os filtros são executados na ordem em que aparecem na lista de filtros. Um filtro pode prevenir a execução da ação e dos demais filtros ainda não executados.

Um filtro pode ser definido como um método na classe do controle. O nome desse método deve começar como `filter`. Por exemplo, o método `filterAccessControl` define um filtro chamado `accessControl`. O método do filtro deve ter a seguinte assinatura:

```
public function filterAccessControl($filterChain)  
{  
    // utilize $filterChain->run() para continuar  
    //o processo de filtragem e executar a ação  
}
```

Acima, `$filterChain` é uma instância da classe [CFilterChain](#), que representa a lista de filtros associados com a ação requisitada. Dentro do método do filtro, podemos utilizar `$filterChain->run()` para continuar o processo de filtragem e executar a ação.

Um filtro também pode ser uma instância de [CFilter](#), ou de uma de suas classes derivadas. No código abaixo, vemos como definir um filtro como uma classe:

```

class PerformanceFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // lógica que será executada antes da ação
        return true; // deve retornar false caso a ação
                     // não deva ser executada
    }

    protected function postFilter($filterChain)
    {
        // lógica que será executada depois da ação
    }
}

```

Para aplicar os filtros às ações, precisamos sobrescrever o método `CController::filters()`. Esse método deve retornar um vetor com as configurações dos filtros. Por exemplo:

```

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

```

No código acima, especificamos dois filtros: `postOnly` e `PerformanceFilter`. O filtro `postOnly` é baseado em um método (o método correspondente já está definido na classe [CController](#)); enquanto o filtro `PerformanceFilter` é baseado em uma classe. O alias `application.filters.PerformanceFilter` especifica que a classe desse filtro está em `protected/filters/PerformanceFilter`. Utilizamos um vetor para configurar o filtro `PerformanceFilter`, assim podemos inicializar devidamente os valores de suas propriedades. Nesse caso, a propriedade `unit` dessa classe será inicializada com o valor `second`.

Utilizando-se os operadores `+` e `-`, podemos especificar a quais ações os filtros devem ou não ser aplicados. No último exemplo, o filtro `postOnly` deverá ser aplicado as ações `edit`

e create, enquanto o `PerformanceFilter` deve ser aplicado a todas as ações, EXCETO nas ações edit e create. Caso nenhum desses operadores seja especificado o filtro será aplicado a todas as ações.

Modelo

Um modelo é uma instância de `CModel` ou de suas classes derivadas. Modelos são utilizados para manter dados e suas regras de negócios relevantes.

Um modelo representa um objeto de dados único. Esse objeto pode ser um registro de uma tabela em um banco de dados ou um formulário com entradas de um usuário. Cada campo do objeto de dados é representado, no modelo, por um atributo. Cada atributo tem um rótulo e pode ser validado por um conjunto de regras.

O Yii implementa dois tipos de modelos: form model (modelo de formulário) e active record. Ambos estendem a mesma classe base `CModel`.

Um form model é uma instância da classe `CFormModel`. Ele é utilizado para manter dados coletados a partir de entradas de usuários. Esse tipo de dado geralmente é coletado, utilizado e, então, descartado. Por exemplo, em uma página de login, podemos utilizar um form model para representar as informações de nome de usuário e senha inseridas pelo usuário. Para mais detalhes, consulte [Trabalhando com formulários](#).

Active Record (AR) é um padrão de projeto utilizado para abstrair o acesso ao banco de dados de uma maneira orientada a objetos. Cada objeto AR é uma instância da classe `CActiveRecord`, ou de suas classes derivadas, representando um registro de uma tabela em um banco de dados. Os campos do registro são representados por propriedades do objeto AR. Mais detalhes sobre AR podem ser encontrados em: [Active Record](#).

Visão

Uma visão é um script em PHP contendo, principalmente, elementos da interface com usuário. Ela pode ter comandos em PHP, mas é recomendado que tais comandos não alterem dados em modelos e sejam simples. De acordo com a ideia da separar a lógica da programação, lógicas complexas devem ser colocadas no controle ou no modelo em vez da visão.

Uma visão tem um nome que é utilizado para identificar seu arquivo de script no momento da renderização. O nome da visão é o mesmo de seu arquivo. Por exemplo, a visão edit refere-se ao arquivo denominado `edit.php`. Para renderizar uma visão, execute [`CController::render\(\)`](#) informando o nome da visão. O método irá procurar pelo arquivo correspondente no diretório `protected/views/ControllerID`.

Dentro do arquivo da visão, podemos acessar a instância do controle a partir da variável `$this`. Dessa forma, podemos acessar qualquer propriedade do controle utilizando `$this->nomeDaPropriedade` dentro da visão.

Podemos também inserir dados na visão da seguinte maneira:


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

No código acima, o método [render\(\)](#) irá extrair os valores do vetor, passado no segundo parâmetro, para variáveis. Como resultado, dentro do script da visão poderemos acessar as variáveis locais \$var1 e \$var2.

Layout

O Layout é uma visão especial, utilizada para decorar outras visões. Normalmente ele contém partes da interface que são comuns entre diversas visões. Por exemplo, um layout pode conter um cabeçalho e um rodapé e inserir o conteúdo de uma visão entre eles:

```
.....cabeçalho aqui.....
<?php echo $content; ?>
.....rodapé aqui.....
```

No trecho acima \$content contém o conteúdo renderizado da visão.

O layout é implicitamente aplicado ao executar o método [render\(\)](#). Por padrão, a visão localizada em `protected/views/layouts/main.php` é utilizada como layout. No entanto, isso pode ser personalizado modificando a propriedade [CWebApplication::layout](#) ou [CController::layout](#). Para renderizar uma visão sem aplicar um layout, utilize o método [renderPartial\(\)](#).

Widget

Um widget é uma instância da classe [CWidget](#), ou de suas classes derivadas. É um componente utilizado principalmente para apresentação. Widgets normalmente são utilizados dentro do código de uma visão para gerar elementos complexos, porém independentes. Por exemplo, um widget pode ser utilizado para renderizar um complexo calendário. Widgets adicionam melhor reutilização na interface com o usuário.

Para utilizar um widget, faça o seguinte na sua visão:

```
<?php $this->beginWidget('path.to.WidgetClass'); ?>
...conteúdo que deve aparecer no corpo do widget...
<?php $this->endWidget(); ?>
```

ou

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

A última forma apresentada é utilizada quando o widget não necessita de conteúdo.

Os Widgets podem ter seus comportamentos personalizados. Para isso, configure os valores iniciais de suas propriedades ao executar [CBaseController::beginWidget](#) ou [CBaseController::widget](#). Por exemplo, quando utilizamos o widget [CMaskedTextField](#), podemos especificar a máscara que será utilizada. Para isso, passamos para o método um vetor onde as chaves são os nomes das propriedades e os valores são os valores iniciais desejados, como no trecho de código a seguir:

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

Para definir um novo widget, estenda a classe [CWidget](#) e sobrescreva os métodos [init\(\)](#) e [run\(\)](#).

```
class MyWidget extends CWidget
{
    public function init()
    {
        // esse método é chamado por CController::beginWidget()
    }

    public function run()
    {
        // esse método é chamado por CController::endWidget()
    }
}
```

Assim como um controle, um widget também pode ter uma visão. Por padrão, os arquivos de visão de um widget estão localizados no diretório views, dentro do diretório onde está localizado o arquivo de classe do widget. Essas visões podem ser renderizadas executando-se o método `CWidget::render()`, assim como o método de mesmo nome existente no controle. A única diferença entre eles é que nenhum layout será aplicado a visão do widget.

Visão do Sistema (System View)

Visões do sistema são as visões utilizadas pelo Yii para exibir informações sobre erros e logs. Por exemplo, quando um usuário requisita uma ação ou controle não existente, o Yii irá lançar uma exceção explicando o erro ocorrido. Esse erro é exibido utilizando uma visão do sistema.

A nomenclatura das visões do sistema seguem algumas regras. Nomes no formato `errorXXX` referem-se à visões utilizadas para exibir erros gerados com a exceção `CHttpException`, onde `XXX` é o código do erro. Por exemplo, se uma `CHttpException` é gerada com o código de erro 404, a visão `error404` será exibida.

O Yii framework já possui um conjunto padrão de visões do sistema, localizadas em `framework/views`. Elas podem ser personalizadas criando-se visões com o mesmo nome que as originais em `protected/views/system`.

Componente

As aplicações feitas com o Yii são construídas por componentes, que são objetos desenvolvidos para um fim específico. Um componente é uma instância da classe `CComponent`, ou uma de suas classes derivadas. A utilização de um componente basicamente envolve o acesso à suas propriedades e a execução/manipulação de seus eventos. A classe base `CComponent` especifica como definir propriedades e eventos.

Propriedade de um Componente

Uma propriedade de um componente é como uma variável membro pública de um objeto. Nós podemos ler seu conteúdo ou lhe atribuir novos valores. Por exemplo:

```
$width=$component->textWidth;    // acessa a propriedade textWidth
$component->enableCaching=true;   // altera a propriedade enableCaching
```

Para definir uma propriedade em um componente, podemos simplesmente declarar uma variável membro pública na classe do componente. No entanto, existe uma maneira mais flexível, que consiste em definir métodos acessores (getters e setters), como no exemplo a seguir:

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

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

No código acima, definimos uma variável alterável chamada `textWidth` (o nome é case-insensitive, não faz diferença entre maiúsculas e minúsculas). Ao acessar a propriedade, o método `getTextWidth()` é executado e seu valor de retorno é o valor da propriedade. De maneira parecida, ao alterar o valor da propriedade, utilizamos o método `setTextWidth()`. Se o método setter não for definido, a propriedade será do tipo somente leitura e a tentativa de alterar seu valor lançará uma exceção.

Nota: Existe uma pequena diferença entre uma propriedade definida via métodos acessores (getters e setters) e variáveis membros de classe. No primeiro caso, o nome da variável é case-insensitive, enquanto que, no último, o nome é case-sensitive (há diferença entre maiúsculas de minúsculas).

Eventos de um Componente

Os eventos de um componente são propriedades especiais que aceitam métodos (chamados de event handlers, manipuladores de eventos) como seus valores. Ao atribuir um método a um evento, fará com que ele seja executado cada vez que o evento for disparado. Portanto, o comportamento de um componente pode ser alterado para funcionar de maneira diferente de como foi desenvolvido.

Um evento pode ser criado definindo-se um método cujo nome inicie com on. Assim como as propriedades definidas via métodos acessores, os nomes de eventos também são case-insensitive. O código abaixo define um evento chamado onCliked:

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

No exemplo acima, \$event é uma instância da classe [CEvent](#), ou de uma de suas classes derivadas, e está representando o parâmetro do evento.

Podemos atribuir um método para esse evento da seguinte maneira:

```
$component->onCliked=$callback;
```

Onde \$callback refere-se a um callback válido em PHP. Ele pode ser uma função global ou um método de classe. No último caso, o callback deve ser passado como um vetor no formato: array(\$objeto, 'nomeDoMetodo').

A assinatura de um manipulador de evento deve ser a seguinte:

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

Onde \$event é o parâmetro descrevendo o evento ocorrido (originado na chamada do método raiseEvent()). O parâmetro \$event é uma instância da classe [CEvent](#), ou uma de suas classes derivadas, e, no mínimo, ele contém a informação sobre quem originou o evento.

Agora, se executarmos o método onCliked(), o evento onCliked será disparado e o manipulador de evento a ele atribuído será invocado automaticamente.

Um evento pode ter diversos manipuladores. Quando o evento é disparado, os manipuladores serão executados, um a um, na ordem em que foram atribuídos ao evento. Um manipulador pode impedir que os manipuladores restantes sejam executados, para isso altere o valor de [\\$event->handled](#) para true.

Comportamento de um Componente

A partir da versão 1.0.2, os componentes passaram a suportar [mixin](#), e, portanto, ganharam a possibilidade de receber um ou mais comportamentos. Um comportamento (behavior) é um objeto cujo os métodos podem ser "herdados" pela classe que o anexou, com a finalidade de coleta de funcionalidades em vez de especialização (por exemplo, a herança normal de classes). Um componente pode receber diversos comportamentos e, assim, utilizar a "herança múltipla".

Classes de comportamento devem implementar a interface [IBehavior]. A maioria dos comportamentos podem utilizar a classe [CBehavior](#) como base, estendendo-a. Se um comportamento precisar ser atribuído a um [modelo](#), ele deve estender as classes [CModelBehavior](#) ou [CActiveRecordBehavior](#), que implementam características específicas para modelos.

Para utilizar um comportamento, primeiro ele deve ser atribuído a um componente, utilizando o método [attach()|Behavior::attach]. Em seguida, podemos executar o comportamento através do componente, da seguinte maneira:

```
// $nome identifica unicamente o comportamento dentro do componente
$componente->attachBehavior($nome, $comportamento);
// test() é um método de $comportamento
$componente->test();
```

Um comportamento atribuído pode ser acessado normalmente, como uma propriedade do componente. Por exemplo, se um comportamento denominado tree é atribuído a um componente, podemos obter uma referência dele da seguinte maneira:

```
$behavior=$component->tree;
// equivalente a:
// $behavior=$component->asa('tree');
```

Podemos desabilitar um comportamento temporariamente, para que seus métodos não estejam disponíveis através do componente:

```
$component->disableBehavior($name);  
// a linha abaixo irá gerar uma exceção  
$component->test();  
$component->enableBehavior($name);  
// agora funciona  
$component->test();
```

É possível que dois comportamentos atribuídos ao mesmo componente tenham métodos com o mesmo nome. Nesse caso, o método do comportamento atribuído primeiro terá precedência.

Os comportamentos são ainda mais poderosos quando utilizado com [eventos](#). Um comportamento, quando atribuído a um componente, pode utilizar alguns de seus métodos como callbacks para os eventos do componente. Dessa forma, o comportamento tem a possibilidade de observar ou alterar o fluxo de execução do componente.

Module

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

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

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

Creating Module

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

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

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

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

Using Module

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

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

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

```

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

```

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

```

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

```

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

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

Nested Module

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

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

Path Alias e Namespace

O Yii utiliza path aliases (apelidos para caminhos) extensivamente. Um path alias, é um apelido associado ao caminho de um diretório ou arquivo. Um path alias utiliza a sintaxe de ponto para separar seus itens, similar a forma largamente adotada em namespaces:


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

Onde `RootAlias` é o nome de um diretório existente. Ao executar o método `YiiBase::setPathOfAlias()`, podemos definir novos apelidos para caminhos. Por conveniência, o Yii já possui predefinidos os seguintes apelidos:

- `system`: refere-se ao diretório do Yii framework;
- `application`: refere-se ao [diretório base](#) da aplicação;
- `webroot`: refere-se ao diretório que contém o arquivo do [script de entrada](#). Esse apelido está disponível desde a versão 1.0.3.
- `ext`: refere-se ao diretório que contém todas as [extensões](#) de terceiros. Esse apelido está disponível desde a versão 1.0.8.

Além disso, se a aplicação utiliza [módulos](#), um apelido de diretório raiz (root alias) é predefinido para cada módulo, apontando para o diretório base do módulo correspondente. Esta funcionalidade está disponível desde a versão 1.0.3.

Ao usar o método `YiiBase::getPathOfAlias()`, um apelido pode ser traduzido para o seu caminho correspondente. Por exemplo, `system.web.CController` seria traduzido para `yii/framework/web/CController`.

A utilização de apelidos é muito conveniente para importar a definição de uma classe. Por exemplo, se quisermos incluir a definição da classe `CController` podemos fazer o seguinte:

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

O método `import` é mais eficiente que o `include` e o `require` do PHP. Com ele, a definição da classe que está sendo importada não é incluída até que seja referenciada pela primeira vez. Importar o mesmo namespace várias vezes, também é muito mais rápido do que utilizar o `include_once` e o `require_once`.

Dica: Quando referenciamos uma das classes do Yii Framework, não precisamos importa-la ou inclui-la. Todas as classes Yii são pré-importadas.

Podemos também utilizar a seguinte sintaxe para importar todo um diretório de uma só vez, de forma que os arquivos de classe dentro dele sejam automaticamente incluídos, quando necessário.

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

Além do método `import`, apelidos são utilizados em vários outros locais para se referir a classes. Por exemplo, um apelido pode ser passado para o método `Yii::createComponent()` para criar uma instância da classe informada, mesmo que o arquivo da classe ainda não tenha sido incluído.

Não confunda um path alias com um namespace. Um namespace refere-se a um agrupamento lógico de nomes de classes para que eles possam ser diferenciadas de

outros nomes das classes, mesmo que eles sejam iguais. Já um path alias é utilizado para referenciar um arquivo de classe ou um diretório. Um path alias não conflita com um namespace.

Dica: Como o PHP, antes da versão 5.3.0, não dá suporte a namespaces, você não pode criar instâncias de duas classes que tenham o mesmo nome, mas definições diferentes. Por isso, todas as classes do Yii framework são prefixadas com uma letra "C" (que significa 'class'), de modo que elas possam ser diferenciadas das classes definidas pelo usuário. Recomenda-se que o prefixo "C" seja reservado somente para utilização do Yii framework, e que classes criadas pelos usuário sejam prefixadas com outras letras.

Convenções

O Yii favorece convenções sobre configurações. Siga as convenções e você poderá criar aplicações sofisticadas sem ter que escrever ou gerenciar configurações complexas. Evidentemente, o Yii ainda podem ser personalizados em quase todos os aspectos, com configurações, quando necessário.

Abaixo descrevemos convenções que são recomendadas para programar com o Yii. Por conveniência, assumimos que WebRoot é o diretório onde está instalada uma aplicação desenvolvida com o Yii framework.

URL

Por padrão, o Yii reconhece URLs com o seguinte formato:

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

A variável r, passada via GET, refere-se a [rota](#), que pode ser interpretada pelo Yii como um controle e uma ação. Se o id da ação (ActionID) for omitido, o controle irá utilizar a ação padrão (definida através da propriedade [CController::defaultAction](#)); e se o id do controle (ControllerID) também for omitido (ou a variável r estiver ausente), a aplicação irá utilizar o controle padrão (definido através da propriedade [CWebApplication::defaultController](#)).

Com a ajuda da classe [CUrlManager](#), é possível criar e reconhecer URLs mais amigáveis, ao estilo SEO, tais como <http://hostname/ControllerID/ActionID.html>. Esta funcionalidade é abordada em detalhes em [Gerenciamento de URL](#).

Código

O Yii recomenda que nomes de variáveis, funções e nomes de classe sejam escritos no formato Camel Case, onde inicia-se cada palavra com letra maiúscula e junta-se todas, sem espaços entre elas. Variáveis e nomes de funções devem ter a sua primeira palavra totalmente em letras minúsculas, a fim de diferencia-los dos nomes das classes (por exemplo, \$basePath, runController(), LinkPager). Para as variáveis privadas membros de classe, é recomendado prefixar seus nomes com um underscore (por exemplo, \$_actionList).

Como não há suporte a namespaces antes do PHP 5.3.0, é recomendado que as classes sejam denominadas de uma forma única, para evitar conflitos com nomes de classes de terceiros. Por esta razão, todas as classes do Yii framework são prefixadas com a letra "C".

Existe uma regra especial para as classes de controle, onde deve-se adicionar o sufixo Controller ao nome da classe. O ID do controle é, então, definido como o nome da classe, com a primeira letra minúscula, e a palavra Controller removida. Por exemplo, a classe PageController terá o ID page. Esta regra torna a aplicação mais segura. Também deixa mais limpas as URLs relacionados aos controles (por exemplo, /index.php?r=page/index em vez de /index.php?r=PageController/index).

Configuração

A configuração é um vetor de pares chave-valor. Cada chave representa o nome de uma propriedade do objeto a ser configurado, e cada valor, o valor inicial da propriedade correspondente. Por exemplo, `array('name'=>'Minha aplicação', 'basePath'=>'/protected')` inicializa as propriedades name e basePath com os valores correspondentes no vetor.

Qualquer propriedades "alterável" de um objeto pode ser configurada. Se não forem configuradas, as propriedades assumirão seus valores padrão. Ao configurar uma propriedade, vale a pena ler a documentação correspondente, para que o valor inicial seja configurado corretamente.

Arquivo

As convenções para nomenclatura e utilização de arquivos dependem seus tipos.

Arquivos de classe devem ser nomeados de acordo com a classe pública que contém.

Por exemplo, a classe [CController](#) está no arquivo CController.php. Uma classe pública é uma classe que pode ser utilizada por qualquer outra. Cada arquivo de classe deve conter, no máximo, uma classe pública. Classes privadas (aquelas que são utilizadas apenas por uma única classe pública) podem residir no mesmo arquivo com a classe que a utiliza.

Os arquivos das visões devem ser nomeados de acordo com o seus nomes. Por exemplo, a visão index está no arquivo index.php. O arquivo de uma visão contém um script com código HTML e PHP, utilizado, principalmente para apresentação de conteúdo.

Arquivos de configuração podem ser nomeadas arbitrariamente. Um arquivo de configuração é um script em PHP cuja única finalidade é a de retornar um vetor associativo representando a configuração.

Diretório

O Yii assume um conjunto predefinido de diretórios utilizados para diversas finalidades. Cada um deles pode ser personalizado, se necessário.

- WebRoot/protected: este é o [diretório base da aplicação](#), onde estão todos os scripts PHP que precisam estar seguros e os arquivos de dados. O Yii tem um apelido (alias)

padrão chamado application, associado a este caminho. Este diretório, e tudo dentro dele, deve estar protegido para não ser acessado via web. Ele pode ser alterado através da propriedade [CWebApplication::basePath](#).

- WebRoot/protected/runtime: este diretório armazena arquivos privados temporários gerados durante a execução da aplicação. Este diretório deve ter permissão de escrita para o processo do servidor Web. Ele pode ser alterado através da propriedade [CApplication::runtimePath](#).
- WebRoot/protected/extensions: este diretório armazena todas as extensões de terceiros. Ele pode ser alterado através da propriedade [CApplication::extensionPath](#).
- WebRoot/protected/modules: este diretório contém todos os [módulos](#) da aplicação, cada um representado como um subdiretório.
- WebRoot/protected/controllers: neste diretório estão os arquivos de classe de todos os controles. Ele pode ser alterado através da propriedade [CWebApplication::controllerPath](#).
- WebRoot/protected/views: este diretório possui todos os arquivos das visões, incluindo as visões dos controles, visões do layout e visões do sistema. Ele pode ser alterado através da propriedade [CWebApplication::viewPath](#).
- WebRoot/protected/views/ControllerID: neste diretório estão os arquivos das visões para um controle específico. Aqui, ControllerID é o ID do controle. Ele pode ser alterado através da propriedade [CController::getViewPath](#).
- WebRoot/protected/views/layouts: este diretório possui todos os arquivos de visão do layout. Ele pode ser alterado através da propriedade [CWebApplication::layoutPath](#).
- WebRoot/protected/views/system: este diretório mantém todos os arquivos de visões do sistema. Visões do sistema são templates utilizados para exibir exceções e erros. Ele pode ser alterado através da propriedade [CWebApplication::systemViewPath](#).
- WebRoot/assets: este diretório mantém os assets publicados. Um asset é um arquivo privado que pode ser publicado para se tornar acessível aos usuários, via web. Este diretório deve ter permissão de escrita para o processo do servidor Web. Ele pode ser alterado através da propriedade [CAssetManager::basePath](#).
- WebRoot/themes: este diretório armazena vários temas que podem ser aplicados à aplicação. Cada subdiretório representa um único tema cujo nome é o nome do tema. Ele pode ser alterado através da propriedade [CThemeManager::basePath](#).

Fluxo de trabalho do desenvolvimento

Após descrever os conceitos fundamentais do Yii, mostraremos o fluxo de trabalho do desenvolvimento de uma aplicação web utilizando o Yii. Esse fluxo assume que já realizamos a análise de requisitos, bem como a análise de design para a aplicação.

1. Crie o esqueleto da estrutura de diretórios. A ferramenta yiic, descrita em criando a primeira aplicação Yii pode ser utilizada para agilizar este passo.
2. Configure a aplicação. Faça isso alterando o arquivo de configuração da aplicação. Neste passo, também pode ser necessário escrever alguns componentes da aplicação (por exemplo, o componente de usuário).
3. Crie um modelo para cada tipo de dado a ser gerenciado. Novamente, podemos utilizar a yiic para gerar automaticamente as classes active record para cada tabela importante do banco de dados.
4. Crie uma classe de controle para cada tipo de requisição do usuário. A classificação dessas requisições depende dos requisitos da aplicação. No geral, se um modelo precisa ser acessado pelo usuário, ele deve ter um controle correspondente. A ferramenta yiic pode automatizar este passo para você.

5. Implemente ações e as visões. Aqui é onde o trabalho de verdade precisa ser feito.
6. Configure os filtros de ações necessários nas classes dos controles.
7. Crie temas, se esta funcionalidade for necessária.
8. Crie mensagens traduzidas se internacionalização for necessária.
9. Identifique dados e visões que possam ser cacheadas e aplique as técnicas apropriadas de caching.
10. Finalmente, faça ajustes de desempenho e a implantação.

Para cada um dos passos acima, testes devem ser criados e executados.

Melhores Práticas em MVC

Although Model-View-Controller (MVC) is known by nearly every Web developer, how to properly use MVC in real application development still eludes many people. The central idea behind MVC is code reusability and separation of concerns. In this section, we describe some general guidelines on how to better follow MVC when developing a Yii application.

To better explain these guidelines, we assume a Web application consists of several sub-applications, such as

- front end: a public-facing website for normal end users;
- back end: a website that exposes administrative functionality for managing the application. This is usually restricted to administrative staff;
- console: an application consisting of console commands to be run in a terminal window or as scheduled jobs to support the whole application;
- Web API: providing interfaces to third parties for integrating with the application.

The sub-applications may be implemented in terms of [modules](#), or as a Yii application that shares some code with other sub-applications.

Model

[Models](#) represent the underlying data structure of a Web application. Models are often shared among different sub-applications of a Web application. For example, a LoginForm model may be used by both the front end and the back end of an application; a News model may be used by the console commands, Web APIs, and the front/back end of an application. Therefore, models

- should contain properties to represent specific data;
- should contain business logic (e.g. validation rules) to ensure the represented data fulfills the design requirement;
- may contain code for manipulating data. For example, a SearchForm model, besides representing the search input data, may contain a search method to implement the actual search.

Sometimes, following the last rule above may make a model very fat, containing too much code in a single class. It may also make the model hard to maintain if the code it contains serves different purposes. For example, a News model may contain a method named `getLatestNews` which is only used by the front end; it may also contain a method named `getDeletedNews` which is only used by the back end. This may be fine for an application of

small to medium size. For large applications, the following strategy may be used to make models more maintainable:

- Define a NewsBase model class which only contains code shared by different sub-applications (e.g. front end, back end);
- In each sub-application, define a News model by extending from NewsBase. Place all of the code that is specific to the sub-application in this News model.

So, if we were to employ this strategy in our above example, we would add a News model in the front end application that contains only the `getLatestNews` method, and we would add another News model in the back end application, which contains only the `getDeletedNews` method.

In general, models should not contain logic that deals directly with end users. More specifically, models

- should not use `$_GET`, `$_POST`, or other similar variables that are directly tied to the end-user request. Remember that a model may be used by a totally different sub-application (e.g. unit test, Web API) that may not use these variables to represent user requests. These variables pertaining to the user request should be handled by the Controller.
- should avoid embedding HTML or other presentational code. Because presentational code varies according to end user requirements (e.g. front end and back end may show the detail of a news in completely different formats), it is better taken care of by views.

View

[Views](#) are responsible for presenting models in the format that end users desire. In general, views

- should mainly contain presentational code, such as HTML, and simple PHP code to traverse, format and render data;
- should avoid containing code that performs explicit DB queries. Such code is better placed in models.
- should avoid direct access to `$_GET`, `$_POST`, or other similar variables that represent the end user request. This is the controller's job. The view should be focused on the display and layout of the data provided to it by the controller and/or model, but not attempting to access request variables or the database directly.
- may access properties and methods of controllers and models directly. However, this should be done only for the purpose of presentation.

Views can be reused in different ways:

- Layout: common presentational areas (e.g. page header, footer) can be put in a layout view.
- Partial views: use partial views (views that are not decorated by layouts) to reuse fragments of presentational code. For example, we use `_form.php` partial view to render the model input form that is used in both model creation and updating pages.
- Widgets: if a lot of logic is needed to present a partial view, the partial view can be turned into a widget whose class file is the best place to contain this logic. For

widgets that generate a lot of HTML markup, it is best to use view files specific to the widget to contain the markup.

- Helper classes: in views we often need some code snippets to do tiny tasks such as formatting data or generating HTML tags. Rather than placing this code directly into the view files, a better approach is to place all of these code snippets in a view helper class. Then, just use the helper class in your view files. Yii provides an example of this approach. Yii has a powerful [CHtml](#) helper class that can produce commonly used HTML code. Helper classes may be put in an [autoloadable directory](#) so that they can be used without explicit class inclusion.

Controller

[Controllers](#) are the glue that binds models, views and other components together into a runnable application. Controllers are responsible for dealing directly with end user requests. Therefore, controllers

- may access `$_GET`, `$_POST` and other PHP variables that represent user requests;
- may create model instances and manage their life cycles. For example, in a typical model update action, the controller may first create the model instance; then populate the model with the user input from `$_POST`; after saving the model successfully, the controller may redirect the user browser to the model detail page. Note that the actual implementation of saving a model should be located in the model instead of the controller.
- should avoid containing embedded SQL statements, which are better kept in models.
- should avoid containing any HTML or any other presentational markup. This is better kept in views.

In a well-designed MVC application, controllers are often very thin, containing probably only a few dozen lines of code; while models are very fat, containing most of the code responsible for representing and manipulating the data. This is because the data structure and business logic represented by models is typically very specific to the particular application, and needs to be heavily customized to meet the specific application requirements; while controller logic often follows a similar pattern across applications and therefore may well be simplified by the underlying framework or the base classes.

Trabalhando com formulários

Visão Geral

Coletar dados do usuário através de formulários HTML é uma das principais tarefas no desenvolvimento de aplicativos Web. Além de conceder formulários, os desenvolvedores precisam preencher o formulário com dados existentes ou valores, validar entrada do usuário, exibir mensagens de erros apropriadas para uma entrada inválida, e salvar o entrada para o armazenamento persistente. Yii simplifica muito este trabalho com a sua arquitetura MVC.

Os seguintes passos são tipicamente necessários ao tratar os formulários em Yii:

1. Crie uma classe modelo que representa os campos de dados a serem coletados;
2. Crie um controlador de ação com o código que responde à submissão do formulário.
3. Crie um arquivo de visualização do formulário associado com o controlador de ação.

Nas próximas subseções, descreveremos cada uma dessas etapas com mais detalhes.

Criando um Modelo

Antes de escrever o código HTML necessário para um formulário, devemos decidir quais tipos de dados esperamos dos usuários e a quais regras eles devem estar de acordo.

Uma classe de modelo pode ser utilizada para registrar essas informações. Um modelo, como descrito em [Modelo](#), é o lugar central para manter as entradas fornecidas pelo usuário e validá-las.

Dependendo da forma como utilizamos as entradas dos usuários, podemos criar dois tipos de modelo. Se os dados são coletados, utilizados e, então, descartados, devemos criar um [modelo de formulário](#) (form model); porém, se a entrada do usuário deve ser coletada e armazenada em uma base de dados, devemos utilizar um [active record](#).

Ambos os tipos de modelo compartilham [CModel](#) como classe base, onde está definida uma interface comum necessária a formulários.

Nota: Nós utilizaremos modelos de formulários nos exemplos desta seção. Entretanto, o mesmo pode ser aplicado para modelos utilizando [active record](#).

Definindo uma Classe de Modelo

No trecho de código abaixo, criamos uma classe de modelo chamada LoginForm que será utilizada para coletar os dados informados pelo usuário em uma página de login. Como essa informação é utilizada somente para autenticar o usuário e não necessita ser armazenada, criaremos a classe LoginForm como um modelo de formulário.


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

Foram declarados três atributos na classe LoginForm: \$username, \$password, e \$rememberMe. Eles são utilizados para manter o nome de usuário e senha informados no formulário, bem como a opção se ele deseja que o sistema se lembre de seu login. Como o valor padrão de \$rememberMe é false, a opção correspondente no formulário de login estará, inicialmente, desmarcada.

Informação: Em vez de chamarmos essas variáveis membro de propriedades, utilizamos o termo atributos para diferenciá-las de propriedades normais. Um atributo é uma propriedade utilizada, basicamente, para armazenar dados originados de entradas de usuários ou do banco de dados.

Declarando Regras de Validação

Uma vez que o usuário envia seus dados e o modelo é preenchido com eles, devemos garantir que essas informações sejam validadas antes de serem utilizadas. Para isso, utilizamos um conjunto de regras que são testadas contra os dados informados. Para especificar essas regras de validação, utilizamos o método rules(), que deve retornar um vetor contendo as configurações de regras.

```

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()) // devemos autenticar o usuário somente
                                // se não existir erros de validação
        {
            $identity=new UserIdentity($this->username,$this->password);
            if($identity->authenticate())
            {
                $duration=$this->rememberMe ? 3600*24*30 : 0; // 30 dias
                Yii::app()->user->login($identity,$duration);
            }
            else
                $this->addError('password','Senha Incorreta.');
```

No código acima, especificamos que username e password são obrigatórios (required). Além disso, definimos que password deve ser autenticado (authenticate).

Cada regra retornada pelo método rules() deve estar no seguinte formato:

```

array('ListaDeAtributos', 'Validador', 'on'=>'ListaDeCenarios', ...opções
adicionais)
```

Onde, ListaDeAtributos é uma string contendo todos os atributos, separados por vírgula, que devem ser validados de acordo com a regra; Validador determina que tipo de

validação deverá ser efetuada; o parâmetro `on` é opcional e é utilizado para especificar uma lista de cenários onde a regra deve ser aplicada; opções adicionais são pares chave-valor, utilizados para iniciar as propriedades do validador.

Existem três maneiras de especificar o Validador em uma regra. Primeira, Validador pode ser o nome de um método na classe do modelo, como o `authenticate` no exemplo acima.

Nesse caso, o método validador deve ter a seguinte assinatura:

```
/**
 * @param string o nome do atributo a ser validado
 * @param array opções especificadas na regra de validação
 */
public function nomeDoValidador($atributo,$parametros) { ... }
```

Segunda, Validador pode ser o nome de uma classe validadora. Dessa maneira, quando a regra é aplicada, uma instância dessa classe será criada para efetuar a validação. As opções adicionais na regra serão utilizadas para iniciar os valores dos atributos da instância. Uma classe validadora deve estender a classe [CValidator](#).

Nota: Quando especificamos regras para um modelo active record, podemos utilizar a opção especial `on`. Os valores dessa opção podem ser `insert` ou `update`, de forma que a regra seja aplicada somente ao inserir ou atualizar um registro. Se não for utilizada, a regra será aplicada em ambos os casos, quando o método `save()` for utilizado.

Terceira, Validador pode ser um alias (apelido) predefinido para uma classe validadora. No exemplo acima, o nome `required` é um alias para a classe [CRequiredValidator](#), a qual valida se o valor do atributo não está vazio. Abaixo, temos uma lista completa dos aliases (apelidos) predefinidos:

- `boolean`: alias para [CBooleanValidator](#), garante que o valor de um atributo seja somente [CBooleanValidator::trueValue](#) ou [CBooleanValidator::falseValue](#).
- `captcha`: alias para [CCaptchaValidator](#), garante que o atributo é igual ao código de verificação exibido em um [CAPTCHA](#).
- `compare`: alias para [CCompareValidator](#), garante que o atributo é igual a outro atributo ou a uma constante.
- `email`: alias para [CEmailValidator](#), garante que o atributo é um endereço de email válido.
- `default`: alias para [CDefaultValueValidator](#), utilizado para atribuir um valor padrão (default) aos atributos especificados.
- `exist`: alias para [CExistValidator](#), garante que o valor do atributo existe na coluna da tabela informada.
- `file`: alias para [CFileValidator](#), garante que o atributo contém o nome de um arquivo enviado via upload.
- `filter`: alias para [CFilterValidator](#), modifica o atributo com um filtro.
- `in`: alias para [CRangeValidator](#), garante que o dado informado está entre uma lista específica de valores.
- `length`: alias para [CStringValidator](#), garante que o tamanho do dado está dentro de um tamanho específico.

- match: alias para [CRegularExpressionValidator](#), garante que o dado informado casa com um expressão regular.
- numerical: alias para [CNumberValidator](#), garante que o dado informado é um número válido.
- required: alias para [CRequiredValidator](#), garante que o valor do atributo não está vazio.
- type: alias para [CTypeValidator](#), garante que o atributo é de um tipo específico.
- unique: alias para [CUniqueValidator](#), garante que o dado informado é único na coluna da tabela do banco de dados informada.
- url: alias para [CUrlValidator](#), garante que o dado informado é uma URL válida.

Abaixo listamos alguns exemplos da utilização de validadores predefinidos:

```
// username é obrigatório
array('username', 'required'),
// username deve ter entre 3 e 12 caracteres
array('username', 'length', 'min'=>3, 'max'=>12),
// quando estiver no cenário register, password deve ser igual password2
array('password', 'compare', 'compareAttribute'=>'password2',
'on'=>'register'),
// quando estiver no cenário login, password deve ser autenticado
array('password', 'authenticate', 'on'=>'login'),
```

Atribuição Segura de Atributos

Nota: a atribuição de atributos baseada em cenários está disponível desde a versão 1.0.2 do framework.

Normalmente, depois que uma instância de um modelo é criada, precisamos popular seus atributos com as informações enviadas pelo usuário. Isso pode ser feito de uma maneira conveniente, utilizando a atribuição em massa, como pode ser visto no código abaixo:

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

Nota: A propriedade [scenario](#) está disponível desde a versão 1.0.4. A atribuição em massa irá utilizar o valor dessa propriedade para determinar quais atributos podem ser atribuídos dessa maneira. Nas versões 1.0.2 e 1.0.3, para fazer a atribuição em massa em um cenário específico, deveríamos proceder da seguinte maneira:

```
$model->setAttributes($_POST['LoginForm'], 'login');
```

Nesse trecho de código, temos uma atribuição em massa que atribui cada entrada em `$_POST['LoginForm']` ao atributo correspondente no modelo, no cenário login. Isso é equivalente a:

```
foreach($_POST['LoginForm'] as $name=>$value)
{
    if($name é um atributo seguro)
        $model->$name=$value;
}
```

A tarefa de decidir se um dado é seguro ou não é baseada no valor de retorno do método `safeAttributes` e o cenário especificado. Por padrão, esse método retorna todas as variáveis membro públicas como atributos seguros para a classe [CFormModel](#), ou todas as colunas de uma tabela, menos a chave primária, como atributos para a classe [CActiveRecord](#). Nós podemos sobrescrever este método para limitar esses atributos seguros de acordo com os cenários. Por exemplo, um modelo usuário deve conter vários atributos, mas no cenário login, precisamos apenas do username e do password. Podemos especificar esses limites da seguinte maneira:

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

Mais precisamente, o valor de retorno do método `safeAttributes` deve ter a seguinte estrutura:

```
array(
    // esses atributos podem ser atribuídos em massa em qualquer cenário
    // isso não ser explicitamente especificado, como vemos abaixo
    'attr1, attr2, ...',
    *
    // esses atributos só podem ser atribuídos em massa no cenário 1
    'cenario1' => 'attr2, attr3, ...',
    *
    // esses atributos só podem ser atribuídos em massa no cenário 2
    'cenario2' => 'attr1, attr3, ...',
)
```

Se os cenários não são importantes para o modelo, ou se todos os cenários tem o mesmo conjunto de atributos, o valor de retorno pode ser simplificado para um simples string:

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

Para dados não seguros, devemos atribui-los individualmente aos atributos, como no exemplo a seguir:

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

Disparando a Validação

Uma vez que o modelo tenha sido populado com os dados enviados pelo usuário, podemos executar o método [CModel::validate\(\)](#) para disparar o processo de validação. Esse método retorna um valor indicando se a validação ocorreu com sucesso ou não.

Para modelos utilizando [CActiveRecord](#), a validação pode ser disparada automaticamente quando o método [CActiveRecord::save\(\)](#) é executado.

Quando chamamos [CModel::validate\(\)](#), podemos especificar um parâmetro com o nome de um cenário. Assim, somente as regras desse cenário serão aplicadas na validação. Uma regra é aplicada a um cenário, se não existir a opção on nela, ou, caso exista, seu valor corresponda ao cenário especificado.

Por exemplo, executamos o código a seguir para executar a validação ao registrar um usuário:

```
$model->scenario='register';  
$model->validate();
```

Nota: A propriedade [scenario](#) está disponível a partir da versão 1.0.4. O método de validação irá utilizar essa propriedade para determinar quais regras irá utilizar. Nas versões 1.0.2 e 1.0.3, devemos informar o cenário da seguinte maneira:

```
$model->validate('register');
```

Devemos declarar as regras de validação na classe do modelo de formulário da seguinte maneira:

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

Como resultado, a primeira regra será aplicada para todos os cenários, enquanto as outras duas serão aplicadas apenas no cenário register.

Nota: validação baseada em cenários está disponível desde a versão 1.0.1.

Recuperando Erros de Validação

Podemos usar o método [CModel::hasErrors\(\)](#) para verificar se existe algum erro de validação e, caso existir, podemos utilizar o método [CModel::getErrors\(\)](#) para obter as mensagens de erro. Ambos os métodos podem ser utilizados para verificar erros em todos os atributos de uma única vez, ou para cada atributo individualmente.

Rótulos de Atributos

Quando desenvolvemos um formulário, normalmente precisamos exibir um rótulo para cada campo. Esse rótulo indica ao usuário que tipo de informação espera-se que ele informe naquele campo. Embora podemos escrever esses rótulos diretamente na visão, seria mais flexível e conveniente poder especificá-los diretamente no modelo correspondente.

Por padrão, a classe [CModel](#) irá retornar o nome do atributo como seu rótulo. Essa característica pode ser alterada sobrescrevendo o método [attributeLabels\(\)](#). Como veremos nas subseções a seguir, especificando rótulos nos modelos nos permite criar formulários poderosos de uma maneira mais rápida.

Criando uma Ação

Uma vez que temos um modelo pronto, podemos começar a escrever a lógica necessária para manipula-lo. Devemos colocar essa lógica dentro de uma ação no controle. Para o exemplo do formulário de login, o código a seguir é necessário:

```

public function actionLogin()
{
    $model=new LoginForm;
    if(isset($_POST['LoginForm']))
    {
        // coleta a informação inserida pelo usuário
        $model->attributes=$_POST['LoginForm'];
        // valida a entrada do usuário e redireciona
        // para a página anterior, caso valide
        if($model->validate())
            $this->redirect(Yii::app()->user->returnUrl);
    }
    // exibe o formulário de login
    $this->render('login',array('model'=>$model));
}

```

No código acima, primeiro criamos uma instância de um LoginForm. Se a requisição for do tipo POST (indicando que um formulário de login foi enviado), nós preenchemos o \$model com os dados enviados via \$_POST['LoginForm']. Em seguida, validamos os dados e, em caso de sucesso, redirecionamos o navegador para a página que requisitou a autenticação. Se a validação falhar, ou se for o primeiro acesso a essa ação, renderizamos o conteúdo da visão 'login', que será descrita na próxima subseção.

Dica: Na ação login, utilizamos a propriedade `Yii::app()->user->returnUrl` para pegar a URL da página que necessitou a autenticação. O componente `Yii::app()->user` é do tipo [CWebUser](#) (ou de uma classe derivada dele) que representa a sessão com as informações do usuário (por exemplo, nome de usuário, status). Para mais detalhes, veja [Autenticação e Autorização](#).

Vamos dar uma atenção especial para o seguinte trecho de código que aparece na ação login:

```

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

```

Como descrevemos em [Atribuição Segura de Atributos](#), essa linha de código preenche um modelo com as informações enviadas pelo usuário. A propriedade `attributes` é definida pela classe [CModel](#) que espera um vetor de pares nome-valor, e atribui cada valor ao atributo correspondente no modelo. Sendo assim, se \$_POST['LoginForm'], contém um vetor desse tipo, o código acima seria o equivalente ao código mais longo abaixo (assumindo que todos os atributos necessários estão presentes no vetor):

```

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

```


Nota: Para fazer com que a variável `$_POST['LoginForm']` nos retorne um vetor em vez de uma string, utilizamos uma convenção ao nomear os campos do formulário na visão. Para um campo correspondente ao atributo `a` de um modelo da classe `C`, seu nome será `C[a]`. Por exemplo, utilizamos `LoginForm[username]` para nomear o campo correspondente ao atributo `username` do modelo `LoginForm`.

O trabalho restante agora é criar a visão login que deve conter um formulário HTML com os campos necessários.

Criando um Formulário

Escrever a visão login é algo bem simples. Devemos começar com uma tag `form`, cujo atributo `action` deve ser a URL da ação login, descrita anteriormente. Em seguida inserimos os rótulos e os campos para os atributos declarados na classe `LoginForm`. Por fim, inserimos um botão de envio (`submit`) que pode ser utilizado pelos usuários para enviar o formulário. Tudo isso pode ser feito puramente com HTML.

O Yii fornece algumas classes auxiliares para facilitar a composição da visão. Por exemplo, para criar um caixa de texto, podemos utilizar o método [CHtml::textField\(\)](#); para criar uma lista do tipo drop-down, utilizamos [CHtml::dropDownList\(\)](#).

Informação: Você deve estar se perguntando qual a vantagem de se utilizar uma classe auxiliar, se elas utilizam a mesma quantidade de código do que o equivalente em HTML. A resposta é que as classes auxiliares geram mais do que somente código HTML. Por exemplo, o código a seguir gera uma caixa de texto que dispara o envio do formulário caso seu valor seja alterado pelo usuário:

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

Se não fosse assim, seria necessário um monte de código em JavaScript espalhado.

No exemplo a seguir, utilizamos a classe [CHtml](#) para criar o formulário de login. Assumimos que a variável `$model` representa uma instância de `LoginForm`.

```

<div class="form">
<?php echo CHtml::beginForm(); ?>
    <?php echo CHtml::errorSummary($model); ?>
    <div class="row">
        <?php echo CHtml::activeLabel($model, 'username'); ?>
        <?php echo CHtml::activeTextField($model, 'username') ?>
    </div>
    <div class="row">
        <?php echo CHtml::activeLabel($model, 'password'); ?>
        <?php echo CHtml::activePasswordField($model, 'password') ?>
    </div>
    <div class="row rememberMe">
        <?php echo CHtml::activeCheckBox($model, 'rememberMe'); ?>
        <?php echo CHtml::activeLabel($model, 'rememberMe'); ?>
    </div>
    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>
<?php echo CHtml::endForm(); ?>
</div><!-- form -->

```

Esse código gera um formulário mais dinâmico. Por exemplo, o método `CHtml::activeLabel()` gera um rótulo associado ao atributo do modelo especificado. Se ocorrer um erro com a validação desse atributo, a classe CSS do rótulo será alterada para `error`, o que mudará a aparência do rótulo. Da mesma forma, o método `CHtml::activeTextField()` gera uma caixa de texto para o atributo especificado e, também, altera sua classe CSS na ocorrência de erros.

Se utilizarmos o arquivo `css form.css`, fornecido pelo script do `yiic`, o formulário gerado terá a seguinte aparência:



A Página de Login

Please fix the following input errors:
 • Username cannot be blank.
 • Password cannot be blank.

Username

 Password

☐ Remember me next time

A Página de Login com Erros

A partir da versão 1.1.1, existe um novo widget chamado [CActiveForm](#), que pode ser utilizado para facilitar a criação de formulários. Esse widget é capaz de realizar validações de forma consistente e transparente, tanto do lado do cliente, quanto do lado do servidor. Utilizando o [CActiveForm](#), o código do último exemplo pode ser reescrito da seguinte maneira:

```
<div class="form">
<?php $form=$this->beginWidget('CActiveForm'); ?>
    <?php echo $form->errorSummary($model); ?>
    <div class="row">
        <?php echo $form->label($model, 'username'); ?>
        <?php echo $form->textField($model, 'username') ?>
    </div>
    <div class="row">
        <?php echo $form->label($model, 'password'); ?>
        <?php echo $form->passwordField($model, 'password') ?>
    </div>
    <div class="row rememberMe">
        <?php echo $form->checkbox($model, 'rememberMe'); ?>
        <?php echo $form->label($model, 'rememberMe'); ?>
    </div>
    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>
<?php $this->endWidget(); ?>
</div><!-- form -->
```

Coletando Entradas Tabulares

As vezes queremos coletar entradas de usuário em modo batch (em lote, vários ao mesmo tempo). Isso é, o usuário entra com informações para diversas instâncias de

modelos e os envia todos de uma só vez. Chamamos isso de Entrada Tabular, porque seus campos normalmente são apresentados em uma tabela HTML.

Para trabalhar com entradas tabulares, devemos primeiro criar e preencher um vetor de instâncias de modelos, dependendo se estamos inserindo ou atualizando os dados.

Podemos então recuperar as entradas do usuário a partir da variável `$_POST` e atribuí-las para cada modelo. Dessa forma, existe uma pequena diferença de quando utilizamos um único modelo para entrada; devemos recuperar os dados utilizando `$_POST['ClasseDoModelo'][$i]` em vez de `$_POST['ClasseDoModelo']`.

```
public function actionBatchUpdate()
{
    // recupera os itens para atualização em lote
    // assumindo que cada item é instância de um Item
    $items=$this->getItemsToUpdate();
    if(isset($_POST['Item']))
    {
        $valid=true;
        foreach($items as $i=>$item)
        {
            if(isset($_POST['Item'][$i]))
                $item->attributes=$_POST['Item'][$i];
            $valid=$valid && $item->validate();
        }
        if($valid) // todos os itens são validos
            // ...faça algo aqui
    }
    // exibe a visão para coletar as entradas tabulares
    $this->render('batchUpdate',array('items'=>$items));
}
```

Com a ação pronta, precisamos criar a visão `batchUpdate` para exibir os campos em uma tabela HTML:

```

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

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

```

Note no código acima que utilizamos "[*\$i*]name" em vez de "name" no segundo parâmetro ao chamar o método [CHtml::activeTextField](#).

Se ocorrer algum erro de validação, os campos correspondentes serão identificados automaticamente, da mesma forma como ocorre quando utilizamos um único modelo.

Trabalhando com Banco de Dados

Visão Geral

O framework Yii fornece um poderoso suporte para programação com Banco de Dados. Construído em cima da extensão PDO (PHP Data Objects). Os objetos de acesso a dados do Yii (DAO), permitem o acesso a diferentes sistemas de gerenciamento de Banco de Dados (SGBD) em uma única relação uniforme. As aplicações desenvolvidas usando Yii DAO podem facilmente ser alteradas para usar um SGBD diferente sem a necessidade de modificar o código de acesso aos dados. A Active Record (AR) do Yii, é implementada como uma abordagem amplamente adotada, a ORM (Object-Relational Mapping), simplificando ainda mais a programação com Banco de Dados. Representando uma tabela em termos de uma classe. Yii AR elimina a tarefa repetitiva de escrever instruções SQL que lidam principalmente com operações CRUD (criar, ler, atualizar e excluir).

Embora Yii DAO e AR possam tratar quase todos os objetos de dados relacionais, você ainda pode usar suas próprias bibliotecas de Banco de Dados na sua aplicação Yii. De fato, o framework Yii é cuidadosamente projetado para ser utilizado em conjunto com bibliotecas de terceiros.

Data Access Objects (DAO)

Data Access Objects (DAO, Objetos de Acesso a Dados), fornecem uma API genérica para acessar dados em diferentes tipos de bancos de dados. Como resultado, pode-se alterar o sistema de banco de dados sem haver a necessidade de alterar o código que utiliza DAO para fazer o acesso.

O DAO do Yii é feito utilizando a extensão PHP Data Objects (PDO), que fornece um acesso de dados unificado para muitos SGBD populares, tais como MySQL e PostgreSQL. Por esse motivo, para utilizar o DAO no Yii, a extensão PDO deve estar instalada, bem como o driver PDO específico para o banco de dados utilizado (por exemplo, PDO_MYSQL).

O DAO no Yii, consiste, basicamente, das quatro classes seguinte:

1. CDbConnection: representa uma conexão com o banco de dados.
2. CDbCommand: representa uma instrução SQL a ser executada no banco de dados.
3. CDbDataReader: representa um conjunto de registros retornados, navegável apenas para frente.
4. CDbTransaction: representa uma transação (transaction) no banco de dados.

A seguir, apresentaremos a utilização do DAO no Yii em diferente cenários.

Estabelecendo uma Conexão com o Banco de Dados

Para estabelecer uma conexão com o banco de dados, criamos uma instância de CDbConnection e a ativamos. É necessário o nome da fonte de dados (data source name, DSN) para conectar-se ao banco. Também podem ser necessários o nome de

usuário e senha para o acesso. Uma exceção será disparada caso ocorra um erro ao estabelecer a conexão (por exemplo, um DSN incorreto, ou usuário/senha inválidos).

```
$connection=new CDbConnection($dsn,$username,$password);  
// estabelece a conexão. Você pode utilizar um try... catch tratar exceções  
$connection->active=true;  
.....  
$connection->active=false; // fecha a conexão
```

O formato do DSN depende do driver PDO em uso. Geralmente, o DSN é formado pelo nome do driver PDO, seguido por ":", seguido pela sintaxe de conexão específica do driver. Veja [PDO documentation](#) para mais informações. Abaixo temos uma lista dos formatos de DSN mais utilizados:

- SQLite: sqlite:/path/to/dbfile
- MySQL: mysql:host=localhost;dbname=testdb
- PostgreSQL: pgsql:host=localhost;port=5432;dbname=testdb
- SQL Server: mssql:host=localhost;dbname=testdb
- Oracle: oci:dbname=//localhost:1521/testdb

Como a classe [CDbConnection](#) estende a classe [CApplicationComponent](#), podemos utiliza-la como um [componente da aplicação](#). Para isso, configure um componente chamado db (ou qualquer outro nome) na [configuração da aplicação](#), como no código a seguir:

```
array(  
    .....  
    'components'=>array(  
        .....  
        'db'=>array(  
            'class'=>'CDbConnection',  
            'connectionString'=>'mysql:host=localhost;dbname=testdb',  
            'username'=>'root',  
            'password'=>'password',  
            'emulatePrepare'=>true,  
            // necessário em algumas instalações do MySQL  
        ),  
    ),  
)
```

Podemos então acessar a conexão com o banco via `Yii::app()->db`, que é ativada automaticamente, a não ser que se configure a propriedade [CDbConnection::autoConnect](#) para false. Dessa maneira, uma única conexão com o banco de dados pode ser compartilhada entre diversas partes de seu código.

Executando Instruções SQL

Uma vez que a conexão com o banco de dados tenha sido estabelecida, comandos SQL podem ser executados utilizando-se a classe [CDbCommand](#). Você pode criar uma instância de [CDbCommand](#) utilizando o método [CDbConnection::createCommand\(\)](#), com a instrução SQL desejada:

```
$connection=Yii::app()->db;
// Aqui, estamos assumind que você configurou
// um conexão com o banco de dados
// Caso contrario, você deverá cria-la explicitamente:
// $connection=new CDbConnection($dsn,$username,$password);
$command=$connection->createCommand($sql);
// se necessário, a instrução SQL pode ser alterada da seguinte maneira:
// $command->text=$newSQL;
```

Uma instrução SQL pode ser executada via [CDbCommand](#) de duas maneiras:

- [execute\(\)](#): executa uma instrução SQL que não retorna resultados, tal como INSERT, UPDATE e DELETE. Em caso de sucesso, retorna a quantidade de registros afetados pela consulta.
- [query\(\)](#): executa uma instrução que retorna registros, tal como SELECT. Em caso de sucesso, retorna uma instância de [CDbDataReader](#), que pode ser utilizada para acessar os registros encontrados. Por conveniência, um conjunto de métodos queryXXX() também está implementado, e retornam diretamente o resultado da consulta.

Uma exceção será disparada caso ocorram erros durante a execução de instruções SQL.

```
$rowCount=$command->execute();
// executa uma consulta que não retorna resultados
$dataReader=$command->query();
// executa uma consulta SQL
$rows=$command->queryAll();
// consulta e retorna todos os resultados encontrados
$row=$command->queryRow();
// consulta e retorna apenas o primeiro registro do resultado
$column=$command->queryColumn();
// consulta e retorna a primeira coluna do resultado
$value=$command->queryScalar();
// consulta e retorna o primeiro campo do primeiro registro
```

Obtendo Resultados de Consultas

Depois que o método [CdbCommand::query\(\)](#) gerar uma instância de [CdbDataReader](#), você pode recuperar os registros do resultado através do método [CdbDataReader::read\(\)](#), repetidamente. Você também pode utilizar um [CdbDataReader](#) dentro de um loop foreach, para recuperar os registros, um a um:

```
$dataReader=$command->query();  
// executando read() repetidamente, até que ele retorne false  
while(($row=$dataReader->read())!==false) { ... }  
// utilizando foreach para "navegar" por cada registro encontrado  
foreach($dataReader as $row) { ... }  
// recuperando todos os registros de uma vez, em um vetor  
$rows=$dataReader->readAll();
```

Nota: Diferente do método [query\(\)](#), todos os métodos [queryXXX\(\)](#), retornam os dados diretamente. Por exemplo, o método [queryRow\(\)](#) retorna um vetor representando o primeiro registro do resultado da consulta.

Utilizando Transações (Transactions)

Quando uma aplicação executa algumas consultas, seja lendo ou gravando informações no banco de dados, é importante garantir que todas as consultas tenham sido executadas. Nesse caso, uma transação, representada por uma instância de [CdbTransaction](#), pode ser iniciada:

- Inicie a transação.
- Execute as consultas, uma a uma. Todas as atualizações no banco de dados não são visíveis aos outros.
- Encerre a transação. Nesse momento, as atualizações tornam-se visíveis, caso a transação tenha encerrado com sucesso.
- Se uma das consultas falhar, toda a transação é desfeita.

O fluxo acima pode ser implementado como no código a seguir:

```

$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //.... outras execuções de comandos SQL
    $transaction->commit();
}
catch(Exception $e) // uma exceção é disparada caso uma das consultas falhe
{
    $transaction->rollBack();
}

```

Vinculando (Binding) Parâmetros

Para evitar ataques de [SQL injection](#) e aumentar a performance ao executar instruções SQL repetidamente, você pode "preparar" um comando SQL com marcadores opcionais que serão substituídos pelos valores reais, durante o processo de vinculação de parâmetros.

Os marcadores de parâmetros podem ser nomeados (representados por tokens únicos) ou anônimos (representados por interrogações). Execute o método [CDbCommand::bindParam\(\)](#) ou [CDbCommand::bindValue\(\)](#) para substituir esses marcadores pelos parâmetros reais. Eles não precisam estar entre aspas: o próprio driver do banco de dados faz isso para você. A vinculação de parâmetros deve ser realizada antes da instrução SQL ser executada.

```

// uma consulta com dois marcadores ":username" e ":email"
$sql="INSERT INTO tbl_users(username, email) VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// substitui o marcador ":username" com o valor atual de $username
$command->bindParam(":username",$username,PDO::PARAM_STR);
// substitui o marcador ":email" com o valor atual de $email
$command->bindParam(":email",$email,PDO::PARAM_STR);
$command->execute();
// insere um novo registro com um novo conjunto de parâmetros
$command->bindParam(":username",$username2,PDO::PARAM_STR);
$command->bindParam(":email",$email2,PDO::PARAM_STR);
$command->execute();

```

Os métodos [bindParam\(\)](#) e [bindValue\(\)](#) são similares. A única diferença entre eles é que o primeiro vincula um parâmetro utilizando uma referência para a variável enquanto o outro utilizar um valor. Para parâmetros que representem uma grande quantidade de dados em memória, o primeiro é mais recomendado, devido a uma melhor performance.

Para mais detalhes sobre a vinculação de parâmetros, veja a [documentação do PHP](#).

Vinculando Colunas

Ao recuperar os resultados de uma consulta, você também pode vincular colunas à variáveis, de forma que elas sejam automaticamente preenchidas cada vez que um novo registro é recuperado:

```
$sql="SELECT username, email FROM tbl_users";
$dataReader=$connection->createCommand($sql)->query();
// vincula a 1ª coluna (username) à variável $username
$dataReader->bindColumn(1,$username);
// vincula a 2ª coluna (email) à variável $email
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // $username e $email contém o nome de
    // usuário e a senha do registro atual
}
```

Utilizando Prefixos de Tabelas

A partir da versão 1.1.0, o Yii framework possui suporte integrado para a utilização de prefixos em nomes de tabela. Um prefixo é uma string que será anexada ao início dos nomes das tabelas do banco de dados conectado. Normalmente, eles são utilizados em ambientes de hospedagem compartilhada, onde múltiplas aplicações utilizam um único banco de dados e é necessário diferenciar as tabelas de cada aplicação. Por exemplo, uma aplicação pode utilizar o prefixo `tbl_`, enquanto outra utiliza `yii_`.

Para utiliza-los, você deverá configurar a propriedade [CDbConnection::tablePrefix](#) com o prefixo desejado. Feito isso, em suas consultas SQL você deverá utilizar `{{NomeDaTabela}}`, onde `NomeDaTabela` é o nome da tabela sem o prefixo. Por exemplo, se um banco de dados contém uma tabela chamada `tbl_user` e `tbl_` é o prefixo configurado, então você pode utilizar o seguinte código para realizar consultas nessa tabela:

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

Gerador de Consultas

The Yii Query Builder provides an object-orient way of writing SQL statements. It allows developers to use class methods and properties to specify individual parts of a SQL statement. It then assembles different parts into a valid SQL statement that can be further executed by calling the DAO methods as described in [Data Access Objects](#). The following shows a typical usage of the Query Builder to build a SELECT SQL statement:

```
$user = Yii::app()->db->createCommand()  
    ->select('id, username, profile')  
    ->from('tbl_user u')  
    ->join('tbl_profile p', 'u.id=p.user_id')  
    ->where('id=:id', array(':id'=>$id))  
    ->queryRow();
```

The Query Builder is best used when you need to assemble a SQL statement procedurally, or based on some conditional logic in your application. The main benefits of using the Query Builder include:

- It allows building complex SQL statements programmatically.
- It automatically quotes table names and column names to prevent conflict with SQL reserved words and special characters.
- It also quotes parameter values and uses parameter binding when possible, which helps reduce risk of SQL injection attacks.
- It offers certain degree of DB abstraction, which simplifies migration to different DB platforms.
- It is not mandatory to use the Query Builder. In fact, if your queries are simple, it is easier and faster to directly write SQL statements.

Preparing Query Builder

The Yii Query Builder is provided in terms of [CDbCommand](#), the main DB query class described in [Data Access Objects](#).

To start using the Query Builder, we create a new instance of [CDbCommand](#) as follows,

```
$command = Yii::app()->db->createCommand();
```

That is, we use `Yii::app()->db` to get the DB connection, and then call [CDbConnection::createCommand\(\)](#) to create the needed command instance.

Note that instead of passing a whole SQL statement to the `createCommand()` call as we do in [Data Access Objects](#), we leave it empty. This is because we will build individual parts of the SQL statement using the Query Builder methods explained in the following.

Building Data Retrieval Queries

Data retrieval queries refer to SELECT SQL statements. The query builder provides a set of methods to build individual parts of a SELECT statement. Because all these methods return the [CDbCommand](#) instance, we can call them using method chaining, as shown in the example at the beginning of this section.

- [select](#): specifies the SELECT part of the query
- [selectDistinct](#): specifies the SELECT part of the query and turns on the DISTINCT flag
- [from](#): specifies the FROM part of the query

- [where](#): specifies the WHERE part of the query
- [join](#): appends an inner join query fragment
- [leftJoin](#): appends a left outer join query fragment
- [rightJoin](#): appends a right outer join query fragment
- [crossJoin](#): appends a cross join query fragment
- [naturalJoin](#): appends a natural join query fragment
- [group](#): specifies the GROUP BY part of the query
- [having](#): specifies the HAVING part of the query
- [order](#): specifies the ORDER BY part of the query
- [limit](#): specifies the LIMIT part of the query
- [offset](#): specifies the OFFSET part of the query
- [union](#): appends a UNION query fragment

In the following, we explain how to use these query builder methods. For simplicity, we assume the underlying database is MySQL. Note that if you are using other DBMS, the table/column/value quoting shown in the examples may be different.

select

```
function select($columns='*')
```

The [select](#) method specifies the SELECT part of a query. The \$columns parameter specifies the columns to be selected, which can be either a string representing comma-separated columns, or an array of column names. Column names can contain table prefixes and/or column aliases. The method will automatically quote the column names unless a column contains some parenthesis (which means the column is given as a DB expression).

Below are some examples:

```
// SELECT *
select()

// SELECT `id`, `username`
select('id, username')

// SELECT `tbl_user`.`id`, `username` AS `name`
select('tbl_user.id, username as name')

// SELECT `id`, `username`
select(array('id', 'username'))

// SELECT `id`, count(*) as num
select(array('id', 'count(*) as num'))
```

selectDistinct

```
function selectDistinct($columns)
```

The [selectDistinct](#) method is similar as [select](#) except that it turns on the DISTINCT flag. For example, `selectDistinct('id, username')` will generate the following SQL:

```
SELECT DISTINCT `id`, `username`
```

from

```
function from($tables)
```

The [from](#) method specifies the FROM part of a query. The \$tables parameter specifies which tables to be selected from. This can be either a string representing comma-separated table names, or an array of table names. Table names can contain schema prefixes (e.g. public.tbl_user) and/or table aliases (e.g. tbl_user u). The method will automatically quote the table names unless it contains some parenthesis (which means the table is given as a sub-query or DB expression).

Below are some examples:

```
// FROM `tbl_user`  
from('tbl_user')  
// FROM `tbl_user` `u`, `public`.`tbl_profile` `p`  
from('tbl_user u, public.tbl_profile p')  
// FROM `tbl_user`, `tbl_profile`  
from(array('tbl_user', 'tbl_profile'))  
// FROM `tbl_user`, (select * from tbl_profile) p  
from(array('tbl_user', '(select * from tbl_profile) p'))
```

where

```
function where($conditions, $params=array())
```

The [where](#) method specifies the WHERE part of a query. The \$conditions parameter specifies query conditions while \$params specifies the parameters to be bound to the whole query. The \$conditions parameter can be either a string (e.g. id=1) or an array of the format:

```
array(operator, operand1, operand2, ...)
```

where operator can be any of the following:

- and: the operands should be concatenated together using AND. For example, array('and', 'id=1', 'id=2') will generate id=1 AND id=2. If an operand is an array, it will be converted into a string using the same rules described here. For example, array('and', 'type=1', array('or', 'id=1', 'id=2')) will generate type=1 AND (id=1 OR id=2). The method will NOT do any quoting or escaping.
- or: similar as the and operator except that the operands are concatenated using OR.
- in: operand 1 should be a column or DB expression, and operand 2 be an array representing the range of the values that the column or DB expression should be in.

For example, `array('in', 'id', array(1,2,3))` will generate `id IN (1,2,3)`. The method will properly quote the column name and escape values in the range.

- not in: similar as the in operator except that IN is replaced with NOT IN in the generated condition.
- like: operand 1 should be a column or DB expression, and operand 2 be a string or an array representing the range of the values that the column or DB expression should be like. For example, `array('like', 'name', 'tester')` will generate `name LIKE '%tester%'`. When the value range is given as an array, multiple LIKE predicates will be generated and concatenated using AND. For example, `array('like', 'name', array('test', 'sample'))` will generate `name LIKE '%test%' AND name LIKE '%sample%'`. The method will properly quote the column name and escape values in the range.
- not like: similar as the like operator except that LIKE is replaced with NOT LIKE in the generated condition.
- or like: similar as the like operator except that OR is used to concatenated several LIKE predicates.
- or not like: similar as the not like operator except that OR is used to concatenated several NOT LIKE predicates.

Below are some examples of using where:

```
// WHERE id=1 or id=2
where('id=1 or id=2')
// WHERE id=:id1 or id=:id2
where('id=:id1 or id=:id2', array(':id1'=>1, ':id2'=>2))
// WHERE id=1 OR id=2
where(array('or', 'id=1', 'id=2'))
// WHERE id=1 AND (type=2 OR type=3)
where(array('and', 'id=1', array('or', 'type=2', 'type=3'))))
// WHERE `id` IN (1, 2)
where(array('in', 'id', array(1, 2)))
// WHERE `id` NOT IN (1, 2)
where(array('not in', 'id', array(1,2)))
// WHERE `name` LIKE '%Qiang%'
where(array('like', 'name', '%Qiang%'))
// WHERE `name` LIKE '%Qiang' AND `name` LIKE '%Xue'
where(array('like', 'name', array('%Qiang', '%Xue'))))
// WHERE `name` LIKE '%Qiang' OR `name` LIKE '%Xue'
where(array('or like', 'name', array('%Qiang', '%Xue'))))
// WHERE `name` NOT LIKE '%Qiang%'
where(array('not like', 'name', '%Qiang%'))
// WHERE `name` NOT LIKE '%Qiang%' OR `name` NOT LIKE '%Xue%'
where(array('or not like', 'name', array('%Qiang', '%Xue'))))
```

Please note that when the operator contains like, we have to explicitly specify the wildcard characters (such as % and _) in the patterns. If the patterns are from user input, we should

also use the following code to escape the special characters to prevent them from being treated as wildcards:

```
$keyword=$_GET['q'];  
// escape % and _ characters  
$keyword=strtr($keyword, array('%'=>'\\%', '_'=>'\\_'));  
$command->where(array('like', 'title', '%'.$keyword.'%'));
```

order

```
function order($columns)
```

The [order](#) method specifies the ORDER BY part of a query. The \$columns parameter specifies the columns to be ordered by, which can be either a string representing comma-separated columns and order directions (ASC or DESC), or an array of columns and order directions. Column names can contain table prefixes. The method will automatically quote the column names unless a column contains some parenthesis (which means the column is given as a DB expression).

Below are some examples:

```
// ORDER BY `name`, `id` DESC  
order('name, id desc')  
// ORDER BY `tbl_profile`.`name`, `id` DESC  
order(array('tbl_profile.name', 'id desc'))
```

limit and offset

```
function limit($limit, $offset=null)  
function offset($offset)
```

The [limit](#) and [offset](#) methods specify the LIMIT and OFFSET part of a query. Note that some DBMS may not support LIMIT and OFFSET syntax. In this case, the Query Builder will rewrite the whole SQL statement to simulate the function of limit and offset.

Below are some examples:

```
// LIMIT 10  
limit(10)  
// LIMIT 10 OFFSET 20  
limit(10, 20)  
// OFFSET 20  
offset(20)
```


join and its variants

```
function join($table, $conditions, $params=array())
function leftJoin($table, $conditions, $params=array())
function rightJoin($table, $conditions, $params=array())
function crossJoin($table)
function naturalJoin($table)
```

The [join](#) method and its variants specify how to join with other tables using INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, CROSS JOIN, or NATURAL JOIN. The \$table parameter specifies which table to be joined with. The table name can contain schema prefix and/or alias. The method will quote the table name unless it contains a parenthesis meaning it is either a DB expression or sub-query. The \$conditions parameter specifies the join condition. Its syntax is the same as that in [where](#). And \$params specifies the parameters to be bound to the whole query.

Note that unlike other query builder methods, each call of a join method will be appended to the previous ones.

Below are some examples:

```
// JOIN `tbl_profile` ON user_id=id
join('tbl_profile', 'user_id=id')
// LEFT JOIN `pub`.`tbl_profile` `p` ON p.user_id=id AND type=:type
leftJoin('pub.tbl_profile p', 'p.user_id=id AND type=:type', array
('/:type'=>1))
```

group

```
function group($columns)
```

The [group](#) method specifies the GROUP BY part of a query. The \$columns parameter specifies the columns to be grouped by, which can be either a string representing comma-separated columns, or an array of columns. Column names can contain table prefixes. The method will automatically quote the column names unless a column contains some parenthesis (which means the column is given as a DB expression).

Below are some examples:

```
// GROUP BY `name`, `id`
group('name, id')
// GROUP BY `tbl_profile`.`name`, `id`
group(array('tbl_profile.name', 'id'))
```

having

```
function having($conditions, $params=array())
```

The [having](#) method specifies the HAVING part of a query. Its usage is the same as [where](#).

Below are some examples:

```
// HAVING id=1 or id=2
having('id=1 or id=2')
// HAVING id=1 OR id=2
having(array('or', 'id=1', 'id=2'))
```

union

```
function union($sql)
```

The [union](#) method specifies the UNION part of a query. It appends \$sql to the existing SQL using UNION operator. Calling union() multiple times will append multiple SQLs to the existing SQL.

Below are some examples:

```
// UNION (select * from tbl_profile)
union('select * from tbl_profile')
```

Executing Queries

After calling the above query builder methods to build a query, we can call the DAO methods as described in [Data Access Objects](#) to execute the query. For example, we can call [CDbCommand::queryRow\(\)](#) to obtain a row of result, or [CDbCommand::queryAll\(\)](#) to get all rows at once. Example:

```
$users = Yii::app()->db->createCommand()
    ->select('*')
    ->from('tbl_user')
    ->queryAll();
```

Retrieving SQLs

Besides executing a query built by the Query Builder, we can also retrieve the corresponding SQL statement. This can be done by calling [CDbCommand::getText\(\)](#).

```
$sql = Yii::app()->db->createCommand()  
    ->select('*')  
    ->from('tbl_user')  
    ->text;
```

If there are any parameters to be bound to the query, they can be retrieved via the [CDbCommand::params](#) property.

Alternative Syntax for Building Queries

Sometimes, using method chaining to build a query may not be the optimal choice. The Yii

Query Builder allows a query to be built using simple object property assignments. In particular, for each query builder method, there is a corresponding property that has the same name. Assigning a value to the property is equivalent to calling the corresponding method. For example, the following two statements are equivalent, assuming `$command` represents a [CDbCommand](#) object:

```
$command->select(array('id', 'username'));  
$command->select = array('id', 'username');
```

Furthermore, the [CDbConnection::createCommand\(\)](#) method can take an array as the parameter. The name-value pairs in the array will be used to initialize the properties of the created [CDbCommand](#) instance. This means, we can use the following code to build a query:

```
$row = Yii::app()->db->createCommand(array(  
    'select' => array('id', 'username'),  
    'from' => 'tbl_user',  
    'where' => 'id=:id',  
    'params' => array(':id'=>1),  
))->queryRow();
```

Building Multiple Queries

A [CDbCommand](#) instance can be reused multiple times to build several queries. Before building a new query, however, the [CDbCommand::reset\(\)](#) method must be invoked to clean up the previous query. For example:

```
$command = Yii::app()->createCommand();  
$users = $command->select('*')->from('tbl_users')->queryAll();  
$command->reset(); // clean up the previous query  
$posts = $command->select('*')->from('tbl_posts')->queryAll();
```

Building Data Manipulation Queries

Data manipulation queries refer to SQL statements for inserting, updating and deleting data in a DB table. Corresponding to these queries, the query builder provides insert, update and delete methods, respectively. Unlike the SELECT query methods described above, each of these data manipulation query methods will build a complete SQL statement and execute it immediately.

- [insert](#): inserts a row into a table
- [update](#): updates the data in a table
- [delete](#): deletes the data from a table

Below we describe these data manipulation query methods.

insert

```
function insert($table, $columns)
```

The [insert](#) method builds and executes an INSERT SQL statement. The \$table parameter specifies which table to be inserted into, while \$columns is an array of name-value pairs specifying the column values to be inserted. The method will quote the table name properly and will use parameter-binding for the values to be inserted.

Below is an example:

```
// build and execute the following SQL:
// INSERT INTO `tbl_user` (`name`, `email`) VALUES (:name, :email)
$command->insert('tbl_user', array(
    'name'=>'Tester',
    'email'=>'tester@example.com',
));
```

update

```
function update($table, $columns, $conditions='', $params=array())
```

The [update](#) method builds and executes an UPDATE SQL statement. The \$table parameter specifies which table to be updated; \$columns is an array of name-value pairs specifying the column values to be updated; \$conditions and \$params are like in [where](#), which specify the WHERE clause in the UPDATE statement. The method will quote the table name properly and will use parameter-binding for the values to be updated.

Below is an example:

```
// build and execute the following SQL:
// UPDATE `tbl_user` SET `name`=:name WHERE id=:id
$command->update('tbl_user', array(
    'name'=>'Tester',
), 'id=:id', array('/:id'=>1));
```

delete

```
function delete($table, $conditions='', $params=array())
```

The [delete](#) method builds and executes a DELETE SQL statement. The \$table parameter specifies which table to be updated; \$conditions and \$params are like in [where](#), which specify the WHERE clause in the DELETE statement. The method will quote the table name properly.

Below is an example:

```
// build and execute the following SQL:
// DELETE FROM `tbl_user` WHERE id=:id
$command->delete('tbl_user', 'id=:id', array('/:id'=>1));
```

Building Schema Manipulation Queries

Besides normal data retrieval and manipulation queries, the query builder also offers a set of methods for building and executing SQL queries that can manipulate the schema of a database. In particular, it supports the following queries:

- [createTable](#): creates a table
- [renameTable](#): renames a table
- [dropTable](#): drops a table
- [truncateTable](#): truncates a table
- [addColumn](#): adds a table column
- [renameColumn](#): renames a table column
- [alterColumn](#): alters a table column
- [dropColumn](#): drops a table column
- [createIndex](#): creates an index
- [dropIndex](#): drops an index

Info: Although the actual SQL statements for manipulating database schema vary widely across different DBMS, the query builder attempts to provide a uniform interface for building these queries. This simplifies the task of migrating a database from one DBMS to another.

Abstract Data Types

The query builder introduces a set of abstract data types that can be used in defining table columns. Unlike the physical data types that are specific to particular DBMS and are quite

different in different DBMS, the abstract data types are independent of DBMS. When abstract data types are used in defining table columns, the query builder will convert them into the corresponding physical data types.

The following abstract data types are supported by the query builder.

- pk: a generic primary key type, will be converted into int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY for MySQL;
- string: string type, will be converted into varchar(255) for MySQL;
- text: text type (long string), will be converted into text for MySQL;
- integer: integer type, will be converted into int(11) for MySQL;
- float: floating number type, will be converted into float for MySQL;
- decimal: decimal number type, will be converted into decimal for MySQL;
- datetime: datetime type, will be converted into datetime for MySQL;
- timestamp: timestamp type, will be converted into timestamp for MySQL;
- time: time type, will be converted into time for MySQL;
- date: date type, will be converted into date for MySQL;
- binary: binary data type, will be converted into blob for MySQL;
- boolean: boolean type, will be converted into tinyint(1) for MySQL.

createTable

```
function createTable($table, $columns, $options=null)
```

The [createTable](#) method builds and executes a SQL statement for creating a table. The \$table parameter specifies the name of the table to be created. The \$columns parameter specifies the columns in the new table. They must be given as name-definition pairs (e.g. 'username'=>'string'). The \$options parameter specifies any extra SQL fragment that should be appended to the generated SQL. The query builder will quote the table name as well as the column names properly.

When specifying a column definition, one can use an abstract data type as described above. The query builder will convert the abstract data type into the corresponding physical data type, according to the currently used DBMS. For example, string will be converted into varchar(255) for MySQL.

A column definition can also contain non-abstract data type or specifications. They will be put in the generated SQL without any change. For example, point is not an abstract data type, and if used in a column definition, it will appear as is in the resulting SQL; and string NOT NULL will be converted into varchar(255) NOT NULL (i.e., only the abstract type string is converted).

Below is an example showing how to create a table:

```
// CREATE TABLE `tbl_user` (
//     `id` int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
//     `username` varchar(255) NOT NULL,
//     `location` point
// ) ENGINE=InnoDB
createTable('tbl_user', array(
    'id' => 'pk',
    'username' => 'string NOT NULL',
    'location' => 'point',
), 'ENGINE=InnoDB')
```

renameTable

```
function renameTable($table, $newName)
```

The [renameTable](#) method builds and executes a SQL statement for renaming a table. The \$table parameter specifies the name of the table to be renamed. The \$newName parameter specifies the new name of the table. The query builder will quote the table names properly.

Below is an example showing how to rename a table:

```
// RENAME TABLE `tbl_users` TO `tbl_user`
renameTable('tbl_users', 'tbl_user')
```

dropTable

```
function dropTable($table)
```

The [dropTable](#) method builds and executes a SQL statement for dropping a table. The \$table parameter specifies the name of the table to be dropped. The query builder will quote the table name properly.

Below is an example showing how to drop a table:

```
// DROP TABLE `tbl_user`
dropTable('tbl_user')
```

truncateTable

```
function truncateTable($table)
```

The [truncateTable](#) method builds and executes a SQL statement for truncating a table. The \$table parameter specifies the name of the table to be truncated. The query builder will quote the table name properly.

Below is an example showing how to truncate a table:

```
// TRUNCATE TABLE `tbl_user`  
truncateTable('tbl_user')
```

addColumn

```
function addColumn($table, $column, $type)
```

The [addColumn](#) method builds and executes a SQL statement for adding a new table column. The \$table parameter specifies the name of the table that the new column will be added to. The \$column parameter specifies the name of the new column. And \$type specifies the definition of the new column. Column definition can contain abstract data type, as described in the subsection of "createTable". The query builder will quote the table name as well as the column name properly.

Below is an example showing how to add a table column:

```
// ALTER TABLE `tbl_user` ADD `email` varchar(255) NOT NULL  
addColumn('tbl_user', 'email', 'string NOT NULL')
```

dropColumn

```
function dropColumn($table, $column)
```

The [dropColumn](#) method builds and executes a SQL statement for dropping a table column. The \$table parameter specifies the name of the table whose column is to be dropped. The \$column parameter specifies the name of the column to be dropped. The query builder will quote the table name as well as the column name properly.

Below is an example showing how to drop a table column:

```
// ALTER TABLE `tbl_user` DROP COLUMN `location`  
dropColumn('tbl_user', 'location')
```

renameColumn

```
function renameColumn($table, $name, $newName)
```

The [renameColumn](#) method builds and executes a SQL statement for renaming a table column. The \$table parameter specifies the name of the table whose column is to be renamed. The \$name parameter specifies the old column name. And \$newName specifies

the new column name. The query builder will quote the table name as well as the column names properly.

Below is an example showing how to rename a table column:

```
// ALTER TABLE `tbl_users` CHANGE `name` `username` varchar(255) NOT NULL  
renameColumn('tbl_user', 'name', 'username')
```

alterColumn

```
function alterColumn($table, $column, $type)
```

The [alterColumn](#) method builds and executes a SQL statement for altering a table column. The \$table parameter specifies the name of the table whose column is to be altered. The \$column parameter specifies the name of the column to be altered. And \$type specifies the new definition of the column. Column definition can contain abstract data type, as described in the subsection of "createTable". The query builder will quote the table name as well as the column name properly.

Below is an example showing how to alter a table column:

```
// ALTER TABLE `tbl_user` CHANGE `username` `username` varchar(255) NOT NULL  
alterColumn('tbl_user', 'username', 'string NOT NULL')
```

addForeignKey

```
function addForeignKey($name, $table, $columns,  
    $refTable, $refColumns, $delete=null, $update=null)
```

The [addForeignKey](#) method builds and executes a SQL statement for adding a foreign key constraint to a table. The \$name parameter specifies the name of the foreign key. The \$table and \$columns parameters specify the table name and column name that the foreign key is about. If there are multiple columns, they should be separated by comma characters. The \$refTable and \$refColumns parameters specify the table name and column name that the foreign key references. The \$delete and \$update parameters specify the ON DELETE and ON UPDATE options in the SQL statement, respectively. Most DBMS support these options: RESTRICT, CASCADE, NO ACTION, SET DEFAULT, SET NULL. The query builder will properly quote the table name, index name and column name(s).

Below is an example showing how to add a foreign key constraint,

```
// ALTER TABLE `tbl_profile` ADD CONSTRAINT `fk_profile_user_id`  
// FOREIGN KEY (`user_id`) REFERENCES `tbl_user` (`id`)  
// ON DELETE CASCADE ON UPDATE CASCADE  
addForeignKey('fk_profile_user_id', 'tbl_profile', 'user_id',  
             'tbl_user', 'id', 'CASCADE', 'CASCADE')
```

dropForeignKey

```
function dropForeignKey($name, $table)
```

The [dropForeignKey](#) method builds and executes a SQL statement for dropping a foreign key constraint. The \$name parameter specifies the name of the foreign key constraint to be dropped. The \$table parameter specifies the name of the table that the foreign key is on. The query builder will quote the table name as well as the constraint names properly.

Below is an example showing how to drop a foreign key constraint:

```
// ALTER TABLE `tbl_profile` DROP FOREIGN KEY `fk_profile_user_id`  
dropForeignKey('fk_profile_user_id', 'tbl_profile')
```

createIndex

```
function createIndex($name, $table, $column, $unique=false)
```

The [createIndex](#) method builds and executes a SQL statement for creating an index. The \$name parameter specifies the name of the index to be created. The \$table parameter specifies the name of the table that the index is on. The \$column parameter specifies the name of the column to be indexed. And the \$unique parameter specifies whether a unique index should be created. If the index consists of multiple columns, they must be separated by commas. The query builder will properly quote the table name, index name and column name(s).

Below is an example showing how to create an index:

```
// CREATE INDEX `idx_username` ON `tbl_user` (`username`)  
createIndex('idx_username', 'tbl_user')
```

dropIndex

```
function dropIndex($name, $table)
```

The [dropIndex](#) method builds and executes a SQL statement for dropping an index. The \$name parameter specifies the name of the index to be dropped. The \$table parameter specifies the name of the table that the index is on. The query builder will quote the table name as well as the index names properly.

Below is an example showing how to drop an index:

```
// DROP INDEX `idx_username` ON `tbl_user`  
dropIndex('idx_username', 'tbl_user')
```

Active Record

Apesar do DAO do Yii ser capaz de cuidar, praticamente, de qualquer tarefa relacionada a banco de dados, há uma grande chance de que, ainda, gastaríamos 90% do nosso tempo escrevendo instruções SQL para efetuar as operações de CRUD (create (criar), read (ler), update (atualizar) e delete (excluir)). Além disso, nosso código é mais difícil de manter quando temos instruções SQL misturadas com ele. Para resolver esses problemas, podemos utilizar Active Record (Registro Ativo).

Active Record (AR) é uma popular técnica de Mapeamento Objeto-Relacional (Object-Relational Mapping, ORM). Cada classe AR representa uma tabela (ou uma view) do banco de dados, cujos campos são representados por propriedades na classe AR. Uma instância de uma AR representa um único registro de uma tabela. As operações de CRUD são implementadas como métodos na classe AR. Como resultado, podemos acessar nossos dados de uma maneira orientada a objetos. Por exemplo, podemos fazer como no código a seguir para inserir um novo registro na tabela Post:

```
$post=new Post;  
$post->title='post de exemplo';  
$post->content='conteúdo do post';  
$post->save();
```

A seguir, descreveremos como configurar AR e como utiliza-lo para executar operações de CRUD. Na próxima seção, iremos mostrar como utilizar AR para trabalhar com relacionamentos. Para simplificar, utilizaremos a seguinte tabela para os exemplos desta seção. Note que, se você estiver utilizando um banco de dados MySQL, você deve substituir o AUTOINCREMENT por AUTO_INCREMENT na instrução abaixo:

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

Nota: A intenção do AR não é resolver todas tarefas relacionadas a banco de dados. Ele é melhor utilizado para modelar tabelas do banco para estruturas no PHP e executar consultas que não envolvem instruções SQL complexas. O DAO do Yii é o recomendado para esses cenários mais complexos.

Estabelecendo uma Conexão com o Banco de Dados

O AR precisa de uma conexão com o banco para executar suas operações. Por padrão, assume-se que o componente de aplicação db possui uma instância da classe [CDbConnection](#) que irá servir esta conexão. Abaixo temos um exemplo da configuração de uma aplicação:

```
return array(  
    'components'=>array(  
        'db'=>array(  
            'class'=>'system.db.CDbConnection',  
            'connectionString'=>'sqlite:path/to/dbfile',  
            // habilita o cache de schema para aumentar a performance  
            // 'schemaCachingDuration'=>3600,  
        ),  
    ),  
);
```

Dica: Como o Active Record depende de metadados sobre as tabelas para determinar informações sobre as colunas, gasta-se tempo lendo esses dados e os analisando. Se o schema do seu banco de dados não irá sofrer alterações, é interessante que você ative o caching de schema, configurando a propriedade `CDbConnection::schemaCachingDuration` para um valor maior de que 0.

O suporte para AR é limitado pelo Sistema de Gerenciamento de Banco de Dados. Atualmente, somente os seguintes SGBDs são suportados:

- [MySQL 4.1 ou maior](#)
- [PostgreSQL 7.3 ou maior](#)
- [SQLite 2 e 3](#)
- [Microsoft SQL Server 2000 ou maior](#)
- [Oracle](#)

Nota: O suporte ao Microsoft SQL Server existe desde a versão 1.0.4; já o suporte ao Oracle está disponível a partir da versão 1.0.5.

Se você deseja utilizar um componente de aplicação diferente de db, ou se quiser trabalhar com vários bancos de dados utilizando AR, você deve sobrescrever o método [CActiveRecord::getDbConnection\(\)](#). A classe [CActiveRecord](#) é a base para todas as classes Active Record.

Dica: Existem duas maneiras de trabalhar como AR utilizando vários bancos de dados. Se os schemas dos bancos são diferentes, você deve criar diferentes classes base AR, com diferentes implementações do método `getDbConnection()`. Caso contrário, alterar dinamicamente a variável estática `CActiveRecord::db` é uma ideia melhor.

Definindo Classes AR

Para acessar uma tabela do banco de dados, precisamos primeiro definir uma classe AR estendendo [CActiveRecord](#). Cada classe Active Record representa uma única tabela do banco, e uma instância dessa classe representa um registro dessa tabela. O exemplo abaixo mostra o código mínimo necessário para uma classe AR que representa a tabela Post:

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

Dica: Como as classes Ar geralmente são utilizadas em diversos lugares, podemos importar todo o diretório onde elas estão localizadas, em vez de fazer a importação uma a uma. Por exemplo, se todos os arquivos de nossas classes estão em `protected/models`, podemos configurar a aplicação da seguinte maneira:

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

Por padrão, o nome de uma classe AR é o mesmo que o da tabela do banco de dados. Sobrescreva o método [tableName\(\)](#) caso eles sejam diferentes. O método [model\(\)](#) deve ser declarado dessa maneira para todas as classes AR (a ser explicado em breve).

Os valores do registro de uma tabela podem ser acessados pelas propriedades da instância AR correspondente. Por exemplo, o código a seguir adiciona um valor ao campo `title`:

```
$post=new Post;
$post->title='um post de exemplo';
```

Embora nunca tenhamos declarado uma propriedade `title` na classe `Post`, ainda assim podemos acessá-la no exemplo acima. Isso acontece porque `title` é uma coluna da tabela `Post`, e a classe [CActiveRecord](#) a deixa acessível por meio de uma propriedade com a ajuda do método mágico `__get()`, do PHP. Ao tentar acessar uma coluna que não existe na tabela será disparada uma exceção.

Informação: Nesse guia, nomeamos as colunas utilizando o estilo camel case (por exemplo, `createTime`). Isso acontece porque acessamos essas colunas através de propriedades de objetos que também utilizam esse estilo para nomeá-las. Embora a utilização de camel case faça nosso código ter uma nomenclatura mais consistente, ele adiciona um problema relacionado aos bancos de dados que diferenciam letras maiúsculas de minúsculas. Por exemplo, o PostgreSQL, por padrão, não faz essa diferenciação nos nomes das colunas, e devemos colocar o nome da coluna entre aspas, em uma consulta, se seu nome conter letras maiúsculas e minúsculas. Por isso, é uma boa ideia nomear as colunas (e as tabelas também) somente com letras minúsculas (por exemplo, `create_time`) para evitar esse tipo de problema.

Criando um Registro

Para inserir um novo registro em uma tabela, criamos uma nova instância da classe `AR` correspondente, inserimos os valores nas propriedades relacionadas as colunas da tabela e, então, utilizamos o método [save\(\)](#) para concluir a inserção.

```
$post=new Post;
$post->title='post de exemplo';
$post->content='conteúdo do post de exemplo';
$post->createTime=time();
$post->save();
```

Se a chave primário da tabela é auto-numérica, após a inserção, a instância da classe `AR` irá conter o valor atualizado da chave primária. No exemplo acima, a propriedade `id` irá refletir o valor da chave primária no novo post inserido, mesmo que não a tenhamos alterado explicitamente.

Se alguma coluna é definida com um valor padrão estático (por exemplo, uma string ou um número) no schema da tabela, a propriedade correspondente na instância `AR` terá, automaticamente, este valor, assim que criada. Uma maneira de alterar esse valor padrão é declarar explicitamente a propriedade na classe `AR`:

```

class Post extends CActiveRecord
{
    public $title='por favor insira um título';
    .....
}

$post=new Post;
echo $post->title; // irá exibir: por favor insira um título

```

A partir da versão 1.0.2, pode-se atribuir a um atributo um valor do tipo [CDbExpression](#), antes que o registro seja salvo (tanto na inserção, quanto na atualização) no banco de dados. Por exemplo, para salvar um timestamp retornado pela função NOW() do MySQL, podemos utilizar o seguinte código:

```

$post=new Post;
$post->createTime=new CDbExpression('NOW()');
// $post->createTime='NOW()'; não irá funcionar porque
// 'NOW()' será tratado como uma string
$post->save();

```

Dica: Embora o Active Record torne possível que sejam realizadas operações no banco de dados sem a necessidade de escrever consultas em SQL, geralmente queremos saber quais consultas estão sendo executadas pelo AR. Para isso, ative o recurso de [registros de logs](#) do Yii. Por exemplo, podemos ativar o componente [CWebLogRoute](#) na configuração da aplicação, e, então poderemos ver as instruções SQL executadas exibidas no final de cada página. Desde a versão 1.0.5, podemos alterar o valor da propriedade [CDbConnection::enableParamLogging](#) para true, na configuração da aplicação, assim os valores dos parâmetros vinculados a instrução também serão registrados.

Lendo um Registro

Para ler dados de uma tabela, podemos utilizar um dos métodos find:

```

// encontra o primeiro registro que atenda a condição especificada
$post=Post::model()->find($condition,$params);
// encontra o registro com a chave primária especificada
$post=Post::model()->findByPk($postId,$condition,$params);
// encontra o registro com os atributos tendo os valores especificados
$post=Post::model()->findByAttributes($attributes,$condition,$params);
// encontra o primeiro registro, utilizando o comando SQL especificado
$post=Post::model()->findBySql($sql,$params);

```

No exemplo acima, utilizamos o método `find` em conjunto com `Post::model()`. Lembre-se que o método estático `model()` é obrigatório em todas as classes AR. Esse método retorna uma instância AR que é utilizada para acessar métodos a nível de classe (algo parecido com métodos estáticos de classe) em um contexto de objeto.

Se o método `find` encontra um registro que satisfaça as condições da consulta, ele irá retornar uma instância cujas propriedades irão conter os valores do registro específico. Podemos então ler os valores carregados normalmente como fazemos com as propriedades de um objeto. Por exemplo, `echo $post->title;`

O método `find` irá retornar `null` se nenhum registro for encontrado.

Ao executar o método `find`, utilizamos os parâmetros `$condition` e `$params` para especificar as condições desejadas. Nesse caso, `$condition` pode ser uma string representando uma cláusula `WHERE`, do SQL, e `$params` é um vetor com parâmetros cujos valores devem ser vinculados a marcadores na `$condition`. Por exemplo:

```
// encontra o registro com postID=10
$post=Post::model()->find('postID=:postID', array(':postID'=>10));
```

Nota: No exemplo acima, precisamos escapar a referência para a coluna `postID`, em certos SGBDs. Por exemplo, se estivermos utilizando o PostgreSQL, deveríamos ter escrito a condição como `"postID"=:postID`, porque este banco de dados, por padrão, não diferencia letras maiúsculas e minúsculas nos nomes de colunas.

Podemos também utilizar o parâmetro `$condition` para especificar condições de pesquisa mais complexas. Em vez de uma string, `$condition` pode ser uma instância de [CdbCriteria](#), o que permite especificar outras condições além do `WHERE`. Por exemplo:

```
$criteria=new CdbCriteria;
$criteria->select='title'; // seleciona apenas a coluna title
$criteria->condition='postID=:postID';
$criteria->params=array(':postID'=>10);
$post=Post::model()->find($criteria); // $params não é necessario
```

Note que, ao utilizar [CdbCriteria](#) como condição para a pesquisa, o parâmetro `$params` não é mais necessário, uma vez que ele pode ser especificado diretamente na instância de [CdbCriteria](#), como no exemplo acima.

Uma maneira alternativa de utilizar [CdbCriteria](#) é passar um vetor para o método `find`. As chaves e valores do vetor correspondem as propriedades e valores da condição, respectivamente. O exemplo acima pode ser reescrito da seguinte maneira:


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

Informação: Quando a condição de uma consulta deve casar com colunas com um determinado valor, utilizamos o método `findByAttributes()`. Fazemos com que o parâmetro `$attributes` seja um vetor, onde os atributos são indexados pelos nomes das colunas. Em alguns frameworks, essa tarefa é feita utilizando-se métodos como `findByNameAndTitle`. Apesar de parecer uma maneira mais atrativa, normalmente esses métodos causam confusão, conflitos e problemas em relação aos nomes de colunas com maiúsculas e minúsculas.

Quando uma condição encontra diversos resultados em uma consulta, podemos trazê-los todos de uma vez utilizando os seguintes métodos `findAll`, cada um com sua contraparte na forma de métodos `find`, como já descrito.

```
// encontra todos os registros que satisfaçam a condição informada
$post=Post::model()->findAll($condition,$params);
// encontra todos os registros com a chave primária informada
$post=Post::model()->findAllByPk($postIDs,$condition,$params);
// encontra todos os registros com campos com o valor informado
$post=Post::model()->findAllByAttributes($attributes,$condition,$params);
// encontra todos os registros utilizando a consulta SQL informada
$post=Post::model()->findAllBySql($sql,$params);
```

Se nenhum registro for encontrado, `findAll` irá retornar um vetor vazio, diferente dos métodos `find` que retornam `null` quando nada é encontrado.

Em conjunto com os métodos `find` e `findAll`, já descritos, os seguintes métodos também são fornecidos:

```
// pega o número de registros que satisfaz a condição informada
$n=Post::model()->count($condition,$params);
// pega o número de registros que satisfaz a instrução SQL informada
$n=Post::model()->countBySql($sql,$params);
// verifica se há pelo menos um registro que satisfaz a condição informada
$exists=Post::model()->exists($condition,$params);
```

Atualizando Registros

Depois que uma instância AR tenha sido preenchida com os valores dos campos da tabela, podemos atualizá-los e salvá-los de volta para o banco de dados.

```
$post=Post::model()->findByPk(10);  
$post->title='novo título do post';  
$post->save(); // salva as alterações para o banco de dados
```

Como podemos ver, utilizamos o mesmo método [save\(\)](#) para fazer a inserção e atualização dos dados. Se uma instância AR é criada por meio do operador new, executar o método [save\(\)](#) irá inserir um novo registro no banco de dados; se a instância é o resultado de um find ou findAll, executar o método [save\(\)](#) irá atualizar o registro existente na tabela. Podemos utilizar a propriedade [CActiveRecord::isNewRecord](#) para verificar se uma instância AR é nova ou não.

Também é possível atualizar um ou vários registros em uma tabela do banco, sem ter que carregá-lo primeiro. Existem os seguintes métodos para efetuar essas operações de uma maneira mais conveniente:

```
// atualiza os registros que satisfaçam a condição informada  
Post::model()->updateAll($attributes,$condition,$params);  
// atualiza os registros que tenham a chave primária informada, e satisfaçam  
a condição  
Post::model()->updateByPk($pk,$attributes,$condition,$params);  
// atualiza uma coluna counter (contagem) que satisfaça a condição informada  
Post::model()->updateCounters($counters,$condition,$params);
```

No exemplo acima, \$attributes é um vetor com os valores das colunas, indexados pelos nomes delas. \$counter é um vetor com as colunas que terão seus valores incrementados, indexadas pelos seus nomes. \$condition e \$params estão descritos nos itens anteriores.

Excluindo um Registro

Podemos também excluir um registro se a instância AR já estiver preenchida com ele.

```
$post=Post::model()->findByPk(10); // assumindo que há um post com ID 10  
$post->delete(); // exclui o registro da tabela no banco de dados.
```

Note que, depois da exclusão, a instância AR continua inalterada, mas o registro correspondente no banco de dados já foi excluído.

Os seguintes métodos são utilizados para excluir registros sem a necessidade de carregá-los primeiro:

```
// exclui os registros que satisfaçam a condição informada  
Post::model()->deleteAll($condition,$params);  
// exclui os registros com a chave primária e condição informada  
Post::model()->deleteByPk($pk,$condition,$params);
```

Validação de Dados

Ao inserir ou atualizar um registro, geralmente precisamos verificar se o valor está de acordo com certas regras. Isso é especialmente importante nos casos em que os valores das colunas são informados pelos usuários. No geral, é bom nunca confiar em nenhum dado vindo do lado do cliente (usuário).

O AR efetua a validação automaticamente quando o método [save\(\)](#) é executado. A validação é baseada em regras especificadas pelo método [rules\(\)](#) da classe AR. Para mais detalhes sobre como especificar regras de validação consulte [Declarando Regras de Validação](#). Abaixo temos o fluxo típico necessário para salvar um registro:

```
if($post->save())
{
    // dados são validos e são inseridos/atualizados no banco
}
else
{
    // dados são inválidos. utilize getErrors() para recuperar as mensagens
    de erro
}
```

Quando os dados para inserção ou atualização são enviados pelos usuários através de um formulário HTML, precisamos atribuí-los as propriedades correspondentes da classe AR. Podemos fazer isso da seguinte maneira:

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

Se existirem muitos campos, teríamos uma longa lista dessas atribuições. Esse trabalho pode ser aliviado, por meio da propriedade [attributes](#), como feito no exemplo abaixo. Mais detalhes podem ser consultados em [Atribuição de Atributos Seguros](#) e [Criando uma Ação](#).

```
// assumindo que $_POST['Post'] é um vetor com os valores das colunas,
indexados pelos seus nomes
$post->attributes=$_POST['Post'];
$post->save();
```

Comparando Registros

Assim como registros de uma tabela, as instâncias AR também são unicamente identificadas pelos valores de suas chaves primárias. Portanto, para comparar duas instâncias AR, precisamos apenas comparar os valores de suas chaves, assumindo que

ambas pertencem a mesma classe. Entretanto, existe uma maneira mais simples de compará-las, que é utilizar o método [CActiveRecord::equals\(\)](#).

Informação: Diferente das implementações de Active Record em outros frameworks, o Yii suporta chaves primárias compostas em seu AR. Uma chave primária composta é formada por duas ou mais colunas. De forma correspondente, a chave primária é representada por um vetor no Yii. A propriedade [primaryKey](#) retorna a chave uma instância AR.

Personalização

A classe [CActiveRecord](#) possui alguns métodos que podem ser sobrescritos por suas classes derivadas, para personalizar seu fluxo de funcionamento.

- [beforeValidate](#) e [afterValidate](#): esses métodos são executados antes e depois que uma validação é executada
- [beforeSave](#) e [afterSave](#): esses métodos são executados antes e depois que um registro é salvo.
- [beforeDelete](#) e [afterDelete](#): esses métodos são executados antes e depois que uma instância AR é excluída.
- [afterConstruct](#): esse método é utilizado para toda instância AR criada por meio do operador new.
- [beforeFind](#): esse método é chamado antes que um objeto AR finder seja utilizado para executar uma consulta (por exemplo, `find()`, `findAll()`). Ele está disponível a partir da versão 1.0.9.
- [afterFind](#): esse método é chamado após cada instância AR criada como resultado de uma consulta.

Utilizando Transações com AR

Todas as instâncias AR contêm uma propriedade chamada [dbConnection](#) que é uma instância da classe [CDbConnection](#). Podemos então, utilizar o recurso de [transações](#) existente no DAO do Yii para trabalhar com Active Record.

```
$model=Post::model();
$transaction=$model->dbConnection->beginTransaction();
try
{
    // find e save são dois passos que podem ser interrompidos por outra
    // requisição
    // portanto utilizamos uma transação para garantir e a consistência a
    // integridade dos dados
    $post=$model->findByPk(10);
    $post->title='novo título para o post';
    $post->save();
    $transaction->commit();
}
catch(Exception $e)
{
    $transaction->rollBack();
}
```

Named Scopes (Escopos com Nomes)

Nota: O suporte a named scopes está disponível a partir da versão 1.0.5. A ideia original dos named scopes veio do Ruby on Rails.

Um named scope representa um critério de consulta com um nome, que pode ser combinado com outros named scopes e ser aplicado em uma consulta com active record.

Named scopes são declarados, normalmente, dentro do método [CActiveRecord::scopes\(\)](#), como pares nome-critério. O código a seguir, declara dois named scopes, published e recently, dentro da classe Post:

```

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

```

Cada named scope é declarado como um vetor que pode ser utilizado para iniciar uma instância da classe [CDbCriteria](#). Por exemplo, o named scope recently especifica que o valor da propriedade order seja createTime DESC e o da propriedade limit seja 5, que será traduzido em um critério de consulta que retornará os 5 posts mais recentes.

Na maioria das vezes, named scopes são utilizados como modificadores nas chamadas aos métodos find. Vários named scopes podem ser encadeados para gerar um conjunto de resultados mais restrito. Por exemplo, para encontrar os posts publicados recentemente, podemos fazer como no código abaixo:

```

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

```

Geralmente, os named scopes aparecem a esquerda da chamada ao método find. Então, cada um deles fornece um critério de pesquisa que é combinado com outros critérios, incluindo o que é passado para o método find. Esse encadeamento é como adicionar uma lista de filtros em um consulta.

A partir da versão 1.0.6, named scopes também podem ser utilizados com os métodos update e delete. Por exemplo, no código a seguir vemos como deletar todos os posts publicados recentemente:

```

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

```

Nota: Named scopes podem ser utilizados somente como métodos a nível de classe. Por esse motivo, o método deve ser executando utilizando NomeDaClasse::model().

Named Scopes Parametrizados

Named scopes podem ser parametrizados. Por exemplo, podemos querer personalizar o número de posts retornados no named scope `recently`. Para isso, em vez de declarar o named scope dentro do método [CActiveRecord::scopes](#), precisamos definir um novo método cujo nome seja o mesmo do escopo:

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

Então, para recuperar apenas os 3 posts publicados recentemente, utilizamos:

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

Se não tivéssemos informado o parâmetro 3 no exemplo acima, iríamos recuperar 5 posts, que é a quantidade padrão definida no método.

Named Scope Padrão

A classe de um modelo pode ter um named scope padrão, que é aplicado para todas as suas consultas (incluindo as relacionais). Por exemplo, um website que suporte vários idiomas, pode querer exibir seu conteúdo somente no idioma que o usuário especificar.

Como devem existir muitas consultas para recuperar esse conteúdo, podemos definir um named scope para resolver esse problema. Para isso sobrescrevemos o método [CActiveRecord::defaultScope](#), como no código a seguir:

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

Assim, a chamada de método a seguir irá utilizar automaticamente o critério definido acima:

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

Note que o named scope padrão é aplicado somente as consultas utilizando SELECT. Ele é ignorado nas consultas com INSERT, UPDATE e DELETE.

Active Record Relacional

Nós já vimos como utilizar Active Record (AR) para selecionar dados de uma tabela em um banco de dados. Nessa seção, descrevemos como utilizar AR para acessar registros relacionados em diversas tabelas e como recuperar esse conjunto de dados.

Para utilizar o AR de forma relacional, é necessário que as relações entre chaves primárias e estrangeiras estejam bem definidas entre as tabelas que farão parte do relacionamento.

Nota: A partir da versão 1.0.1, você pode utilizar AR relacional mesmo que você não tenha definido nenhuma chave estrangeira em suas tabelas.

Para simplificar, os exemplos desta seção serão baseados na estrutura de tabelas exibida no seguinte diagrama de entidade-relacionamento:

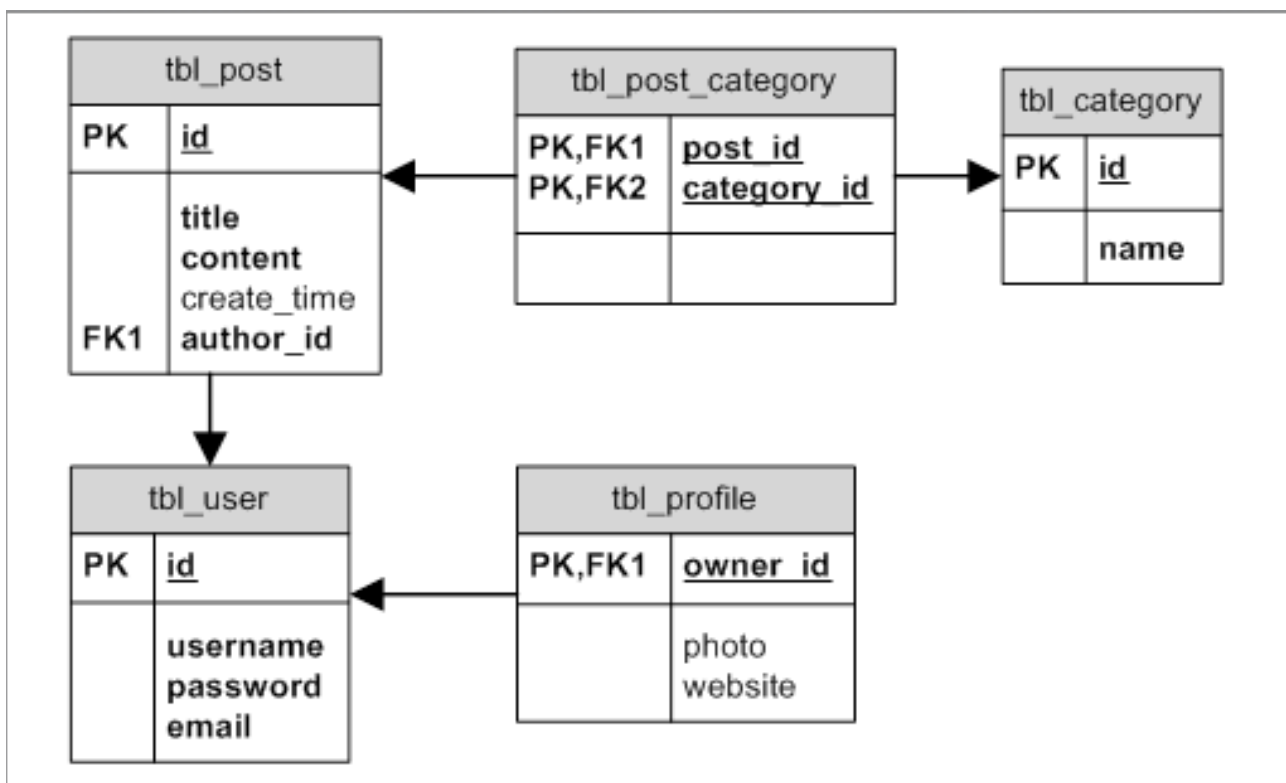


Diagrama Entidade Relacionamento

Informação: O suporte à chaves estrangeiras é diferente dependendo do SGBD.

O SQLite não tem suporte chaves estrangeiras, mas mesmo assim você pode declará-las quando estiver criando as tabelas. O AR é capaz de tirar proveito dessas declarações para efetuar corretamente as consultas relacionais.

O MySQL suporta chaves estrangeiras apenas utilizando a engine InnoDB. Por isso recomendamos utilizar InnoDB em seus bancos de dados MySQL. Quando utilizar a engine MyISAM, você pode utilizar o seguinte truque para realizar consultas relacionais utilizando 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)'
);
```

No código acima, utilizamos a palavra-chave COMMENT para descrever a chave estrangeira. O AR pode ler o COMMENT e reconhecer o relacionamento descrito nele.

Declarando Relacionamentos

Antes de utilizar AR para executar consultas relacionais, precisamos fazer com que uma classe AR saiba que está relacionada a outra.

O relacionamento entre duas classes AR está diretamente relacionado com o relacionamento entre as tabelas no banco, representadas pelas classes. Do ponto de vista do banco de dados, um relacionamento entre duas tabelas, A e B, pode ser de três tipos: um-para-muitos (exemplo, User e Post), um-para-um (exemplo, User e Profile) e muitos-para-muitos (exemplo, Category e Post). No AR, existem 4 tipos de relacionamentos:

- BELONGS_TO: se o relacionamento entre as tabelas A e B for um-para-muitos, então B pertence a A. (por exemplo, Post pertence a User);
- HAS_MANY: se o relacionamento entre as tabelas A e B for um-para-muitos, então A tem vários B (por exemplo, User tem vários Post);
- HAS_ONE: esse é um caso especial de HAS_MANY, onde A tem no máximo um B. (por exemplo, User tem no máximo um Profile);
- MANY_MANY: corresponde ao relacionamento muitos-para-muitos. É necessária uma terceira tabela associativa para quebrar o relacionamento muitos-para-muitos em relacionamentos um-para-muitos, já que a maioria dos bancos de dados não suporta esse tipo de relacionamento. No diagrama de entidade-relacionamento, a tabela PostCategory tem essa finalidade. Na terminologia AR, podemos explicar o

tipo MANY_MANY como a combinação do BELONGS_TO com um HAS_MANY. Por exemplo, Post pertence a várias Category e Category tem vários Post.

A declaração de relacionamentos no AR é feita sobrescrevendo o método [relations\(\)](#) da classe [CActiveRecord](#). Esse método retorna um vetor com as configurações do relacionamento. Cada elemento do vetor representa um relacionamento com o seguinte formato:

```
NomeRel'=>array('TipoDoRelacionamento', 'NomeDaClasse',  
'ChaveEstrangeira', ...opções adicionais)
```

Onde NomeRel é o nome do relacionamento; TipoDoRelacionamento especifica o tipo do relacionamento, que pode ser uma dessas quatro constantes: self::BELONGS_TO, self::HAS_ONE, self::HAS_MANY e self::MANY_MANY; NomeDaClasse é o nome da classe AR relacionada com essa classe; eChaveEstrangeira especifica a(s) chave(s) estrangeira(s) envolvidas no relacionamento. Opções adicionais podem ser especificadas ao final de cada vetor de relacionamento (a ser explicado).

O código a seguir mostra como declaramos os relacionamentos para as classes User e Post:

```
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'),  
        );  
    }  
}
```

Informação: Uma chave estrangeira pode ser composta, consistindo de duas ou mais colunas. Nesse caso, devemos concatenar os nomes das colunas da chave estrangeira e separá-los por um espaço ou uma vírgula. Para relacionamentos do tipo MANY_MANY, a tabela associativa também deve ser especificada na chave estrangeira. Por exemplo, o relacionamento categories em Post é especificado com a chave estrangeira PostCategory(postID, categoryID).

A declaração de relacionamentos em uma classe AR adiciona, implicitamente, uma propriedade para cada relacionamento declarado. Depois que a consulta relacional for executada, a propriedade correspondente será preenchida com as instâncias AR relacionadas. Por exemplo, se \$author representa uma instância de User, podemos utilizar \$author->posts para acessar as instâncias de seus Post relacionados.

Executando Consultas Relacionais

A maneira mais simples de executar uma consulta relacional é acessar uma propriedade relacional em uma instância AR. Se a propriedade não foi acessada antes, uma consulta relacional será iniciada, que irá unir as duas tabelas relacionadas e filtra-las pela chave primária da instância. O resultado da consulta será armazenado na propriedade como instância(s) da classe AR relacionada. Essa técnica é conhecida por lazy loading (carregamento retardado). Ou seja, a pesquisa relacional é executada somente quando os objetos relacionados são acessados. O exemplo abaixo mostra como utilizar essa técnica:

```
// recupera post com ID 10
$post=Post::model()->findPk(10);
// recupera o autor do post. Uma consulta relacional será executada aqui
$author=$post->author;
```

Informação: Se não existirem instâncias para um relacionamento, a propriedade correspondente poderá ser null ou um vetor vazio. Para relacionamentos do tipo BELONGS_TO e HAS_ONE, o retorno é null; para HAS_MANY e MANY_MANY, o retorno é um vetor vazio. Note que HAS_MANY e MANY_MANY retornam um vetor de objetos. Sendo assim, você precisará primeiro acessar seus elementos para acessar suas propriedades. Caso contrário, você poderá gerar o erro "Trying to get property of non-object".

A técnica do lazy loading é bastante conveniente, mas não é eficiente em alguns cenários. Por exemplo, se queremos acessar informações do autor para N posts, a utilização de lazy loading irá executar N consultas. Nessas circunstâncias devemos recorrer a técnica de eager loading.

Nessa técnica, recuperamos as instâncias AR relacionadas junto com a instância AR principal. Isso é feito utilizando-se o método [with\(\)](#), junto com um dos métodos [find](#) ou [findAll](#). Por exemplo:

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

O código acima retornará um vetor de instâncias de Post. Diferente do lazy loading, a propriedade `author` de cada instância de Post já está preenchida com a instância de User relacionada, antes de acessarmos a propriedade. Em vez de executar uma consulta de junção para cada post, a técnica de eager loading traz todos os posts, junto com seus autores, em uma única consulta.

Podemos especificar o nome de múltiplos relacionamentos na chamada do método `with()` e o eager loading se encarregará de trazê-los todos de uma só vez. Por exemplo, o código a seguir irá recuperar todos os posts, juntos com seus autores e suas categorias:

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

Podemos também fazer eager loadings aninhados. Em vez de uma lista com nomes de relacionamentos, podemos passar uma representação hierárquica de nomes de relacionamentos para o método `with()`, como no exemplo a seguir:

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

O exemplo acima irá recuperar todos os posts, junto com seus autores e suas categorias. Ele trará também o perfil de cada autor e seus posts.

Nota: O uso do método `with()` foi alterado a partir da versão 1.0.2. Por favor, leia a documentação correspondente cuidadosamente.

A implementação do AR no Yii é bastante eficiente. Quando utilizamos eager loading para carregar uma hierarquia de objetos relacionados envolvendo N relacionamentos do tipo HAS_MANY e MANY_MANY, serão necessárias N+1 consultas SQL para obter os resultados necessários. Isso significa que serão executadas 3 consultas SQL no último exemplo, por causa das propriedades `posts` e `categories`. Outros frameworks preferem uma técnica mais radical, utilizando somente uma consulta. A primeira vista, essa técnica parece mais eficiente porque menos consultas estão sendo executadas pelo banco de dados. Mas na realidade, isso é impraticável por duas razões. Primeira, existem muitas colunas com dados repetidos nos resultados, que precisam de mais tempo para serem transmitidos e processados. Segunda, o número de registros em um resultado cresce exponencialmente de acordo com o número de tabelas envolvidas, de forma a ficarem simplesmente intratáveis quanto mais relacionamentos estão envolvidos.

A partir da versão 1.0.2, você também pode forçar que a consulta relacional seja feita com uma única consulta SQL. Simplesmente adicione uma chamada ao método `together()` depois do método `with()`. Por exemplo:

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

A consulta acima será executada em um único comando SQL. Sem o método [together](#), serão necessárias três consultas: uma irá fazer a junção das tabelas Post, User e Profile, outra irá fazer a junção de User e Post e a última irá fazer a junção de Post, PostCategory e Category.

Opções de Consultas Relacionais

Nós mencionamos que pode-se especificar algumas opções adicionais na declaração de um relacionamento. Essas opções, especificadas na forma de pares nome-valor, são utilizadas para personalizar as consultas relacionais. Elas estão resumidas abaixo:

- **select**: uma lista de colunas que serão selecionadas nas classes AR relacionadas. Por padrão, seu valor é '*', que significa todas as colunas. Quando utilizada em expressões os nomes das colunas devem se identificados com um aliasToken (apelido para tabela), (por exemplo COUNT(??.name) AS nameCount).
- **condition**: representa a cláusula WHERE. Não tem nenhum valor por padrão. Note que, para evitar conflitos entre nomes de colunas iguais, referencias a colunas precisam ser identificadas por um aliasToken, (por exemplo, ??id=10).
- **params**: os parâmetros que serão vinculados ao comando SQL gerado. Eles devem ser informados em um vetor, com pares de nome-valor. Essa opção está disponível desde a versão 1.0.3.
- **on**: representa a cláusula ON. A condição especificada aqui será adicionada à condição de junção, utilizando-se o operador AND. Note que, para evitar conflitos entre nomes de colunas iguais, os nomes delas devem ser diferenciados com a utilização de um aliasToken, (por exemplo, ??id=10). Essa opção não pode ser utilizada em relações do tipo MANY_MANY. Ela está disponível desde a versão 1.0.2.
- **order**: representa a cláusula ORDER BY. Não tem nenhum valor por padrão. Note que, para evitar conflitos entre nomes de colunas iguais, referencias a colunas precisam ser identificadas por um aliasToken, (por exemplo, ??age DESC).
- **with**: uma lista de objetos filhos relacionados que deverão ser carregados juntos com esse objeto. Seja cuidadoso, pois utilizar esta opção de forma inapropriada poderá gerar um loop infinito nos relacionamentos.
- **joinType**: o tipo de junção nesse relacionamento. Por padrão é LEFT OUTER JOIN.
- **aliasToken**: é o marcador do prefixo da coluna. Ele será substituído pelo apelido da tabela correspondente para diferenciar as referencias às colunas. Seu padrão é '??.'.
- **alias**: o apelido para a tabela associada a esse relacionamento. Essa opção está disponível desde a versão 1.0.1. Por padrão é null, indicando que o apelido para a tabela será gerado automaticamente. Um alias é diferente de uma aliasToken, uma vez que esse último é só um marcador que será substituído pelo verdadeiro apelido para a tabela.
- **together**: especifica se a tabela associada com esse relacionamento deverá ser forçada a juntar-se a tabela primária. Essa opção só faz sentido em relacionamentos do tipo HAS_MANY e MANY_MANY. Se esta opção não for utilizada, ou seu valor for false, cada relacionamento HAS_MANY e MANY_MANY terão suas próprias

instruções JOIN, para aumentar a performance. Essa opção está disponível desde a versão 1.0.3.

- **group**: representa a cláusula GROUP BY. Não tem nenhum valor por padrão. Note que, para evitar conflitos entre nomes de colunas iguais, referências a colunas precisam ser identificadas por um aliasToken, (por exemplo, ??age).
- **having**: representa a cláusula HAVING. Não tem nenhum valor por padrão. Note que, para evitar conflitos entre nomes de colunas iguais, referências a colunas precisam ser identificadas por um aliasToken, (por exemplo, ??age DESC). Esta opção está disponível desde a versão 1.0.1.
- **index**: o nome de uma coluna cujos valores devem ser utilizados como chaves no vetor que retorna os objetos relacionados. Sem a utilização dessa opção, o vetor retornado tem índices numéricos iniciando em 0. Esta opção só pode ser utilizada em relacionamentos do tipo HAS_MANY e MANY_MANY. Ela está disponível desde a versão 1.0.7.

Além dessas, as opções abaixo podem ser utilizadas para certos relacionamentos quando utilizado o lazy loading:

- **limit**: limite de registros a ser selecionado. Essa opção NÃO pode ser utilizada em relacionamentos do tipo BELONGS_TO.
- **offset**: posição, em relação aos resultados encontrados, que os registros serão selecionados. Essa opção NÃO pode ser utilizada em relacionamentos do tipo BELONGS_TO.

Abaixo alteramos a declaração do relacionamento posts em User, incluindo algumas das opções descritas acima:

```
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'),
        );
    }
}
```

Agora, se acessarmos \$author->post, obteremos os posts desse autor, ordenados pela sua data de criação em ordem descendente. Cada instância de post virá com suas categorias já carregadas.

Informação: Quando o nome de uma coluna aparece em duas ou mais tabelas em uma junção, eles precisam ser diferenciados. Isso é feito prefixando-se o nome da tabela ao nome da coluna. Por exemplo, id se torna Team.id. Entretanto, nas consultas relacionais utilizando AR, não temos essa liberdade porque as instruções de relacionamentos são geradas automaticamente pelo Active Record, que dá a cada tabela um apelido (alias). Por isso, para evitar evitar conflitos entre nomes de colunas devemos utilizar um marcador para identificar a existência de uma coluna que precisa ser diferenciada. O AR irá substituir esse marcador pelo apelido da tabela, diferenciando apropriadamente a coluna.

Opções Dinâmicas em Consultas Relacionais

A partir da versão 1.0.2, podemos utilizar opções dinâmicas em consultas relacionais, tanto no método `with()` quanto na opção `with`. As opções dinâmicas irão sobrescrever as opções existentes, declaradas no método `relations()`. Por exemplo, com o modelo `User` acima, podemos utilizar eager loading para retornar os posts de um determinado autor em ordem crescente (a opção `order` especificada na relação utiliza ordem decrescente), como no exemplo abaixo:

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

A partir da versão 1.0.5, opções dinâmicas também podem ser utilizadas com o lazy loading. Para isso, devemos chamar o método cujo nome é o mesmo da relação e passar as opções dinâmicas como parâmetros para ele. Por exemplo, o código a seguir retorna os posts de um usuário cujo status é 1:

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

Consultas Estatísticas

Nota: Consultas estatísticas são suportadas a partir da versão 1.0.4.

Além das consultas relacionais descritas acima, o Yii também suporta as chamadas consultas estatísticas (statistical query, ou aggregational query). Essas consultas retornam informações agregadas sobre objetos relacionados, tais como o número de comentários em cada post, a avaliação média para cada produto, etc. Consultas estatísticas só podem ser utilizadas em objetos relacionadas com HAS_MANY (por exemplo, um post tem vários comentários) ou MANY_MANY (por exemplo, um post pertence a várias categorias e uma categoria tem vários posts).

Executar consultas estatísticas é bem parecido com a execução de consultas relacionais que descrevemos anteriormente. Primeiro precisamos declarar a consulta estatística no método `relations()` da classe `CActiveRecord`, como fazemos com as consultas relacionais.


```

class Post extends ActiveRecord
{
    public function relations()
    {
        return array(
            'commentCount'=>array(self::STAT, 'Comment', 'postID'),
            'categoryCount'=>array(self::STAT, 'Category', 'PostCategory
(postID, categoryID)'),
        );
    }
}

```

No código acima, declaramos duas consultas estatísticas: `commentCount`, que calcula o número de comentários em um post e `categoryCount`, que calcula o número de categorias a quem um post pertence. Note que o relacionamento entre Post e Comment é do tipo HAS_MANY, enquanto o relacionamento entre Post e Category é MANY_MANY (com a tabela associativa PostCategory). Como podemos ver a declaração é bastante similar a das relações descritas anteriormente. A única diferença é que o tipo de relação agora é STAT.

Com a declaração acima, podemos recuperar o número de comentários para um post acessando a propriedade `$post->commentCount`. Ao acessá-la pela primeira vez, uma instrução SQL será executada, implicitamente, para retornar o resultado correspondente. Como já sabemos, essa técnica é chamada de lazy loading. Também podemos utilizar a técnica eager loading se precisarmos determinar a quantidade de comentários para vários posts.

```

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

```

O código acima irá executar três instruções SQL para retornar todos os posts juntamente com a quantidade de comentários e categorias em cada um deles. Utilizando lazy loading, acabaríamos executando $2*N+1$ instruções SQL, se existirem N posts.

Por padrão, uma consulta estatística utiliza a expressão COUNT para calcular os resultados. Podemos personaliza-la especificando opções adicionais, no método [relations\(\)](#). As opções disponíveis estão resumidas abaixo:

- **select:** a expressão estatística. Por padrão é COUNT(*), que contará todos os objetos.
- **defaultValue:** o valor a ser atribuído aos registros que não receberem um resultado em uma consulta estatística. Por exemplo, se um post não tiver comentários, o valor de commentCount será o especificado nesta propriedade. Por padrão, seu valor é 0.
- **condition:** representa a cláusula WHERE. Não tem nenhum valor por padrão.
- **params:** representa os parâmetros que devem ser vinculados na instrução SQL gerada.

Deve ser um vetor com pares nome-valor.

- order: representa a cláusula ORDER BY. Não tem nenhum valor por padrão.
- group: representa a cláusula GROUP BY. Não tem nenhum valor por padrão.
- having: representa a cláusula HAVING. Não tem nenhum valor por padrão.

Consultas Relacionais Utilizando Named Scopes

Nota: O suporte a named scopes está disponível a partir da versão 1.0.5.

As consultas relacionais também podem ser utilizadas em conjunto com [named scopes](#). Isso pode ser feito de duas formas. Na primeira, os named scopes são aplicados aos modelos principais. Na segunda forma os named scopes são aplicados aos modelos relacionados.

No código a seguir vemos como aplicar named scopes no modelo principal:

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

Vemos que é bem parecido com a forma não-relacional. A única diferença é uma chamada ao método `with()` após a cadeia de named scopes. Essa consulta retornará os posts publicados recentemente, junto com seus comentários.

O código a seguir mostra como aplicar named scopes em objetos relacionados.

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

A consulta acima irá retornar todos os posts, junto com os seus comentário aprovados. Note que `comments` refere-se ao nome do relacionamento, enquanto `recently` e `approved` referem-se a dois named scopes declarados na classe do modelo `Comment`. O nome do relacionamento e os named scopes devem ser separados por dois-pontos (":").

Named scopes também podem ser especificados com a opção `with`, descrita acima. No exemplo a seguir, a propriedade `$user->posts` retorna todos os comentários aprovados dos posts.

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

A consulta acima irá retornar todos os posts, junto com os seus comentário aprovados. Note que `comments` refere-se ao nome do relacionamento, enquanto `recently` e `approved` referem-se a dois named scopes declarados na classe do modelo `Comment`. O nome do relacionamento e os named scopes devem ser separados por dois-pontos (":").

Named scopes também podem ser especificados com a opção `with`, descrita acima. No exemplo a seguir, a propriedade `$user->posts` retorna todos os comentários aprovados dos posts.

```

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

```

Nota: Named scopes aplicados a modelos relacionados devem ser declarados no método `CActiveRecord::scopes`. Como resultado, eles não podem ser parametrizados.

Migração de Bases de Dados

Note: The database migration feature has been available since version 1.1.6.

Like source code, the structure of a database is evolving as we develop and maintain a database-driven application. For example, during development, we may want to add a new table; or after the application is put into production, we may realize the need of adding an index on a column. It is important to keep track of these structural database changes (called migration) like we do with our source code. If the source code and the database are out of sync, it is very likely the whole system may break. For this reason, Yii provides a database migration tool that can keep track of database migration history, apply new migrations, or revert existing ones.

The following steps show how we can use database migration during development:

- Tim creates a new migration (e.g. create a new table)
- Tim commits the new migration into source control system (e.g. SVN, GIT)
- Doug updates from source control system and receives the new migration
- Doug applies the migration to his local development database

Yii supports database migration via the `yiic migrate` command line tool. This tool supports creating new migrations, applying/reverting/redone migrations, and showing migration history and new migrations.

In the following, we will describe how to use this tool.

Creating Migrations

To create a new migration (e.g. create a news table), we run the following command:

```

yiic migrate create <name>

```

The required name parameter specifies a very brief description of the migration (e.g. `create_news_table`). As we will show in the following, the name parameter is used as part of a PHP class name. Therefore, it should only contain letters, digits and/or underscore characters.

```
yii migrate create create_news_table
```

The above command will create under the `protected/migrations` directory a new file named `m101129_185401_create_news_table.php` which contains the following initial code:

```
class m101129_185401_create_news_table extends CDbMigration
{
    public function up()
    {
    }

    /**
     * public function down()
     * {
     * }
     */
}
```

Notice that the class name is the same as the file name which is of the pattern `m<timestamp>_<name>`, where `<timestamp>` refers to the UTC timestamp (in the format of `yymmdd_hhmmss`) when the migration is created, and `<name>` is taken from the command's name parameter.

The `up()` method should contain the code implementing the actual database migration, while the `down()` method may contain the code reverting what is done in `up()`.

Sometimes, it is impossible to implement `down()`. For example, if we delete table rows in `up()`, we will not be able to recover them in `down()`. In this case, the migration is called irreversible, meaning we cannot roll back to a previous state of the database.

As an example, let's show the migration about creating a news table.

```

class m101129_185401_create_news_table extends CDbMigration
{
    public function up()
    {
        $this->createTable('tbl_news', array(
            'id' => 'pk',
            'title' => 'string NOT NULL',
            'content' => 'text',
        ));
    }

    public function down()
    {
        $this->dropTable('tbl_news');
    }
}

```

The base class [CDbMigration](#) provides a set of methods for manipulating data and schema of a database. For example, [CDbMigration::createTable](#) will create a database table, while [CDbMigration::insert](#) will insert a row of data. These methods all use the database connection returned by [CDbMigration::getDbConnection\(\)](#), which by default returns `Yii::app()->db`.

Info: You may notice that the database methods provided by `CDbMigration` are very similar to those in `CDbCommand`. Indeed they are nearly the same except that `CDbMigration` methods will measure the time used by their methods and print some messages about the method parameters.

Applying Migrations

To apply all available new migrations (i.e., make the local database up-to-date), run the following command:

```
yiic migrate
```

The command will show the list of all new migrations. If you confirm to apply the migrations, it will run the `up()` method in every new migration class, one after another, in the order of the timestamp value in the class name.

After applying a migration, the migration tool will keep a record in a database table named `tbl_migration`. This allows the tool to identify which migrations have been applied and which are not. If the `tbl_migration` table does not exist, the tool will automatically create it in the database specified by the db application component.

Sometimes, we may only want to apply one or a few new migrations. We can use the following command:

```
yiic migrate up 3
```

This command will apply the 3 new migrations. Changing the value 3 will allow us to change the number of migrations to be applied.

We can also migrate the database to a specific version with the following command:

```
yiic migrate to 101129_185401
```

That is, we use the timestamp part of a migration name to specify the version that we want to migrate the database to. If there are multiple migrations between the last applied migration and the specified migration, all these migrations will be applied. If the specified migration has been applied before, then all migrations applied after it will be reverted (to be described in the next section).

Reverting Migrations

To revert the last one or several applied migrations, we can use the following command:

```
yiic migrate down [step]
```

where the optional step parameter specifies how many migrations to be reverted back. It defaults to 1, meaning reverting back the last applied migration.

As we described before, not all migrations can be reverted. Trying to revert such migrations will throw an exception and stop the whole reverting process.

Redoing Migrations

Redoing migrations means first reverting and then applying the specified migrations. This can be done with the following command:

```
yiic migrate redo [step]
```

where the optional step parameter specifies how many migrations to be redone. It defaults to 1, meaning redoing the last migration.

Showing Migration Information

Besides applying and reverting migrations, the migration tool can also display the migration history and the new migrations to be applied.

```
yiic migrate history [limit]  
yiic migrate new [limit]
```

where the optional parameter `limit` specifies the number of migrations to be displayed. If `limit` is not specified, all available migrations will be displayed.

The first command shows the migrations that have been applied, while the second command shows the migrations that have not been applied.

Modifying Migration History

Sometimes, we may want to modify the migration history to a specific migration version without actually applying or reverting the relevant migrations. This often happens when developing a new migration. We can use the the following command to achieve this goal.

```
yii migrate mark 101129_185401
```

This command is very similar to `yii migrate to` command, except that it only modifies the migration history table to the specified version without applying or reverting the migrations.

Customizing Migration Command

There are several ways to customize the migration command.

Use Command Line Options

The migration command comes with four options that can be specified in command line:

- `interactive`: boolean, specifies whether to perform migrations in an interactive mode. Defaults to `true`, meaning the user will be prompted when performing a specific migration. You may set this to `false` should the migrations be done in a background process.
- `migrationPath`: string, specifies the directory storing all migration class files. This must be specified in terms of a path alias, and the corresponding directory must exist. If not specified, it will use the `migrations` sub-directory under the application base path.
- `migrationTable`: string, specifies the name of the database table for storing migration history information. It defaults to `tbl_migration`. The table structure is `version varchar (255) primary key, apply_time integer`.
- `connectionID`: string, specifies the ID of the database application component. Defaults to `'db'`.
- `templateFile`: string, specifies the path of the file to be served as the code template for generating the migration classes. This must be specified in terms of a path alias (e.g. `application.migrations.template`). If not set, an internal template will be used. Inside the template, the token `{ClassName}` will be replaced with the actual migration class name.

To specify these options, execute the migrate command using the following format

```
yii migrate up --option1=value1 --option2=value2 ...
```

For example, if we want to migrate for a forum module whose migration files are located within the `module'smigrations` directory, we can use the following command:

```
yiic migrate up --migrationPath=ext.forum.migrations
```

Configure Command Globally

While command line options allow us to configure the migration command on-the-fly, sometimes we may want to configure the command once for all. For example, we may want to use a different table to store the migration history, or we may want to use a customized migration template. We can do so by modifying the console application's configuration file like the following,

```
return array(  
    .....  
    'commandMap'=>array(  
        'migrate'=>array(  
            'class'=>'system.cli.commands.MigrateCommand',  
            'migrationPath'=>'application.migrations',  
            'migrationTable'=>'tbl_migration',  
            'connectionID'=>'db',  
            'templateFile'=>'application.migrations.template',  
        ),  
        .....  
    ),  
    .....  
);
```

Now if we run the migrate command, the above configurations will take effect without requiring us to enter the command line options every time.

Caching

Visão Geral

Caching é uma maneira rápida e efetiva de aumentar a performances de aplicações Web. Ao armazenar em cache dados relativamente estáticos, e utiliza-lo quando esses dados forem requisitados, economizamos o tempo necessário para gerar esses dados.

A utilização de cache no Yii consiste em configurar e acessar um componente de cache.

A configuração de aplicação exibida a seguir, especifica um componente de cache que utiliza memcache com dois servidores:

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

Quando a aplicação está em execução, o componente pode ser acessado via: `Yii::app()->cache`.

O Yii fornece diversos componentes de cache que podem armazenar dados em diferentes meios. Por exemplo, o componente [CMemCache](#) encapsula a extensão memcache do PHP e utiliza a memória como meio para armazenar os dados; o componente [CApcCache](#) encapsula a extensão APC; e o componente [CDbCache](#) armazena os dados do cache em um banco de dados. Abaixo temos um resumo dos componentes de cache disponíveis:

- [CMemCache](#): utiliza a extensão PHP [memcache](#).
- [CApcCache](#): utiliza a extensão PHP [APC](#).
- [CXCACHE](#): utiliza a extensão PHP [XCache](#). Nota: esse componente está disponível a partir da versão 1.0.1.
- [CEAcceleratorCache](#): utiliza a extensão PHP [EAccelerator](#).
- [CDbCache](#): utiliza uma tabela no banco de dados para armazenar os dados. Por padrão, ele irá criar e utilizar um banco de dados SQLite3 no diretório runtime de sua aplicação. Você pode especificar explicitamente um banco de dados por meio da propriedade [connectionID](#).

- [CZendDataCache](#): utiliza o Zend Data Cache como meio de armazenamento. Nota: esse componente está disponível a partir da versão 1.0.4.
- [CFileCache](#): utiliza arquivos para armazenar os dados em cache. Esse método é particularmente útil para armazenar grandes quantidades de dados em cache (por exemplo, páginas). Nota: esse componente está disponível a partir da versão 1.0.6.
- [CDummyCache](#): é um cache falso, que na realidade não realiza caching algum. A finalidade deste componente é simplificar o desenvolvimento de código que precisa trabalhar com dados em cache. Por exemplo, durante o desenvolvimento ou quando o servidor atual não tem suporte a cache, podemos utilizar esse componente. Assim, quando o suporte a cache estiver disponível basta apenas trocar o componente. Em ambos os casos, podemos utilizar utilizar o `Yii::app()->cache->get($key)` para recuperar dados do cache, sem se preocupar se `Yii::app()->cache` é null. Este componente está disponível a partir da versão 1.0.5;

Dica: Como todos os componentes de cache são derivados da classe `CCache`, você pode alterar entre diversos tipos de cache sem modificar o código que os utilizam.

O caching pode ser utilizado em diferentes níveis. No mais baixo nível, podemos utilizar cache para armazenar pequenos dados, tal como uma variável. Chamamos isso de data caching (Caching de dados). No próximo nível, podemos utiliza-lo para armazenar fragmentos de uma página que é gerada por uma visão. No nível mais alto, armazenamos toda a página no cache e a servimos de lá, quando necessário.

Nas próximas subseções, falaremos mais sobre como utilizar o cache nesses níveis.

Nota: Por definição, o cache é um meio de armazenamento volátil. Ele não garante a existência dos dados em cache, mesmo que eles não expirem. Portanto, não utilize cache como um meio de armazenamento persistente (por exemplo, não o utilize para armazenar dados da sessão).

Data Caching (Caching de Dados)

O caching de dados consiste em armazenar e recuperar variáveis PHP no cache. Para essa finalidade, a classe base [CCache](#) fornece dois métodos que são utilizados na maioria dos casos: [set\(\)](#) e [get\(\)](#).

Para armazenar a variável `$value` em cache, escolhemos um ID único e utilizamos o método [set\(\)](#) para armazená-lo:

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

O dado armazenado ficará em cache para sempre, a não ser que ele seja eliminado por alguma regra de caching (por exemplo, o espaço para caching esteja cheio e, então, os dados mais velhos são removidos). Para alterar esse comportamento, ao executar o método [set\(\)](#), podemos especificar também um parâmetro de tempo de expiração, de forma a indicar que o dado deve ser removido do cache depois de um certo período de tempo.

```
// Mantém o valor no cache por 30 segundos
Yii::app()->cache->set($id, $value, 30);
```

Mais tarde, quando precisarmos acessar essa variável (nessa requisição, ou em outras) utilizamos o método [get\(\)](#) informando o ID da variável desejada. Se o valor retornado for false, significa que a variável não está disponível no cache, e devemos armazená-la novamente.

```
$value=Yii::app()->cache->get($id);
if($value===false)
{
    // re-armazena $value porque seu valor não foi encontrado
    // Yii::app()->cache->set($id,$value);
}
```

Ao escolher o ID para variável a ser armazenada, tenha certeza de que ele seja único entre todos os outros utilizados pelas demais variáveis em cache na aplicação. NÃO é necessário que o ID seja único entre aplicações diferentes, uma vez que o componente de cache é inteligente o suficiente para diferenciar IDs de aplicações diferentes.

Alguns sistemas de cache, tais como MemCache e APC, suportam a recuperação em lote de vários valores em cache, podendo reduzir a sobrecarga envolvida nesse processo. A partir da versão 1.0.8, um novo método chamado [mget\(\)](#) explora essa funcionalidade. Caso o sistema de cache utilizado não suporte este recurso, o método [mget\(\)](#) irá simulá-lo.

Para remover uma variável do cache, utilizamos o método [delete\(\)](#). Para remover tudo do cache, utilizamos o método [flush\(\)](#). Tenha muito cuidado ao utilizar o método [flush\(\)](#) porque ele também irá remover dados de outras aplicações.

Dica: Como a classe CCache implementa ArrayAccess, um componente de cache pode ser utilizado como um vetor. Abaixo, alguns exemplos:

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

Dependências do Cache

Além do tempo de expiração, os dados em cache podem ser invalidados de acordo com a alteração de algumas dependências. Por exemplo, se estamos fazendo cache do conteúdo de um arquivo e ele é alterado, devemos invalidar a cópia em cache e recuperar o conteúdo atualizado diretamente do arquivo, em vez do cache.

Representamos uma dependência como uma instância de [CCacheDependency](#), ou uma de suas classes derivadas. Passamos essa instância, juntamente com os dados que devem ser armazenados, para o método [set\(\)](#).

```
// value irá expirar em 30 segundos
// ela também deverá se tornar inválida se o conteúdo de 'FileName' for
alterado
Yii::app()->cache->set($id, $value, 30, new CFileCacheDependency
('FileName'));
```

Agora, ao recuperar a variável `$value` do cache, com o método [get\(\)](#), a dependência será verificada e, se o arquivo foi alterado, retornará um `false`, indicando que a informação precisa ser re-armazenada.

Abaixo temos um resumo com todas as dependências do cache:

- [CFileCacheDependency](#): a dependência é alterada caso a hora de última modificação do arquivo tenha sido alterada.
- [CDirectoryCacheDependency](#): a dependência é alterada se qualquer um dos arquivos ou subdiretórios do diretório informado sofrer alterações.
- [CDbCacheDependency](#): a dependência é alterada se o resultado da consulta informada sofre alterações.
- [CGlobalStateCacheDependency](#): a dependência é alterada se o valor do estado global informado for alterado. Um estado global é uma variável que é persistente entre múltiplas requisições e sessões de uma aplicação. Ele é definido através do método [CApplication::setGlobalState\(\)](#).
- [CChainedCacheDependency](#): a dependência é alterada se qualquer uma das dependências na cadeia informada sofrer alteração.
- [CExpressionDependency](#): a dependência é alterada se o resultado da expressão PHP informada for alterado. Essa classe está disponível a partir da versão 1.0.4.

Caching de Fragmentos

Caching de fragmentos refere-se ao ato de armazenar em cache apenas um fragmento de uma página. Por exemplo, se uma página exibe em uma tabela o resumo anual de vendas, podemos armazenar essa tabela em cache para eliminar o tempo necessário para gera-la em cada requisição.

Para utilizar o caching de fragmentos, utilizamos os métodos [CController::beginCache\(\)](#) e [CController::endCache\(\)](#) na visão de um controle. Esses dois métodos marcam, respectivamente, o início e termino do conteúdo da página que deve ser armazenado em cache. Assim como no [caching de dados](#), precisamos de um ID para identificar o fragmento armazenado.

```
...outro conteúdo HTML...
<?php if($this->beginCache($id)) { ?>
...conteúdo que deve ser armazenado...
<?php $this->endCache(); } ?>
...outro conteúdo HTML...
```

No código acima, se o método [beginCache\(\)](#) retornar false, o conteúdo já armazenado no cache será exibido, caso contrário, o conteúdo dentro do if será executado e, então, armazenado quando o método [endCache\(\)](#) for executado. Na realidade, os métodos [beginCache\(\)](#) e [endCache\(\)](#) encapsulam os métodos de mesmo nome existentes no widget [COutputCache](#). Sendo assim, as opções de caching podem ser os valores iniciais para qualquer uma das propriedades de [COutputCache](#).

Duração

Provavelmente a opção mais utilizada seja a [duration](#), que especifica por quanto tempo o conteúdo deve ser mantido válido no cache. Seu funcionamento é similar ao parâmetro de tempo de expiração do método [CCache::set\(\)](#). O código a seguir armazena em cache o fragmento por, no máximo, uma hora:

```
...outro conteúdo HTML...
<?php if($this->beginCache($id, array('duration'=>3600))) { ?>
...conteúdo a ser armazenado...
<?php $this->endCache(); } ?>
...outro conteúdo HTML...
```

Se não informamos a duração, seu valor padrão será 60, indicando que o conteúdo do cache deve ser invalidado depois de 60 segundos.

Dependência

Assim com o [caching de dados](#), fragmentos de conteúdo armazenados em cache também podem ter dependências. Por exemplo, o conteúdo de um post pode se exibido, caso ele tenha sido alterado ou não.

Para especificar uma dependência, devemos utilizar a opção [dependency](#), que pode ser um objeto implementando a interface `[ICacheDependency]` ou um vetor de configuração que pode ser utilizado para gerar a dependência. O código a seguir especifica que o fragmento depende da alteração do valor da coluna `lastModified`:

```
...outro conteúdo HTML...
<?php if($this->beginCache($id, array('dependency'=>array(
    'class'=>'system.caching.dependencies.CDbCacheDependency',
    'sql'=>'SELECT MAX(lastModified) FROM Post')))) { ?>
...conteúdo a ser armazenado no cache...
<?php $this->endCache(); } ?>
...outro conteúdo HTML...
```

Variação

O conteúdo armazenado em cache pode sofrer variações de acordo com alguns parâmetros. Por exemplo, o perfil pessoal pode ter aparências diferentes para diferentes

usuários. Para armazenar em cache o conteúdo do perfil, seria interessante que a cópia em cache varie de acordo com os IDs dos usuários. Isso significa que devemos utilizar IDs diferentes ao chamar o método [beginCache\(\)](#).

Em vez de deixar para os desenvolvedores o controle sobre a variação desses IDs, a classe [COutputCache](#) já possui esse recurso. Abaixo, um resumo:

- [varyByRoute](#): mudando seu valor para true, o conteúdo em cache irá variar de acordo com a [rota](#). Dessa forma, cada combinação de controle/ação terá seu conteúdo armazenado em cache separadamente.
- [varyBySelection](#): mudando seu valor para true, podemos fazer com que o conteúdo em cache varie de acordo com os IDs da sessão. Dessa forma, cada sessão de usuário pode ter conteúdos diferentes e todos servidos através do cache.
- [varyByParam](#): ao atribuir um vetor de nomes a essa opção, podemos fazer com que o conteúdo do cache varie de acordo com valores passados através de GET. Por exemplo, se uma página exibe o conteúdo de um post de acordo com a variável GET id, podemos definir [varyByParam](#) como `array('id')`, assim podemos armazenar em cache o conteúdo de cada post. Sem esse tipo de variação, poderíamos apenas armazenar um único post.
- [varyByExpression](#): ao atribuir uma expressão PHP a essa opção, podemos fazer com que o conteúdo do cache varie de acordo com o resultado dessa expressão. Essa opção está disponível a partir da versão 1.0.4.

Tipos de Requisição

As vezes, queremos que o cache de fragmentos esteja habilitado somente para certos tipos de requisições. Por exemplo, para uma página exibindo um formulário, queremos armazenar o formulário apenas na primeira vez em que a página é requisitada (via GET). Nas exibições seguintes (via POST), o formulário não deve ser armazenado porque ele pode conter os dados informados pelos usuários. Para isso, podemos utilizar a opção [requestTypes](#):

```
...outro conteúdo HTML...
<?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>
...conteúdo a ser armazenado...
<?php $this->endCache(); } ?>
...outro conteúdo armazenado...
```

Caching Aninhado

O caching de fragmentos pode ser aninhado. Isso é, um fragmento em cache que está dentro de um outro fragmento, também em cache. Por exemplo, os comentários estão em cache, junto do conteúdo do post que também está em cache.

```

...outro conteúdo HTML...
<?php if($this->beginCache($id1)) { ?>
...conteúdo externo em cache...
    <?php if($this->beginCache($id2)) { ?>
        ...conteúdo interno em cache...
        <?php $this->endCache(); } ?>
    ...conteúdo externo em cache...
<?php $this->endCache(); } ?>
...outro conteúdo em cache...

```

Diferentes opções de caching podem ser utilizadas para os caches aninhados. Por exemplo, os caches interno e o externo, no exemplo acima, podem ter durações diferentes. Quando os dados armazenados em cache do conteúdo externo tornarem-se inválidos, o cache do conteúdo interno ainda vai conter seu fragmento válido. Entretanto, o inverso não é verdadeiro. Se o conteúdo externo contém dados válidos, ele sempre irá fornecer a cópia em cache, mesmo que o cache do conteúdo interno já tenha expirado.

Caching de Páginas

O caching de páginas é aquele que armazena o conteúdo de uma página inteira. Ele pode ocorrer em diferentes lugares. Por exemplo, ao ajustar corretamente os cabeçalhos HTTP para uma página, o navegador do cliente pode armazenar em seu cache o conteúdo por um tempo limitado. A aplicação web também pode armazenar o conteúdo da página em cache. Nessa subseção, veremos como fazer.

O cache de páginas pode ser considerado um caso especial de [caching de fragmentos](#). Como o conteúdo de uma página geralmente é gerado aplicando-se um layout a uma visão, o cache não irá funcionar apenas utilizando os métodos [beginCache\(\)](#) e [endCache\(\)](#) no layout. A razão para isso é que o layout é aplicado dentro do método [CController::render\(\)](#) DEPOIS que o conteúdo da visão foi gerado.

Para armazenar a página inteira, devemos pular a execução da ação que gera o conteúdo da página. Para isso, podemos utilizar [COutputCache](#) como um [filtro](#). O código abaixo mostra como configuramos o filtro para o cache:

```

public function filters()
{
    return array(
        array(
            'COutputCache',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}

```

A configuração acima, faz com que o filtro seja aplicado a todas as ações do controle. Podemos limita-lo para uma ou algumas ações utilizando o operador +. Mais detalhes podem ser encontrados em [filtro](#).

Dica: Podemos utilizar [COutputCache](#) como um filtro porque ela estende a classe [CFilterWidget](#), o que faz com que ela seja tanto um widget quanto um filtro. De fato, o jeito como um widget funciona é bastante similar a um filtro: um widget (filtro) é iniciado antes que conteúdo por ele delimitado (ação) seja gerado, e termina depois que seu conteúdo delimitado (ação) foi gerado.

Conteúdo Dinâmico

Quando utilizamos [caching de fragmentos](#) ou [caching de páginas](#), por diversas vezes nos encontramos em uma situação em que todo o conteúdo de uma página é estático, exceto em alguns lugares. Por exemplo, uma página de ajuda pode exibir a informação de ajuda, estática, com o nome do usuário atualmente logado, no topo.

Para resolver esse problema, podemos variar o cache de acordo com o nome do usuário, mas isso seria um desperdício de precioso espaço em cache, uma vez que, exceto pelo nome do usuário, todo o conteúdo é o mesmo. Poderíamos também dividir a página em vários fragmentos e armazená-los individualmente, mas isso tornaria nossa visão mais complexa. Uma técnica melhor é utilizar o recurso de conteúdo dinâmico fornecido pela classe [CController](#).

Um conteúdo dinâmico é um fragmento que não deve ser armazenado, mesmo que o conteúdo que o contém seja armazenado em cache. Para tornar o conteúdo dinâmico para sempre, ele deve ser gerado todas as vezes, mesmo quando o conteúdo é servido do cache. Por esse motivo, precisamos que esse conteúdo seja gerado por algum método ou função.

Utilizamos os método [CController::renderDynamic\(\)](#) para inserir o conteúdo dinâmico no lugar desejado.

```
...outro conteúdo HTML...
<?php if($this->beginCache($id)) { ?>
...fragmento que deve ser armazenado em cache...
    <?php $this->renderDynamic($callback); ?>
...fragmento que deve ser armazenado em cache...
<?php $this->endCache(); } ?>
...outro conteúdo HTML...
```

No exemplo acima, \$callback é um callback válido em PHP. Ele pode ser uma string com o nome de um método na classe do controle ou uma função global. Ele também pode ser um vetor indicando uma método na classe. Qualquer parâmetro adicional para o método [renderDynamic\(\)](#) será passado para o callback. O callback deve retornar o conteúdo dinâmico em vez de exibi-lo.

Estendendo o Yii

Visão Geral

Estender o Yii é uma atividade comum durante o desenvolvimento. Por exemplo, quando você cria um novo controle, você estende o framework herdando da classe [CController](#); quando você cria um novo widget, você está estendendo a classe [CWidget](#) ou outro widget existente. Se o código estendido for projetado para a reutilização por terceiros, o chamamos de extensão.

Uma extensão normalmente atende a um único propósito. No Yii, ela pode ser classificada como:

- [componente da aplicação](#)
- [comportamento](#)
- [widget](#)
- [controle](#)
- [ação](#)
- [filtro](#)
- [comando de console](#)
- validador: um validador é uma classe que estende de [CValidator](#).
- helper: um helper é uma classe somente com métodos estáticos. São como funções globais, que utilizam o nome da classe como seu namespace.
- [módulo](#): um módulo é uma unidade de software independente, que contém [modelos](#), [visões](#), [controles](#) e outros componentes de suporte. Em diversos aspectos, um módulo lembra uma [aplicação](#). A principal diferença é que um módulo está dentro de uma aplicação. Por exemplo, podemos ter um módulo com funcionalidades para o gerenciamento de usuários.

Uma extensão também pode ser um componente que não se encaixe em nenhuma das categorias acima. Na verdade, o Yii é cuidadosamente projetado de forma que, praticamente todo seu código possa ser estendido e customizado para atender necessidades individuais.

Usando Extensões

A utilização de uma extensão normalmente envolve os seguintes passos:

- Faça o download da extensão no [repositório](#) do Yii.
- Descompacte a extensão no diretório `extensions/xyz`, dentro do [diretório base da aplicação](#), onde `xyz` é o nome da extensão.
- Importe, configure e utilize a extensão.

Cada extensão tem um nome que a identifica unicamente. Dada uma extensão chamada `xyz`, podemos sempre utilizar o path `alias.ext.xyz` para localizar seu diretório base, que contém todos os arquivos de `xyz`.

Nota: O path alias `ext` está disponível a partir da versão 1.0.8. Nas versões anteriores, precisávamos utilizar `application.extensions` para nos referir ao diretório base das extensões. Nos exemplos a seguir, vamos assumir que `ext` está definido, Caso você utilize a versão 1.0.7, ou anterior, substitua o path alias por `application.extensions`.

Extensões diferentes tem requisitos diferentes para importação, configuração e utilização. Abaixo, resumimos os tipos mais comuns de utilização de extensões, de acordo com as categorias descritas na visão geral.

Extensões Zii

Antes de descrever a utilização de extensões de terceiros, gostaríamos de apresentar a biblioteca de extensões Zii. Trata-se de um conjunto de extensões criadas pelo time de desenvolvedores do Yii e é incluída em todos os lançamentos do framework, a partir da versão 1.1.0. Essa biblioteca está hospedada no Google Code, no projeto chamado `zii`.

Ao utilizar uma das extensões da Zii, você deve utilizar um path alias para fazer referências às classes correspondentes, no formato `zii.caminho.para.NomeDaClasse`. O alias `zii` é definido pelo framework e aponta para o diretório raiz da biblioteca Zii. Por exemplo, para utilizar a extensão `CGridView`, devemos utilizar o seguinte código em uma view:

```
$this->widget('zii.widgets.grid.CGridView', array(
    'dataProvider'=>$dataProvider,
));
```

Componente de Aplicação

Para utilizar um [componente de aplicação](#), primeiro precisamos alterar a [configuração da aplicação](#) adicionando uma nova entrada na propriedade `components`, como no código abaixo:

```
return array(
    // 'preload'=>array('xyz',...),
    'components'=>array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // configurações de outros componentes
    ),
);
```

Dessa forma, podemos acessar o componente em qualquer lugar utilizando `Yii::app()->xyz`. O componente será criado somente quando for acessado pela primeira vez, a não ser que ele tenha sido adicionado na propriedade `preload`.

Comportamento

[Comportamentos](#) podem ser utilizados em todos os tipos de componentes. O processo é realizado em dois passos. No primeiro, um comportamento é atribuído a um componente. No segundo, um método do comportamento é executado através do componente. Por exemplo:

```
// $nome identifica o comportamento dentro do componente
$componente->attachBehavior($nome, $comportamento);
// test() é um método de $comportamento
$componente->test();
```

Na maioria das vezes, um comportamento é atribuído a um componente através de configurações, em vez de utilizar o método `attachBehavior`. Por exemplo, para atribuir um comportamento a um [componente da aplicação](#), podemos utilizar a seguinte [configuração](#):

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'CDbConnection',
            'behaviors'=>array(
                'xyz'=>array(
                    'class'=>'ext.xyz.XyzComportamento',
                    'propriedade1'=>'valor1',
                    'propriedade2'=>'valor2',
                ),
            ),
        ),
        //....
    ),
);
```

No exemplo acima, o comportamento `xyz` é atribuído ao componente `db`. Essa forma de atribuição é possível porque a classe [CApplicationComponent](#) define uma propriedade chamada `behaviors`. Ao atribuir a ela uma lista de configurações de comportamentos, o componente irá anexá-los quando for inicializado.

Para as classes [CController](#), [CFormModel](#) e [CActiveModel](#), que, normalmente, necessitam ser estendidas, a atribuição de comportamentos é feita sobrescrevendo-se o método `behaviors()`. Qualquer comportamento declarado nesse método será automaticamente anexo à classe. Por exemplo:

```

public function behaviors()
{
    return array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzComportamentos',
            'propriedade1'=>'valor1',
            'propriedade2'=>'valor2',
        ),
    );
}

```

Widget

[Widgets](#) são utilizados principalmente nas [visões](#). Dada uma classe widget, chamada XyzClass, pertencente a extensão xyz, podemos utiliza-la da seguinte maneira:

```

// um widget que não precisa de conteúdo para seu corpo
<?php $this->widget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

// um widget que precisa de conteúdo para o seu corpo
<?php $this->beginWidget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

...conteúdo do corpo do widget...

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

```

Ação

[Ações](#) são utilizadas por um [controle](#) para responder à uma requisição específica do usuário. Dada a classe da ação XyzClass, pertencente a extensão xyz, podemos utiliza-la sobrescrevendo o método [CController::actions](#) na classe de nosso controle:

```

class TestController extends CController
{
    public function actions()
    {
        return array(
            'xyz'=>array(
                'class'=>'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // outras ações
        );
    }
}

```

Dessa forma, a ação pode ser acessada através da [rota](#) test/xyz.

Filtro

[Filtros](#) também são utilizados por um [controle](#). Basicamente eles pré e pós processam a requisição do usuário manuseada por uma [ação](#). Dada a classe do filtro XyzClass, pertencente a extensão xyz, podemos utiliza-la sobrescrevendo o método [CController::filters](#), na classe de nosso controle.

```

class TestController extends CController
{
    public function filters()
    {
        return array(
            array(
                'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // outros filtros
        );
    }
}

```

No exemplo acima, podemos utilizar no primeiro elemento do vetor os operadores + e -, para limitar as ações onde o filtro será aplicado. Para mais detalhes, veja a documentação da classe [CController](#).

Controle

Um [controle](#), fornece um conjunto de ações que podem ser requisitadas pelos usuários. Para utilizar uma extensão de um controle, precisamos configurar a propriedade `CWebApplication::controllerMap` na [configuração da aplicação](#):

```
return array(  
    'controllerMap'=>array(  
        'xyz'=>array(  
            'class'=>'ext.xyz.XyzClass',  
            'property1'=>'value1',  
            'property2'=>'value2',  
        ),  
        // outros controles  
    ),  
);
```

Dessa forma, uma ação a no controle pode ser acessada pela [rota](#) xyz/a.

Validador

Um validador é utilizado principalmente na classe de um [modelo](#) (que estenda de [CFormModel](#) ou [CActiveRecord](#)). Dada a classe de um validador chamada XyzClass, pertencente a extensão xyz, podemos utiliza-la sobrescrevendo o método [CModel::rules](#) na classe de nosso modelo:

```
class MyModel extends CActiveRecord // ou CFormModel  
{  
    public function rules()  
    {  
        return array(  
            array(  
                'attr1, attr2',  
                'ext.xyz.XyzClass',  
                'property1'=>'value1',  
                'property2'=>'value2',  
            ),  
            // outras regras de validação  
        );  
    }  
}
```

Comando de Console

Uma extensão do tipo [comando de console](#), normalmente é utilizada para adicionar comandos à ferramenta yiic. Dado um comando de console XyzClass, pertencente à extensão xyz, podemos utilizá-lo adicionando nas configurações da aplicação de console:

```
return array(  
    'commandMap'=>array(  
        'xyz'=>array(  
            'class'=>'ext.xyz.XyzClass',  
            'property1'=>'value1',  
            'property2'=>'value2',  
        ),  
        // outros comandos  
    ),  
);
```

Dessa forma, podemos utilizar o comando xyz na ferramenta yiic.

Nota: Uma aplicação de console normalmente utiliza um arquivo de configuração diferente do utilizado pela aplicação web. Se uma aplicação foi criada utilizando o comando yiic webapp, o arquivo de configurações para o console estará em `protected/config/console.php`, enquanto o arquivo de configuração para a aplicação web estará em `protected/config/main.php`.

Módulo

Para utilizar módulos, por favor, veja a seção sobre [módulos](#).

Componente Genérico

Para utilizar um [componente](#), primeiro precisamos incluir seu arquivo de classe, utilizando:

```
Yii::import('ext.xyz.XyzClass');
```

Feito isso, podemos criar uma instância dessa classe, configurar suas propriedades e chamar seus métodos. Podemos também estendê-lo para criar novas classes.

Criando Extensões

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.

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.

Behavior

To create a behavior, one must implement the `[IBehavior]` interface. For convenience, Yii provides a base class [CBehavior](#) that already implements this interface and provides some additional convenient methods. Child classes mainly need to implement the extra methods that they intend to make available to the components being attached to.

When developing behaviors for [CModel](#) and [CActiveRecord](#), one can also extend [CModelBehavior](#) and [CActiveRecordBehavior](#), respectively. These base classes offer additional features that are specifically made for [CModel](#) and [CActiveRecord](#). For example, the [CActiveRecordBehavior](#) class implements a set of methods to respond to the life cycle events raised in an ActiveRecord object. A child class can thus override these methods to put in customized code which will participate in the AR life cycles.

The following code shows an example of an ActiveRecord behavior. When this behavior is attached to an AR object and when the AR object is being saved by calling `save()`, it will automatically sets the `create_time` and `update_time` attributes with the current timestamp.

```

class TimestampBehavior extends CActiveRecordBehavior
{
    public function beforeSave($event)
    {
        if($this->owner->isNewRecord)
            $this->owner->create_time=time();
        else
            $this->owner->update_time=time();
    }
}

```

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.

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]`.

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.

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 theviews 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.

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);
    }
}

```

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';
    }
}
```

Module

Please refer to the section about [modules](#) on how to create a module.

A general guideline for developing a module is that it should be self-contained. Resource files (such as CSS, JavaScript, images) that are used by a module should be distributed together with the module. And the module should publish them so that they can be Web-accessible.

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.

Utilizando Bibliotecas de Terceiros

Yii is carefully designed so that third-party libraries can be easily integrated to further extend Yii's functionalities. When using third-party libraries in a project, developers often encounter issues about class naming and file inclusion. Because all Yii classes are prefixed with letter C, it is less likely class naming issue would occur; and because Yii relies on [SPL autoload](#) to perform class file inclusion, it can play nicely with other libraries if they use the same autoloading feature or PHP include path to include class files.

Below we use an example to illustrate how to use the [Zend Search Lucene](#) component from the [Zend framework](#) in a Yii application.

First, we extract the Zend framework release file to a directory under protected/vendors, assuming protected is the [application base directory](#). Verify that the file protected/vendors/Zend/Search/Lucene.php exists.

Second, at the beginning of a controller class file, insert the following lines:

```
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::importbefore 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));
```

Testes

Visão Geral

Note: The testing support described in this chapter requires Yii version 1.1 or higher. This does not mean, however, that you cannot test applications developed using Yii 1.0.x. There are many great testing frameworks available to help you accomplish this task, such as [PHPUnit](#), [SimpleTest](#).

Testing is an indispensable process of software development. Whether we are aware of it or not, we conduct testing all the time when we are developing a Web application. For example, when we write a class in PHP, we may use some echo or die statement to show that we implement a method correctly; when we implement a Web page containing a complex HTML form, we may try entering some test data to ensure the page interacts with us as expected. More advanced developers would write some code to automate this testing process so that each time when we need to test something, we just need to call up the code and let the computer to perform testing for us. This is known as automated testing, which is the main topic of this chapter.

The testing support provided by Yii includes unit testing and functional testing.

A unit test verifies that a single unit of code is working as expected. In object-oriented programming, the most basic code unit is a class. A unit test thus mainly needs to verify that each of the class interface methods works properly. That is, given different input parameters, the test verifies the method returns expected results. Unit tests are usually developed by people who write the classes being tested.

A functional test verifies that a feature (e.g. post management in a blog system) is working as expected. Compared with a unit test, a functional test sits at a higher level because a feature being tested often involves multiple classes. Functional tests are usually developed by people who know very well the system requirements (they could be either developers or quality engineers).

Test-Driven Development

Below we show the development cycles in the so-called [test-driven development \(TDD\)](#):

1. Create a new test that covers a feature to be implemented. The test is expected to fail at its first execution because the feature has yet to be implemented.
2. Run all tests and make sure the new test fails.
3. Write code to make the new test pass.
4. Run all tests and make sure they all pass.
5. Refactor the code that is newly written and make sure the tests still pass.

Repeat step 1 to 5 to push forward the functionality implementation.

Test Environment Setup

The testing supported provided by Yii requires PHPUnit 3.4+ and Selenium Remote Control 1.0+. Please refer to their documentation on how to install PHPUnit and Selenium Remote Control.

When we use the `yiic webapp` console command to create a new Yii application, it will generate the following files and directories for us to write and perform new tests:

```
testdrive/
  protected/           containing protected application files
  tests/               containing tests for the application
    fixtures/          containing database fixtures
    functional/         containing functional tests
    unit/              containing unit tests
    report/            containing coverage reports
  bootstrap.php        the script executed at the very beginning
  phpunit.xml          the PHPUnit configuration file
  WebTestCase.php      the base class for Web-based functional tests
```

As shown in the above, our test code will be mainly put into three directories: fixtures, functional and unit, and the directory report will be used to store the generated code coverage reports.

To execute tests (whether unit tests or functional tests), we can execute the following commands in a console window:

```
% cd testdrive/protected/tests
% phpunit functional/PostTest.php    // executes an individual test
% phpunit --verbose functional      // executes all tests under
'functional'
% phpunit --coverage-html ./report unit
```

In the above, the last command will execute all tests under the unit directory and generate a code-coverage report under the report directory. Note that [xdebug extension](#) must be installed and enabled in order to generate code-coverage reports.

Test Bootstrap Script

Let's take a look what may be in the bootstrap.php file. This file is so special because it is like the entry script and is the starting point when we execute a set of tests.

```
$yiit='path/to/yii/framework/yiit.php';
$config=dirname(__FILE__).'../config/test.php';
require_once($yiit);
require_once(dirname(__FILE__).'../WebTestCase.php');
Yii::createWebApplication($config);
```

In the above, we first include the yiit.php file from the Yii framework, which initializes some global constants and includes necessary test base classes. We then create a Web application instance using the test.php configuration file. If we check test.php, we shall find that it inherits from the main.php configuration file and adds a fixture application component whose class is [CDbFixtureManager](#). We will describe fixtures in detail in the next section.

```
return CMap::mergeArray(
    require(dirname(__FILE__).'../main.php'),
    array(
        'components'=>array(
            'fixture'=>array(
                'class'=>'system.test.CDbFixtureManager',
            ),
            /* uncomment the following to provide test database connection
            'db'=>array(
                'connectionString'=>'DSN for test database',
            ),
            */
        ),
    );
```

When we run tests that involve database, we should provide a test database so that the test execution does not interfere with normal development or production activities. To do so, we just need to uncomment the db configuration in the above and fill in the connectionString property with the DSN (data source name) to the test database.

With such a bootstrap script, when we run unit tests, we will have an application instance that is nearly the same as the one that serves for Web requests. The main difference is that it has the fixture manager and is using the test database.

Definindo Fixtures

Automated tests need to be executed many times. To ensure the testing process is repeatable, we would like to run the tests in some known state called fixture. For example, to test the post creation feature in a blog application, each time when we run the tests, the tables storing relevant data about posts (e.g. the Post table, the Comment table) should be

restored to some fixed state. The [PHPUnit documentation](#) has described well about generic fixture setup. In this section, we mainly describe how to set up database fixtures, as we just described in the example.

Setting up database fixtures is perhaps one of the most time-consuming parts in testing database-backed Web applications. Yii introduces the [CDbFixtureManager](#) application component to alleviate this problem. It basically does the following things when running a set of tests:

- Before all tests run, it resets all tables relevant to the tests to some known state.
- Before a single test method runs, it resets the specified tables to some known state.
- During the execution of a test method, it provides access to the rows of the data that contribute to the fixture.

To use [CDbFixtureManager](#), we configure it in the [application configuration](#) as follows,

```
return array(  
    'components'=>array(  
        'fixture'=>array(  
            'class'=>'system.test.CDbFixtureManager',  
        ),  
    ),  
);
```

We then provide the fixture data under the directory `protected/tests/fixtures`. This directory may be customized to be a different one by configuring the [CDbFixtureManager::basePath](#) property in the application configuration. The fixture data is organized as a collection of PHP files called fixture files. Each fixture file returns an array representing the initial rows of data for a particular table. The file name is the same as the table name. The following is an example of the fixture data for the Post table stored in a file named `Post.php`:


```
<?php
return array(
    'sample1'=>array(
        'title'=>'test post 1',
        'content'=>'test post content 1',
        'createTime'=>1230952187,
        'authorId'=>1,
    ),
    'sample2'=>array(
        'title'=>'test post 2',
        'content'=>'test post content 2',
        'createTime'=>1230952287,
        'authorId'=>1,
    ),
);
```

As we can see, two rows of data are returned in the above. Each row is represented as an associative array whose keys are column names and whose values are the corresponding column values. In addition, each row is indexed by a string (e.g. sample1, sample2) which is called row alias. Later when we write test scripts, we can conveniently refer to a row by its alias. We will describe this in detail in the next section.

You may notice that we do not specify the id column values in the above fixture. This is because the idcolumn is defined to be an auto-incremental primary key whose value will be filled up when we insert new rows.

When [CdbFixtureManager](#) is referenced for the first time, it will go through every fixture file and use it to reset the corresponding table. It resets a table by truncating the table, resetting the sequence value for the table's auto-incremental primary key, and then inserting the rows of data from the fixture file into the table.

Sometimes, we may not want to reset every table which has a fixture file before we run a set of tests, because resetting too many fixture files could take very long time. In this case, we can write a PHP script to do the initialization work in a customized way. The script should be saved in a file named init.php under the same directory that contains other fixture files. When [CdbFixtureManager](#) detects the existence of this script, it will execute this script instead of resetting every table.

It is also possible that we do not like the default way of resetting a table, i.e., truncating it and inserting it with the fixture data. If this is the case, we can write an initialization script for the specific fixture file. The script must be named as the table name suffixed with .init.php. For example, the initialization script for the Posttable would be Post.init.php.

When [CdbFixtureManager](#) sees this script, it will execute this script instead of using the default way to reset the table.

Tip: Having too many fixture files could increase the test time dramatically. For this reason, you should only provide fixture files for those tables whose content may change during the test. Tables that serve as look-ups do not change and thus do not need fixture files.

In the next two sections, we will describe how to make use of the fixtures managed by [CDbFixtureManager](#) in unit tests and functional tests.

Testes Unitários

Because the Yii testing framework is built on top of [PHPUnit](#), it is recommended that you go through the [PHPUnit documentation](#) first to get the basic understanding on how to write a unit test. We summarize in the following the basic principles of writing a unit test in Yii:

- A unit test is written in terms of a class `XyzTest` which extends from [CTestCase](#) or [CDbTestCase](#), where `Xyz` stands for the class being tested. For example, to test the `Post` class, we would name the corresponding unit test as `PostTest` by convention. The base class [CTestCase](#) is meant for generic unit tests, while [CDbTestCase](#) is suitable for testing [active record](#) model classes. Because `PHPUnit_Framework_TestCase` is the ancestor class for both classes, we can use all methods inherited from this class.
- The unit test class is saved in a PHP file named as `XyzTest.php`. By convention, the unit test file may be stored under the directory `protected/tests/unit`.
- The test class mainly contains a set of test methods named as `testAbc`, where `Abc` is often the name of the class method to be tested.
- A test method usually contains a sequence of assertion statements (e.g. `assertTrue`, `assertEquals`) which serve as checkpoints on validating the behavior of the target class.

In the following, we mainly describe how to write unit tests for [active record](#) model classes. We will extend our test classes from [CDbTestCase](#) because it provides the database fixture support that we introduced in the previous section.

Assume we want to test the `Comment` model class in the [blog demo](#). We start by creating a class named `CommentTest` and saving it as `protected/tests/unit/CommentTest.php`:

```
class CommentTest extends CDbTestCase
{
    public $fixtures=array(
        'posts'=>'Post',
        'comments'=>'Comment',
    );

    .....
}
```

In this class, we specify the fixtures member variable to be an array that specifies which fixtures will be used by this test. The array represents a mapping from fixture names to model class names or fixture table names (e.g. from fixture name posts to model class Post). Note that when mapping to fixture table names, we should prefix the table name with a colon (e.g. :Post) to differentiate it from model class name. And when using model class names, the corresponding tables will be considered as fixture tables. As we described earlier, fixture tables will be reset to some known state each time when a test method is executed.

Fixture names allow us to access the fixture data in test methods in a convenient way. The following code shows its typical usage:

```
// return all rows in the 'Comment' fixture table  
$comments = $this->comments;  
// return the row whose alias is 'sample1' in the 'Post' fixture table  
$post = $this->posts['sample1'];  
// return the AR instance representing the 'sample1' fixture data row  
$post = $this->posts('sample1');
```

Note: If a fixture is declared using its table name (e.g. 'posts'=>':Post'), then the third usage in the above is not valid because we have no information about which model class the table is associated with.

Next, we write the testApprove method to test the approve method in the Comment model class. The code is very straightforward: we first insert a comment that is pending status; we then verify this comment is in pending status by retrieving it from database; and finally we call the approve method and verify the status is changed as expected.

```

public function testApprove()
{
    // insert a comment in pending status
    $comment=new Comment;
    $comment->setAttributes(array(
        'content'=>'comment 1',
        'status'=>Comment::STATUS_PENDING,
        'createTime'=>time(),
        'author'=>'me',
        'email'=>'me@example.com',
        'postId'=>$this->posts['sample1']['id'],
    ),false);
    $this->assertTrue($comment->save(false));

    // verify the comment is in pending status
    $comment=Comment::model()->findByPk($comment->id);
    $this->assertTrue($comment instanceof Comment);
    $this->assertEquals(Comment::STATUS_PENDING,$comment->status);

    // call approve() and verify the comment is in approved status
    $comment->approve();
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
    $comment=Comment::model()->findByPk($comment->id);
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
}

```

Testes Funcionais

Before reading this section, it is recommended that you read the [Selenium documentation](#) and the [PHPUnit documentation](#) first. We summarize in the following the basic principles of writing a functional test in Yii:

- Like unit test, a functional test is written in terms of a class `XYZTest` which extends from [CWebTestCase](#), where `XYZ` stands for the class being tested. Because `PHPUnit_Extensions_SeleniumTestCase` is the ancestor class for [CWebTestCase](#), we can use all methods inherited from this class.
- The functional test class is saved in a PHP file named as `XYZTest.php`. By convention, the functional test file may be stored under the directory `protected/tests/functional`.
- The test class mainly contains a set of test methods named as `testAbc`, where `Abc` is often the name of a feature to be tested. For example, to test the user login feature, we can have a test method named `testLogin`.

- A test method usually contains a sequence of statements that would issue commands to Selenium RC to interact with the Web application being tested. It also contains assertion statements to verify that the Web application responds as expected.

Before we describe how to write a functional test, let's take a look at the WebTestCase.php file generated by the yiic webapp command. This file defines WebTestCase that may serve as the base class for all functional test classes.

```
define('TEST_BASE_URL', 'http://localhost/yii/demos/blog/index-test.php/');

class WebTestCase extends CWebTestCase
{
    /**
     * Sets up before each test method runs.
     * This mainly sets the base URL for the test application.
     */
    protected function setUp()
    {
        parent::setUp();
        $this->setBrowserUrl(TEST_BASE_URL);
    }

    .....
}
```

The class WebTestCase mainly sets the base URL of the pages to be tested. Later in test methods, we can use relative URLs to specify which pages to be tested.

We should also pay attention that in the base test URL, we use index-test.php as the entry script instead of index.php. The only difference between index-test.php and index.php is that the former uses test.php as the application configuration file while the latter main.php.

We now describe how to test the feature about showing a post in the [blog demo](#). We first write the test class as follows, noting that the test class extends from the base class we just described:

```

class PostTest extends WebTestCase
{
    public $fixtures=array(
        'posts'=>'Post',
    );

    public function testShow()
    {
        $this->open('post/1');
        // verify the sample post title exists
        $this->assertTextPresent($this->posts['sample1']['title']);
        // verify comment form exists
        $this->assertTextPresent('Leave a Comment');
    }

    .....
}

```

Like writing a unit test class, we declare the fixtures to be used by this test. Here we indicate that the Postfixture should be used. In the testShow test method, we first instruct Selenium RC to open the URL post/1. Note that this is a relative URL, and the complete URL is formed by appending it to the base URL we set in the base class (i.e. <http://localhost/yii/demos/blog/index-test.php/post/1>). We then verify that we can find the title of the sample1 post can be found in the current Web page. And we also verify that the page contains the text Leave a Comment.

Tip: Before running functional tests, the Selenium-RC server must be started. This can be done by executing the command `java -jar selenium-server.jar` under your Selenium server installation directory.

Tópicos Especiais

Gerenciamento de URL

Complete URL management for a Web application involves two aspects. First, when a user request comes in terms of a URL, the application needs to parse it into understandable parameters. Second, the application needs to provide a way of creating URLs so that the created URLs can be understood by the application. For a Yii application, these are accomplished with the help of [CUrlManager](#).

Creating URLs

Although URLs can be hardcoded in controller views, it is often more flexible to create them dynamically:

```
$url=$this->createUrl($route,$params);
```

Where `$this` refers to the controller instance; `$route` specifies the [route](#) of the request; and `$params` is a list of GET parameters to be appended to the URL.

By default, URLs created by [createUrl](#) is in the so-called get format. For example, given `$route='post/read'` and `$params=array('id'=>100)`, we would obtain the following URL:

```
/index.php?r=post/read&id=100
```

Where parameters appear in the query string as a list of Name=Value concatenated with the ampersand characters, and the `r` parameter specifies the request [route](#). This URL format is not very user-friendly because it requires several non-word characters.

We could make the above URL look cleaner and more self-explanatory by using the so-called path format which eliminates the query string and puts the GET parameters into the path info part of URL:

```
/index.php/post/read/id/100
```

To change the URL format, we should configure the [urlManager](#) application component so that [createUrl](#) can automatically switch to the new format and the application can properly understand the new URLs:

```

array(
    .....
    'components'=>array(
        .....
        'urlManager'=>array(
            'urlFormat'=>'path',
        ),
    ),
);

```

Note that we do not need to specify the class of the urlManager component because it is pre-declared as CUrlManager in CWebApplication.

Tip: The URL generated by the createUrl method is a relative one. In order to get an absolute URL, we can prefix it with `Yii::app()->request->hostInfo`, or call `createAbsoluteUrl`.

User-friendly URLs

When path is used as the URL format, we can specify some URL rules to make our URLs even more user-friendly. For example, we can generate a URL as short as `/post/100`, instead of the lengthy `/index.php/post/read/id/100`. URL rules are used by CUrlManager for both URL creation and parsing purposes.

To specify URL rules, we need to configure the rules property of the urlManager application component:

```

array(
    .....
    'components'=>array(
        .....
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
                'pattern1'=>'route1',
                'pattern2'=>'route2',
                'pattern3'=>'route3',
            ),
        ),
    ),
);

```


The rules are specified as an array of pattern-route pairs, each corresponding to a single rule. The pattern of a rule is a string used to match the path info part of URLs. And the route of a rule should refer to a valid controller [route](#).

Besides the above pattern-route format, a rule may also be specified with customized options, like the following:

```
'pattern1'=>array('route1', 'urlSuffix'=>'.xml', 'caseSensitive'=>false)
```

In the above, the array contains a list of customized options. As of version 1.1.0, the following options are available:

- [urlSuffix](#): the URL suffix used specifically for this rule. Defaults to null, meaning using the value of [CUrlManager::urlSuffix](#).
- [caseSensitive](#): whether this rule is case sensitive. Defaults to null, meaning using the value of [CUrlManager::caseSensitive](#).
- [defaultParams](#): the default GET parameters (name=>value) that this rule provides. When this rule is used to parse the incoming request, the values declared in this property will be injected into \$_GET.
- [matchValue](#): whether the GET parameter values should match the corresponding sub-patterns in the rule when creating a URL. Defaults to null, meaning using the value of [CUrlManager::matchValue](#). If this property is false, it means a rule will be used for creating a URL if its route and parameter names match the given ones. If this property is set true, then the given parameter values must also match the corresponding parameter sub-patterns. Note that setting this property to true will degrade performance.

Using Named Parameters

A rule can be associated with a few GET parameters. These GET parameters appear in the rule's pattern as special tokens in the following format:

```
&lt;ParamName:ParamPattern>;
```

Where ParamName specifies the name of a GET parameter, and the optional ParamPattern specifies the regular expression that should be used to match the value of the GET parameter. In case when ParamPattern is omitted, it means the parameter should match any characters except the slash /. When creating a URL, these parameter tokens will be replaced with the corresponding parameter values; when parsing a URL, the corresponding GET parameters will be populated with the parsed results.

Let's use some examples to explain how URL rules work. We assume that our rule set consists of three rules:

```
array(  
    'posts'=>'post/list',  
    'post/<id:\d+>'=>'post/read',  
    'post/<year:\d{4}>/<title>'=>'post/read',  
)
```

- Calling `$this->createUrl('post/list')` generates `/index.php/posts`. The first rule is applied.
- Calling `$this->createUrl('post/read',array('id'=>100))` generates `/index.php/post/100`. The second rule is applied.
- Calling `$this->createUrl('post/read',array('year'=>2008,'title'=>'a sample post'))` generates `/index.php/post/2008/a%20sample%20post`. The third rule is applied.
- Calling `$this->createUrl('post/read')` generates `/index.php/post/read`. None of the rules is applied.

In summary, when using [createUrl](#) to generate a URL, the route and the GET parameters passed to the method are used to decide which URL rule to be applied. If every parameter associated with a rule can be found in the GET parameters passed to [createUrl](#), and if the route of the rule also matches the route parameter, the rule will be used to generate the URL.

If the GET parameters passed to [createUrl](#) are more than those required by a rule, the additional parameters will appear in the query string. For example, if we call `$this->createUrl('post/read',array('id'=>100,'year'=>2008))`, we would obtain `/index.php/post/100?year=2008`. In order to make these additional parameters appear in the path info part, we should append `/*` to the rule. Therefore, with the rule `post/<id:\d+>/*`, we can obtain the URL as `/index.php/post/100/year/2008`.

As we mentioned, the other purpose of URL rules is to parse the requesting URLs. Naturally, this is an inverse process of URL creation. For example, when a user requests for `/index.php/post/100`, the second rule in the above example will apply, which resolves in the route `post/read` and the GET parameter `array('id'=>100)` (accessible via `$_GET`).

Note: Using URL rules will degrade application performance. This is because when parsing the request URL, [CUrlManager](#) will attempt to match it with each rule until one can be applied. The more the number of rules, the more the performance impact. Therefore, a high-traffic Web application should minimize its use of URL rules.

Parameterizing Routes

Starting from version 1.0.5, we may reference named parameters in the route part of a rule. This allows a rule to be applied to multiple routes based on matching criteria. It may also help reduce the number of rules needed for an application, and thus improve the overall performance.

We use the following example rules to illustrate how to parameterize routes with named parameters:

```
array(
    '<_c:(post|comment)>/<id:\d+>/<_a:(create|update|delete)>' => '<_c>/<_a>',
    '<_c:(post|comment)>/<id:\d+>' => '<_c>/read',
    '<_c:(post|comment)>s' => '<_c>/list',
)
```

In the above, we use two named parameters in the route part of the rules: `_c` and `_a`. The former matches a controller ID to be either `post` or `comment`, while the latter matches an action ID to be `create`, `update` or `delete`. You may name the parameters differently as long as they do not conflict with GET parameters that may appear in URLs.

Using the aboving rules, the URL `/index.php/post/123/create` would be parsed as the route `post/create` with GET parameter `id=123`. And given the route `comment/list` and GET parameter `page=2`, we can create a URL `/index.php/comments?page=2`.

Parameterizing Hostnames

Starting from version 1.0.11, it is also possible to include hostname into the rules for parsing and creating URLs. One may extract part of the hostname to be a GET parameter. For example, the URL `http://admin.example.com/en/profile` may be parsed into GET parameters `user=admin` and `lang=en`. On the other hand, rules with hostname may also be used to create URLs with paratermized hostnames.

In order to use parameterized hostnames, simply declare URL rules with host info, e.g.:

```
array(
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
)
```

The above example says that the first segment in the hostname should be treated as `user` parameter while the first segment in the path info should be `lang` parameter. The rule corresponds to the `user/profile` route.

Note that [CUrlManager::showScriptName](#) will not take effect when a URL is being created using a rule with parameterized hostname.

Also note that the rule with parameterized hostname should NOT contain the sub-folder if the application is under a sub-folder of the Web root. For example, if the application is under `http://www.example.com/sandbox/blog`, then we should still use the same URL rule as described above without the sub-folder `sandbox/blog`.

Hiding index.php

There is one more thing that we can do to further clean our URLs, i.e., hiding the entry script `index.php` in the URL. This requires us to configure the Web server as well as the [urlManager](#) application component.

We first need to configure the Web server so that a URL without the entry script can still be handled by the entry script. For [Apache HTTP server](#), this can be done by turning on the URL rewriting engine and specifying some rewriting rules. We can create the file `/wwwroot/blog/.htaccess` with the following content. Note that the same content can also be put in the Apache configuration file within the Directory element for `/wwwroot/blog`.

```
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.

Autenticação e Autorização

Authentication and authorization are required for a Web page that should be limited to certain users. Authentication is about verifying whether someone is who they claim to be. 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 (i.e. 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.

Defining Identity Class

As mentioned above, authentication is about validating the identity of the user. A typical Web application authentication implementation usually involves using a username and password combination to verify a user's identity. However, it may include other methods and different implementations may be required. To accommodate varying authentication methods, the Yii auth framework introduces the identity class.

We define an identity class which contains the actual authentication logic. The identity class should implement the `[IUserIdentity]` interface. Different identity classes can be implemented for different authentication approaches (e.g. OpenID, LDAP, Twitter OAuth, Facebook Connect). A good start when writing your own implementation is to extend [CUserIdentity](#) which is a base class for the authentication approach using a username and password.

The main work in defining an identity class is the implementation of the `[IUserIdentity::authenticate]` method. This is the method used to encapsulate the main details of the authentication approach. An identity class may also declare additional identity information that needs to be persistent during the user session.

An Example

In the following example, we use an identity class to demonstrate using a database approach to authentication. This is very typical of most Web applications. A user will enter their username and password into a login form, and then we validate these credentials, using [ActiveRecord](#), against a user table in the database . There are actually a few things being demonstrated in this single example:

- The implementation of the `authenticate()` to use the database to validate credentials.
- Overriding the `CUserIdentity::getId()` method to return the `_id` property because the default implementation returns the username as the ID.
- Using the `setState()` ([CBaseUserIdentity::setState](#)) method to demonstrate storing other information that can easily be retrieved upon subsequent requests.

```

class UserIdentity extends CUserIdentity
{
    private $_id;

    public function authenticate()
    {
        $record=User::model()->findByAttributes(array(
            'username'=>$this->username
        ));
        if($record===null)
            $this->errorCode=self::ERROR_USERNAME_INVALID;
        else if($record->password!==md5($this->password))
            $this->errorCode=self::ERROR_PASSWORD_INVALID;
        else
        {
            $this->_id=$record->id;
            $this->setState('title', $record->title);
            $this->errorCode=self::ERROR_NONE;
        }
        return !$this->errorCode;
    }

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

```

When we cover login and logout in the next section, we'll see that we pass this identity class into the login method for a user. Any information that we store in a state (by calling [CBaseUserIdentity::setState](#)) will be passed to [CWebUser](#), which in turn will store them in a persistent storage, such as session. This information can then be accessed like properties of [CWebUser](#). In our example, we stored the user title information via `$this->setState('title', $record->title);`. Once we complete our login process, we can obtain the title information of the current user by simply using `Yii::app()->user->title` (This has been available since version 1.0.3. In prior versions, one must instead use `Yii::app()->user->getState('title');`.)

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.

Login and Logout

Now that we have seen an example of creating a user identity, we use this to help ease the implementation of our needed login and logout actions. The following code demonstrates how this is accomplished:

```
// 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();
```

Here we are creating a new `UserIdentity` object and passing in the authentication credentials (i.e. the `$username` and `$password` values submitted by the user) to its constructor. We then simply call the `authenticate()` method. If successful, we pass the identity information into the `CWebUser::login` method, which will store the identity information into persistent storage (PHP session by default) for retrieval upon subsequent requests. If the authentication fails, we can interrogate the `errorMessage` property for more information as to why it failed.

Whether or not a user has been authenticated can easily be checked throughout the application by using `Yii::app()->user->isGuest`. If using persistent storage like session (the default) and/or a cookie (discussed below) to store the identity information, the user can remain logged in upon subsequent requests. In this case, we don't need to use the `UserIdentity` class and the entire login process upon each request. Rather `CWebUser` will automatically take care of loading the identity information from this persistent storage and will use it to determine whether `Yii::app()->user->isGuest` returns true or false.

Cookie-based Login

By default, a user will be logged out after a certain period of inactivity, depending on the session configuration. To change this behavior, we can set the `allowAutoLogin` property of the user component to be true and pass a duration parameter to the `CWebUser::login` method. The user will then remain logged in for the specified duration, even if he closes his browser window. Note that this feature requires the user's browser to accept cookies.

```
// Keep the user logged in for 7 days.
// Make sure allowAutoLogin is set true for the user component.
Yii::app()->user->login($identity,3600*24*7);
```

As we mentioned above, when cookie-based login is enabled, the states stored via `CBaseUserIdentity::setState` will be saved in the cookie as well. The next time when the user is logged in, these states will be read from the cookie and made accessible via `Yii::app()->user`.

Although Yii has measures to prevent the state cookie from being tampered on the client side, we strongly suggest that security sensitive information be not stored as states. Instead, these information should be restored on the server side by reading from some persistent storage on the server side (e.g. database).

In addition, for any serious Web applications, we recommend using the following strategy to enhance the security of cookie-based login.

When a user successfully logs in by filling out a login form, we generate and store a random key in both the cookie state and in persistent storage on server side (e.g. database).

Upon a subsequent request, when the user authentication is being done via the cookie information, we compare the two copies of this random key and ensure a match before logging in the user.

If the user logs in via the login form again, the key needs to be re-generated.

By using the above strategy, we eliminate the possibility that a user may re-use an old state cookie which may contain outdated state information.

To implement the above strategy, we need to override the following two methods:

- `CUserIdentity::authenticate()`: this is where the real authentication is performed. If the user is authenticated, we should re-generate a new random key, and store it in the database as well as in the identity states via `CBaseUserIdentity::setState`.
- `CWebUser::beforeLogin()`: this is called when a user is being logged in. We should check if the key obtained from the state cookie is the same as the one from the database.

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 more complex access control you may use role-based access (RBAC), which we will cover in the next subsection.

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 other name-value pairs specify the pattern parameters of the rule. The rules defined above are interpreted as follows: the create and edit actions cannot be executed by anonymous users; the delete action can be executed by users with admin role; and the delete action cannot be executed by anyone.

The access rules are evaluated one by one in the order they are specified. The first rule that matches the current pattern (e.g. username, roles, client IP, address) determines the authorization result. If this rule is an allow rule, the action can be executed; if it is a deny rule, the action cannot be executed; if none of the rules matches the context, the action can still be executed.

Tip: To ensure an action does not get executed under certain contexts, it is beneficial to always specify a matching-all deny rule at the end of rule set, like the following:

```
return array(  
    // ... other rules...  
    // the following rule denies 'delete' action for all contexts  
    array('deny',  
        'actions'=>array('delete'),  
    ),  
);
```

The reason for this rule is because if none of the rules matches a context, then the action will continue to be executed.

An access rule can match the following context parameters:

- [actions](#): specifies which actions this rule matches. This should be an array of action IDs. The comparison is case-insensitive.
- [controllers](#): specifies which controllers this rule matches. This should be an array of controller IDs. The comparison is case-insensitive. This option has been available since version 1.0.4.
- [users](#): specifies which users this rule matches. The current user's [name](#) is used for matching. The comparison is case-insensitive. Three special characters can be used here:
 - *: any user, including both anonymous and authenticated users.
 - ?: anonymous users.
 - @: authenticated users.
- [roles](#): specifies which roles that this rule matches. This makes use of the [role-based access control](#) feature to be described in the next subsection. In particular, the rule is applied if [CWebUser::checkAccess](#) returns true for one of the roles. Note, you should mainly use roles in an allowrule because by definition, a role represents a permission to do something. Also note, although we use the term roles here, its value can actually be any auth item, including roles, tasks and operations.
- [ips](#): specifies which client IP addresses this rule matches.
- [verbs](#): specifies which request types (e.g. GET, POST) this rule matches. The comparison is case-insensitive.
- [expression](#): specifies a PHP expression whose value indicates whether this rule matches. In the expression, you can use variable `$user` which refers to `Yii::app()->user`. This option has been available since version 1.0.3.

Handling Authorization Result

When authorization fails, i.e., the user is not allowed to perform the specified action, one of the following two scenarios may happen:

If the user is not logged in and if the [loginUrl](#) property of the user component is configured to be the URL of the login page, the browser will be redirected to that page. Note that by default, [loginUrl](#) points to the site/login page.

Otherwise an HTTP exception will be displayed with error code 403.

When configuring the [loginUrl](#) property, one can provide a relative or absolute URL. One can also provide an array which will be used to generate a URL by calling [CWebApplication::createUrl](#). The first array element should specify the [route](#) to the login controller action, and the rest name-value pairs are GET parameters. For example,

```
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);
```

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');

```

Once we have established this hierarchy, the authManager component (e.g. CPhpAuthManager, CDbAuthManager) will load the authorization items automatically. Therefore, we only need to run the above code one time, and NOT for every request.

Info: While the above example looks long and tedious, it is mainly for demonstrative purpose. Developers will usually need to develop some administrative user interfaces so that end users can use to establish an authorization hierarchy more intuitively.

Using Business Rules

When we are defining the authorization hierarchy, we can associate a role, a task or an operation with a so-called business rule. We may also associate a business rule when we assign a role to a user. A business rule is a piece of PHP code that is executed when we perform access checking. The returning value of the code is used to determine if the role or assignment applies to the current user. In the example above, we associated a business rule with the `updateOwnPost` task. In the business rule we simply check if the current user ID is the same as the specified post's author ID. The post information in the `$params` array is supplied by developers when performing access checking.

Access Checking

To perform access checking, we first need to know the name of the authorization item. For example, to check if the current user can create a post, we would check if he has the permission represented by the `createPost` operation. We then call [`CWebUser::checkAccess`](#) to perform the access checking:

```
if(Yii::app()->user->checkAccess('createPost'))
{
    // create post
}
```

If the authorization rule is associated with a business rule which requires additional parameters, we can pass them as well. For example, to check if a user can update a post, we would pass in the post data in the `$params`:

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

Using Default Roles

Note: The default role feature has been available since version 1.0.3

Many Web applications need some very special roles that would be assigned to every or most of the system users. For example, we may want to assign some privileges to all authenticated users. It poses a lot of maintenance trouble if we explicitly specify and store these role assignments. We can exploit default roles to solve this problem.

A default role is a role that is implicitly assigned to every user, including both authenticated and guest. We do not need to explicitly assign it to a user. When [`CWebUser::checkAccess`](#) is invoked, default roles will be checked first as if they are assigned to the user.

Default roles must be declared in the [CAuthManager::defaultRoles](#) property. For example, the following configuration declares two roles to be default roles: authenticated and guest.

```
return array(
    'components'=>array(
        'authManager'=>array(
            'class'=>'CDBAuthManager',
            'defaultRoles'=>array('authenticated', 'guest'),
        ),
    ),
);
```

Because a default role is assigned to every user, it usually needs to be associated with a business rule that determines whether the role really applies to the user. For example, the following code defines two roles, authenticated and guest, which effectively apply to authenticated users and guest users, respectively.

```
$bizRule='return !Yii::app()->user->isGuest;';
$auth->createRole('authenticated', 'authenticated user', $bizRule);

$bizRule='return Yii::app()->user->isGuest;';
$auth->createRole('guest', 'guest user', $bizRule);
```

Temas

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.

Using a Theme

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.

Creating a Theme

Contents under a theme directory should be organized in the same way as those under the [application base path](#). For example, all view files must be located under views, layout view files under views/layouts, and system view files under views/system. For example, if we want to replace the create view of `PostController` with a view in the classic theme, we should save the new view file as `WebRoot/themes/classic/views/post/create.php`.

For views belonging to controllers in a [module](#), the corresponding themed view files should also be placed under the views directory. For example, if the aforementioned `PostController` is in a module named `forum`, we should save the create view file as `WebRoot/themes/classic/views/forum/post/create.php`. If the `forum` module is nested in another module named `support`, then the view file should be `WebRoot/themes/classic/views/support/forum/post/create.php`.

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

When we call [render](#) or [renderPartial](#) to display a view, the corresponding view file as well as the layout file will be looked for in the currently active theme. And if found, those files will be rendered. Otherwise, it falls back to the default location as specified by [viewPath](#) and [layoutPath](#).

Tip: Inside a theme view, we often need to link other theme resource files. For example, we may want to show an image file under the theme's images directory. Using the `baseUrl` property of the currently active theme, we can generate the URL for the image as follows,

```
Yii::app()->theme->baseUrl . '/images/FileName.gif'
```

Below is an example of directory organization for an application with two themes `basic` and `fancy`.

```
WebRoot/
  assets
  protected/
    .htaccess
    components/
    controllers/
    models/
    views/
      layouts/
        main.php
      site/
        index.php
  themes/
    basic/
      views/
        .htaccess
        layouts/
          main.php
        site/
          index.php
    fancy/
      views/
        .htaccess
        layouts/
          main.php
        site/
          index.php
```

In the application configuration, if we configure

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

then the basic theme will be in effect, which means the application's layout will use the one under the directory `themes/basic/views/layouts`, and the site's index view will use the one under `themes/basic/views/site`. In case a view file is not found in the theme, it will fall back to the one under the `protected/views` directory.

Theming Widgets

Starting from version 1.1.5, views used by a widget can also be themed. In particular, when we call [CWidget::render\(\)](#) to render a widget view, Yii will attempt to search under the theme folder as well as the widget view folder for the desired view file.

To theme the view `xyz` for a widget whose class name is `Foo`, we should first create a folder named `Foo` (same as the widget class name) under the currently active theme's view folder. If the widget class is namespaced (available in PHP 5.3.0 or above), such as `\app\widgets\Foo`, we should create a folder named `app_widgets_Foo`. That is, we replace the namespace separators with the underscore characters.

We then create a view file named `xyz.php` under the newly created folder. To this end, we should have a file `themes/basic/views/Foo/xyz.php`, which will be used by the widget to replace its original view, if the currently active theme is basic.

Customizing Widgets Globally

Note: this feature has been available since version 1.1.3.

When using a widget provided by third party or Yii, we often need to customize it for specific needs. For example, we may want to change the value of [CLinkPager::maxButtonCount](#) from 10 (default) to 5. We can accomplish this by passing the initial property values when calling [CBaseController::widget](#) to create a widget. However, it becomes troublesome to do so if we have to repeat the same customization in every place we use [CLinkPager](#).

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>5,
    'cssFile'=>false,
));
```

Using the global widget customization feature, we only need to specify these initial values in a single place, i.e., the application configuration. This makes the customization of widgets more manageable.

To use the global widget customization feature, we need to configure the [widgetFactory](#) as follows:

```

return array(
    'components'=>array(
        'widgetFactory'=>array(
            'widgets'=>array(
                'CLinkPager'=>array(
                    'maxButtonCount'=>5,
                    'cssFile'=>false,
                ),
                'CJuiDatePicker'=>array(
                    'language'=>'ru',
                ),
            ),
        ),
    ),
);

```

In the above, we specify the global widget customization for both [CLinkPager](#) and [CJuiDatePicker](#) widgets by configuring the [CWidgetFactory::widgets](#) property. Note that the global customization for each widget is represented as a key-value pair in the array, where the key refers to the widget class name while the value specifies the initial property value array.

Now, whenever we create a [CLinkPager](#) widget in a view, the above property values will be assigned to the widget, and we only need to write the following code in the view to create the widget:

```

$this->widget('CLinkPager', array(
    'pages'=>$pagination,
));

```

We can still override the initial property values when necessary. For example, if in some view we want to set `maxButtonCount` to be 2, we can do the following:

```

$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>2,
));

```

Skin

Note: this feature has been available since version 1.1.0.

While using a theme we can quickly change the outlook of views, we can use skins to systematically customize the outlook of the [widgets](#) used in the views.

A skin is an array of name-value pairs that can be used to initialize the properties of a widget. A skin belongs to a widget class, and a widget class can have multiple skins identified by their names. For example, we can have a skin for the [CLinkPager](#) widget and the skin is named as classic.

In order to use the skin feature, we first need to modify the application configuration by configuring the [CWidgetFactory::enableSkin](#) property to be true for the widgetFactory application component:

```
return array(  
    'components'=>array(  
        'widgetFactory'=>array(  
            'enableSkin'=>true,  
        ),  
    ),  
);
```

Please note that in versions prior to 1.1.3, you need to use the following configuration to enable widget skinning:

```
return array(  
    'components'=>array(  
        'widgetFactory'=>array(  
            'class'=>'CWidgetFactory',  
        ),  
    ),  
);
```

We then create the needed skins. Skins belonging to the same widget class are stored in a single PHP script file whose name is the widget class name. All these skin files are stored under protected/views/skins, by default. If you want to change this to be a different directory, you may configure the skinPath property of the widgetFactory component. As an example, we may create under protected/views/skins a file named CLinkPager.php whose content is as follows,

```

<?php
return array(
    'default'=>array(
        'nextPageLabel'=>'&gt;&gt; ',
        'prevPageLabel'=>'&lt;&lt; ',
    ),
    'classic'=>array(
        'header'=>' ',
        'maxButtonCount'=>5,
    ),
);

```

In the above, we create two skins for the [CLinkPager](#) widget: default and classic. The former is the skin that will be applied to any [CLinkPager](#) widget that we do not explicitly specify its skin property, while the latter is the skin to be applied to a [CLinkPager](#) widget whose skin property is specified as classic. For example, in the following view code, the first pager will use the default skin while the second the classic skin:

```

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

<?php $this->widget('CLinkPager', array('skin'=>'classic')); ?>

```

If we create a widget with a set of initial property values, they will take precedence and be merged with any applicable skin. For example, the following view code will create a pager whose initial values will be `array('header'=>'', 'maxButtonCount'=>6, 'cssFile'=>false)`, which is the result of merging the initial property values specified in the view and the classic skin.

```

<?php $this->widget('CLinkPager', array(
    'skin'=>'classic',
    'maxButtonCount'=>6,
    'cssFile'=>false,
)); ?>

```

Note that the skin feature does NOT require using themes. However, when a theme is active, Yii will also look for skins under the skins directory of the theme's view directory (e.g. `WebRoot/themes/classic/views/skins`). In case a skin with the same name exists in both the theme and the main application view directories, the theme skin will take precedence.

If a widget is using a skin that does not exist, Yii will still create the widget as usual without any error.

Info: Using skin may degrade the performance because Yii needs to look for the skin file the first time a widget is being created.

Skin is very similar to the global widget customization feature. The main differences are as follows.

- Skin is more related with the customization of presentational property values;
- A widget can have multiple skins;
- Skin is themeable;
- Using skin is more expensive than using global widget customization.

Registros (Logs)

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.

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.

Message Routing

Messages logged using [Yii::log](#) or [Yii::trace](#) are kept in memory. We usually need to display them in browser windows, or save them in some persistent storage such as files, emails. This is called message routing, i.e., sending messages to different destinations. In Yii, message routing is managed by a [CLogRouter](#) application component. It manages a set of the so-called log routes. Each log route represents a single log destination.

Messages sent along a log route can be filtered according to their levels and categories.

To use message routing, we need to install and preload a [CLogRouter](#) application component. We also need to configure its [routes](#) property with the log routes that we want.

The following shows an example of the needed [application configuration](#):

```
array(  
    .....  
    'preload'=>array('log'),  
    'components'=>array(  
        .....  
        'log'=>array(  
            'class'=>'CLogRouter',  
            'routes'=>array(  
                array(  
                    'class'=>'CFileLogRoute',  
                    'levels'=>'trace, info',  
                    'categories'=>'system.*',  
                ),  
                array(  
                    'class'=>'CEmailLogRoute',  
                    'levels'=>'error, warning',  
                    'emails'=>'admin@example.com',  
                ),  
            ),  
        ),  
    ),  
)
```

In the above example, we have two log routes. The first route is [CFileLogRoute](#) which saves messages in a file under the application runtime directory. Only messages whose level is trace or info and whose category starts with system. are saved. The second route is [CEmailLogRoute](#) which sends messages to the specified email addresses. Only messages whose level is error or warning are sent.

The following log routes are available in Yii:

- [CDbLogRoute](#): saves messages in a database table.
- [CEmailLogRoute](#): sends messages to specified email addresses.
- [CFileLogRoute](#): saves messages in a file under the application runtime directory.
- [CWebLogRoute](#): displays messages at the end of the current Web page.
- [CProfileLogRoute](#): displays profiling messages at the end of the current Web page.

Info: Message routing occurs at the end of the current request cycle when the `onEndRequest` event is raised. To explicitly terminate the processing of the current request, call `CApplication::end()` instead of `die()` or `exit()`, because `CApplication::end()` will raise the `onEndRequest` event so that the messages can be properly logged.

Message Filtering

As we mentioned, messages can be filtered according to their levels and categories before they are sent along a log route. This is done by setting the [levels](#) and [categories](#) properties of the corresponding log route. Multiple levels or categories should be concatenated by commas.

Because message categories are in the format of `xxx.yyy.zzz`, we may treat them as a category hierarchy. In particular, we say `xxx` is the parent of `xxx.yyy` which is the parent of `xxx.yyy.zzz`. We can then use `xxx.*` to represent category `xxx` and all its child and grandchild categories.

Logging Context Information

Starting from version 1.0.6, we can specify to log additional context information, such as PHP predefined variables (e.g. `$_GET`, `$_SERVER`), session ID, user name, etc. This is accomplished by specifying the [CLogRoute::filter](#) property of a log route to be a suitable log filter.

The framework comes with the convenient [CLogFilter](#) that may be used as the needed log filter in most cases. By default, [CLogFilter](#) will log a message with variables like `$_GET`, `$_SERVER` which often contains valuable system context information. [CLogFilter](#) can also be configured to prefix each logged message with session ID, username, etc., which may greatly simplify the global search when we are checking the numerous logged messages.

The following configuration shows how to enable logging context information. Note that each log route may have its own log filter. And by default, a log route does not have a log filter.

```

array(
    .....
    'preload'=>array('log'),
    'components'=>array(
        .....
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error',
                    'filter'=>'CLogFilter',
                ),
                ...other log routes...
            ),
        ),
    ),
),
)

```

Starting from version 1.0.7, Yii supports logging call stack information in the messages that are logged by calling `Yii::trace`. This feature is disabled by default because it lowers performance. To use this feature, simply define a constant named `YII_TRACE_LEVEL` at the beginning of the entry script (before including `yii.php`) to be an integer greater than 0. Yii will then append to every trace message with the file name and line number of the call stacks belonging to application code. The number `YII_TRACE_LEVEL` determines how many layers of each call stack should be recorded. This information is particularly useful during development stage as it can help us identify the places that trigger the trace messages.

Performance Profiling

Performance profiling is a special type of message logging. Performance profiling can be used to measure the time needed for the specified code blocks and find out what the performance bottleneck is.

To use performance profiling, we need to identify which code blocks need to be profiled. We mark the beginning and the end of each code block by inserting the following methods:

```

Yii::beginProfile('blockID');
...code block being profiled...
Yii::endProfile('blockID');

```

where `blockID` is an ID that uniquely identifies the code block.

Note, code blocks need to be nested properly. That is, a code block cannot intersect with another. It must be either at a parallel level or be completely enclosed by the other code block.

To show profiling result, we need to install a [CLogRouter](#) application component with a [CProfileLogRoute](#) log route. This is the same as we do with normal message routing. The [CProfileLogRoute](#) route will display the performance results at the end of the current page.

Profiling SQL Executions

Profiling is especially useful when working with database since SQL executions are often the main performance bottleneck of an application. While we can manually insert `beginProfile` and `endProfile` statements at appropriate places to measure the time spent in each SQL execution, starting from version 1.0.6, Yii provides a more systematic approach to solve this problem.

By setting [CDbConnection::enableProfiling](#) to be true in the application configuration, every SQL statement being executed will be profiled. The results can be readily displayed using the aforementioned [CProfileLogRoute](#), which can show us how much time is spent in executing what SQL statement. We can also call [CDbConnection::getStats\(\)](#) to retrieve the total number SQL statements executed and their total execution time.

Errors

Yii provides a complete error handling framework based on the PHP 5 exception mechanism. When the application is created to handle an incoming user request, it registers its [handleError](#) method to handle PHP warnings and notices; and it registers its [handleException](#) method to handle uncaught PHP exceptions. Consequently, if a PHP warning/notice or an uncaught exception occurs during the application execution, one of the error handlers will take over the control and start the necessary error handling procedure.

Tip: The registration of error handlers is done in the application's constructor by calling PHP functions `set_exception_handler` and `set_error_handler`. If you do not want Yii to handle the errors and exceptions, you may define constant `YII_ENABLE_ERROR_HANDLER` and `YII_ENABLE_EXCEPTION_HANDLER` to be false in the entry script.

By default, [errorHandler](#) (or [exceptionHandler](#)) will raise an [onError](#) event (or [onException](#) event). If the error (or exception) is not handled by any event handler, it will call for help from the [errorHandler](#) application component.

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.');
```

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:

- WebRoot/themes/ThemeName/views/system: this is the system view directory under the currently active theme.
- WebRoot/protected/views/system: this is the default system view directory for an application.
- yii/framework/views: this is the standard system view directory provided by the Yii framework.

Therefore, if we want to customize the error display, we can simply create error view files under the system view directory of our application or theme. Each view file is a normal PHP script consisting of mainly HTML code. For more details, please refer to the default view files under the framework's view directory.

Handling Errors Using an Action

Starting from version 1.0.6, Yii allows using a [controller action](#) to handle the error display work. To do so, we should configure the error handler in the application configuration as follows:

```

return array(
    .....
    'components'=>array(
        'errorHandler'=>array(
            'errorAction'=>'site/error',
        ),
    ),
);

```

In the above, we configure the [CErrorHandler::errorAction](#) property to be the route site/error which refers to the error action in SiteController. We may use a different route if needed.

We can write the error action like the following:

```

public function actionError()
{
    if($error=Yii::app()->errorHandler->error)
        $this->render('error', $error);
}

```

In the action, we first retrieve the detailed error information from [CErrorHandler::error](#). If it is not empty, we render the error view together with the error information. The error information returned from [CErrorHandler::error](#) is an array with the following fields:

- code: the HTTP status code (e.g. 403, 500);
- type: the error type (e.g. [CHttpException](#), PHP Error);
- message: the error message;
- file: the name of the PHP script file where the error occurs;
- line: the line number of the code where the error occurs;
- trace: the call stack of the error;
- source: the context source code where the error occurs.

Tip: The reason we check if `CErrorHandler::error` is empty or not is because the error action may be directly requested by an end user, in which case there is no error. Since we are passing the `$error` array to the view, it will be automatically expanded to individual variables. As a result, in the view we can access directly the variables such as `$code`, `$type`.

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.

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.

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.

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.

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.

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;

    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }
}

```

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'
                ),
            ),
        );
    }
    .....
}

```

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).

Internacionalização (i18n)

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.

Locale and Language

Locale is a set of parameters that defines the user's language, country and any special variant preferences that the user wants to see in their user interface. It is usually identified by an ID consisting of a language ID and a region ID. For example, the ID `en_US` stands for the locale of English and United States. For consistency, all locale IDs in Yii are

canonicalized to the format of LanguageID or LanguageID_RegionID in lower case (e.g. en, en_us).

Locale data is represented as a [CLocale](#) instance. It provides locale-dependent information, including currency symbols, number symbols, currency formats, number formats, date and time formats, and date-related names. Since the language information is already implied in the locale ID, it is not provided by [CLocale](#). For the same reason, we often interchangeably using the term locale and language.

Given a locale ID, one can get the corresponding [CLocale](#) instance by `CLocale::getInstance($localeID)` or `CApplication::getLocale($localeID)`.

Info: Yii comes with locale data for nearly every language and region. The data is obtained from Common Locale Data Repository (CLDR). For each locale, only a subset of the CLDR data is provided as the original data contains a lot of rarely used information. Starting from version 1.1.0, users can also supply their own customized locale data. To do so, configure the `CApplication::localeDataPath` property with the directory that contains the customized locale data. Please refer to the locale data files under `framework/i18n/data` in order to create customized locale data files.

For a Yii application, we differentiate its target language from source language. The target language is the language (locale) of the users that the application is targeted at, while the source language refers to the language (locale) that the application source files are written in. Internationalization occurs only when the two languages are different.

One can configure target language in the application configuration, or change it dynamically before any internationalization occurs.

Tip: Sometimes, we may want to set the target language as the language preferred by a user (specified in user's browser preference). To do so, we can retrieve the user preferred language ID using `CHttpRequest::preferredLanguage`.

Translation

The most needed I18N feature is perhaps translation, including message translation and view translation. The former translates a text message to the desired language, while the latter translates a whole file to the desired language.

A translation request consists of the object to be translated, the source language that the object is in, and the target language that the object needs to be translated to. In Yii, the source language is default to the [application source language](#) while the target language is default to the [application language](#). If the source and target languages are the same, translation will not occur.

Message Translation

Message translation is done by calling `Yii::t()`. The method translates the given message from [source language](#) to [target language](#).

When translating a message, its category has to be specified since a message may be translated differently under different categories (contexts). The category `yii` is reserved for messages used by the Yii framework core code.

Messages can contain parameter placeholders which will be replaced with the actual parameter values when calling [Yii::t\(\)](#). For example, the following message translation request would replace the `{alias}` placeholder in the original message with the actual alias value.

```
Yii::t('app', 'Path alias "{alias}" is redefined.',  
    array('{alias}'=>$alias))
```

Note: Messages to be translated must be constant strings. They should not contain variables that would change message content (e.g. "Invalid {\$message} content."). Use parameter placeholders if a message needs to vary according to some parameters.

Translated messages are stored in a repository called message source. A message source is represented as an instance of [CMessageSource](#) or its child class. When [Yii::t\(\)](#) is invoked, it will look for the message in the message source and return its translated version if it is found.

Yii comes with the following types of message sources. You may also extend [CMessageSource](#) to create your own message source type.

- [CPhpMessageSource](#): the message translations are stored as key-value pairs in a PHP array. The original message is the key and the translated message is the value. Each array represents the translations for a particular category of messages and is stored in a separate PHP script file whose name is the category name. The PHP translation files for the same language are stored under the same directory named as the locale ID. And all these directories are located under the directory specified by [basePath](#).
- [CGettextMessageSource](#): the message translations are stored as [GNU Gettext](#) files.
- [CDbMessageSource](#): the message translations are stored in database tables. For more details, see the API documentation for [CDbMessageSource](#).

A message source is loaded as an [application component](#). Yii pre-declares an application component named [messages](#) to store messages that are used in user application. By default, the type of this message source is [CPhpMessageSource](#) and the base path for storing the PHP translation files is `protected/messages`.

In summary, in order to use message translation, the following steps are needed:

- Call [Yii::t\(\)](#) at appropriate places;
- Create PHP translation files as `protected/messages/LocaleID/CategoryName.php`. Each file simply returns an array of message translations. Note, this assumes you are using the default [CPhpMessageSource](#) to store the translated messages.
- Configure [CApplication::sourceLanguage](#) and [CApplication::language](#).

Tip: The yiic tool in Yii can be used to manage message translations when CPhpMessageSource is used as the message source. Its message command can automatically extract messages to be translated from selected source files and merge them with existing translations if necessary. For more details of using the message command, please run `yiic help message`.

Starting from version 1.0.10, when using [CPhpMessageSource](#) to manage message source, messages for an extension class (e.g. a widget, a module) can be specially managed and used. In particular, if a message belongs to an extension whose class name is `XYZ`, then the message category can be specified in the format of `XYZ.categoryName`. The corresponding message file will be assumed to be `BasePath/messages/LanguageID/categoryName.php`, where `BasePath` refers to the directory that contains the extension class file. And when using `Yii::t()` to translate an extension message, the following format should be used, instead:

```
Yii::t('XYZ.categoryName', 'message to be translated')
```

Since version 1.0.2, Yii has added the support for [choice format](#). Choice format refers to choosing a translated according to a given number value. For example, in English the word 'book' may either take a singular form or a plural form depending on the number of books, while in other languages, the word may not have different form (such as Chinese) or may have more complex plural form rules (such as Russian). Choice format solves this problem in a simple yet effective way.

To use choice format, a translated message must consist of a sequence of expression-message pairs separated by `|`, as shown below

```
'expr1#message1|expr2#message2|expr3#message3'
```

where `exprN` refers to a valid PHP expression which evaluates to a boolean value indicating whether the corresponding message should be returned. Only the message corresponding to the first expression that evaluates to true will be returned. An expression can contain a special variable named `n` (note, it is not `$n`) which will take the number value passed as the first message parameter. For example, assuming a translated message is:

```
'n==1#one book|n>1#many books'
```

and we are passing a number value 2 in the message parameter array when calling `Yii::t()`, we would obtain many books as the final translated message:

```
Yii::t('app', 'n==1#one book|n>1#many books', array(1));  
//or since 1.1.6  
Yii::t('app', 'n==1#one book|n>1#many books', 1);
```

As a shortcut notation, if an expression is a number, it will be treated as `n==Number`. Therefore, the above translated message can be also be written as:

```
'1#one book|n>1#many books'
```

Plural forms format

Since version 1.1.6 CLDR-based plural choice format can be used with a simpler syntax that. It is handy for languages with complex plural form rules.

The rule for English plural forms above can be written in the following way:

```
Yii::t('test', 'cucumber|cucumbers', 1);  
Yii::t('test', 'cucumber|cucumbers', 2);  
Yii::t('test', 'cucumber|cucumbers', 0);
```

The code above will give you:

```
cucumber  
cucumbers  
cucumbers
```

If you want to include number you can use the following code.

```
echo Yii::t('test', '{n} cucumber|{n} cucumbers', 1);
```

Here {n} is a special placeholder holding number passed. It will print 1 cucumber.

You can pass additional parameters:

```
Yii::t('test', '{username} has a cucumber|{username} has {n} cucumbers',  
array(5, '{username}' => 'samdark'));
```

and even replace number parameter with something else:

```
function convertNumber($number)  
{  
    // convert number to word  
    return $number;  
}  
  
Yii::t('test', '{n} cucumber|{n} cucumbers',  
array(5, '{n}' => convertNumber(5)));
```

For Russian it will be:

```
Yii::t('app', '{n} cucumber|{n} cucumbers', 62);  
Yii::t('app', '{n} cucumber|{n} cucumbers', 1.5);  
Yii::t('app', '{n} cucumber|{n} cucumbers', 1);  
Yii::t('app', '{n} cucumber|{n} cucumbers', 7);
```

with translated message

```
'{n} cucumber|{n} cucumbers' => '{n} огурец|{n} огурца|{n} огурцов|{n}  
огурца',
```

and will give you

```
62 огурца  
1.5 огурца  
1 огурец  
7 огурцов
```

Info: to learn about how many values you should supply and in which order they should be, please refer to [CLDR Language Plural Rules page](#).

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.

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).

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.

Sintaxe Alternativa de Templates

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.

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,

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

where clipID is an identifier that uniquely identifies the clip content. The clip tags will be translated as

```
<?php $this->beginClip('clipID'); ?>
    // content for this clip
<?php $this->endClip(); ?>
```

Comment Tags

Comment tags are used to write view comments that should only be visible to developers. Comment tags will be stripped off when the view is displayed to end users. The syntax for comment tags is as follows,

```
<!--
view comments that will be stripped off
-->
```

Mixing Template Formats

Starting from version 1.1.2, it is possible to mix the usage of some alternative template syntax with the normal PHP syntax. To do so, the `CViewRenderer::fileExtension` property of the installed view renderer must be configured with a value other than `.php`. For example, if the property is set as `.tpl`, then any view file ending with `.tpl` will be rendered using the installed view renderer, while all other view files ending with `.php` will be treated as normal PHP view script.

Aplicativos de Console

Console applications are mainly used to perform offline work needed by an online Web application, such as code generation, search index compiling, email sending, etc. Yii provides a framework for writing console applications in an object-oriented way. It allows a console application to access the resources (e.g. DB connections) that are used by an online Web application.

Overview

Yii represents each console task in terms of a command. A console command is written as a class extending from `CConsoleCommand`.

When we use the `yiic webapp` tool to create an initial skeleton Yii application, we may find two files under the protected directory:

- `yiic`: this is an executable script used on Linux/Unix;
- `yiic.bat`: this is an executable batch file used on Windows.

In a console window, we can enter the following commands:

```
cd protected  
yiic help
```

This will display a list of available console commands. By default, the available commands include those provided by Yii framework (called system commands) and those developed by users for individual applications (called user commands).

To see how to use a command, we can execute

```
yiic help <command-name>
```

And to execute a command, we can use the following command format:

```
yiic <command-name> [parameters...]
```

Creating Commands

Console commands are stored as class files under the directory specified by [`CConsoleApplication::commandPath`](#). By default, this refers to the directory `protected/commands`.

A console command class must extend from [`CConsoleCommand`](#). The class name must be of format `XYZCommand`, where `XYZ` refers to the command name with the first letter in upper case. For example, a sitemap command must use the class name `SitemapCommand`. Console command names are case-sensitive.

Tip: By configuring `CConsoleApplication::commandMap`, one can also have command classes in different naming conventions and located in different directories.

To create a new command, one often needs to override [`CConsoleCommand::run\(\)`](#) or develop one or several command actions (to be explained in the next section).

When executing a console command, the [`CConsoleCommand::run\(\)`](#) method will be invoked by the console application. Any console command parameters will be passed to the method as well, according to the following signature of the method:

```
public function run($args) { ... }
```

where \$args refers to the extra parameters given in the command line.

Within a console command, we can use `Yii::app()` to access the console application instance, through which we can also access resources such as database connections (e.g. `Yii::app()->db`). As we can tell, the usage is very similar to what we can do in a Web application.

Info: Starting from version 1.1.1, we can also create global commands that are shared by all Yii applications on the same machine. To do so, define an environment variable named `YII_CONSOLE_COMMANDS` which should point to an existing directory. We can then put our global command class files under this directory.

Console Command Action

Note: The feature of console command action has been available since version 1.1.5.

A console command often needs to handle different command line parameters, some required, some optional. A console command may also need to provide several sub-commands to handle different sub-tasks. These work can be simplified using console command actions.

A console command action is a method in a console command class. The method name must be of the format `actionXyz`, where `Xyz` refers to the action name with the first letter in upper-case. For example, a method `actionIndex` defines an action named `index`.

To execute a specific action, we use the following console command format:

```
yiic <command-name> <action-name> --option1=value --option2=value2 ...
```

The additional option-value pairs will be passed as named parameters to the action method. The value of a `xyz` option will be passed as the `$xyz` parameter of the action method. For example, if we define the following command class:

```
class SitemapCommand extends CConsoleCommand
{
    public function actionIndex($type, $limit=5) { ... }
    public function actionInit() { ... }
}
```

Then, the following console commands will all result in calling `actionIndex('News', 5)`:

```
yiic sitemap index --type=News --limit=5

// $limit takes default value
yiic sitemap index --type=News

// $limit takes default value
// because 'index' is a default action, we can omit the action name
yiic sitemap --type=News

// the order of options does not matter
yiic sitemap index --limit=5 --type=News
```

If an option is given without value (e.g. `--type` instead of `--type=News`), the corresponding action parameter value will be assumed to be boolean true.

Note: The feature of console command action has been available since version 1.1.5.

A parameter can take an array value by declaring it with array type hinting:

```
public function actionIndex(array $types) { ... }
```

To supply the array value, we simply repeat the same option in the command line as needed:

```
yiic sitemap index --types=News --types=Article
```

The above command will call `actionIndex(array('News', 'Article'))` ultimately.

Starting from version 1.1.6, Yii also supports using anonymous action parameters and global options.

Anonymous parameters refer to those command line parameters not in the format of options. For example, in a command `yiic sitemap index --limit=5 News`, we have an anonymous parameter whose value is `News` while the named parameter `limit` is taking the value `5`.

To use anonymous parameters, a command action must declare a parameter named as `$args`. For example,

```
public function actionIndex($limit=10, $args=array()) {...}
```

The `$args` array will hold all available anonymous parameter values.

Global options refer to those command line options that are shared by all actions in a command. For example, in a command that provides several actions, we may want every action to recognize an option named as verbose. While we can declare \$verbose parameter in every action method, a better way is to declare it as a public member variable of the command class, which turns verbose into a global option:

```
class SitemapCommand extends CConsoleCommand
{
    public $verbose=false;
    public function actionIndex($type) {...}
}
```

The above code will allow us to execute a command with a verbose option:

```
yiic sitemap index --verbose=1 --type=News
```

Customizing Console Applications

By default, if an application is created using the yiic webapp tool, the configuration for the console application will be protected/config/console.php. Like a Web application configuration file, this file is a PHP script which returns an array representing the property initial values for a console application instance. As a result, any public property of [CConsoleApplication](#) can be configured in this file.

Because console commands are often created to serve for the Web application, they need to access the resources (such as DB connections) that are used by the latter. We can do so in the console application configuration file like the following:

```
return array(
    .....
    'components'=>array(
        'db'=>array(
            .....
        ),
    ),
);
```

As we can see, the format of the configuration is very similar to what we do in a Web application configuration. This is because both [CConsoleApplication](#) and [CWebApplication](#) share the same base class.

Segurança

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(); ?>
```

Cross-site Request Forgery Prevention

Cross-Site Request Forgery (CSRF) attacks occur when a malicious web site causes a user's web browser to perform an unwanted action on a trusted site. For example, a malicious web site has a page that contains an image tag whose src points to a banking site: <http://bank.example/withdraw?transfer=10000&to=someone>. If a user who has a login cookie for the banking site happens to visit this 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.

Cookie Attack Prevention

Protecting cookies from being attacked is of extreme importance, as session IDs are commonly stored in cookies. If one gets hold of a session ID, he essentially owns all relevant session information.

There are several countermeasures to prevent cookies from being attacked.

- An application can use SSL to create a secure communication channel and only pass the authentication cookie over an HTTPS connection. Attackers are thus unable to decipher the contents in the transferred cookies.
- Expire sessions appropriately, including all cookies and session tokens, to reduce the likelihood of being attacked.
- Prevent cross-site scripting which causes arbitrary code to run in a user's browser and expose his cookies.
- Validate cookie data and detect if they are altered.

Yii implements a cookie validation scheme that prevents cookies from being modified. In particular, it does HMAC check for the cookie values if cookie validation is enabled.

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 `$_COOKIES`:

```
// 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;
```

Ajustes no Desempenho

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.

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.

Disabling Debug Mode

Disabling debug mode is another easy way to improve performance. A Yii application runs in debug mode if the constant `YII_DEBUG` is defined as true. Debug mode is useful during development stage, but it would impact performance because some components cause extra burden in debug mode. For example, the message logger may record additional debug information for every message being logged.

Using yiilite.php

When the [PHP APC extension](#) is enabled, we can replace `yii.php` with a different Yii bootstrap file named `yiilite.php` to further boost the performance of a Yii-powered application.

The file `yiilite.php` comes with every Yii release. It is the result of merging some commonly used Yii class files. Both comments and trace statements are stripped from the merged file. Therefore, using `yiilite.php` would reduce the number of files being included and avoid execution of trace statements.

Note, using `yiilite.php` without APC may actually reduce performance, because `yiilite.php` contains some classes that are not necessarily used in every request and would take extra parsing time. It is also observed that using `yiilite.php` is slower with some server configurations, even when APC is turned on. The best way to judge whether to use `yiilite.php` or not is to run a benchmark using the included hello world demo.

Using Caching Techniques

As described in the [Caching](#) section, Yii provides several caching solutions that may improve the performance of a Web application significantly. If the generation of some data takes long time, we can use the [data caching](#) approach to reduce the data generation frequency; If a portion of page remains relatively static, we can use the [fragment caching](#) approach to reduce its rendering frequency; If a whole page remains relative static, we can use the [page caching](#) approach to save the rendering cost for the whole page.

If the application is using [Active Record](#), we should turn on the schema caching to save the time of parsing database schema. This can be done by configuring the [CDbConnection::schemaCachingDuration](#) property to be a value greater than 0.

Besides these application-level caching techniques, we can also use server-level caching solutions to boost the application performance. As a matter of fact, the [APC caching](#) we described earlier belongs to this category. There are other server techniques, such as [Zend Optimizer](#), [eAccelerator](#), [Squid](#), to name a few.

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 modeling 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.

Minimizing Script Files

Complex pages often need to include many external JavaScript and CSS files. Because each file would cause one extra round trip to the server and back, we should minimize the number of script files by merging them into fewer ones. We should also consider reducing the size of each script file to reduce the network transmission time. There are many tools around to help on these two aspects.

For a page generated by Yii, chances are that some script files are rendered by components that we do not want to modify (e.g. Yii core components, third-party components). In order to minimizing these script files, we need two steps.

Note: The scriptMap feature described in the following has been available since version 1.0.3.

First, we declare the scripts to be minimized by configuring the scriptMap property of the clientScript application component. This can be done either in the application configuration or in code. For example,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>'/js/all.js',
    'jquery.ajaxqueue.js'=>'/js/all.js',
    'jquery.metadata.js'=>'/js/all.js',
    .....
);
```

What the above code does is that it maps those JavaScript files to the URL /js/all.js. If any of these JavaScript files need to be included by some components, Yii will include the URL (once) instead of the individual script files.

Second, we need to use some tools to merge (and perhaps compress) the JavaScript files into a single one and save it as js/all.js.

The same trick also applies to CSS files.

We can also improve page loading speed with the help of [Google AJAX Libraries API](#). For example, we can include jquery.js from Google servers instead of our own server. To do so, we first configure the scriptMap as follows,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>false,
    'jquery.ajaxqueue.js'=>false,
    'jquery.metadata.js'=>false,
    .....
);
```

By mapping these script files to false, we prevent Yii from generating the code to include these files. Instead, we write the following code in our pages to explicitly include the script files from Google,

```
<head>
<?php echo CGoogleApi::init(); ?>

<?php echo CHtml::script(
    CGoogleApi::load('jquery','1.3.2') . "\n" .
    CGoogleApi::load('jquery.ajaxqueue.js') . "\n" .
    CGoogleApi::load('jquery.metadata.js')
); ?>
.....
</head>
```