



Полное руководство на русском языке

Руководство доступно по адресу <http://yiiframework.ru/doc/guide/ru/index>

Начало

Полное руководство по Yii

Данное руководство выпущено в соответствии с положениями о документации Yii.

Переводчики

- Константин Мирин, Konstantin Mirin (programmersnotes.info)
- Александр Макаров, Sam Dark (rmcreative.ru)
- Алексей Лукьяненко, Caveman (caveman.ru)
- Евгений Халецкий, xenon
- multif
- Сергей Кузнецов, cr0t (summercode.ru)
- Александр Кожевников, Bethrezen (bethrezen.ru)

© 2008—2010, Yii Software LLC.

Новые возможности

На этой странице кратко излагаются новые возможности, внесённые в каждом релизе Yii.

Версия 1.1.5

- Добавлена поддержка действий и параметров действий в консольных командах
- Добавлена поддержка загрузки классов из пространств имён
- Добавлена поддержка темизации виджетов

Версия 1.1.4

- Добавлена поддержка автоматической привязки параметров действий контроллера

Версия 1.1.3

- Добавлена возможность настройки виджета через файл конфигурации приложения

Версия 1.1.2

- Добавлен веб-кодогенератор Gii

Версия 1.1.1

- Добавлен виджет CActiveForm, упрощающий написание кода формы и поддерживающий прозрачную валидацию как на стороне клиента, так и на стороне сервера.
- Произведён рефакторинг кода, генерируемого yiiс. Приложение-каркас теперь генерируется с поддержкой нескольких главных разметок, использован виджет меню, добавлена возможность сортировать данные в административном интерфейсе, для отображения форм используется CActiveForm.
- Добавлена поддержка глобальных консольных команд.

Версия 1.1.0

- Добавлена возможность использования модульного и функционального тестирования.
- Добавлена возможность использования скинов виджета.
- Добавлен гибкий инструмент для построения форм.
- Улучшен способ объявления безопасных атрибутов модели:
 - Безопасное присваивание значений атрибутам.
- Изменён алгоритм жадной загрузки по умолчанию для зависимых запросов AR так, что все таблицы объединяются в одном SQL запросе.
- Изменён псевдоним таблицы по умолчанию на имя отношений AR.
- Добавлена поддержка использования префикса таблиц.
- Добавлен набор новых расширений — библиотека Zii.
- Псевдоним для главной таблицы в AR запросе теперь всегда равен 't'.

Версия 1.0.11

- Добавлена поддержка разбора и создания URL с параметризованными именами хостов:
 - Параметризация имен хостов

Версия 1.0.10

- Улучшен CPhpMessageSource. Теперь можно использовать для перевода сообщений модулей:

- Перевод сообщений
- Добавлена поддержка анонимных функций в качестве обработчиков событий:
 - События компонента

Версия 1.0.8

- Добавлена поддержка получения нескольких кэшированных значений одновременно:
 - Кэширование данных
- Введен новый корневой псевдоним `ext` для директории, содержащей сторонние расширения:
 - Использование расширений

Версия 1.0.7

- Добавлена поддержка отображения информации стека вызовов в трассирующих сообщениях:
 - Сохранение контекста сообщений
 - В отношениях AR добавлена опция `index`, позволяющая использовать значения столбца в качестве ключей массива связанных объектов:
 - Параметры реляционного запроса

Версия 1.0.6

- Добавлена поддержка использования именованной группы условий с методами `update` и `delete`:
 - Именованные группы условий
- Добавлена поддержка использования именованной группы условий в параметре `with` реляционных правил:
 - Реляционные запросы с именованными группами условий
- Добавлена поддержка профилирования SQL-запросов:
 - Профилирование SQL-запросов
- Добавлена поддержка журналирования дополнительной информации контекста сообщений:
 - Сохранение контекста сообщений
- Добавлена поддержка настройки одиночного URL-правила установкой его параметров `urlFormat` и `caseSensitive`:
 - Человекопонятные URL
- Добавлена возможность отображения ошибок приложения в действии контроллера:
 - Управление отображением ошибок в действии контроллера

Версия 1.0.5

- Active Record расширена поддержкой именованных групп условий:
 - Именованные группы условий
 - Именованная группа условий по умолчанию
 - Реляционные запросы с именованными группами условий
- Active Record расширена поддержкой отложенной загрузки с динамическими параметрами реляционного запроса:
 - Динамические параметры реляционного запроса
- Расширен класс `CUrlManager` поддержкой параметризованных маршрутов в URL-правилах:
 - Параметризация маршрутов

Обновление с версии 1.0 на версию 1.1

Изменения, связанные со сценариями модели

- Удален метод `safeAttributes()`. Теперь, чтобы быть валированными, безопасные атрибуты определяются правилами, определенными в методе `rules()` для конкретного сценария.
- Изменены методы `validate()`, `beforeValidate()`, `afterValidate()`. `setAttributes()`, `getSafeAttributeNames()` и параметр `'scenario'` удалены. Вы должны получать и устанавливать сценарий модели через свойство `CModel::scenario`.
- Изменён метод `getValidators()` и удалён `getValidatorsForAttribute()`. `CModel::getValidators()` теперь возвращает только валидаторы, применяемые к сценарию, определенному свойством сценария модели (`CModel::scenario`).
- Изменены методы `isAttributeRequired()` и `CModel::getValidatorsForAttribute()`. Параметр сценария удален. Вместо него следует использовать свойство сценария модели.
- Удалено свойство `CHtml::scenario`. `CHtml` теперь использует сценарий, указанный в модели.

Изменения, связанные с 'жадной' загрузкой для отношений Active Record

- По умолчанию для всех отношений, включенных в 'жадную' загрузку, будет сгенерировано и выполнено одно выражение с использованием `JOIN`. Если в основной таблице есть опции запроса

`LIMIT` или `OFFSET`, то сначала будет выполнен этот запрос, а затем другой SQL-запрос, который возвращает все связанные объекты. Раньше, в версии 1.0.x, по умолчанию вызывалось `n+1` SQL-запросов, если 'жадная' загрузка включала `n` отношений `HAS_MANY` или `MANY_MANY`.

Изменения, связанные с псевдонимами таблиц в отношениях Active Record

- Теперь псевдоним по умолчанию для связанной таблицы такой же, как и соответствующее имя отношения. Ранее, в версии 1.0.x, по умолчанию Yii автоматически генерировал псевдоним таблицы для каждой связанной таблицы, и мы должны были использовать префикс `??` для ссылки на этот автоматически сгенерированный псевдоним.
- Псевдоним для главной таблицы в AR запросе теперь всегда равен `t`. В версии 1.0.x, он соответствовал имени таблицы. Данное изменение ломает код существующих запросов AR в том случае, если в качестве псевдонима было использовано имя таблицы. Решение — заменить такие псевдонимы на `'t'`.

Изменения, связанные с табличным (пакетным) вводом данных

- Для имен полей использование записи вида `поле[$i]` больше неверно. Они должны выглядеть так — `[$i]поле`, чтобы была возможность поддержки множественного ввода одностипных полей (например, `[$i]поле[$index]`).

Другие изменения

- Изменён конструктор `CActiveRecord`. Первый параметр (список атрибутов) убран.

Что такое Yii

Yii — это высокоэффективный основанный на компонентной структуре PHP-фреймворк для разработки масштабных веб-приложений. Он позволяет максимально применить концепцию повторного использования кода и может существенно ускорить процесс веб-разработки. Название Yii (произносится как *Yee* или *[ji:]*) означает *простой (easy)*, *эффективный (efficient)* и *расширяемый (extensible)*.

Требования

Для запуска веб-приложений, построенных на Yii, вам понадобится веб-сервер с поддержкой PHP версии 5.1.0 или выше.

Для разработчиков, желающих использовать Yii, крайне полезным будет понимание концепции объектно-ориентированного программирования (ООП), так как Yii — это строго объектно-ориентированный фреймворк.

Для чего Yii будет лучшим выбором?

Yii — это фреймворк для веб-программирования общего назначения, который может быть использован для разработки практически любых веб-приложений. Благодаря своей легковесности и наличию продвинутых средств кэширования, Yii особенно подходит для разработки приложений с большим потоком трафика, таких как порталы, форумы, системы управления контентом (CMS), системы электронной коммерции и др.

Yii в сравнении с другими фреймворками

Подобно большинству других PHP-фреймворков, Yii — это MVC-фреймворк.

Превосходство Yii над другими фреймворками заключается в эффективности, широких возможностях и качественной документации. Yii изначально спроектирован очень тщательно для соответствия всем требованиям при разработке серьезных веб-приложений. Yii не является ни побочным продуктом какого-либо проекта, ни сборкой сторонних решений. Он является результатом большого опыта авторов в разработке веб-приложений, а также исследований наиболее популярных веб-фреймворков и приложений.

Установка

Для установки Yii, как правило, необходимо выполнить два шага:

1. Скачать Yii Framework с yiiframework.com;
2. Распаковать релиз Yii в директорию, доступную из веб.

Подсказка: На самом деле, Yii содержит входной скрипт, который обычно является единственным файлом со свободным доступом к нему из веб. Другие PHP-скрипты, включая и файлы Yii, должны быть защищены от прямого доступа, так как могут быть использованы для взлома.

Требования

После установки Yii проверьте, соответствует ли ваш веб-сервер необходимым требованиям для использования Yii. Вы можете сделать это, воспользовавшись специальным скриптом, доступным из веб-браузера по адресу:

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

Минимальные требования для Yii — поддержка вашим веб-сервером PHP версии 5.1.0 или выше. Yii тестировался с Apache HTTP server на платформах Windows и Linux. Скорее всего, он будет работать и на других веб-серверах и платформах с поддержкой PHP 5.

Создание первого приложения

В этом разделе мы расскажем, как создать наше первое приложение. Для создания нового приложения будем использовать `yiic` (консольную утилиту), для генерации кода — `gii` (мощный веб кодогенератор). Будем считать для удобства, что `yiiRoot` — это директория, куда установлен Yii, а `webRoot` — корневая директория вашего веб-сервера.

Запускаем `yiic` в консоли со следующими параметрами:

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

Примечание: При использовании `yiic` на Mac OS, Linux или Unix вам может понадобиться изменить права доступа для файла `yiic`, чтобы сделать его исполняемым. Альтернативный вариант запуска утилиты представлен ниже:

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

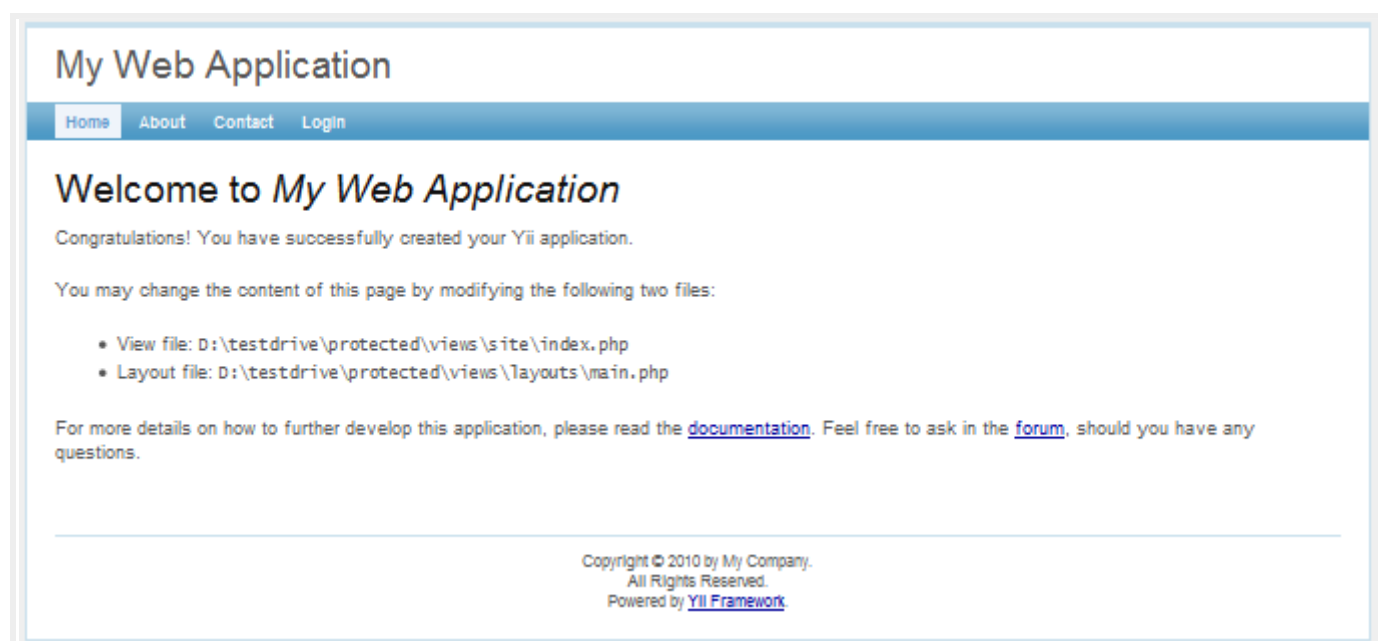
В результате в директории `webRoot/testdrive` будет создан каркас приложения.

Созданное приложение — хорошая отправная точка для добавления необходимого функционала, так как оно уже содержит все необходимые директории и файлы. Не написав ни единой строчки кода, мы уже можем протестировать наше первое Yii-приложение, перейдя в браузере по следующему URL:

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

Приложение содержит четыре страницы: главную, страницу «о проекте», страницу обратной связи и страницу авторизации. Страница обратной связи содержит форму для отправки вопросов и пожеланий, а страница авторизации позволяет пользователю аутентифицироваться и получить доступ к закрытой части сайта (см. рисунки ниже).

Главная страница



Страница обратной связи

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

 [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](#)

Страница обратной связи с ошибками ввода

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](#)

Страница обратной связи с успешно отправленной формой

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](#)

Страница авторизации

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

Login

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

Наше приложение имеет следующую структуру папок. Подробное описание этой структуры можно найти в соглашениях.

testdrive/	
index.php	скрипт инициализации приложения
index-test.php	скрипт инициализации функциональных тестов
assets/	содержит файлы ресурсов
css/	содержит CSS-файлы
images/	содержит картинки
themes/	содержит темы оформления приложения
protected/	содержит защищенные файлы приложения
yiic	скрипт yiic
yiic.bat	скрипт yiic для Windows
yiic.php	PHP-скрипт yiic
commands/	содержит команды 'yiic'
shell/	содержит команды 'yiic shell'
components/	содержит компоненты для повторного использования
Controller.php	класс базового контроллера
UserIdentity.php	класс 'UserIdentity' для аутентификации
config/	содержит конфигурационные файлы
console.php	файл конфигурации консоли
main.php	файл конфигурации веб-приложения
test.php	файл конфигурации функциональных тестов
controllers/	содержит файлы классов контроллеров

SiteController.php	класс контроллера по умолчанию
data/	содержит пример базы данных
schema.mysql.sql	схема БД для MySQL
schema.sqlite.sql	схема БД для SQLite
testdrive.db	файл БД для SQLite
extensions/	содержит сторонние расширения
messages/	содержит переведенные сообщения
models/	содержит файлы классов моделей
LoginForm.php	модель формы для действия 'login'
ContactForm.php	модель формы для действия 'contact'
runtime/	содержит временные файлы
tests/	содержит тесты
views/	содержит файлы представлений контроллеров и файлы макетов
(layout)	
layouts/	содержит файлы представлений макетов
main.php	общая для всех страниц разметка
column1.php	разметка для страниц с одной колонкой
column2.php	разметка для страниц с двумя колонками
site/	содержит файлы представлений для контроллера 'site'
pages/	статические страницы
about.php	страница «о проекте»
contact.php	файл представления для действия 'contact'
error.php	файл представления для действия 'error' (отображение
ошибок)	
index.php	файл представления для действия 'index'
login.php	файл представления для действия 'login'

Соединение с базой данных

Большинство веб-приложений используют базы данных, и наше приложение не исключение. Для использования базы данных необходимо объяснить приложению, как к ней подключиться. Это делается в конфигурационном файле `WebRoot/testdrive/protected/config/main.php`. Например, так:

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

В приведённом выше коде указано, что приложение должно подключиться к базе данных SQLite `WebRoot/testdrive/protected/data/testdrive.db` как только это понадобится. Отметим, что база данных SQLite уже включена в сгенерированное приложение. В этой базе имеется только одна таблица `tbl_user`:

```
CREATE TABLE tbl_user (
```

```

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

```

Если вы хотите использовать базу данных MySQL, вы можете использовать файл `WebRoot/testdrive/protected/data/schema.mysql.sql` для её создания.

Примечание: Для работы с базой данных Yii требуется расширение PHP PDO и соответствующий драйвер PDO. Для тестового приложения необходимо подключить расширения `php_pdo` и `php_pdo_sqlite`.

Реализация операций CRUD

А теперь самое веселое. Мы бы хотели добавить операции CRUD (создание, чтение, обновление и удаление) для только что созданной таблицы `tbl_user` — это часто необходимо при создании реальных приложений. Вместо написания кода мы воспользуемся веб-кодогенератором `Gii`.

Информация: `Gii` доступен начиная с версии 1.1.2. Ранее для тех же целей использовался уже упомянутый `yiic`. Подробнее `yiic` описан в разделе «генерация CRUD при помощи `yiic shell`».

Настройка Gii

Для того, чтобы использовать `Gii`, нужно отредактировать файл конфигурации приложения `WebRoot/testdrive/protected/config/main.php`:

```

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

    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',
            'password'=>'задайте свой пароль',
        ),
    ),
);

```

После этого зайдите по URL `http://hostname/testdrive/index.php?r=gii` и введите указанный в конфигурации пароль.

Генерация модели User

После входа зайдите в раздел `Model Generator`:

Model Generator

Model Generator

This generator generates a model class for the specified database table.

*Fields with * are required. Click on highlighted fields to edit them.*

Table Prefix
[empty]

Table Name *

Model Class *

Base Class *
CActiveRecord

Model Path *
application.models

Code Template *
default (D:\yii\framework\gii\generators\model\templates\default)

В поле `Table Name` введите `tbl_user`. В поле `Model Class` — `User`. Затем нажмите на кнопку `Preview`. Будет показан новый файл, который будет сгенерирован. После нажатия кнопки `Generate` в `protected/models` будет создан файл `User.php`. Как будет описано далее в руководстве, класс модели `User` позволяет работать с данными в таблице `tbl_user` в стиле ООП.

Генерация CRUD

После генерации класса модели мы сгенерируем код, реализующий для неё операции CRUD. Выбираем `Crud Generator`:

CRUD Generator

Crud Generator

This generator generates a controller and views that implement CRUD operations for the specified data model.

*Fields with * are required. Click on highlighted fields to edit them.*

Model Class *

Controller ID *

Base Controller Class *
Controller

Code Template *
default (D:\yii\framework\gii\generators\crud\templates\default)

В поле `Model Class` вводим `User`. В поле `Controller ID` — `user` (в нижнем регистре). Теперь нажимаем `Preview` и затем `Generate`. Генерация кода CRUD завершена.

Доступ к страницам CRUD

Давайте порадуемся нашим трудам, перейдя по следующему URL:

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

Мы увидим страницу со списком пользователей из таблицы `tbl_user`. Поскольку наша таблица пустая, то записей в ней не будет. Кликнем по кнопке `Create User` и, если мы еще не авторизованы, отобразится страница авторизации. Затем загрузится форма добавления нового пользователя. Заполним её и нажмем кнопку `Create`. Если при заполнении формы были допущены ошибки, мы увидим аккуратное сообщение об ошибке.

Возвращаясь к списку пользователей, мы должны увидеть в списке только что созданного пользователя. Добавим еще несколько пользователей. Обратите внимание, что при значительном количестве пользователей для их отображения на одной странице список будет автоматически разбиваться на страницы. Авторизовавшись в качестве администратора (`admin/admin`), можно увидеть страницу управления пользователями по адресу:

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

Появится аккуратная таблица пользователей. Можно кликнуть на заголовок таблицы, чтобы упорядочить записи по значениям соответствующего столбца. Для просмотра, редактирования или удаления соответствующей строки можно воспользоваться кнопками. Также можно переходить на разные страницы, фильтровать результаты и производить поиск по ним.

Всё это не требует написания кода!

Страница управления пользователями

My Web Application

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

[Home](#) » [Users](#) » Manage

Manage Users

[Advanced Search](#)

Id

Username





















Email

Search

Operations

[List User](#)
[Create User](#)

Displaying 1-10 of 21 result(s).

Id	Username	Password	Email	
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](#).

Страница добавления нового пользователя

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 *

Password *

Password cannot be blank.

Email *

Email cannot be blank.

Operations

[List User](#)
[Manage User](#)

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

Основы

Модель-Представление-Контроллер (MVC)

Yii использует шаблон проектирования Модель-Представление-Контроллер (MVC, Model-View-Controller), который широко применяется в веб-программировании.

MVC направлен на отделение бизнес-логики от пользовательского интерфейса, чтобы разработчики могли легко изменять отдельные части приложения не затрагивая другие. В архитектуре MVC модель предоставляет данные и правила бизнес-логики, представление отвечает за пользовательский интерфейс (например, текст, поля ввода), а контроллер обеспечивает взаимодействие между моделью и представлением.

Помимо этого, Yii также использует фронт-контроллер, называемый приложением (application), который выступает в роли контекста выполнения запроса. Приложение производит обработку запроса пользователя и передает его для дальнейшей обработки соответствующему контроллеру.

Следующая диаграмма отображает структуру приложения Yii:

Статическая структура приложения Yii



Типичная последовательность работы приложения Yii

Следующая диаграмма описывает типичную последовательность процесса обработки пользовательского запроса приложением:

Типичная последовательность работы приложения Yii



1. Пользователь осуществляет запрос посредством URL `http://www.example.com/index.php?r=post/show&id=1`, а веб-сервер обрабатывает его, запуская выполнение скрипта инициализации `index.php`;
2. Скрипт инициализации создает экземпляр приложения и запускает его на выполнение;
3. Приложение получает подробную информацию о запросе пользователя от компонента приложения `request`;
4. Приложение определяет запрошенные контроллер и действие при помощи компонента `urlManager`. В данном примере контроллером будет `post`, относящийся к классу `PostController`, а действием — `show`, суть которого определяется контроллером;
5. Приложение создает экземпляр запрашиваемого контроллера для дальнейшей обработки запроса пользователя. Контроллер определяет соответствие действия `show` методу `actionShow` в классе контроллера. Далее создаются и применяются фильтры (например, `access control`, `benchmarking`), связанные с данным действием, и, если фильтры позволяют, действие выполняется;
6. Действие считывает из базы данных модель `Post` с ID равным 1;
7. Действие формирует представление `show` с данными модели `Post`;
8. Представление получает и отображает атрибуты модели `Post`;
9. Представление выполняет некоторые виджеты;
10. Сформированное представление добавляется в макет страницы;
11. Действие завершает формирование представления и выводит результат пользователю.

Входной скрипт

Входной скрипт — это PHP-скрипт, выполняющий первоначальную обработку пользовательского запроса. Это единственный PHP-скрипт, который доступен для исполнения конечному пользователю по прямому запросу.

В большинстве случаев входной скрипт приложения Yii содержит простой код:

```
// для производственного режима эту строку удалите
defined('YII_DEBUG') or define('YII_DEBUG',true);
// подключаем файл инициализации Yii
require_once('path/to/yii/framework/yii.php');
// создаем экземпляр приложения и запускаем его
$configFile='path/to/config/file.php';
Yii::createWebApplication($configFile)->run();
```

Сначала скрипт подключает файл инициализации фреймворка `yii.php`, затем создает экземпляр приложения с установленными параметрами и запускает его на исполнение.

Режим отладки

Приложение может выполняться в отладочном (`debug`) или рабочем (`production`) режиме в зависимости от значения константы `YII_DEBUG`.

По умолчанию её значение установлено в `false`, что означает рабочий режим. Для запуска в режиме отладки установите значение константы в `true` до подключения файла `yii.php`. Работа приложения в режиме отладки не столь эффективна из-за ведения множества внутренних логов. С другой стороны, данный режим очень полезен на стадии разработки, т.к. предоставляет большее количество отладочной информации при возникновении ошибок.

Приложение

Приложение (`application`) — это контекст выполнения запроса. Основная задача приложения — предварительная обработка запроса (`request resolving`) пользователя и передача его соответствующему контроллеру для дальнейшей обработки. Другой задачей приложения является хранение параметров конфигурации уровня приложения (`application-level configuration`). Поэтому приложение также называют **фронт-контроллером**.

Приложение создается входным скриптом как **одиночка** (`singleton`). Экземпляр приложения доступен из любой его точки посредством `Yii::app()`.

Конфигурация приложения

По умолчанию, приложение — это экземпляр класса `CWebApplication`, который может быть настроен с использованием конфигурационного файла (или массива). Необходимые значения свойств устанавливаются в момент создания экземпляра приложения. Альтернативный путь настройки приложения — расширение класса `CWebApplication`.

Конфигурация — это массив пар ключ-значение, где каждый ключ представляет собой имя свойства экземпляра приложения, а значение — начальное значение соответствующего свойства. Например, следующая конфигурация устанавливает значения свойств приложения `name` и `defaultController`:

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

Обычно конфигурация хранится в отдельном PHP-скрипте (например, `protected/config/main.php`). Скрипт возвращает конфигурационный массив так:

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

Чтобы воспользоваться конфигурацией, необходимо передать имя конфигурационного файла как параметр конструктору приложения или классу `Yii::createWebApplication()`, как показано ниже. Обычно это делается во входном скрипте:

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

Подсказка: Если конфигурация очень громоздкая, можно разделить ее на несколько файлов, каждый из которых возвращает часть конфигурационного массива. Затем, в основном конфигурационном файле, необходимо подключить эти файлы, используя `include()`, и слить массивы-части в единый конфигурационный массив.

Базовая директория приложения

Базовой директорией приложения называется корневая директория, содержащая все основные с точки зрения безопасности PHP-скрипты и данные. По умолчанию, это поддиректория с названием `protected` в директории, содержащей входной скрипт. Изменить местоположение можно изменив свойство `basePath` в конфигурации приложения.

Содержимое базовой директории должно быть закрыто от доступа из веб. При использовании веб-сервера Apache HTTP server, это можно сделать путем добавления в базовую директорию файла `.htaccess` следующего содержания:

```
deny from all
```

Компоненты приложения

Функциональность приложения может быть легко модифицирована и расширена благодаря компонентной архитектуре. Приложение управляет набором компонентов, каждый из которых имеет специфические возможности. Например, приложение производит предварительную обработку запроса пользователя, используя компоненты `CUrlManager` и `CHttpRequest`.

Изменяя значение свойства `components`, можно настроить классы и значения свойств любого компонента, используемого приложением. Например, можно сконфигурировать компонент `CMemCache` так, чтобы он использовал несколько `memcache`-серверов для кэширования:

```
array(
    ...
    'components'=>array(
        ...
        'cache'=>array(
```

```

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

```

В данном примере мы добавили элемент `cache` к массиву `components`. Элемент `cache` указывает, что классом компонента является `CMemCache`, а также устанавливает его свойство `servers`.

Для доступа к компоненту приложения используйте `Yii::app()->ComponentID`, где `ComponentID` — это идентификатор компонента (например, `Yii::app()->cache`).

Компонент может быть деактивирован путем установки параметра `enabled` в его конфигурации равным `false`. При обращении к деактивированному компоненту будет возвращен `null`.

Подсказка: По умолчанию, компоненты приложения создаются по требованию. Это означает, что экземпляр компонента может быть не создан вообще, в случае, если это не требуется при обработке пользовательского запроса. Как результат — общая производительность приложения не пострадает, даже если конфигурация приложения включает множество компонентов.

При необходимости обязательного создания экземпляров компонентов (например, `CLogRouter`) вне зависимости от того, используются они или нет, укажите их идентификаторы в значении конфигурационного свойства `preload`.

Ключевые компоненты приложения

Yii предопределяет набор компонентов ядра, которые предоставляют возможности, необходимые для большинства веб-приложений. Например, компонент `request` используется для предварительной обработки запросов пользователя и предоставления различной информации, такой как URL и cookies. Задавая свойства компонентов, можно изменять стандартное поведение Yii практически как угодно.

Далее перечислены ключевые компоненты, предопределенные классом `CWebApplication`:

- `assetManager`: `CAssetManager` — управляет публикацией файлов ресурсов (asset files);
- `authManager`: `CAuthManager` — контролирует доступ на основе ролей (RBAC);
- `cache`: `CCache` — предоставляет возможности кэширования данных; учтите, что вы должны указать используемый класс (например, `CMemCache`, `CDbCache`), иначе при обращении к компоненту будет возвращен `null`;
- `clientScript`: `CClientScript` — управляет клиентскими скриптами (javascripts и CSS);
- `coreMessages`: `CPhpMessageSource` — предоставляет переводы системных сообщений Yii-фреймворка;
- `db`: `CDbConnection` — обслуживает соединение с базой данных; обратите внимание, что для использования компонента необходимо установить свойство `connectionString`;
- `errorHandler`: `CErrorHandler` — обрабатывает не пойманные ошибки и исключения PHP;
- `format`: `CFormatter` - форматирует данные для их последующего отображения. Доступно с версии 1.1.0;
- `messages`: `CPhpMessageSource` — предоставляет переводы сообщений, используемых в Yii-приложении;
- `request`: `CHttpRequest` — предоставляет информацию, относящуюся к пользовательскому запросу;
- `securityManager`: `CSecurityManager` — предоставляет функции, связанные с безопасностью (например, хеширование, шифрование);
- `session`: `CHttpSession` — обеспечивает функциональность, связанную с сессиями;
- `statePersister`: `CStatePersister` — предоставляет метод для сохранения глобального состояния;
- `urlManager`: `CUrlManager` — предоставляет функции парсинга и формирования URL;
- `user`: `CWebUser` — предоставляет идентификационную информацию текущего пользователя;
- `themeManager`: `CThemeManager` — управляет темами оформления.

Жизненный цикл приложения

Жизненный цикл приложения при обработке пользовательского запроса:

1. Предварительная инициализация приложения через `CApplication::preinit()`.
2. Инициализация автозагрузчика классов и обработчика ошибок.
3. Регистрация компонентов ядра.
4. Загрузка конфигурации приложения.
5. Инициализация приложения `CApplication::init()`:
 - регистрация поведений приложения;
 - загрузка статических компонентов приложения;
6. Вызов события `onBeginRequest`.
7. Обработка запроса:
 - разбор запроса;
 - создание контроллера;
 - запуск контроллера на исполнение;
8. Вызов события `onEndRequest`.

Контроллер

Контроллер (`controller`) — это экземпляр класса `CController` или производного от него. Контроллер создается приложением в случае, когда пользователь его запрашивает. При запуске контроллер выполняет соответствующее действие, что обычно подразумевает создание соответствующих моделей и рендеринг необходимых представлений. В самом простом случае **действие** — это метод класса контроллера, название которого начинается на `action`.

У контроллера есть действие по умолчанию, которое выполняется в случае, когда пользователь не указывает действие при запросе. По умолчанию это действие называется `index`. Изменить его можно путем установки значения `CController::defaultAction`.

Ниже приведен минимальный код класса контроллера. Поскольку этот контроллер не определяет ни одного действия, обращение к нему приведет к вызову исключения.

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

Маршрут

Контроллеры и действия опознаются по их идентификаторам. Идентификатор контроллера — это запись формата `path/to/xyz`, соответствующая файлу класса контроллера

`protected/controllers/path/to/XYZController.php`, где `xyz` следует заменить реальным названием класса (например, `post` соответствует `protected/controllers/PostController.php`).

Идентификатор действия — это название метода без префикса `action`. Например, если класс контроллера содержит метод `actionEdit`, то идентификатор соответствующего действия — `edit`.

Примечание: До версии 1.0.3 идентификатор контроллера указывался как `path.to.xyz` вместо `path/to/xyz`.

Пользователь обращается к контроллеру и действию посредством маршрута (`route`). Маршрут формируется путем объединения идентификаторов контроллера и действия, отделенных косой чертой. Например, маршрут `post/edit` указывает на действие `edit` контроллера `PostController` и, по умолчанию, URL `http://hostname/index.php?r=post/edit` приведет к вызову именно этих контроллера и действия.

Примечание: По умолчанию маршруты чувствительны к регистру. Начиная с версии 1.0.1 это возможно изменить путем установки свойства `CUrlManager::caseSensitive` в конфигурации приложения равным `false`. В режиме нечувствительном к регистру убедитесь, что названия директорий, содержащих файлы классов контроллеров написаны в нижнем регистре, а также что `controller map` и `action map` используют ключи в нижнем регистре.

Создание экземпляра контроллера

Экземпляр контроллера создается, когда `CWebApplication` обрабатывает входящий запрос. Получив идентификатор контроллера, приложение использует следующие правила для определения класса контроллера и его местоположения:

- если установлено свойство `CWebApplication::catchAllRequest`, контроллер будет создан на основании этого свойства, а контроллер, запрошенный пользователем, будет проигнорирован. Как правило, это используется для установки приложения в режим технического обслуживания и отображения статической страницы с соответствующим сообщением;
- если идентификатор контроллера обнаружен в `CWebApplication::controllerMap`, то для создания экземпляра контроллера будет использована соответствующая конфигурация контроллера;
- если идентификатор контроллера соответствует формату `'path/to/xyz'`, то имя класса контроллера определяется как `XYZController`, а соответствующий класс как `protected/controllers/path/to/XYZController.php`. Например, идентификатор контроллера `admin/user` будет распознан как класс контроллера — `UserController` и файл класса — `protected/controllers/admin/UserController.php`. Если файл класса не существует, будет вызвано исключение `CHttpException` с кодом ошибки 404.

В случае использования модулей (доступны, начиная с версии 1.0.3), процесс описанный выше будет выглядеть несколько иначе. В частности, приложение проверит, если идентификатор соответствует контроллеру внутри модуля. В случае, если это имеет место, будет создан экземпляр модуля вместе с экземпляром контроллера.

Действие

Как было упомянуто выше, действие — это метод, имя которого начинается на `action`. Более продвинутый способ — создать класс действия и указать контроллеру создавать экземпляр этого класса при необходимости. Такой подход позволяет использовать действия повторно.

Для создания класса действия необходимо выполнить следующее:

```
class UpdateAction extends CAction
{
    public function run()
    {
        // некоторая логика действия
    }
}
```

Чтобы контроллер знал об этом действии, необходимо переопределить метод `actions()` в классе контроллера:

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

В приведенном коде мы используем псевдоним маршрута `application.controllers.post.UpdateAction` для указания на файл класса действия `protected/controllers/post/UpdateAction.php`. Создавая действия, основанные на классах, можно организовать приложение в модульном стиле. Например, следующая структура директорий может быть использована для расположения кода контроллеров:

```
protected/
    controllers/
        PostController.php
        UserController.php
    post/
```

```
CreateAction.php
ReadAction.php
```

```
UpdateAction.php
user/
    CreateAction.php
    ListAction.php
    ProfileAction.php
    UpdateAction.php
```

Привязка параметров действий

Начиная с версии 1.1.4, в Yii появилась поддержка автоматической привязки параметров для действий контроллера. То есть можно задать именованные параметры, в которые автоматически будет попадать соответствующее значение из `$_GET`.

Для того, чтобы лучше объяснить данную возможность, допустим, что нам нужно реализовать действие `create` контроллера `PostController`. Действие требует двух параметров:

- `category`: ID категории, в которой будет создаваться запись. Целое число;
- `language`: строка, содержащая код языка, который будет использоваться в записи.

Скорее всего в результате для получения параметров из `$_GET` у нас получится приведённый ниже скучный код:

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

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

        // ... действительно полезная часть кода ...
    }
}
```

Теперь, используя параметры действий, мы можем получить более приятный код:

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

        // ... действительно полезная часть кода ...
    }
}
```

```

    }
}

```

Мы добавляем два параметра методу `actionCreate`. Имя каждого должно в точности совпадать с одним из ключей в `$_GET`. Параметру `$language` задано значение по умолчанию `en`, которое используется, если в запросе соответствующий параметр отсутствует. Так как `$category` не имеет значения по умолчанию, в случае отсутствия соответствующего параметра в запросе будет автоматически выброшено исключение `CHttpException` (с кодом ошибки 400).

Начиная с версии 1.1.5, Yii поддерживает массивы как параметры действий. Использовать их можно следующим образом:

```

class PostController extends CController
{
    public function actionCreate(array $categories)
    {
        // Yii приведёт $categories к массиву
    }
}

```

Мы добавляем ключевое слово `array` перед параметром `$categories`. После этого, если параметр `$_GET['categories']` является простой строкой, то он будет приведён к массиву, содержащему исходную строку.

Примечание: Если параметр объявлен без указания типа `array`, то он должен быть скалярным (т.е. не массивом). В этом случае передача массива через `$_GET`-параметр приведёт к исключению HTTP.

Фильтры

Фильтр (filter) — это часть кода, который может выполняться до или после выполнения действия контроллера в зависимости от конфигурации. Например, фильтр контроля доступа может проверять, аутентифицирован ли пользователь перед тем, как будет выполнено запрошенное действие. Фильтр, контролирующий производительность, может быть использован для определения времени, затраченного на выполнение действия.

Действие может иметь множество фильтров. Фильтры запускаются в том порядке, в котором они указаны в списке фильтров, при этом фильтр может предотвратить выполнение действия и следующих за ним фильтров.

Фильтр может быть определен как метод класса контроллера. Имя метода должно начинаться на `filter`. Например, существование метода `filterAccessControl` означает, что определен фильтр `accessControl`. Метод фильтра оформляется следующим образом:

```

public function filterAccessControl($filterChain)
{
    // для выполнения последующих фильтров и выполнения действия вызовите метод
    $filterChain->run()
}

```

где `$filterChain` — экземпляр класса `CFilterChain`, представляющего собой список фильтров, ассоциированных с запрошенным действием. В коде фильтра можно вызвать `$filterChain->run()` для того, чтобы продолжить выполнение последующих фильтров и действия.

Фильтр также может быть экземпляром класса `CFilter` или его производного. Следующий код определяет новый класс фильтра:

```

class PerformanceFilter extends CFilter

```

```

{
    protected function preFilter($filterChain)
    {
        // код, выполняемый до выполнения действия
        return true; // false — для случая, когда действие не должно быть выполнено
    }

    protected function postFilter($filterChain)
    {
        // код, выполняемый после выполнения действия
    }
}

```

Для того, чтобы применить фильтр к действию, необходимо переопределить метод `CController::filters()`, возвращающий массив конфигураций фильтров. Например:

```

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

```

Данный код определяет два фильтра: `postOnly` и `PerformanceFilter`. Фильтр `postOnly` задан как метод (соответствующий метод уже определен в `CController`), в то время как `PerformanceFilter` — фильтр на базе класса. Псевдоним `application.filters.PerformanceFilter` указывает на файл класса фильтра — `protected/filters/PerformanceFilter`. Для конфигурации `PerformanceFilter` использован массив, поэтому возможно инициализировать значения свойства фильтра. В данном случае свойство `unit` фильтра `PerformanceFilter` будет инициализировано значением `'second'`. Используя операторы `'+'` и `'-'` можно указать, к каким действиям должен и не должен быть применен фильтр. В приведенном примере `postOnly` должен быть применен к действиям `edit` и `create`, а `PerformanceFilter` — ко всем действиям, кроме `edit` и `create`. Если операторы `'+'` и `'-'` не указаны, фильтр будет применен ко всем действиям.

Модель

Модель (model) — это экземпляр класса `CModel` или производного от него. Модель используется для хранения данных и применимых к ним бизнес-правил.

Модель представляет собой отдельный объект данных. Это может быть запись таблицы базы данных или форма пользовательского ввода. Каждое поле объекта данных представляется атрибутом модели. Каждый атрибут имеет метку и может быть проверен на корректность, используя набор правил.

Yii предоставляет два типа моделей: модель формы (form model) и Active Record. Оба типа являются расширением базового класса CModel.

Модель формы — это экземпляр класса CFormModel. Она используется для хранения данных, введенных пользователем. Как правило, мы получаем эти данные, обрабатываем, а затем избавляемся от них. Например, на странице авторизации модель такого типа может быть использована для представления информации об имени пользователя и пароле. Подробное описание работы с формами приведено в разделе Создание формы.

Active Record (AR) — это шаблон проектирования, используемый для абстрагирования доступа к базе данных в объектно-ориентированной форме. Каждый объект AR является экземпляром класса CActiveRecord или производного от него, представляя отдельную запись в таблице базы данных. Поля строки представлены свойствами AR-объекта. Подробнее с AR-моделью можно ознакомиться в разделе Active Record.

Представление

Представление — это PHP-скрипт, состоящий преимущественно из элементов пользовательского интерфейса. Он может включать выражения PHP, однако рекомендуется, чтобы эти выражения не изменяли данные и оставались относительно простыми. Следуя концепции разделения логики и представления, большая часть кода логики должна быть помещена в контроллер или модель, а не в скрипт представления. У представления есть имя, которое используется, чтобы идентифицировать файл скрипта представления в процессе рендеринга.

Имя представления должно совпадать с названием файла представления. К примеру, для представления `edit` соответствующий файл скрипта должен называться `edit.php`. Чтобы отрендерить представление необходимо вызвать метод `CController::render()`, указав имя представления. При этом метод попытается обнаружить соответствующий файл представления в директории `protected/views/ControllerID`. Внутри скрипта представления экземпляр контроллера доступен через `$this`. Таким образом, мы можем обратиться к свойству контроллера из кода представления: `$this->propertyName`.

Кроме того, мы можем использовать следующий способ для передачи данных представлению:

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

В приведенном коде метод `render()` извлечет второй параметр — массив — в переменные. Как результат, в коде скрипта представления можно обращаться к локальным переменным `$var1` и `$var2`.

Макет

Макет (layout) — это специальное представление для декорирования других представлений. Макет обычно содержит части пользовательского интерфейса, часто используемые другими представлениями. Например, макет может содержать верхнюю и нижнюю части страницы, заключая между ними содержание другого представления.

```
...здесь верхняя часть...
<?php echo $content; ?>
...здесь нижняя...
```

Здесь `$content` хранит результаты рендеринга представления основного содержания.

Макет применяется неявно при вызове метода `render()`. По умолчанию, в качестве макета используется представление `protected/views/layouts/main.php`. Это можно изменить путем установки значений `CWebApplication::layout` или `CController::layout`. Для рендеринга представления без применения макета необходимо вызвать `renderPartial()`.

Виджет

Виджет (widget) — это экземпляр класса CWidget или производного от него. Это компонент, применяемый в основном с целью оформления. Виджеты обычно встраивают в скрипт представления для генерации некоторой комплексной самодостаточной части пользовательского интерфейса. К примеру, виджет календаря может быть использован для рендеринга комплексного интерфейса календаря. Виджеты служат цели повторного использования кода пользовательского интерфейса.

Для использования виджета необходимо выполнить в коде:


```
<?php $this->beginWidget('path.to.WidgetClass'); ?>
...некое содержимое, которое может быть использовано виджетом...
<?php $this->endWidget(); ?>
```

или

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

В последнем варианте для использования виджета не требуется дополнительного содержимого. В случае, если поведение виджета необходимо изменить, можно сделать это путём установки начальных значений его свойств при вызове `CBaseController::beginWidget` или `CBaseController::widget`. Например, при использовании виджета `CMaskedTextField` можно указать используемую маску, передав массив значений свойств, как показано ниже, где ключи массива являются названиями свойств, а значения — требуемыми начальными значениями соответствующих свойств виджета:

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

Чтобы создать новый виджет необходимо расширить класс `CWidget` и перегрузить его методы `init()` и `run()`:

```
class MyWidget extends CWidget
{
    public function init()
    {
        // этот метод будет вызван методом CController::beginWidget()
    }

    public function run()
    {
        // этот метод будет вызван методом CController::endWidget()
    }
}
```

Как и у контроллера, у виджета может быть собственное представление. По умолчанию, файлы представлений виджета находятся в поддиректории `views` директории, содержащей файл класса виджета. Эти представления можно рендерить при помощи вызова `CWidget::render()`, точно так же, как и в случае с контроллером. Единственная разница состоит в том, что для представления виджета не используются макеты. Также, `$this` в представлении указывает на экземпляр виджета, а не на экземпляр контроллера.

Системные представления

Системные представления относятся к представлениям, используемым Yii для отображения ошибок и информации лога. Например, когда пользователь запрашивает несуществующий контроллер или действие, Yii сгенерирует исключение, раскрывающее суть ошибки. Такое исключение будет отображено с помощью системного представления.

Именование системных представлений подчиняется некоторым правилам. Имена типа `errorxxx` относятся к представлениям, служащим для отображения `CHttpException` с кодом ошибки `xxx`. Например, если исключение `CHttpException` сгенерировано с кодом ошибки 404, будет использовано представление `error404`.

Yii предоставляет стандартный набор системных представлений, расположенных в `framework/views`. Их можно видоизменить путем создания файлов представлений с теми же названиям в директории `protected/views/system`.

Компонент

Yii-приложения состоят из компонентов—объектов, созданных согласно спецификациям. Компонент (component) — это экземпляр класса CComponent или производного от него. Использование компонента в основном включает доступ к его свойствам, а также вызов и обработку его событий. Базовый класс CComponent устанавливает то, как определяются свойства и события.

Свойство компонента

Свойство компонента схоже с публичной переменной-членом класса (public member variable). Мы можем читать или устанавливать его значение. Например:

```
$width=$component->textWidth; // получаем значение свойства textWidth
$component->enableCaching=true; // устанавливаем значение свойства enableCaching
```

Для того, чтобы создать свойство компонента необходимо просто объявить публичную переменную в классе компонента. Более гибкий вариант — определить методы, считывающие (getter) и записывающие (setter) это свойство, например:

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

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

В приведенном коде определено свойство `textWidth` (имя нечувствительно к регистру), доступное для записи.

При чтении вызывается метод чтения `getTextWidth()`, возвращающий значение свойства. Соответственно, при записи будет вызван метод записи `setTextWidth()`. Если метод записи не определен, свойство будет доступно только для чтения, а при попытке записи будет вызвано исключение. Использование методов чтения и записи имеет дополнительное преимущество: при чтении или записи значения свойства могут быть выполнены дополнительные действия (такие как проверка на корректность, вызов события и др.).

Примечание: Есть небольшая разница в определении свойства через методы и через простое объявление переменной. В первом случае имя свойства нечувствительно к регистру, во втором — чувствительно.

События компонента

События компонента — это специальные свойства, в качестве значений которых выступают методы (называемые обработчиками событий). Прикрепление метода к событию приведет к тому, что метод будет вызван автоматически при возникновении события. Поэтому поведение компонента может быть изменено совершенно отлично от закладываемого при разработке.

Событие компонента задается путем создания метода с именем, начинающимся на `on`. Подобно именам свойств, заданных через методы чтения и записи, имена событий не чувствительны к регистру. Следующий код задает событие `onClicked`:

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

```
}
```

где `$event` — это экземпляр класса `CEvent` или производного от него, представляющего параметр события. К событию можно подключить обработчик, как показано ниже:

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

где `$callback` — это корректный callback-вызов PHP (см. PHP-функцию `call_user_func`). Это может быть либо глобальная функция, либо метод класса. В последнем случае вызову должен передаваться массив: `array($object, 'methodName')`.

Обработчик события должен быть определен следующим образом:

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

где `$event` — это параметр, описывающий событие (происходит из вызова `raiseEvent()`). Параметр `$event` — это экземпляр класса `CEvent` или его производного. Как минимум, он содержит информацию о том, кто вызвал событие.

Начиная с версии 1.0.10 обработчик события может быть анонимной функцией, требующей наличия версии PHP 5.3+. Например,

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

Если теперь вызвать `onClicked()`, событие `onClicked` будет вызвано (внутри `onClicked()`), и прикрепленный обработчик события будет запущен автоматически.

К событию может быть прикреплено несколько обработчиков. При возникновении события обработчики будут вызваны в том порядке, в котором они были прикреплены к событию. Если в обработчике необходимо предотвратить вызов последующих обработчиков, необходимо установить `$event->handled` в `true`.

Поведение компонента

Начиная с версии 1.0.2, к компоненту была добавлена поддержка примесей (`mixin`) и теперь к компоненту можно прикрепить одно или несколько поведений. *Поведение* — это объект, чьи методы могут быть «унаследованы» компонентом, к которому прикреплены, посредством объединения функционала вместо четкой специализации (как в случае обычного наследования класса). К компоненту можно прикрепить несколько поведений и таким образом получить множественное наследование.

Поведение классов должно реализовывать интерфейс `IBehavior`. Большинство поведений могут быть созданы путем расширения базового класса `CBehavior`. В случае, если поведение необходимо прикрепить к модели, его можно создать на основе класса `CModelBehavior` или класса `CActiveRecordBehavior`, который реализует дополнительные, специфические для модели возможности.

Чтобы воспользоваться поведением, его необходимо прикрепить к компоненту путем вызова метода поведения `attach()`. Далее мы вызываем метод поведения через компонент:

```
// $name уникально идентифицирует поведения в компоненте
$component->attachBehavior($name,$behavior);
// test() является методом $behavior
$component->test();
```

К прикрепленному поведению можно обращаться, как к обычному свойству компонента. Например, если поведение с именем `tree` прикреплено к компоненту, мы можем получить ссылку на этот объект поведения следующим образом:

```
$behavior=$component->tree;
// эквивалентно выражению:
```

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

Поведение можно временно деактивировать таким образом, чтобы его методы были недоступны через компонент. Например:

```
$component->disableBehavior($name);
// выражение ниже приведет к вызову исключения
$component->test();
$component->enableBehavior($name);
// здесь все будет работать нормально
$component->test();
```

В случае, когда два поведения, прикрепленные к одному компоненту, имеют методы с одинаковыми именами, преимущество будет иметь метод поведения, которое было прикреплено раньше. Использование поведений совместно с событиями дает дополнительные возможности. Поведение, прикрепленное к компоненту, может присваивать некоторые свои методы событиям компонента. В этом случае, поведение получает возможность следить или менять нормальный ход выполнения компонента. Начиная с версии 1.1.0, свойства поведения также могут быть доступны из компонента, к которому оно привязано. Свойства включают в себя как открытые, так и определенные через геттеры и/или сеттеры поведения. Например, поведение имеет свойство с именем `xyz` и привязано к компоненту `$a`. Тогда мы можем использовать выражение `$a->xyz` для доступа к свойству.

Модуль

Примечание: Поддержка модулей доступна, начиная с версии 1.0.3.

Модуль — это самодостаточная программная единица, состоящая из моделей, представлений, контроллеров и иных компонентов. Во многом модуль схож с приложением. Основное различие заключается в том, что модуль не может использоваться сам по себе — только в составе приложения. Пользователи могут обращаться к контроллерам внутри модуля абсолютно так же, как и в случае работы с обычными контроллерами приложения.

Модули могут быть полезными в нескольких ситуациях. Если приложение очень объемное, мы можем разделить его на несколько модулей, разрабатываемых и поддерживаемых по отдельности. Кроме того, некоторый часто используемый функционал, например, управление пользователями, комментариями и пр., может разрабатываться как модули, чтобы впоследствии можно было с легкостью воспользоваться им вновь.

Создание модуля

Модуль организован как директория, имя которой выступает в качестве уникального идентификатора модуля. Структура директории модуля похожа на структуру базовой директории приложения. Ниже представлена типовая структура директории модуля с именем `forum`:

<code>forum/</code>	
<code>ForumModule.php</code>	файл класса модуля
<code>components/</code>	содержит компоненты пользователя
<code>views/</code>	содержит файлы представлений для виджетов
<code>controllers/</code>	содержит файлы классов контроллеров
<code>DefaultController.php</code>	файл класса контроллера по умолчанию
<code>extensions/</code>	содержит сторонние расширения
<code>models/</code>	содержит файлы классов моделей

views/ layouts/	содержит файлы представлений контроллера и макетов содержит файлы макетов
default/ index.php	содержит файлы представлений для контроллера по умолчанию файл представления 'index'

Модуль должен иметь класс модуля, унаследованный от класса `CWebModule`. Имя класса определяется с использованием выражения `ucfirst($id) . 'Module'`, где `$id` соответствует идентификатору модуля (или названию директории модуля). Класс модуля выполняет роль центрального места для хранения совместно используемой информации. Например, мы можем использовать `CWebModule::params` для хранения параметров модуля, а также `CWebModule::components` для совместного использования компонентов приложения на уровне модуля.

Подсказка: Для создания простого каркаса модуля можно воспользоваться генератором модулей, входящим в состав Gii.

Использование модуля

Для использования модуля необходимо поместить папку модуля в директорию `modules` базовой директории приложения. Далее необходимо объявить идентификатор модуля в свойстве приложения `modules`. Например, чтобы воспользоваться модулем `forum`, приведенным выше, можно использовать следующую конфигурацию приложения:

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

Кроме того, модулю можно задать начальные значения свойств. Порядок использования аналогичен порядку с компонентами приложения. Например, модуль `forum` может иметь в своем классе свойство с именем `postPerPage`, которое может быть установлено в конфигурации приложения следующим образом:

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

К экземпляру модуля можно обращаться посредством свойства `module` активного в настоящий момент контроллера. Через экземпляр модуля можно получить доступ к совместно используемой информации на уровне модуля. Например, для того, чтобы обратиться к упомянутому выше свойству `postPerPage`, мы можем воспользоваться следующим выражением:

```
$postPerPage = Yii::app()->controller->module->postPerPage;
// или таким, если $this ссылается на экземпляр контроллера
// $postPerPage = $this->module->postPerPage;
```

Обратиться к действию контроллера в модуле можно, используя маршрут `moduleID/controllerID/actionID`. Например, предположим, что все тот же модуль `forum` имеет контроллер с именем `PostController`, тогда мы можем использовать маршрут `forum/post/create` для того, чтобы обратиться к действию `create` этого контроллера. Адрес URL, соответствующий этому маршруту, будет таким: `http://www.example.com/index.php?r=forum/post/create`.

Подсказка: Если контроллер находится в подпапке папки `controllers`, мы также можем использовать формат маршрута, приведенный выше. Например, предположим, что контроллер `PostController` находится в папке `forum/controllers/admin`, тогда мы можем обратиться к действию `create` через `forum/admin/post/create`.

Вложенные модули

Модули могут быть вложенными друг в друга сколько угодно раз, т.е. один модуль может содержать в себе другой, который содержит в себе ещё один. Первый мы будем называть *модуль-родитель*, второй — *модуль-потомок*. Модули-потомки должны быть описаны в свойстве `modules` модуля-родителя точно так же, как мы описываем модули в файле конфигурации приложения. Для обращения к действию контроллера в дочернем модуле используется маршрут `parentModuleID/childModuleID/controllerID/actionID`.

Псевдоним пути и пространство имен

Псевдонимы пути широко используются в Yii. Псевдоним ассоциируется с директорией или путем к файлу. При его указании используется точечный синтаксис, схожий с широко используемым форматом пространств имен:

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

где `RootAlias` — псевдоним существующей директории.

При помощи `YiiBase::getPathOfAlias()` мы можем преобразовать псевдоним в соответствующий ему путь. К примеру, `system.web.CController` будет преобразован в `yii/framework/web/CController`.

Также мы можем использовать `YiiBase::setPathOfAlias()` для определения новых корневых псевдонимов.

Корневой псевдоним

Для удобства, следующие системные псевдонимы уже предопределены:

- **system**: соответствует директории фреймворка;
- **zii**: соответствует директории библиотеки расширений Zii;
- **application**: соответствует базовой директории приложения;
- **webroot**: соответствует директории, содержащей входной скрипт. Этот псевдоним доступен, начиная с версии 1.0.3.
- **ext**: соответствует директории, содержащей все сторонние расширения. Этот псевдоним доступен, начиная с версии 1.0.8.

Кроме того, в случае, если приложение использует модули, системный псевдоним предопределяется для каждого идентификатора модуля и соответствует пути к этому модулю. Эта возможность доступна, начиная с версии 1.0.3.

Importing Classes

Используя псевдонимы, очень удобно импортировать описания классов. К примеру, для подключения класса `CController` можно вызвать:

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

Использование метода `import` более эффективно, чем `include` и `require`, поскольку описание импортируемого класса не будет включено до первого обращения (реализовано через механизм автозагрузки классов PHP). Импорт одного и того же пространства имён также происходит намного быстрее, чем при использовании `include_once` и `require_once`.

Подсказка: Если мы ссылаемся на класс фреймворка, то нет необходимости импортировать или включать их. Все системные классы Yii уже импортированы заранее.

Использование таблицы классов

Начиная с версии 1.1.5, Yii позволяет предварительно импортировать пользовательские классы через тот же механизм, что используется для классов ядра. Такие классы могут использоваться где угодно в приложении без необходимости их предварительного импорта или подключения. Данная возможность отлично подходит для фреймворка или библиотеки, использующих Yii.

Для импорта набора классов, выполните следующий код до вызова `CWebApplication::run()`:

```
Yii::$classMap=array(
    'ClassName1' => 'path/to/ClassName1.php',
    'ClassName2' => 'path/to/ClassName2.php',
    .....
);
```

Импорт директорий

Можно использовать следующий синтаксис для того, чтобы импортировать целую директорию, а файлы классов, содержащиеся в директории, будут подключены автоматически при необходимости.

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

Помимо `import`, псевдонимы также используются во многих других местах, где есть ссылки на классы. Например, псевдоним может быть передан методу `Yii::createComponent()` для создания экземпляра соответствующего класса, даже если этот класс не был предварительно включен.

Пространство имён

Пространства имен служат для логической группировки имен классов, чтобы их можно было отличить от других, даже если их имена совпадают. Не путайте псевдоним пути с пространством имён. Псевдоним пути — всего лишь удобный способ именования файлов и директорий. К пространствам имён он не имеет никакого отношения.

Подсказка: Так как версии PHP до 5.3.0 не поддерживают пространства имен, вы не можете создать экземпляры классов с одинаковыми именами, но различными описаниями. По этой причине все названия классов Yii-фреймворка имеют префикс 'C' (означающий 'class'), чтобы их можно было отличить от пользовательских классов. Для пользовательских классов рекомендуется использовать другие префиксы, сохранив префикс 'C' зарезервированным для Yii-фреймворка.

Классы в пространствах имён

Класс в пространстве имён — любой класс, описанный в неглобальном пространстве имён. К примеру, класс `applicationcomponentsGoogleMap` описан в пространстве имён `applicationcomponents`.

Использование пространств имён требует PHP 5.3.0 и выше.

Начиная с версии 1.1.5 стало возможным использование класс из пространства имён без его предварительного подключения. К примеру, мы можем создать новый экземпляр `applicationcomponentsGoogleMap` без явного подключения соответствующего файла. Это реализуется при помощи улучшенного загрузчика классов Yii.

Для того, чтобы автоматически подгрузить класс из пространства имён, пространство имён должно быть названо в том же стиле, что и псевдонимы пути. Например, класс `applicationcomponentsGoogleMap` должен храниться в файле, которому соответствует псевдоним `application.components.GoogleMap`.

Соглашения

Yii ставит соглашения выше конфигураций. Следуя соглашениям, вы сможете создавать серьезные приложения без необходимости написания и поддержки сложных конфигураций. Конечно же, при необходимости Yii может быть изменен с помощью конфигураций практически как угодно.

Ниже представлены соглашения, рекомендуемые для программирования под Yii. Для удобства примем, что `webRoot` — это директория, в которую установлено приложение.

URL

По умолчанию, Yii понимает адреса URL следующего формата:


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

GET-переменная `r` представляет маршрут, из которого Yii извлекает информацию о контроллере и действии. Если `ActionID` не указан, контроллер будет использовать действие по умолчанию (определенный в свойстве `CController::defaultAction`). Если же `ControllerID` также не указан (либо переменная `r` отсутствует), будет использован контроллер по умолчанию (определенный в свойстве `CWebApplication::defaultController`).

Используя `CUrlManager` можно создавать и применять более SEO-дружественные адреса URL, такие как `http://hostname/ControllerID/ActionID.html`. Эта возможность подробно описана в разделе Красивые адреса URL.

Код

Yii рекомендует именовать переменные, функции и классы, используя ГорбатыйРегистр, что подразумевает написание каждого слова в имени с большой буквы и соединение их без пробелов. Первое слово в имени переменных и функций должно быть написано в нижнем регистре, чтобы отличать их от имен классов (например, `$basePath`, `runController()`, `LinkPager`). Для имён свойств класса с видимостью `private` рекомендуется использовать знак подчеркивания в качестве префикса (например, `$_actionList`). Поскольку пространства имен не поддерживаются версиями PHP до 5.3.0, рекомендуется, чтобы имена классов были уникальными во избежание конфликта имен с классами сторонних производителей. По этой причине все имена классов фреймворка имеют префикс "C".

Особое правило для имен классов контроллеров — они должны быть дополнены словом `Controller`. При этом идентификатором контроллера будет имя класса с первой буквой в нижнем регистре и без слова `Controller`. Например, для класса `PageController` идентификатором будет `page`. Данное правило делает приложение более защищенным. Оно также делает адреса URL более понятными (к примеру, `/index.php?r=page/index` вместо `/index.php?r=PageController/index`).

Конфигурация

Конфигурация — это массив пар ключ-значение, где каждый ключ представляет собой имя свойства конфигурируемого объекта, а значение — начальное значение соответствующего свойства. К примеру, `array('name'=>'My application', 'basePath'=>'./protected')` инициализирует свойства `name` и `basePath` соответствующими значениями.

Любые свойства объекта, которые доступны для записи, могут быть сконфигурированы. Если некоторые свойства не сконфигурированы, для них будут использованы значения по умолчанию. При конфигурировании свойств рекомендуется изучить соответствующий раздел документации, чтобы избежать задания некорректных значений.

Файл

Соглашения для именования и использования файлов зависят от их типов. Файлы классов должны быть названы так же, как и общие классы, содержащиеся в них. Например, класс `CController` находится в файле `CController.php`. Общий класс — это класс, который может быть использован любыми другими классами. Каждый файл классов должен содержать максимум один общий класс. Приватные классы (классы, которые могут быть использованы только одним общим классом) должны находиться в одном файле с общим классом.

Файлы представлений должны иметь такие же имена, как и содержащиеся в них представления. К примеру, представление `index` находится в файле `index.php`. Файл представления — это PHP-скрипт, содержащий HTML и PHP-код, в основном для задач отображения пользовательского интерфейса.

Конфигурационные файлы могут именоваться произвольным образом. Файл конфигурации — это PHP-скрипт, чье единственное назначение — возвращать ассоциативный массив, представляющий конфигурацию.

Директория

В Yii предопределен набор директорий для различных целей. Каждая из них может быть изменена при необходимости.

- `WebRoot/protected`: это базовая директория приложения, содержащая все наиболее важные с точки зрения безопасности PHP-скрипты и файлы данных. Псевдоним по умолчанию для этого пути — `application`. Эта директория и ее содержимое должно быть защищено от прямого доступа из веб. Директория может быть настроена через `CWebApplication::basePath`;
- `WebRoot/protected/runtime`: эта директория содержит приватные временные файлы, сгенерированные во время выполнения приложения. Эта директория должна быть доступна для записи вебсервером. Она может быть настроена через `CApplication::runtimePath`;

- `WebRoot/protected/extensions`: эта директория содержит все сторонние расширения. Она может быть настроена через `CApplication::extensionPath`;
- `WebRoot/protected/modules`: эта директория содержит все модули приложения, каждый из которых представлен в отдельной поддиректории. Директория может быть настроена через `CWebApplication::modulePath`;
- `WebRoot/protected/controllers`: эта директория содержит файлы всех классов контроллеров. Она может быть настроена через `CWebApplication::controllerPath`;
- `WebRoot/protected/views`: эта директория содержит файлы всех представлений, включая представления контроллеров, макеты и системные представления. Она может быть настроена через `CWebApplication::viewPath`;
- `WebRoot/protected/views/ControllerID`: эта директория содержит файлы представлений для отдельного класса контроллера. Здесь `ControllerID` является идентификатором контроллера. Директория может быть настроена через `CController::viewPath`;
- `WebRoot/protected/views/layouts`: эта директория содержит файлы макетов. Она может быть настроена через `CWebApplication::layoutPath`;
- `WebRoot/protected/views/system`: эта директория содержит файлы системных представлений (используются для отображения сообщений об ошибках и исключениях). Она может быть настроена через `CWebApplication::systemViewPath`;
- `WebRoot/assets`: эта директория содержит файлы ресурсов (приватные файлы, которые могут быть опубликованы для доступа к ним из веб). Директория должна быть доступна для записи процессами веб-сервера. Она может быть настроена через `CAssetManager::basePath`;
- `WebRoot/themes`: эта директория содержит различные темы оформления, которые доступны приложению. Каждая поддиректория содержит отдельную тему с именем, совпадающим с названием поддиректории. Директория может быть настроена через `CThemeManager::basePath`.

База данных

Большинство приложений хранят данные в БД. Мы предлагаем соглашения для таблиц и атрибутов БД. Стоит отметить, что Yii не требует строгого следования им.

- Таблицы и атрибуты именуются в нижнем регистре.
- Слова в названии разделяются подчёркиванием (например, `product_order`).
- В именах таблиц используется либо единственное число, либо множественное, но не оба сразу. Мы рекомендуем использовать единственное число.
- Имена таблиц могут содержать префикс. Например, `tbl_`. Это особенно полезно когда таблицы нашего приложения находятся в БД, используемой одновременно другими приложениями.

Процесс разработки

Рассказав фундаментальные концепции Yii, мы опишем общий процесс создания веб-приложений с использованием фреймворка. Процесс подразумевает, что анализ требований уже проведен, так же, как и необходимый анализ устройства приложения.

1. Создание структуры директорий. Утилита `yii`, описанная в разделе «создание первого приложения», может быть использована для того, чтобы ускорить этот процесс;
2. Конфигурирование приложения путем модификации файла конфигурации приложения. Этот этап также может потребовать написания некоторых компонентов приложения (например, компонент управления пользователями);
3. Создание класса модели для каждого используемого типа данных. Для автоматической генерации всех интересующих вас моделей `active record` можно воспользоваться инструментом `Gii`, описанным в разделах «создание первого приложения» и «автоматическая генерация кода»;
4. Создание класса контроллера для каждого типа пользовательского запроса. Классификация пользовательских запросов зависит от текущих требований. В общем случае, если класс модели используется пользователем, должен существовать соответствующий класс контроллера. Утилита `Gii` также может автоматизировать этот процесс;
5. Создание действий и их представлений. Именно здесь и делается основная работа;
6. Конфигурирование необходимых фильтров для действий в классах контроллеров;
7. Создание тем оформления при необходимости;
8. Перевод сообщений в случае, когда требуется локализация;
9. Выявление данных и представлений, которые могут быть закешированы и применение соответствующих техник кэширования.
10. Настройка производительности и развёртывание.

Для каждого из представленных этапов может потребоваться создание и применение тестов.

Работа с формами

Создание модели

Прежде, чем писать HTML код для формы, нам нужно определить, какие данные мы будем получать от пользователей и каким правилам они должны соответствовать. Для фиксации этой информации можно использовать класс модели данных. Модель данных, как говорится в разделе Модель, это центральное место для хранения и проверки данных, вводимых пользователем.

В зависимости от того, каким образом используются данные ввода, мы можем создать два типа модели данных. Если мы получаем данные, обрабатываем их, а затем удаляем, то используем модель формы; если же получаем данные и сохраняем их в базу данных, используем Active Record. Оба типа моделей данных используют один и тот же базовый класс CModel, который определяет общий интерфейс используемый формой.

Примечание: В примерах этого раздела используются модели формы. Тем не менее, всё может быть в равной степени применено и к моделям Active Record.

Определение класса модели

Ниже мы создадим класс модели `LoginForm`, который будет использоваться для получения данных, вводимых пользователем на странице авторизации. Поскольку эти данные используются исключительно в целях аутентификации пользователя и сохранять их не требуется, то создадим модель `LoginForm` как модель формы.

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

Мы объявили в `LoginForm` три атрибута: `$username`, `$password` и `$rememberMe`. Они используются для хранения имени пользователя, пароля, а также значения опции сохранения имени пользователя. Так как `$rememberMe` по умолчанию имеет значение `false`, то изначально в форме авторизации галочка этой опции будет снята.

Информация: Термин «атрибут» используется, чтобы отделить свойства, определяемые этими переменными-членами класса, от прочих свойств. Здесь атрибут — это свойство, которое используется в основном для хранения данных, вводимых пользователем, или данных, получаемых из базы данных.

Определение правил проверки

В момент, когда пользователь отправляет данные формы, а модель их получает, нам необходимо удостовериться, что эти данные корректны, прежде, чем мы будем их использовать. Это осуществляется посредством проверки данных в соответствии с набором правил. Правила проверки задаются в методе `rules()`, который возвращает массив сконфигурированных правил.

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

    private $_identity;
```

```

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

public function authenticate($attribute,$params)
{
    $this->_identity=new UserIdentity($this->username,$this->password);
    if(!$this->_identity->authenticate())
        $this->addError('password','Неправильное имя пользователя или пароль.');
```

В коде, представленном выше, `username` и `password` — обязательные для заполнения поля, поле `password` должно быть проверено также на соответствие указанному имени пользователя. Поле `rememberMe` может принимать значения `true` или `false`.

Каждое правило, возвращаемое `rules()`, должно быть задано в следующем формате:

```
array('AttributeList', 'Validator', 'on'=>'ScenarioList', ...дополнительные параметры)
```

где `AttributeList` — строка с именами атрибутов, отделенных запятыми, которые должны быть проверены в соответствии с правилами; `Validator` указывает на тип используемой проверки; параметр `on` — необязательный параметр, устанавливающий список сценариев, где должно использоваться правило; а также прочие параметры — пары имя-значение, которые используются для инициализации значений свойств соответствующего валидатора.

Есть три способа указать `Validator` в правиле проверки. Во-первых, `Validator` может быть именем метода в классе модели данных, аналогично `authenticate` в примере выше. Метод проверки оформляется следующим образом:

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

Второй способ — указать `Validator` в качестве имени класса. В этом случае для проверки данных в момент применения правила создается экземпляр класса проверки. Дополнительные параметры в правиле используются для инициализации значений атрибутов экземпляра. Класс проверки должен быть производным классом от `CValidator`.

Третий вариант — предопределить псевдоним класса валидатора. В примере выше, имя `required` — это псевдоним класса `CRequiredValidator`, который проверяет, чтобы проверяемое значение атрибута не было пустым. Ниже приведен полный список предопределенных псевдонимов валидаторов, включенных в состав Yii:

- `boolean`: псевдоним класса `CBooleanValidator`, который проверяет, чтобы атрибут имел значение либо `CBooleanValidator::trueValue` либо `CBooleanValidator::falseValue`;

- `captcha`: псевдоним класса `CCaptchaValidator`, который проверяет, чтобы значение атрибута было равно коду верификации на капче;
- `compare`: псевдоним класса `CCompareValidator`, который проверяет, чтобы значение атрибута совпадало со значением другого атрибута или константой;
- `email`: псевдоним класса `CEmailValidator`, который отвечает за проверку корректности email адреса;
- `default`: псевдоним класса `CDefaultValueValidator`, который присваивает значение по умолчанию выбранным атрибутам;
- `exist`: псевдоним класса `CExistValidator`, который проверяет наличие значения атрибута в указанном столбце таблицы;
- `file`: псевдоним класса `CFileValidator`, отвечающего за проверку атрибута на наличие в нем имени загруженного файла;
- `filter`: псевдоним класса `CFilterValidator`, преобразовывающего атрибут с использованием фильтра;
- `in`: псевдоним класса `CRangeValidator`, который проверяет, содержатся ли данные в заданном наборе значений;
- `length`: псевдоним класса `CStringValidator`, который проверяет соответствует ли длина данных заданному интервалу;
- `match`: псевдоним класса `CRegularExpressionValidator`, проверяющего данные на соответствие регулярному выражению;
- `numerical`: псевдоним класса `CNumberValidator`, проверяющего, являются ли данные корректным числовым значением;
- `required`: псевдоним класса `CRequiredValidator`, который проверяет, чтобы значение атрибута не было пустым;
- `type`: псевдоним класса `CTypeValidator`, проверяющего соответствие атрибута заданному типу данных;
- `unique`: псевдоним класса `CUniqueValidator`, который проверяет, являются ли данные уникальными в пределах поля базы данных;
- `url`: псевдоним класса `CUrlValidator`, отвечающего за проверку корректности URL.

Ниже представлены несколько примеров использования предопределенных валидаторов:

```
// имя пользователя – обязательное поле формы
array('username', 'required'),
// длина имени пользователя должна быть от 3 до 12 символов включительно
array('username', 'length', 'min'=>3, 'max'=>12),
// в сценарии регистрации значения полей «password» и «password2» должны быть равны
array('password', 'compare', 'compareAttribute'=>'password2', 'on'=>'register'),
// в сценарии авторизации поле `password` должно быть проверено на соответствие указанному
// имени пользователя
array('password', 'authenticate', 'on'=>'login'),
```

Безопасное присваивание значений атрибутам

После того, как создан экземпляр модели данных, нам часто требуется заполнить его данными, которые ввел пользователь. Это можно проделать легко и непринужденно, используя массовое присваивание:

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

Последнее выражение в примере как раз и является *массовым присваиванием*, где значение каждой переменной в `$_POST['LoginForm']` присваивается соответствующему атрибуту модели. Это эквивалентно следующей операции:

```
foreach($_POST['LoginForm'] as $name=>$value)
{
```

```
if($name является безопасным атрибутом)
    $model->$name=$value;
```

```
}
```

Очень важно определить, какие атрибуты являются безопасными. Например, если мы сделаем первичный ключ таблицы безопасным, злоумышленник сможет получить шанс его изменить и, таким образом, изменить данные, которые он не должен менять, поскольку не авторизован для этого. Политика выбора безопасных атрибутов отличается в версиях 1.0 и 1.1. В дальнейшем мы расскажем о них по отдельности.

Безопасные атрибуты в версии 1.1

В версии 1.1 атрибут считается безопасным, если он появляется в правиле валидации, применяемом в данном сценарии. Например,

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

В коде выше атрибуты `username` и `password` необходимы в сценарии `login`, а атрибуты `username`, `password` и `email` — в сценарии `register`. В результате, если мы проводим массовое присваивание в сценарии `login`, то только атрибуты `username` и `password` будут массово присвоены, т.к. только они входят в правило валидации для сценария `login`. С другой стороны, если текущим сценарием является `register`, то все три атрибута могут быть массово присвоены.

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

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

Так почему же мы используем именно такую политику для определения, является ли атрибут безопасным или нет? Если атрибут уже есть в одном или нескольких правилах валидации, зачем беспокоиться о чём-то ещё?

Важно помнить, что правила валидации используются для проверки введенных пользователем данных, а не данных, которые мы генерируем в коде (например, текущее время или автоматически сгенерированный первичный ключ). Поэтому, НЕ ДОБАВЛЯЙТЕ правила валидации для атрибутов, не являющихся введенными конечным пользователем.

Иногда мы хотим объявить атрибут безопасным даже если в действительности не имеем правила для него. Пример — атрибут содержания статьи, который может принимать любые введенные пользователем данные. Для этого мы можем использовать специальное правило `safe`:

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

Для полноты картины, существует также правило `unsafe`, используемое для явного указания небезопасного атрибута:

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

Правило `unsafe` используется редко и является противоположностью описанному нами ранее определению безопасных атрибутов.

Безопасные атрибуты в версии 1.0

В версии 1.0 безопасность входных данных задается при помощи метода `safeAttributes` и конкретного сценария. По умолчанию метод возвращает как безопасные все публичные переменные-члены для `CFormModel` и все поля таблицы, кроме первичного ключа, — для `CActiveRecord`. Мы можем переопределить этот метод, чтобы ограничить безопасные атрибуты в соответствии с требованиями сценария. Например, модель, описывающая пользователя, может содержать множество различных атрибутов, но в сценарии авторизации нам требуются только атрибуты `username` и `password`. Обозначить это можно следующим образом:

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

Будет аккуратнее, если структуру значения, возвращаемого методом `safeAttributes` оформить так:

```
array(
    // эти атрибуты могут быть *массово присвоены* в любом сценарии,
    // кроме специально обозначенных ниже
    'attr1, attr2, ...',
    *
    // эти атрибуты можно *присвоить массово* только в сценарии 'scenario1'
    'scenario1' => 'attr2, attr3, ...',
    *
    // а эти — только в сценарии 'scenario2'
    'scenario2' => 'attr1, attr3, ...',
)
```

Если модель не завязана на сценарии (т.е. используется только в одном сценарии или же все сценарии работают с единым набором безопасных атрибутов), тогда возвращаемое значение можно упростить до одной строки:

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

В случае небезопасных входных данных, мы должны присваивать значения соответствующим атрибутам, используя отдельные операции присваивания, как представлено ниже:

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

Выполнение проверки

Как только модель заполнена пользовательскими данными, вызываем метод `CModel::validate()`, чтобы запустить процесс проверки. По итогам проверки метод возвращает положительный или отрицательный результат. Для моделей `CActiveRecord` проверку можно выполнять автоматически в момент вызова метода `CActiveRecord::save()`.

Мы можем задать сценарий через свойство `scenario` и, с его помощью, задать, какой набор правил использовать для проверки.

Проверка выполняется в зависимости от сценария. Свойство `scenario` задаёт сценарий, в котором применяется модель и определяет какой набор правил валидации использовать. К примеру, в сценарии `login`, мы хотим проверить только поля модели пользователя `username` и `password`. В сценарии

`register` нам необходимо проверять большее количество данных: `email`, `address` и других. Ниже показано, как провести проверку для сценария `register`:

```
// создаём модель User и задаём её сценарий как `register`. Выражение ниже
эквивалентно следующему:
// $model=new User;
// $model->scenario='register';
$model=new User('register');

// наполняем модель данными
$model->attributes=$_POST['User'];

// проводим валидацию
if($model->validate()) // если данные верны
    ...
else
    ...
```

Сценарий для правил проверки задаётся в свойстве `on` правила. Если `on` не определено, правило используется для всех сценариев. Например,

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

Первое правило будет распространяться на любые сценарии, а два последующих будут применяться только к сценарию `register`.

Информация об ошибках

После проверки все возможные ошибки находятся в объекте модели. Мы можем получить их через `CModel::getErrors()` и `CModel::getError()`. Первый метод возвращает все ошибки для указанного атрибута модели, второй — *только первую* ошибку.

Чтобы узнать, возникли ли во время выполнения проверки какие-либо ошибки, можно воспользоваться методом `CModel::hasErrors()`. И если ошибки действительно есть, то получить их можно с помощью метода `CModel::getErrors()`. Оба метода могут быть использованы как для всех, так и для конкретного атрибута.

Метки атрибутов

Часто при работе с формами для каждого поля требуется отображать его метку. Она подсказывает пользователю, какие данные ему требуется ввести в поле. Мы, конечно, можем задать метки полей в представлении, но, если указать их непосредственно в модели данных, мы выиграем в удобстве и гибкости. По умолчанию `CModel` в качестве меток возвращает названия атрибутов. Изменить их можно, если переопределить метод `attributeLabels()`.

Далее мы увидим, что возможность указания меток в модели данных позволяет быстро создавать сложные формы.

Создание формы

Написание формы не должно вызвать никаких затруднений. Мы начинаем с тега `form`, атрибут `action` которого должен содержать URL действия `login`, рассмотренного ранее. Затем добавляем метки и поля

ввода для атрибутов, объявленных в классе `LoginForm`. В завершении мы вставляем кнопку отправки данных формы. Все это без проблем пишется на чистом HTML коде. Для упрощения процесса создания формы Yii предоставляет несколько классов помощников (helper). Например, для создания текстового поля, можно вызвать метод `CHtml::textField()`, для выпадающего списка — `CHtml::dropDownList()`.

Информация: Безусловно, может возникнуть справедливый вопрос, а в чем преимущество использования помощника, если объем используемого кода сравним с чистым HTML кодом? Ответ прост: использование помощника дает большие возможности. Например, код, приведенный ниже, создает текстовое поле, отправляющее данные формы на сервер, когда пользователь меняет её значение.

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

Заметьте, что все реализовано без единой строчки JavaScript.

Ниже мы создаем представление — форму авторизации — с помощью класса `CHtml`. Здесь переменная `$model` — экземпляр класса `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('Войти'); ?>
</div>

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

Форма, которую мы создали выше, обладает куда большей динамичностью. К примеру, `CHtml::activeLabel()` создает метку, соответствующую атрибуту модели, а если при вводе данных была допущена ошибка, то CSS класс метки сменится на `error`, сменив внешний вид метки на соответствующий CSS стиль. Похожим образом метод `CHtml::activeTextField()` создает текстовое поле для соответствующего атрибута модели и графически выделяет ошибки ввода.

Если использовать CSS стиль `form.css`, предоставляемый скриптом `yii`, созданная форма будет выглядеть так:

Страница авторизации

Username

Password

☐ Remember me next time

Login

Страница авторизации с сообщением об ошибке

Please fix the following input errors:

- Username cannot be blank.
- Password cannot be blank.

Username

Password

☐ Remember me next time

Login

Начиная с версии 1.1.1, для создания форм можно воспользоваться новым виджетом CActiveForm, который позволяет реализовать валидацию как на клиенте, так и на сервере. При использовании CActiveForm код отображения будет выглядеть следующим образом:

```
<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('Войти'); ?>

    </div>
```

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

Использование построителя форм

При создании HTML форм часто приходится писать довольно большое количество повторяющегося кода, который почти невозможно использовать в других проектах. К примеру, для каждого поля ввода нам необходимо вывести описание и возможные ошибки валидации. Для того, чтобы сделать возможным повторное использование подобного кода, можно использовать построитель форм, доступный с версии Yii 1.1.0.

Общая идея

Построитель форм использует объект CForm для описания параметров, необходимых для создания HTML формы, таких как модели и поля, используемые в форме, а также параметры построения самой формы. Разработчику достаточно создать объект CForm, задать его параметры и вызвать его метод для построения формы.

Параметры формы организованы в виде иерархии элементов формы. Корнем является объект CForm. Корневой объект формы включает в себя две коллекции, содержащие другие элементы: CForm::buttons и CForm::elements. Первая содержит кнопки (такие как «Сохранить» или «Очистить»), вторая — поля ввода, статический текст и вложенные формы — объекты CForm, находящиеся в коллекции CForm::elements другой формы. Вложенная форма может иметь свою модель данных и коллекции CForm::buttons и CForm::elements.

Когда пользователи отправляют форму, данные, введенные в поля ввода всей иерархии формы, включая вложенные формы, передаются на сервер. CForm включает в себя методы, позволяющие автоматически присвоить данные полям соответствующей модели и провести валидацию.

Создание простой формы

Ниже будет показано, как построить форму входа на сайт.

Сначала реализуем действие login:

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

Вкратце, здесь мы создали объект CForm, используя конфигурацию, найденную по пути, который задан псевдонимом application.views.site.loginForm. Объект CForm, как описано в разделе «Создание модели», использует модель LoginForm.

Если форма отправлена и все входные данные прошли проверку без ошибок, перенаправляем пользователя на страницу site/index. Иначе, выводим представление login, описывающее форму. Псевдоним пути application.views.site.loginForm указывает на файл PHP protected/views/site/loginForm.php. Этот файл возвращает массив, описывающий настройки, необходимые для CForm:

```
return array(
    'title'=>'Пожалуйста, представьтесь',

    'elements'=>array(
```

```

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

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

```

Настройки, приведённые выше являются ассоциативным массивом, состоящем из пар имя-значение, используемых для инициализации соответствующих свойств CForm. Самыми важными свойствами, как мы уже упомянули, являются CForm::elements и CForm::buttons. Каждое из них содержит массив, определяющий элементы формы. Более детальное описание элементов формы будет приведено в следующем подразделе.

Опишем шаблон представления `login`:

```

<h1>Вход</h1>

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

```

Подсказка: Приведённый выше код `echo $form;` эквивалентен `echo $form->render();`. Использование более компактной записи возможно так как CForm реализует магический метод `__toString`, в котором вызывается метод `render()`, возвращающий код формы.

Описание элементов формы

При использовании построителя форм, вместо написания разметки мы, главным образом, описываем элементы формы. В данном подразделе мы опишем, как задать свойство CForm::elements. Мы не будем описывать CForm::buttons так как конфигурация этого свойства практически ничем не отличается от CForm::elements.

Свойство CForm::elements является массивом, каждый элемент которого соответствует элементу формы. Это может быть поле ввода, статический текст или вложенная форма.

Описание поля ввода

Поле ввода, главным образом, состоит из заголовка, самого поля, подсказки и текста ошибки и должно соответствовать определённому атрибуту модели. Описание поля ввода содержится в экземпляре класса CFormInputElement. Приведённый ниже код массива CForm::elements описывает одно поле ввода:

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

Здесь указано, что атрибут модели называется `username`, тип поля — `text` и его атрибут `maxlength` равен 32.

Любое доступное для записи свойство `CFormInputElement` может быть настроено показанным выше способом. К примеру, можно задать свойство `hint` для того, чтобы показывать подсказку или свойство `items`, если поле является выпадающим списком или группой элементов `checkbox` или `radio`. Если имя опции не является свойством `CFormInputElement`, оно будет считаться атрибутом соответствующего HTML-тега `input`. К примеру, так как выше опция `maxlength` не является свойством `CFormInputElement`, она будет использована как атрибут `maxlength` HTML-элемента `input`.

Мы можем задать и другие доступные для записи свойства `[CFormInputElement]`.

Следует отдельно остановиться на свойстве `type`. Оно определяет тип поля ввода. К примеру, тип `text` означает, что будет использован элемент формы `input`, а `password` — поле для ввода пароля. В `CFormInputElement` реализованы следующие типы полей ввода:

- `text`
- `hidden`
- `password`
- `textarea`
- `file`
- `radio`
- `checkbox`
- `listbox`
- `dropdownlist`
- `checkboxlist`
- `radiolist`

Отдельно следует описать использование "списочных" типов `dropdownlist`, `checkboxlist` и `radiolist`. Для них необходимо задать свойство `items` соответствующего элемента `input`. Сделать это можно так:

```
'gender'=>array(
    'type'=>'dropdownlist',
    'items'=>User::model()->getGenderOptions(),
    'prompt'=>'Выберите значение:',
),
```

...

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

```
}
```

Данный код сгенерирует выпадающий список с текстом «Выберите значение:» и опциями «Мужчина» и «Женщина», которые мы получаем из метода `getGenderOptions` модели `User`.

Кроме данных типов полей, в свойстве `type` можно указать класс или псевдоним пути виджета. Класс виджета должен наследовать `CInputWidget` или `CJuiInputWidget`. При генерации элемента формы, будет создан и выполнен экземпляр класса виджета. Виджет будет использовать конфигурацию, переданную через настройки элемента формы.

Описание статического текста

Довольно часто в форме, помимо полей ввода, содержится некоторая декоративная HTML разметка. К примеру, горизонтальный разделитель для выделения определённых частей формы или изображение, улучшающее внешний вид формы. Подобный HTML код можно описать в коллекции `CForm::elements` как статический текст. Для этого в `CForm::elements` в нужном нам месте вместо массива необходимо использовать строку. К примеру:

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

        '<hr />',

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

В приведённом коде мы вставили горизонтальный разделитель между полями `password` и `rememberMe`. Статический текст лучше всего использовать в том случае, когда разметка и её расположение достаточно уникальны. Если некоторую разметку должен содержать каждый элемент формы, лучше всего переопределить непосредственно построение разметки формы, как будет описано далее.

Описание вложенных форм

Вложенные формы используются для разделения сложных форм на несколько связанных простых. К примеру, мы можем разделить форму регистрации пользователя на две вложенные формы: данные для входа и данные профиля. Каждая вложенная форма может, хотя и не обязана, быть связана с моделью данных. В примере с формой регистрации, если мы храним данные для входа и данные профиля в двух разных таблицах (и соответственно в двух моделях), то каждая вложенная форма будет сопоставлена соответствующей модели данных. Если же все данные хранятся в одной таблице, ни одна из вложенных форм не будет привязана к модели и обе будут использовать модель главной формы.

Вложенная форма, как и главная, описывается объектом `CForm`. Для того, чтобы описать вложенную форму, необходимо определить элемент типа `form` в свойстве `CForm::elements`:

```
return array(
    'elements'=>array(
        .....
        'user'=>array(
            'type'=>'form',
```

```

        'title'=>'Данные для входа',
        'elements'=>array(
            'username'=>array(
                'type'=>'text',
            ),
            'password'=>array(
                'type'=>'password',
            ),
            'email'=>array(
                'type'=>'text',
            ),
        ),
    ),
    'profile'=>array(
        'type'=>'form',
        .....
    ),
    .....
),
.....
);

```

Также, как и у главной формы, у вложенной формы необходимо задать свойство `CForm::elements`. Если вложенной форме необходимо сопоставить модель данных, можно сделать это задав свойство `CForm::model`.

В некоторых случаях бывает полезно определить форму в объекте класса, отличного от `CForm`. К примеру, как будет показано ниже, можно расширить `CForm` для реализации своего алгоритма построения разметки. При указании типа элемента `form`, вложенная форма будет автоматически использовать объект того же класса, что и у главной формы. Если указать тип элемента как, например, `XYZForm` (строка, оканчивающаяся на `Form`), то вложенная форма будет использовать объект класса `XYZForm`.

Доступ к элементам формы

Обращаться к элементам формы так же просто, как и к элементам массива. Свойство `CForm::elements` возвращает объект `CFormElementCollection`, наследуемый от `CMap` и позволяет получить доступ к своим элементам как к элементам массива. К примеру, для того, чтобы обратиться к элементу `username` формы `login` из примера, можно использовать следующий код:

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

Для доступа к элементу `email` формы регистрации пользователя из примера, можно использовать:

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

Так как `CForm` реализует доступ к элементам `CForm::elements` как к массиву, можно упростить код до:

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

Создание вложенной формы

Ранее мы уже описывали вложенные формы. Форма в которой имеются вложенные формы называется главной. В данном разделе мы будем использовать форму регистрации пользователя в качестве примера

создания вложенных форм, соответствующих нескольким моделям данных. Далее данные для входа пользователя хранятся в модели `User`, а данные профиля — в модели `Profile`.

Реализуем действие `register` следующим образом:

```
public function actionRegister()
{
    $form = new CForm('application.views.user.registerForm');
    $form['user']->model = new User;
    $form['profile']->model = new Profile;
    if($form->submitted('register') && $form->validate())
    {
        $user = $form['user']->model;
        $profile = $form['profile']->model;
        if($user->save(false))
        {
            $profile->userID = $user->id;
            $profile->save(false);
            $this->redirect(array('site/index'));
        }
    }

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

Выше мы создаём форму, используя настройки из `application.views.user.registerForm`. После отправки данных формы и их успешной валидации, мы пытаемся сохранить модели пользовательских данных и профиля. Мы получаем модели через свойство `model` соответствующего объекта вложенной формы. Так как валидация уже пройдена, вызываем `$user->save(false)` с параметром `false`, позволяющим её не проводить повторно. Точно так же поступаем с моделью профиля.

Далее описываем настройки формы в файле `protected/views/user/registerForm.php`:

```
return array(
    'elements'=>array(
        'user'=>array(
            'type'=>'form',
            'title'=>'Данные для входа',
            'elements'=>array(
                'username'=>array(
                    'type'=>'text',
                ),
                'password'=>array(
                    'type'=>'password',
                ),
                'email'=>array(
                    'type'=>'text',
                )
            ),
        ),
    ),
),
```

```

        'profile'=>array(
            'type'=>'form',
            'title'=>'Профиль',
            'elements'=>array(
                'firstName'=>array(
                    'type'=>'text',
                ),
                'lastName'=>array(
                    'type'=>'text',
                ),
            ),
        ),
    ),
);

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

```

При задании каждой вложенной формы, мы указываем свойство `CForm::title`. По умолчанию, при построении HTML-формы каждая вложенная форма будет выведена в `fieldset` с заданным нами заголовком. Описываем очень простой код шаблона представления `register`:

```

<h1>Регистрация</h1>

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

```

Свой рендеринг формы

Главное преимущество при использовании построителя форм — разделение логики (конфигурация формы хранится в отдельном файле) и отображения (метод `CForm::render`). В результате, мы можем настроить рендеринг формы либо переопределением метода `CForm::render`, либо своим отображением. Оба варианта позволяют не менять конфигурацию и позволяют использовать её повторно.

При переопределении `CForm::render`, необходимо, главным образом, обойти коллекции `CForm::elements` и `CForm::buttons` и вызвать метод `CFormElement::render` для каждого элемента. К примеру:

```

class MyForm extends CForm
{
    public function render()
    {
        $output = $this->renderBegin();
    }
}

```



```

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

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

        return $output;
    }
}

```

Также можно использовать отображение `_form`:

```

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

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

echo $form->renderEnd();

```

Для этого достаточно:

```

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

```

Если стандартный рендеринг формы не подходит (к примеру, в форме нужны уникальные декоративные элементы для определённых полей), в отображении можно поступить следующим образом:

какие-нибудь сложные элементы интерфейса

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

какие-нибудь сложные элементы интерфейса

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

какие-нибудь сложные элементы интерфейса

В этом случае построитель форм не очень эффективен так как нам приходится описывать те же объёмы кода формы. Тем не менее, преимущество есть. Оно в том, что форма, описанная в отдельном файле конфигурации, позволяет разработчику сфокусироваться на логике.

Работа с БД

Объекты доступа к данным (DAO)

Объекты доступа к данным (DAO) предоставляют общий API для доступа к данным, хранящимся в различных СУБД. Это позволяет без проблем поменять используемую СУБД на любую другую без необходимости изменения кода, использующего DAO для доступа к данным.

Yii DAO является надстройкой над PHP Data Objects (PDO) - расширением, которое предоставляет унифицированный доступ к данным многих популярных СУБД, таких, как MySQL, PostgreSQL. Для

использования Yii DAO необходимо, чтобы были установлены расширение PDO и драйвер PDO, соответствующий используемой базе данных (например, `PDO_MYSQL`).

Yii DAO состоит из четырех основных классов:

- `CDbConnection`: представляет подключение к базе данных;
- `CDbCommand`: представляет выражение SQL и его исполнение;
- `CDbDataReader`: представляет однонаправленный поток строк из данных, возвращаемых в ответ на SQL-запрос;
- `CDbTransaction`: представляет транзакции базы данных.

Ниже мы проиллюстрируем использование Yii DAO.

Соединение с базой данных

Для установления соединения с базой необходимо создать экземпляр класса `CDbConnection` и активировать его. Дополнительную информацию, необходимую для подключения к БД (хост, порт, имя пользователя, пароль и пр.), указываем в DSN. В случае возникновения ошибки в процессе соединения с БД, будет вызвано исключение (например, неверный DSN или неправильные имя пользователя/пароль).

```
$connection=new CDbConnection($dsn,$username,$password);
// устанавливаем соединение. Можно попробовать try...catch возможных исключений
$connection->active=true;
...
$connection->active=false; // close connection
```

Формат DSN зависит от используемого драйвера PDO. Как правило, DSN состоит из имени драйвера PDO, за которым следует двоеточие, а далее указываются параметры подключения, соответствующие синтаксису подключения используемого драйвера. Подробнее с этим можно ознакомиться в документации по PDO. Ниже представлены несколько основных форматов DSN:

- 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`

Поскольку `CDbConnection` является подклассом `CApplicationComponent`, то мы можем использовать его в качестве компонента. Для этого нужно настроить компонент `db` в конфигурации приложения следующим образом:

```
array(
    ...
    'components'=>array(
        ...
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'mysql:host=localhost;dbname=testdb',
            'username'=>'root',
            'password'=>'password',
            'emulatePrepare'=>true, // необходимо для некоторых версий инсталляций MySQL
        ),
    ),
),
)
```

Теперь мы можем получить доступ к соединению с БД через `Yii::app()->db`. Чтобы соединение не активировалось автоматически, необходимо установить значение `CDbConnection::autoConnect` в `false`. Этот способ дает нам возможность использования одного подключения к БД в любом месте кода.

Исполнение SQL-выражений

Когда соединение с БД установлено, мы можем выполнять SQL-выражения, используя `CDbCommand`. Для этого создаем экземпляр `CDbCommand` путем вызова `CDbConnection::createCommand()` с указанием SQL-выражения:

```
$connection=Yii::app()->db; // так можно сделать, если в конфигурации описан компонент
соединения "db"
// Если не описан — можно создать соединение явно:
// $connection=new CDbConnection($dsn,$username,$password);
$command=$connection->createCommand($sql);
// если необходимо, SQL-выражение можно обновить:
// $command->text=$newSQL;
```

Существует два варианта исполнения SQL-выражения с использованием `CDbCommand`:

- `execute()`: выполняет SQL-выражения типа `INSERT`, `UPDATE` и `DELETE`. В случае успешного исполнения, возвращает количество обработанных строк;
- `query()`: выполняет SQL-выражения, возвращающие наборы данных, например, типа `SELECT`. В случае успешного исполнения, возвращает экземпляр класса `CDbDataReader`, через который доступны полученные данные. Для удобства также реализованы методы `queryXXX()`, возвращающие результаты запроса напрямую.

Если в процессе выполнения SQL-выражения возникнет ошибка, будет вызвано исключение.

```
$rowCount=$command->execute(); // исполнение выражения типа `INSERT`, `UPDATE` и `DELETE`
$dataReader=$command->query(); // исполнение выражения типа `SELECT`
$rows=$command->queryAll(); // запрос и возврат всех строк результата
$row=$command->queryRow(); // запрос и возврат первой строки результата
$column=$command->queryColumn(); // запрос и возврат первого столбца результата
$value=$command->queryScalar(); // запрос и возврат первого поля в первой строке
```

Обработка результатов запроса

После того, как `CDbCommand::query()` создает экземпляр класса `CDbDataReader`, мы можем получить данные построчно путем повторного вызова метода `CDbDataReader::read()`. Для получения данных строка за строкой можно также использовать `CDbDataReader` в конструкциях `foreach`.

```
$dataReader=$command->query();
// многократно вызываем read() до возврата методом значения false
while(($row=$dataReader->read())!==false) { ... }
// используем foreach для построчного обхода данных
foreach($dataReader as $row) { ... }
// получаем все строки разом в одном массиве
```

```
$rows=$dataReader->readAll();
```

Примечание: Все методы `queryXXX()`, в отличие от `query()`, возвращают все данные напрямую. Например, метод `queryRow()` возвращает массив с первой строкой результата запроса.

Использование транзакций

В случае, когда приложение выполняет несколько запросов, каждый из которых что-то пишет или читает из БД, важно удостовериться, что набор запросов выполнен полностью, а не наполовину. В этой ситуации можно воспользоваться транзакциями, представляемыми экземпляром класса `CDbTransaction`:

- начало транзакции;
- исполнение запросов по очереди, все изменения данных в БД недоступны вне базы;

- подтверждение транзакции, если результат транзакции положительный, то изменения становятся доступны;
 - если возникает ошибка при выполнении какого-либо запроса, то вся транзакция откатывается назад.
- Эту последовательность действий можно реализовать следующим образом:

```
$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //... прочие SQL запросы
    $transaction->commit();
}
catch(Exception $e) // в случае ошибки при выполнении запроса выбрасывается исключение
{
    $transaction->rollBack();
}
```

Связывание параметров

С целью избежания SQL-инъекций и улучшения производительности при выполнении повторно используемых SQL-выражений, мы можем «подготавливать» SQL-выражение с маркерами параметров (placeholder), которые в процессе привязки будут заменяться на реальные значения. Маркеры параметров могут быть именованными (уникальные маркеры) или неименованными (вопросительные знаки). Для замены маркеров на реальные значения нужно вызвать `CDbCommand::bindParam()` или `CDbCommand::bindValue()`. Добавлять кавычки к параметрам нет необходимости, т.к. используемый драйвер базы данных все сделает сам. Связывание параметров должно быть завершено до исполнения SQL-выражения.

```
// выражение SQL с двумя маркерами «:username» и «:email»
$sql="INSERT INTO tbl_user(username, email) VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// меняем маркер «:username» на соответствующее значение имени пользователя
$command->bindParam(":username",$username,PDO::PARAM_STR);
// меняем маркер «:email» на соответствующее значение электронной почты
$command->bindParam(":email",$email,PDO::PARAM_STR);
$command->execute();
// вставляем следующую строку с новыми параметрами
```

```
$command->bindParam(":username",$username2,PDO::PARAM_STR);
```

```
$command->bindParam(":email",$email2,PDO::PARAM_STR);
$command->execute();
```

Методы `bindParam()` и `bindValue()` очень похожи, единственное различие состоит в том, что первый привязывает параметр к ссылке на переменную PHP, а второй — к значению. Для параметров, представляющих большой объем данных, с точки зрения производительности предпочтительнее использовать метод `bindParam()`.

Подробнее о связывании параметров можно узнать в соответствующей документации PHP.

Связывание полей

При обработке результатов запроса мы также можем привязать поля таблицы к переменным PHP. Это позволяет автоматически подставлять значения переменных для каждой строки:

```
$sql="SELECT username, email FROM tbl_user";
```

```

$dataReader=$connection->createCommand($sql)->query();
// привязываем первое поле (username) к переменной $username
$dataReader->bindColumn(1,$username);
// привязываем второе поле (email) к переменной $email
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // переменные $username и $email получают значения полей username и email текущей строки
}

```

Использование префикса таблиц

Начиная с версии 1.1.0, Yii обеспечивает встроенную поддержку префиксов таблиц. Префикс таблиц — это строка, предваряющая имя таблиц в текущей подключенной БД. В основном используется на общем хостинге, где к одной БД подключается несколько приложений, использующих различные префиксы таблиц для различения принадлежности таблиц к приложению. Например, одно приложение использует префикс `tbl_`, а другое — `yii_`.

Для использования префикса таблиц, установите в свойстве `CDbConnection::tablePrefix` желаемый префикс таблиц. Затем, в SQL выражении, используйте `{{TableName}}` для ссылки на имена таблиц, где `TableName` — имя таблицы без префикса. Например, если БД содержит таблицу с именем `tbl_user`, где `tbl_` — это префикс таблиц, тогда мы можем использовать следующий код для запроса пользователей:

```

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

```

Active Record

Хотя Yii DAO справляется практически с любыми задачами, касающимися работы с БД, почти наверняка 90% времени уйдут на написание SQL-выражений, реализующих общие операции CRUD (создание, чтение, обновление и удаление). Кроме того, код, перемешанный с SQL-выражениями, поддерживать проблематично. Для решения этих проблем мы можем воспользоваться Active Record.

Active Record реализует популярный подход объектно-реляционного проецирования (ORM). Каждый класс AR отражает таблицу (или представление) базы данных, экземпляр AR — строку в этой таблице, а общие операции CRUD реализованы как методы AR. В результате, мы можем работать с большей объектно-ориентированностью. Например, используя следующий код, можно вставить новую строку в таблицу `tbl_post`.

```

$post=new Post;
$post->title='тестовая запись';

$post->content='содержимое записи';

$post->save();

```

Ниже мы покажем, как настроить и использовать AR для реализации CRUD-операций, а в следующем разделе — как использовать AR для работы со связанными таблицами. Для примеров в этом разделе мы будем использовать следующую таблицу. Обратите внимание, что при использовании БД MySQL в SQL-выражении ниже `AUTOINCREMENT` следует заменить на `AUTO_INCREMENT`.

```

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

```

```
create_time INTEGER NOT NULL
);
```

Примечание: AR не дает решения для всех задач, касающихся работы с базами данных. Лучше всего его использовать для моделирования таблиц в конструкциях PHP и для несложных SQL-запросов. Для сложных случаев следует использовать Yii DAO.

Соединение с базой данных

Для работы AR требуется подключение к базе данных. По умолчанию, предполагается, что компонент приложения `db` предоставляет необходимый экземпляр класса `CDbConnection`, который отвечает за подключение к базе. Ниже приведен пример конфигурации приложения:

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:path/to/dbfile',
            // включить кэширование схем для улучшения производительности
            // 'schemaCachingDuration'=>3600,
        ),
    ),
);
```

Подсказка: Поскольку для получения информации о полях таблицы AR использует метаданные, требуется некоторое время для их чтения и анализа. Если не предполагается, что схема базы данных будет меняться, то следует включить кэширование схемы установив для атрибута `CDbConnection::schemaCachingDuration` любое значение больше нуля.

В настоящий момент AR поддерживает следующие СУБД:

- MySQL 4.1 и выше
- PostgreSQL 7.3 и выше
- SQLite 2 и 3
- Microsoft SQL Server 2000 и выше
- Oracle

Примечание: Microsoft SQL Server поддерживается начиная с версии 1.0.4; Oracle поддерживается начиная с версии 1.0.5.

Если вы хотите использовать другой компонент нежели `db` или предполагаете, используя AR, работать с несколькими БД, то следует переопределить метод `CActiveRecord::getDbConnection()`. Класс `CActiveRecord` является базовым классом для всех классов AR.

Подсказка: Есть несколько способов для работы AR с несколькими БД. Если схемы используемых баз различаются, то можно создать разные базовые классы AR с различной реализацией метода `getDbConnection()`. В противном случае, проще будет динамически менять статическую переменную `CActiveRecord::db`.

Определение AR-класса

Для доступа к таблице БД нам прежде всего требуется определить класс AR путем наследования класса `CActiveRecord`. Каждый класс AR представляет одну таблицу базы данных, а экземпляр класса — строку в

этой таблице. Ниже приведен минимальный код, требуемый для определения класса AR, представляющего таблицу `tbl_post`.

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

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

Подсказка: Поскольку классы AR часто появляются во многих местах кода, мы можем вместо включения классов по одному, добавить всю папку с AR-классами. К примеру, если AR-классы находятся в папке `protected/models`, мы можем сконфигурировать приложение следующим образом:

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

По умолчанию имя AR-класса совпадает с названием таблицы в базе данных. Если они различаются, потребуется переопределить метод `tableName()`. Метод `model()` объявляется для каждого AR-класса.

Информация: Для использования префиксов таблиц, появившихся в версии 1.1.0, метод `tableName()` AR-класса может быть переопределен как показано ниже:

```
public function tableName()

{

    return '{{post}}';
}
```

Вместо того, чтобы возвращать полное имя таблицы, мы возвращаем имя таблицы без префикса и заключаем его в двойные фигурные скобки.

Значения полей в строке таблицы доступны как атрибуты соответствующего экземпляра AR-класса. Например, код ниже устанавливает значение для атрибута `title`:

```
$post=new Post;
$post->title='тестовая запись';
```

Хотя мы никогда не объявляем заранее свойство `title` класса `Post`, мы, тем не менее, можем обратиться к нему как в коде выше. Это возможно по причине того, что `title` является полем в таблице `tbl_post` и

CActiveRecord делает его доступным в качестве свойства благодаря волшебному методу PHP `__get()`. Если аналогичным образом обратиться к несуществующему полю, будет вызвано исключение.

Информация: В данном руководстве мы называем столбцы и таблицы в нижнем регистре так как различные СУБД работают с регистрозависимыми именами по-разному. Например, PostgreSQL считает имя столбцов регистронезависимыми по умолчанию, и мы должны заключать имя столбца в кавычки в условиях запроса, если имя столбца имеет заглавные буквы. Использование нижнего регистра помогает избежать данной проблемы.

AR опирается на правильно определённые первичные ключи таблиц БД. Если в таблице нет первичного ключа, требуется указать в соответствующем классе AR столбцы, которые будут использоваться как первичный ключ. Сделать это можно путём перекрытия метода `primaryKey()`:

```
public function primaryKey()
{
    return 'id';
    // Для составного первичного ключа следует использовать массив:
    // return array('pk1', 'pk2');
}
```

Создание записи

Для добавления новой строки в таблицу БД, нам необходимо создать новый экземпляр соответствующего класса, присвоить значения атрибутам, ассоциированным с полями таблицы, и вызвать метод `save()` для завершения добавления.

```
$post=new Post;
$post->title='тестовая запись';
$post->content='содержимое тестовой записи';
$post->create_time=time();
$post->save();
```

Если первичный ключ таблицы автоинкрементный, то после добавления экземпляра AR будет содержать обновлённое значение первичного ключа. В примере выше, свойство `id` всегда будет содержать первичный ключ для новой записи.

Если поле задано в схеме таблицы с некоторым статическим значением по умолчанию (например, строка или число), то после создания экземпляра соответствующее свойство экземпляра AR будет автоматически содержать это значение. Один из способов поменять это значение — прописать его в AR-классе.

```
class Post extends CActiveRecord
{
    public $title='пожалуйста, введите заголовок';
    ...
}

$post=new Post;
echo $post->title; // отобразится: пожалуйста, введите заголовок
```

Начиная с версии 1.0.2, до сохранения записи (добавления или обновления) атрибуту может быть присвоено значение типа `CDbExpression`. Например, для сохранения текущей даты, возвращаемой функцией MySQL `now()`, можно использовать следующий код:

```
$post=new Post;
$post->create_time=new CDbExpression('NOW()');
// $post->create_time='NOW()'; этот вариант работать не будет
```



```
// т.к. значение 'NOW()' будет воспринято как строка
$post->save();
```

Подсказка: Несмотря на то, что AR позволяет производить различные операции без написания громоздкого SQL, часто необходимо знать, какой SQL выполняется на самом деле. Этого можно добиться, включив журналирование. К примеру, чтобы вывести запросы SQL в конце каждой страницы, мы можем включить CWebLogRoute в настройках приложения. Начиная с версии 1.0.5, можно задать параметр CDbConnection::enableParamLogging в true и получить также значения параметров запросов.

Чтение записи

Для чтения данных из таблицы базы данных нужно вызвать метод `find`:

```
// найти первую строку, удовлетворяющую условию
$post=Post::model()->find($condition,$params);
// найти строку с указанным значением первичного ключа
$post=Post::model()->findByPrimaryKey($postID,$condition,$params);
// найти строку с указанными значениями атрибута
$post=Post::model()->findByAttributes($attributes,$condition,$params);
// найти первую строку, используя некоторое выражение SQL
$post=Post::model()->findBySql($sql,$params);
```

Выше мы вызываем метод `find` через `Post::model()`. Запомните, что статический метод `model()` обязателен для каждого AR-класса. Этот метод возвращает экземпляр AR, используемый для доступа к методам уровня класса (что-то схожее со статическими методами класса) в контексте объекта. Если метод `find` находит строку, соответствующую условиям запроса, он возвращает экземпляр класса `Post`, свойства которого содержат значения соответствующих полей строки таблицы. Далее мы можем читать загруженные значения аналогично обычным свойствам объектам, например, `echo $post->title;`. В случае, если в базе нет данных, соответствующих условиям запроса, метод `find` вернет значение `null`. Параметры `$condition` и `$params` используются для уточнения запроса. В данном случае `$condition` может быть строкой, соответствующей оператору `WHERE` в SQL-выражении, а `$params` — массивом параметров, значения которых должны быть привязаны к маркерам, указанным в `$condition`. Например:

```
// найдем строку, где postID=10

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

Примечание: В примере выше, нам может понадобиться заключить в кавычки обращение к столбцу `postID` для некоторых СУБД. Например, если мы используем СУБД PostgreSQL, нам следует писать условие как `"postID"=:postID`, потому что PostgreSQL по умолчанию считает имя столбца регистронезависимым.

Кроме того, можно использовать `$condition` для указания более сложных условий запроса. Вместо строки параметр `$condition` может быть экземпляром класса `CDbCriteria`, который позволяет указать иные условия нежели `WHERE` выражение. К примеру:

```
$criteria=new CDbCriteria;
$criteria->select='title'; // выбираем только поле 'title'
$criteria->condition='postID=:postID';
$criteria->params=array(':postID'=>10);
```

```
$post=Post::model()->find($criteria); // $params не требуется
```

Обратите внимание, если в качестве условия запроса используется CDbCriteria, то параметр `$params` уже не нужен, поскольку его можно указать непосредственно в CDbCriteria, как показано выше. Помимо использования CDbCriteria, есть другой способ указать условие — передать методу массив ключей и значений, соответствующих именам и значениям свойств критерия. Пример выше можно переписать следующим образом:

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

Информация: В случае, когда условие заключается в соответствии значениям некоторых полей, можно воспользоваться методом `findByAttributes()`, где параметр `$attributes` представляет собой массив значений, проиндексированных по имени поля. В некоторых фреймворках эта задача решается путем использования методов типа `findByNameAndTitle`. Хотя такой способ и выглядит привлекательно, часто он вызывает путаницу и проблемы, связанные с чувствительностью имен полей к регистру.

В случае, если условию запроса отвечает множество строк, мы можем получить их все, используя методы `findAll`, приведенные ниже. Как мы отметили ранее, каждый из этих методов `findAll` имеет `find` аналог.

```
// найдем все строки, удовлетворяющие условию
$post=Post::model()->findAll($condition,$params);
// найдем все строки с указанными значениями первичного ключа
$post=Post::model()->findAllByPk($postIDs,$condition,$params);
// найдем все строки с указанными значениями атрибута
$post=Post::model()->findAllByAttributes($attributes,$condition,$params);
// найдем все строки, используя SQL-выражение
$post=Post::model()->findAllBySql($sql,$params);
```

В отличие от `find`, метод `findAll` в случае, если нет ни одной строки, удовлетворяющей запросу, возвращает не `null`, а пустой массив. Помимо методов `find` и `findAll` описанных выше, для удобства также доступны следующие методы:

```
// получим количество строк, удовлетворяющих условию
$n=Post::model()->count($condition,$params);
// получим количество строк с использованием указанного SQL-выражения
$n=Post::model()->countBySql($sql,$params);
// проверим, есть ли хотя бы одна строка, удовлетворяющая условию
$exists=Post::model()->exists($condition,$params);
```

Обновление записи

Заполнив экземпляр AR значениями полей, мы изменяем эти значения и сохраняем их обратно в БД.

```
$post=Post::model()->findByPk(10);
$post->title='new post title';
$post->save(); // сохраняем изменения в базу данных
```

Как можно было заметить, мы используем метод `save()` для добавления и обновления записей. Если экземпляр AR создан с использованием оператора `new`, то вызов метода `save()` приведет к добавлению новой строки в базу данных. В случае же, если экземпляр AR создан как результат вызова методов `find` и `findAll`, вызов метода `save()` обновит данные существующей строки в таблице. На самом деле, можно использовать свойство `CActiveRecord::isNewRecord` для указания, является экземпляр AR новым или нет. Кроме того, можно обновить одну или несколько строк в таблице без их предварительной загрузки. Для этого в AR существуют следующие методы уровня класса:

```
// обновим строки, отвечающие заданному условию
Post::model()->updateAll($attributes,$condition,$params);
// обновим строки, удовлетворяющие заданному условию и первичному ключу (или несколькими
ключам)
Post::model()->updateByPk($pk,$attributes,$condition,$params);
// обновим поля-счетчики в строках, удовлетворяющих заданным условиям
Post::model()->updateCounters($counters,$condition,$params);
```

Здесь `$attributes` — это массив значений полей, проиндексированных по имени поля, `$counters` — массив инкрементных значений, проиндексированных по имени поля, `$condition` и `$params` аналогично описанию выше.

Удаление записи

Мы можем удалить строку, если экземпляр AR был заполнен значениями этой строки.

```
$post=Post::model()->findByPk(10); // предполагаем, что запись с ID=10 существует
$post->delete(); // удаляем строку из таблицы
```

Обратите внимание, что после удаления экземпляра AR не меняется, но соответствующей записи в таблице уже нет.

Следующие методы используются для удаления строк без их предварительной загрузки:

```
// удалим строки, соответствующие указанному условию
Post::model()->deleteAll($condition,$params);
// удалим строки, соответствующие указанному условию и первичному ключу (или несколькими
ключам)

Post::model()->deleteByPk($pk,$condition,$params);
```

Проверка данных

Часто во время добавления или обновления строки, нам требуется проверить, соответствуют ли значения полей некоторым правилам. Особенно это важно в случае данных, поступающих со стороны клиента, — в подавляющем большинстве случаев этим данным доверять нельзя.

AR осуществляет проверку данных автоматически в момент вызова метода `save()`. Проверка основана на правилах, заданных в методе AR-класса `rules()`. Детально ознакомиться с тем, как задаются правила проверки, можно в разделе [Определение правил проверки](#). Ниже приведем типичный порядок обработки в момент сохранения записи:

```
if($post->save())
{
    // данные корректны и успешно добавлены/обновлены
}
else
{
```

```
// данные некорректны, сообщения об ошибках могут быть получены путем вызова метода
getErrors()
}
```

В момент, когда данные для добавления или обновления отправляются пользователем через форму ввода, нам требуется присвоить их соответствующим свойствам AR. Это можно проделать следующим образом:

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

Если полей будет много, мы получим простыню из подобных присваиваний. Этого можно избежать, если использовать свойство `attributes` как показано ниже. Подробности можно найти в разделах [Безопасное присваивание значений атрибутам](#) и [Создание действия](#).

```
// предполагаем, что $_POST['Post'] является массивом значений полей, проиндексированных по
имени поля
$post->attributes=$_POST['Post'];
$post->save();
```

Сравнение записей

Экземпляры AR идентифицируются уникальным образом по значениям первичного ключа, аналогично строкам таблицы, поэтому для сравнения двух экземпляров нам нужно просто сравнить значения их первичных ключей, предполагая, что оба экземпляра одного AR-класса. Однако можно проделать это еще проще, вызвав метод `CActiveRecord::equals()`.

Информация: В отличие от реализации AR в других фреймворках, Yii поддерживает в AR составные первичные ключи. Составной первичный ключ состоит из двух и более полей таблицы. Соответственно, первичный ключ в Yii представлен как массив, а свойство `primaryKey` содержит значение первичного ключа для экземпляра AR.

Тонкая настройка

Класс `CActiveRecord` предоставляет несколько методов, которые могут быть переопределены в дочерних классах для тонкой настройки работы AR.

- `beforeValidate` и `afterValidate`: методы вызываются до и после осуществления проверки;
- `beforeSave` и `afterSave`: методы вызываются до и после сохранения экземпляра AR;
- `beforeDelete` и `afterDelete`: методы вызываются до и после удаления экземпляра AR;
- `afterConstruct`: метод вызывается для каждого экземпляра AR, созданного с использованием оператора `new`;
- `beforeFind`: метод вызывается перед тем, как finder AR выполнит запрос (например, `find()`, `findAll()`). Данный метод доступен с версии 1.0.9.
- `afterFind`: метод вызывается для каждого экземпляра AR, созданного в результате выполнения запроса.

Использование транзакций с AR

Каждый экземпляр AR содержит свойство `dbConnection`, которое является экземпляром класса `CDbConnection`. Соответственно, в случае необходимости можно использовать возможность транзакций, предоставляемую Yii DAO:

```
$model=Post::model();
$transaction=$model->dbConnection->beginTransaction();
try
{
    // поиск и сохранение — шаги, которые можно разбить третьим запросом
```

```
// соответственно, мы используем транзакцию, чтобы удостовериться в целостности
$post=$model->findByPk(10);
$post->title='new post title';
$post->save();
$transaction->commit();
}
catch(Exception $e)
{
    $transaction->rollBack();
}
```

Именованные группы условий

Примечание: именованные группы условий поддерживаются, начиная с версии 1.0.5. Идея групп условий позаимствована у Ruby on Rails.

Именованная группа условий представляет собой *именованный* критерий запроса, который можно использовать с другими группами и применять к запросам AR. Именованные группы чаще всего описываются в методе `CActiveRecord::scopes()` парами имя-условие. Приведённый ниже код описывает две именованные группы условий для модели `Post`: `published` и `recently`:

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

Каждая группа описывается массивом, который может быть использован для инициализации экземпляра `CDbCriteria`. К примеру, `recently` определяет, что условие `order` будет `create_time DESC`, а `limit` будет равен 5. Вместе эти условия означают, что будут выбраны 5 последних публикаций.

Именованные группы условий обычно используются в качестве модификаторов для метода `find`. Можно использовать несколько групп для получения более специфичного результата. К примеру, чтобы найти последние опубликованные записи можно использовать следующий код:

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

В общем случае, именованные группы условий должны располагаться левее вызова `find`. Каждая группа определяет критерий запроса, который совмещается с остальными критериями, включая переданные непосредственно методу `find`. Конечный результат получается применением к запросу набора фильтров. Начиная с версии 1.0.6, именованные группы условий могут быть использованы в методах `update` и `delete`. К примеру, следующий код удалит все недавно опубликованные записи:

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

Примечание: Именованные группы могут быть использованы только с методами класса. Таким образом, метод должен вызываться при помощи `ClassName::model()`.

Именованные группы условий с параметрами

Именованные группы условий могут быть параметризованы. К примеру, нам понадобилось задать число публикаций для группы `recently`. Чтобы это сделать, вместо того, чтобы описывать группу в методе `CActiveRecord::scopes`, нам необходимо описать новый метод с таким же именем, как у группы условий:

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

После этого, для того, чтобы получить 3 последних опубликованных записи, можно использовать следующий код:

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

Если не передать параметром 3, то по умолчанию будут выбраны 5 последних опубликованных записей.

Именованная группа условий по умолчанию

Класс модели может содержать именованную группу условий по умолчанию, которая будет применяться ко всем запросам (включая реляционные). К примеру, на сайте реализована поддержка нескольких языков и содержимое отображается на языке, выбранном пользователем. Так как запросов, связанных с получением данных скорее всего достаточно много, для решения этой задачи мы можем определить группу условий по умолчанию. Для этого мы перекрываем метод `CActiveRecord::defaultScope` следующим образом:

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

Теперь к показанному ниже вызову будет автоматически применены наши условия:

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

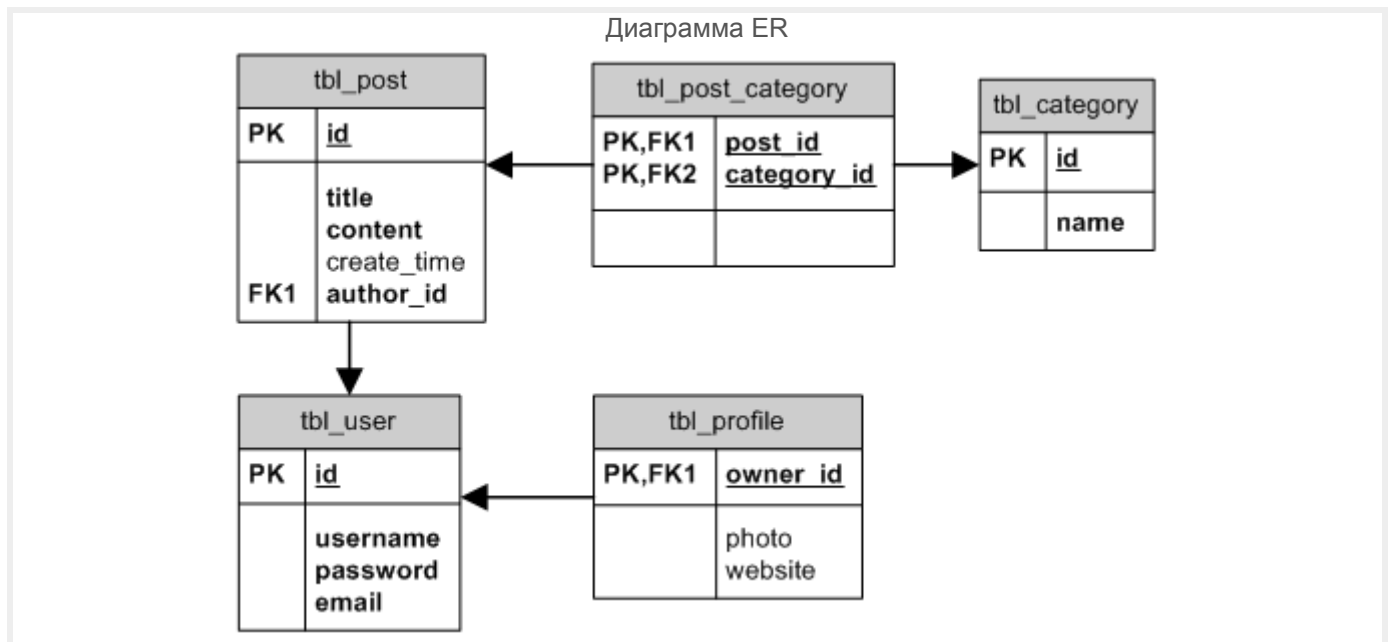
Заметим, что именованная группа условий по умолчанию применяется только к `SELECT` запросам. Она игнорируется в `INSERT`, `UPDATE` и `DELETE` запросах.

Реляционная Active Record

Мы уже рассмотрели использование Active Record (AR) для выбора данных из одной таблицы базы данных. В этом разделе мы расскажем, как использовать AR для объединения нескольких связанных таблиц и получить объединенный набор данных.

Для использования реляционной AR рекомендуется, чтобы все связи отношения первичный-внешний ключ были четко определены для объединяемых таблиц. Это помогает поддерживать связность и целостность данных.

Для наглядности примеров в данном разделе мы будем использовать схему базы данных, представленную на этой диаграмме сущность-отношение (ER).



Информация: Поддержка ограничений по внешнему ключу различна в разных СУБД. SQLite < 3.6.19 не поддерживает ограничений, но вы, тем не менее, можете их объявить при создании таблиц.

Объявление отношения

Перед тем, как использовать AR для реляционных запросов, нам необходимо объяснить AR, как AR-классы связаны друг с другом.

Отношение между двумя AR-классами напрямую зависит от отношений между соответствующими таблицами базы данных. С точки зрения БД, отношение между таблицами A и B может быть трех типов: один-ко-многим (например, `tbl_user` и `tbl_post`), один-к-одному (например, `tbl_user` и `tbl_profile`) и многие-ко-многим (например, `tbl_category` и `tbl_post`). В AR существует четыре типа отношений:

- **BELONGS_TO**: если отношение между A и B один-ко-многим, значит B принадлежит A (например, `Post` принадлежит `User`);
- **HAS_MANY**: если отношение между таблицами A и B один-ко-многим, значит у A есть много B (например, у `User` есть много `Post`);
- **HAS_ONE**: это частный случай **HAS_MANY**, где A может иметь максимум одно B (например, у `User` есть только один `Profile`);
- **MANY_MANY**: это отношение соответствует типу отношения многие-ко-многим в БД. Поскольку многие СУБД не поддерживают непосредственно тип отношения многие-ко-многим, требуется ассоциированная таблица для преобразования отношения многие-ко-многим в отношения один-ко-многим. В нашей схеме базы данных, этой цели служит таблица `tbl_post_category`. В терминологии AR отношение **MANY_MANY** можно описать как комбинацию **BELONGS_TO** и **HAS_MANY**. Например, `Post` принадлежит многим `Category`, а у `Category` есть много `Post`.

Объявляя отношение в AR, мы переопределяем метод `relations()` класса `CActiveRecord`. Этот метод возвращает массив с конфигурацией отношений. Каждый элемент массива представляет одно отношение в следующем формате:

```
'VarName'=>array('RelationType', 'ClassName', 'ForeignKey', ...дополнительные параметры)
```

где `VarName` — имя отношения, `RelationType` указывает на один из четырех типов отношения, `ClassName` — имя AR-класса, связанного с данным AR-классом, а `ForeignKey` обозначает один или несколько внешних ключей, используемых для связи. Кроме того, можно указать ряд дополнительных параметров, о которых расскажем чуть позже.

В коде ниже показано, как объявить отношение между классами `User` и `Post`.

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

class User extends ActiveRecord
{
    ...
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
        );
    }
}
```

Информация: Если внешний ключ составной, мы должны объединить имена полей внешнего ключа и отделить друг от друга пробелом или запятой. Для типа отношения `MANY_MANY` имя ассоциативной таблицы также должно быть указано во внешнем ключе. Например, отношение `categories` в модели `Post` обозначено внешним ключом `tbl_post_category(post_id, category_id)`.

При объявлении отношения в AR-классе для каждого отношения в класс неявно добавляется свойство. После выполнения реляционного запроса соответствующее свойство будет заполнено связанным(-и) экземпляром(-ами) AR. Например, если `$author` представляет AR-экземпляр `User`, то можно использовать `$author->posts` для доступа к связанным экземплярам `Post`.

Выполнение реляционного запроса

Самый простой способ выполнить реляционный запрос — считать реляционное свойство AR-класса. Если ранее к этому свойству никто не обращался, то будет инициирован реляционный запрос, который соединит

связанные таблицы и оставит только данные, соответствующие первичному ключу текущего экземпляра AR. Результат запроса будет сохранен в свойстве как экземпляр(-ы) связанного класса. Этот подход также известен, как «отложенная загрузка» (lazy loading), т.е. реляционный запрос осуществляется только в момент первого обращения к связанным объектам. Пример ниже показывает использование этого подхода:

```
// получаем запись с ID=10
$post=Post::model()->findByPk(10);
// Получаем автора записи. Здесь будет выполнен реляционный запрос.
$author=$post->author;
```

Информация: Если для отношения не существует связанного экземпляра, то соответствующее свойство будет null для отношений `BELONGS_TO` и `HAS_ONE` или пустым массивом для `HAS_MANY` и `MANY_MANY`. Стоит отметить, что отношения `HAS_MANY` и `MANY_MANY` возвращают массивы объектов и обращаться к их свойствам необходимо в цикле, иначе можно получить ошибку «Trying to get property of non-object».

Способ отложенной загрузки удобен, но не всегда эффективен. Например, если мы захотим получить информацию об авторе `n` записей, использование отложенной загрузки потребует выполнения `n` запросов для объединения. В данной ситуации, нам поможет метод «жадной загрузки» (eager loading). Этот подход заключается в загрузке всех связанных экземпляров AR одновременно с основным экземпляром AR. Реализуется этот подход путем использования в AR метода `with()` в связке с методом `find` или `findAll`. Например:

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

Код выше вернет массив экземпляров `Post`. В отличие от отложенной загрузки, свойство `author` в каждой записи заполнено связанным экземпляром `User` еще до обращения к свойству. Вместо выполнения объединяющего запроса для каждой записи, жадная загрузка получает все записи вместе с авторами в одном объединяющем запросе!

В методе `with()` можно указать множество имен отношений и жадная загрузка вернет их за один раз. Например, следующий код вернет записи вместе с их авторами и категориями:

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

Кроме того, можно осуществлять вложенную жадную загрузку. Для этого вместо простого списка имен отношений, мы передаем методу `with()` имена отношений, упорядоченных иерархически, как в примере ниже:

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

    'author.posts',

    'categories')->findAll();
```

Пример выше вернет нам все записи с их авторами и категориями, а также профиль каждого автора и все его записи.

Начиная с версии 1.1.0, жадная загрузка может быть выполнена путём указания свойства `CDbCriteria::with`:

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

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

или

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

Параметры реляционного запроса

Выше мы упоминали о том, что в реляционном запросе можно указать дополнительные параметры. Эти параметры — пары имя-значение — используются для тонкой настройки реляционного запроса. Список параметров представлен ниже.

- **select**: список выбираемых полей для связанного AR-класса. По умолчанию значение параметра равно `*`, т.е. выбираются все поля таблицы. Для используемых столбцов должны быть разрешены конфликты имён.
- **condition**: соответствует оператору **WHERE**, по умолчанию значение параметра пустое. Для используемых столбцов должны быть разрешены конфликты имён.
- **params**: параметры для связывания в генерируемом SQL-выражении. Параметры передаются как массив пар имя-значение. Параметр доступен, начиная с версии 1.0.3;
- **on**: соответствует оператору **ON**. Условие, указываемое в этом параметре, будет добавлено к условию объединения с использованием оператора **AND**. Для используемых столбцов должны быть разрешены конфликты имён. Данный параметр неприменим для отношений типа **MANY_MANY**. Параметр доступен, начиная с версии 1.0.2;
- **order**: соответствует оператору **ORDER BY**, по умолчанию значение параметра пустое. Для используемых столбцов должны быть разрешены конфликты имён.
- **with**: список дочерних связанных объектов, которые должны быть загружены с самим объектом. Неправильное использование данной возможности может привести к бесконечному циклу.
- **joinType**: тип объединения для отношения. По умолчанию значение параметра равно **LEFT OUTER JOIN**;
- **alias**: псевдоним таблицы, ассоциированной с отношением. Этот параметр доступен с версии 1.0.1. По умолчанию значение параметра равняется `null`, что означает, что псевдоним соответствует имени отношения.
- **together**: параметр, устанавливающий необходимость принудительного объединения таблицы, ассоциированной с этим отношением, с другими таблицами. Этот параметр имеет смысл только для отношений типов **HAS_MANY** и **MANY_MANY**. Если параметр не установлен или равен `false`, тогда каждое отношение **HAS_MANY** или **MANY_MANY** будет использовать отдельный SQL запрос для связанных данных, что может улучшить скорость выполнения запроса т.к. уменьшается количество выбираемых данных. Если данный параметр равен `true`, зависимая таблица при запросе будет всегда объединяться с основной, то есть будет сделан один запрос даже в том случае, если к основной таблице применяется страничная разбивка. Если данный параметр не задан, зависимая таблица будет объединена с основной только в случае, когда не к основной таблице не применяется страничная разбивка. Более подробное описание можно найти в разделе «производительность реляционного запроса». Параметр доступен, начиная с версии 1.0.3;
- **join**: дополнительный оператор **JOIN**. По умолчанию пуст. Этот параметр доступен с версии 1.1.3.
- **group**: соответствует оператору **GROUP BY**, по умолчанию значение параметра пустое. Для используемых столбцов должны быть разрешены конфликты имён.
- **having**: соответствует оператору **HAVING**, по умолчанию значение параметра пустое. Для используемых столбцов должны быть разрешены конфликты имён. Параметр доступен, начиная с версии 1.0.1.
- **index**: имя столбца, значения которого должны быть использованы в качестве ключей массива, хранящего связанные объекты. Без установки этого параметра, массив связанных объектов

использует целочисленный индекс, начинающийся с нуля. Параметр может быть установлен только для отношений `HAS_MANY` и `MANY_MANY`. Параметр доступен с версии 1.0.7.

Кроме того, для отложенной загрузки некоторых типов отношений доступен ряд дополнительных параметров:

- `limit`: параметр для ограничения количества строк в выборке. Параметр не применим для отношений `BELONGS_TO`;
- `offset`: параметр для указания начальной строки выборки. Параметр не применим для отношений `BELONGS_TO`.

Ниже мы изменим определение отношения `posts` в модели `User`, добавив несколько вышеприведенных параметров:

```
class User extends ActiveRecord
{
    public function relations()
    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                'order'=>'posts.create_time DESC',
                'with'=>'categories'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
        );
    }
}
```

Теперь при обращении к `$author->posts`, мы получим записи автора, отсортированные в обратном порядке по времени их создания. Для каждой записи будут загружены её категории.

Устранение конфликта имён столбцов

При совпадении имён столбцов в двух и более соединяемых таблицах, приходится решать конфликт имён. Это делается при помощи добавления псевдонима таблицы к имени столбца.

В реляционном запросе псевдоним главной таблицы всегда равен `t`. Имя псевдонима относящейся к ней таблице по умолчанию соответствует имени отношения. К примеру, в коде ниже псевдонимы для `Post` и `Comment` соответственно `t` и `comments`:

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

Допустим, что и в `Post` и в `Comment` есть столбец `create_time`, в котором хранится время создания записи или комментария, и нам необходимо получить записи вместе с комментариями к ним, отсортированные сначала по времени создания записи, а затем по времени написания комментария. Для этого нам понадобится устранить конфликт столбцов `create_time` следующим образом:

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

Примечание: разрешение конфликта имён изменилось по сравнению с версией 1.1.0. До этого по умолчанию Yii генерировал псевдоним таблицы для каждой связанной таблицы, а для его использования необходимо было использовать префикс `??` в нужных местах запроса. Также в версиях 1.0.x псевдоним главной таблицы соответствовал имени таблицы.

Динамические параметры реляционного запроса

Начиная с версии 1.0.2, мы можем использовать динамические параметры как для параметра `with()`, так и для параметра `with`. Динамические параметры переопределяют существующие параметры в соответствии с описанием метода `relations()`. К примеру, если для модели `User`, приведенной выше, мы хотим воспользоваться жадной загрузкой для получения записей автора в порядке возрастания (параметр `order` в определении отношения задает убывающий порядок), можно сделать это следующим образом:

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

Начиная с версии 1.0.5 динамические параметры в реляционных запросах можно использовать вместе с отложенной загрузкой. Для этого необходимо вызвать метод с тем же именем, что и имя связи, и передать параметры как его аргумент. К примеру, следующий код вернёт публикации пользователя, у которых `status` равен 1:

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

Производительность реляционного запроса

Как было описано выше, жадная загрузка используется, главным образом, когда требуется получить множество связанных объектов. В этом случае соединением всех таблиц генерируется большой сложный SQL-запрос. Такой запрос во многих случаях является предпочтительным т.к. упрощает фильтрацию по значению столбца связанной таблицы. Тем не менее, в некоторых случаях такие запросы не являются эффективными.

Рассмотрим пример, в котором нам надо найти последние записи вместе с их комментариями. Учитывая, что у каждой записи 10 комментариев, при использовании одного большого SQL-запроса мы получим множество лишних данных так как каждая запись будет повторно выбираться с каждым её комментарием. Теперь попробуем по-другому: сначала выберем последние записи, а затем комментарии к ним. В данном случае нам необходимо выполнить два SQL запроса. Плюс в том, что в полученных данных не будет ничего лишнего.

Так какой подход более эффективен? Абсолютно верного ответа на этот вопрос нет. Выполнение одного большого SQL запроса может быть более эффективным так как СУБД не приходится лишний раз разбирать и выполнять дополнительные запросы. С другой стороны, используя один SQL запрос, мы получаем больше лишних данных и соответственно нам требуется больше времени на их передачу и обработку.

По этой причине в Yii имеется параметр запроса `together`, позволяющий выбрать между двумя описанным подходами. По умолчанию Yii использует первый подход, то есть генерирует один SQL запрос для жадной загрузки. Если выставить параметр `together` в `false`, некоторые таблицы будут объединены отдельными SQL запросами. К примеру, для того, чтобы использовать второй подход для выборки последних записей с комментариями к ним, мы можем описать отношение `comments` в классе `Post` следующим образом:

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

Для жадной загрузки мы можем задать эту опцию динамически:

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

Примечание: В версии 1.0.x по умолчанию генерировалось и выполнялось $n+1$ SQL запросов при наличии n связей типа `HAS_MANY` или `MANY_MANY`. Каждая связь `HAS_MANY` или `MANY_MANY` выполнялась в своём запросе. Вызовом метода `together()` после `with()` можно было сгенерировать и выполнить единый SQL запрос:

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

Статистический запрос

Примечание: статистические запросы доступны, начиная с версии 1.0.4.

Помимо реляционных запросов, описанных выше, Yii также поддерживает так называемые статистические запросы (или запросы агрегирования). Этот тип запросов используется для получения агрегированных данных, относящихся к связанным объектам, например количество комментариев к каждой записи, средний рейтинг для каждого наименования продукции и т.д. Статистические запросы могут быть использованы только для объектов, связанных отношениями `HAS_MANY` (например, у записи есть много комментариев) или `MANY_MANY` (например, запись принадлежит многим категориям, а к категории относится множество записей).

Выполнение статистического запроса аналогично выполнению реляционного запроса в соответствии с описанием выше. Первым делом необходимо объявить статистический запрос в методе `relations()` класса `CActiveRecord`.

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

Выше мы объявили два статистических запроса: `commentCount` подсчитывает количество комментариев к записи, а `categoryCount` считает количество категорий, к которым относится запись. Обратите внимание, что отношение между `Post` и `Comment` — типа `HAS_MANY`, а отношение между `Post` и `Category` — типа `MANY_MANY` (с использованием преобразующей таблицы `post_category`). Как можно видеть, порядок объявления очень схож с объявлением отношений, описанных выше. Единственное различие состоит в том, что в данном случае тип отношения равен `STAT`.

За счет объявленных отношений мы можем получить количество комментариев для записи, используя выражение `$post->commentCount`. В момент первого обращения к данному свойству для получения соответствующего результата неявным образом выполняется SQL-выражение. Как мы уже говорили, это называется подходом *отложенной загрузки*. Можно также использовать *жадный* вариант загрузки, если необходимо получить количество комментариев к нескольким записям:

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

Выражение выше выполняет три SQL-запроса для получения всех записей вместе с значениями количества комментариев к ним и количества категорий. В случае отложенной загрузки нам бы понадобилось выполнить $2 \cdot n + 1$ SQL-запросов для n записей.

По умолчанию статистический запрос считает количество с использованием выражения `count`. Его можно уточнить путем указания дополнительных параметров в момент объявления в методе `relations()`. Доступные параметры перечислены ниже:

- `select`: статистическое выражение, по умолчанию равно `count (*)`, что соответствует количеству дочерних объектов;
- `defaultValue`: значение, которое присваивается в случае, если результат статистического запроса для записи отрицателен. Например, если запись не имеет ни одного комментария, то свойству `commentCount` будет присвоено это значение. По умолчанию значение данного параметра равно 0;
- `condition`: соответствует оператору `WHERE`, по умолчанию значение параметра пустое;
- `params`: параметры для связывания в генерируемом SQL-выражении. Параметры передаются как массив пар имя-значение;
- `order`: соответствует оператору `ORDER BY`, по умолчанию значение параметра пустое;
- `group`: соответствует оператору `GROUP BY`, по умолчанию значение параметра пустое;
- `having`: соответствует оператору `HAVING`, по умолчанию значение параметра пустое.

Реляционные запросы с именованными группами условий

Примечание: Группы условий поддерживаются, начиная с версии 1.0.5.

В реляционном запросе именованные группы условий могут быть использованы двумя способами. Их можно применить к основной модели и к связанным моделям.

Следующий код показывает случай с основной моделью:

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

Данный код очень похож на нереляционные запросы. Единственное отличие в том, что у нас присутствует вызов `with()` после вызовов групп условий. Данный запрос вернёт недавно опубликованные записи вместе с комментариями к ним.

В следующем примере показано, как применить группы условий к связанным моделям:

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

Этот запрос вернёт все записи вместе с одобренными комментариями. Здесь `comments` относится к имени отношения. `recently` и `approved` — именованные группы, описанные в модели `Comment`. Имя отношения и группы параметров разделяются двоеточием.

Именованные группы могут быть использованы при описании отношений модели в методе `CActiveRecord::relations()` в параметре `with`. В следующем примере при обращении к `$user->posts` вместе с публикациями будут получены все одобренные комментарии.

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

Примечание: Именованные группы параметров, применяемые к реляционным моделям, должны описываться в методе `CActiveRecord::scopes`, поэтому они не могут быть параметризованы.

Кэширование

Кэширование — простой и эффективный способ улучшения производительности веб-приложения.

Сохраняя относительно статичные данные в кэше и используя эти данные из кэша когда потребуется, мы экономим время генерации данных.

Использование кэша в Yii — это, главным образом, конфигурирование и вызов компонента кэша. Ниже показана конфигурация, определяющая компонент кэша, использующий memcache с двумя кэш-серверами.

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

Во время работы приложения обратиться к компоненту, отвечающему за кэширование, можно так:

```
Yii::app()->cache.
```

Yii обеспечивает разные кэш-компоненты, кэширующие данные в различных хранилищах. Например, компонент CMemCache инкапсулирует дополнение memcache для PHP и использует память как хранилище кэша; компонент CApcCache инкапсулирует дополнение APC для PHP; и компонент CDbCache сохраняет кэшируемые данные в базе данных. Далее приведен список доступных компонентов кэширования:

- CMemCache: использует дополнение memcache для PHP;
- CApcCache: использует дополнение APC для PHP;
- CXCache: использует дополнение XCache для PHP. Доступно с версии 1.0.1;
- CEAcceleratorCache: использует дополнение EAccelerator для PHP;
- CDbCache: использует таблицы базы данных для хранения кэшируемых данных. По умолчанию создает и использует базу данных SQLite3 во временном каталоге. Мы можем явно определить базу данных установкой свойства connectionID;
- CZendDataCache: использует Zend Data Cache. Доступно с версии 1.0.4;
- CFileCache: для хранения кэшированных данных используются файлы. Хорошо подходит для больших единиц данных, таких как целые страницы. Доступно с версии 1.0.6;
- CDummyCache: кэш-пустышка. Ничего не кэширует. Нужен для упрощения кода, необходимого при проверке доступности кэша. Мы можем воспользоваться данным компонентом во время разработки или в случае, если сервер не поддерживает кэширование. Когда осуществление кэширования будет возможно, мы сможем его применить. В обоих случаях будет использован идентичный код для получения данных `Yii::app()->cache->get($key)`. При этом можно не беспокоиться о том, что `Yii::app()->cache` может быть равен `null`. Данный компонент доступен с версии 1.0.5.

Подсказка: Все эти компоненты кэширования наследуются от базового класса CCache, поэтому можно переключаться между различными типами кэширования без изменения кода, использующего кэш.

Кэширование может использоваться на различных уровнях. На низшем уровне мы используем кэширование для хранения «атомарных» (одиночных) кусков данных, таких как переменные, и называем это *кэшированием данных*. На следующем уровне, мы храним в кэше фрагменты страницы, генерируемые частью скрипта представления. И на высшем уровне мы храним в кэше целую страницу и получаем ее из кэша при необходимости.

В следующих нескольких подразделах мы подробно разберем, как использовать кэширование на этих уровнях.

Примечание: По определению, кэш — энергозависимая среда. Поэтому нет гарантий сохранения кэшируемых данных, даже если они не устаревают. Поэтому не используйте кэш как постоянное хранилище данных (например, не используйте кэш для хранения данных сессий).

Кэширование данных

Кэширование данных — это хранение некоторой переменной PHP в кэше и получение её оттуда. Для этой цели базовый класс CCache компонента кэша имеет два наиболее используемых метода: `set()` и `get()`. Для кэширования переменной `$value` мы выбираем уникальный идентификатор (ID) и вызываем метод `set()` для её сохранения в кэше:

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

Данные будут оставаться в кэше до тех пор, пока не будут удалены из-за некоторых условий функционирования кэша (например, места для кэширования не осталось, тогда более старые данные удаляются). Для изменения такого поведения мы можем установить значения срока действия кэша при вызове метода `set()`. Тогда данные будут удалены из кэша после определенного периода времени:

```
// храним значение переменной в кэше не более 30 секунд
Yii::app()->cache->set($id, $value, 30);
```

Позже, когда нам понадобится доступ к этой переменной (в этом же или другом запросе), мы вызываем метод `get()` с идентификатором переменной. Если возвращенное значение — `false`, то переменная не доступна в кэше и мы должны регенерировать ее (обновить в кэше).

```
$value=Yii::app()->cache->get($id);
if($value===false)
{
    // обновляем $value, т.к. переменная не найдена в кэше,
    // и сохраняем в кэш для дальнейшего использования:
    // Yii::app()->cache->set($id,$value);
}
```

При выборе идентификатора для кэшируемой переменной, учитывайте, что он должен быть уникальным для каждой переменной из тех, что могут быть кэшированы в приложении. *НЕ* требуется, чтобы идентификатор был уникальным между разными приложениями, потому что компонент кэша достаточно умен для различения идентификаторов разных приложений.

Некоторые кэш-хранилища, такие как MemCache, APC, поддерживают загрузку нескольких кэшированных значений в пакетном режиме, которая может уменьшить накладные расходы на получение данных, сохраненных в кэше. Начиная с версии 1.0.8, новый метод `mget()` позволяет использовать эту особенность. В случае, когда кэш-хранилище не поддерживает эту функцию, `mget()` будет по-прежнему имитировать ее. Для удаления кэшированного значения из кэша надо вызвать метод `delete()`, а для очистки всего кэша — вызвать метод `flush()`. Надо быть осторожным при вызове метода `flush()`, т.к. он также удаляет кэшированные данные других приложений.

Подсказка: класс CCache реализует `ArrayAccess`, поэтому компонент кэша может использоваться как массив. Ниже приведены примеры:

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

Зависимость кэша

Помимо установки срока действия, кэшируемые данные также могут стать недействительными в соответствии с некоторыми изменениями зависимости (dependency). Например, если мы кэшируем содержимое некоторого файла, и файл изменился, мы должны принять кэшированную копию как недействительную и считать свежее содержимое из файла, а не из кэша. Мы представляем зависимость как экземпляр класса `CCacheDependency` или классов, его наследующих. Мы передаем экземпляр зависимости вместе с кэшируемыми данными, когда вызываем метод `set()`.

```
// значение действительно не более 30 секунд
// также, значение может стать недействительным раньше, если зависимый файл изменен
Yii::app()->cache->set($id, $value, 30, new CFileCacheDependency('FileName'));
```

Теперь, если мы попытаемся получить значение `$value` из кэша, вызвав метод `get()`, зависимость будет проверена и, если она изменилась, мы получим значение `false`, показывающее, что данные требуют обновления.

Ниже приведен список доступных зависимостей кэша:

- `CFileCacheDependency`: зависимость меняется, если время модификации файла изменено;
- `CDirectoryCacheDependency`: зависимость меняется, если любой файл в каталоге или в подкаталогах изменен;
- `CDbCacheDependency`: зависимость меняется, если результат запроса некоторого определенного SQL выражения изменен;
- `CGlobalStateCacheDependency`: зависимость меняется, если значение определенного глобального состояния изменено. Глобальное состояние — это переменная, являющаяся постоянной в многократных запросах и сессиях приложения. Устанавливается методом `CApplication::setGlobalState()`;
- `CChainedCacheDependency`: зависимость меняется, если любая зависимость цепочки изменена;
- `CExpressionDependency`: зависимость меняется, если результат определенного выражения PHP изменен. Данный класс доступен с версии 1.0.4.

Кэширование фрагментов

Кэширование фрагментов относится к кэшированию фрагментов страницы. Например, если страница отображает в таблице суммарные годовые продажи, мы можем сохранить эту таблицу в кэше с целью экономии времени, требуемого для генерации таблицы при каждом запросе.

Для использования кэширования фрагментов мы вызываем методы `CController::beginCache()` и `CController::endCache()` в скрипте представления контроллера. Эти два метода являются метками начала и конца содержимого страницы, которое должно быть кэшировано. Как и в кэшировании данных, нам нужен идентификатор для определения кэшируемого фрагмента.

```
...другое HTML-содержимое...
<?php if($this->beginCache($id)) { ?>
...кэшируемое содержимое...
<?php $this->endCache(); } ?>
...другое HTML-содержимое...
```

В коде выше, если метод `beginCache()` возвращает `false`, то кэшированное содержимое будет автоматически вставлено в данное место, иначе, содержимое внутри выражения `if` будет выполнено и сохранено в кэше, когда будет вызван метод `endCache()`.

Параметры кэширования

Вызывая метод `beginCache()`, мы можем передать в качестве второго параметра массив, содержащий параметры кэширования для управления кэшированием фрагмента. Фактически, методы `beginCache()` и `endCache()` являются удобной оберткой виджета `COutputCache`. Поэтому, параметры кэширования могут быть начальными значениями для любых свойств виджета `COutputCache`.

Длительность (срок хранения)

Наверное, наиболее часто используемым параметром является `duration`, который определяет, насколько долго содержимое кэша будет оставаться действительным (валидным). Это похоже на параметр срока действия метода `CCache::set()`. Код ниже кэширует фрагмент на время не более часа:

```

...другое HTML-содержимое...
<?php if($this->beginCache($id, array('duration'=>3600))) { ?>
...кэшируемое содержимое...
<?php $this->endCache(); } ?>
...другое HTML-содержимое...

```

Если мы не установим длительность (срок хранения), она будет равна значению по умолчанию (60 секунд). Это значит, что кэшированное содержимое станет недействительным через 60 секунд.

Зависимость

Как и кэширование данных, кэшируемое содержимое фрагмента тоже может иметь зависимости (dependency). Например, отображение содержимого сообщения зависит от того, изменено или нет это сообщение.

Для определения зависимости, мы устанавливаем параметр `dependency`, который может быть либо объектом, реализующим интерфейс `ICacheDependency`, либо массивом настроек, который может быть использован для генерации объекта зависимости. Следующий код определяет содержимое фрагмента, зависящее от изменения значения столбца `lastModified`:

```

...другое HTML-содержимое...
<?php if($this->beginCache($id, array('dependency'=>array(
    'class'=>'system.caching.dependencies.CDbCacheDependency',
    'sql'=>'SELECT MAX(lastModified) FROM Post')))) { ?>
...кэшируемое содержимое...
<?php $this->endCache(); } ?>
...другое HTML-содержимое...

```

Вариации (изменения)

Кэшируемое содержимое может быть изменено в соответствии с некоторыми параметрами. Например, личный профиль может по-разному выглядеть для разных пользователей. Для кэширования содержимого профиля мы бы хотели, чтобы кэшированная копия была различной в соответствии с идентификатором пользователя. По-существу, это значит, что мы должны использовать разные идентификаторы при вызове метода `beginCache()`.

Вместо того, чтобы спрашивать у разработчика различные идентификаторы, соответствующие некоторой схеме, существует класс `COutputCache`, включающий в себя такую возможность. Ниже приведен список встроенных вариаций:

- `varyByRoute`: если установлено в значение `true`, кэшированное содержимое будет изменяться в соответствии с настройками маршрута. Поэтому, каждая комбинация запрашиваемого контроллера и действия будут иметь разное кэшированное содержимое;
- `varyBySession`: если установлено в значение `true`, кэшированное содержимое будет изменяться в соответствии с идентификатором сессии. Поэтому, каждая пользовательская сессия может видеть различное содержимое и получать его из кэша;
- `varyByParam`: если установлено в качестве массива имен, кэшированное содержимое будет изменяться в соответствии с определенными GET параметрами. Например, если страница отображает содержимое сообщения в зависимости от GET-параметра `id`, мы можем определить `varyByParam` в виде массива `array('id')` и затем кэшировать содержимое каждого сообщения. Без такой вариации, мы могли бы кэшировать только одно сообщение;
- `varyByExpression`: если установлено в качестве выражения PHP, кэшированное содержимое будет изменяться в соответствии с результатом данного выражения PHP. Доступно с версии 1.0.4.

Типы запросов

Иногда мы хотим, чтобы кэширование фрагмента было включено только для некоторых типов запроса. Например, страницу с формой мы хотим кэшировать только тогда, когда она инициализирована (GET запросом) (не заполнена). Любое последующее отображение формы (получившееся POST запросом) (заполненная форма) не должно быть кэшировано, потому что может содержать данные, введенные пользователем. Чтобы так сделать, мы определяем параметр `requestTypes`:

```

...другое HTML содержимое...

```

```
<?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>
...кэшируемое содержимое...
<?php $this->endCache(); } ?>
...другое HTML содержимое...
```

Вложенное кэширование

Кэширование фрагментов может быть вложенным. Это значит, что кэшированный фрагмент окружен более крупным фрагментом (содержится в нем), который тоже кэшируется. Например, комментарии кэшированы во внутреннем фрагменте кэша, и они же кэшированы вместе с содержимым сообщения во внешнем фрагменте кэша.

```
...другое HTML содержимое...
<?php if($this->beginCache($id1)) { ?>
...внешнее кэшируемое содержимое...
    <?php if($this->beginCache($id2)) { ?>
        ...внутреннее кэшируемое содержимое...
        <?php $this->endCache(); } ?>
    ...внешнее кэшируемое содержимое...
<?php $this->endCache(); } ?>
...другое HTML содержимое...
```

Параметры кэширования могут быть различными для вложенных кэшей. Например, внутренний и внешний кэши в вышеприведенном примере могут иметь разные сроки хранения. Когда кэшированные данные во внешнем кэше становятся недействительны, внутренний кэш все еще может выдавать действительные фрагменты. Тем не менее, это *неверно* в обратном случае. Если внешний кэш содержит действительные данные, он всегда будет давать кэшированную копию, даже если у содержимого внутреннего кэша истек срок действия (оно устарело).

Кэширование страниц

Кэширование страниц — это кэширование всего содержимого страницы. Кэширование страниц может встречаться в различных местах. Например, выбрав соответствующий странице заголовок, браузер пользователя может кэшировать просматриваемую страницу на некоторое время. Веб-приложение также может само хранить содержимое страницы в кэше. В данном подразделе мы рассмотрим именно второй вариант.

Кэширование страницы может быть рассмотрено как частный случай кэширования фрагмента. Из-за того, что содержимое страницы часто генерируется применением макета к представлению, кэширование не будет работать, если мы просто вызовем в макете методы `beginCache()` и `endCache()`. Причина этого в том, что макет применяется при вызове метода `CController::render()` *после* оценки формирования содержимого представления.

Для кэширования всей страницы мы должны пропустить этап формирования содержимого страницы. Для выполнения этой задачи мы можем использовать класс `COutputCache` как фильтр действия. В коде ниже показано, как можно сконфигурировать фильтр кэша:

```
public function filters()
{
    return array(
        array(
            'COutputCache',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}
```

```
}
```

Вышеприведенная конфигурация фильтра создает фильтр, применяемый ко всем действиям контроллера. Мы можем ограничить это поведение одним или несколькими действиями только используя оператор +. Подробнее с работой фильтров можно ознакомиться в теме фильтры.

Подсказка: Мы можем использовать класс `COutputCache` в качестве фильтра, поскольку он наследует класс `CFilterWidget`, т.е. оба этих класса одновременно являются и виджетами и фильтрами. Фактически, способ работы виджета очень похож на работу фильтра: виджет (фильтр) выполняется до того, как любое вложенное содержимое (действие) будет сформировано (выполнено), а выполнение виджета (фильтра) заканчивается после того, как вложенное содержимое (действие) будет сформировано (выполнено).

Кэширование динамического содержимого

Когда мы используем кэширование фрагментов или кэширование страниц, то часто попадаем в ситуацию, когда целые части содержимого на выходе относительно статичны, кроме одного или нескольких мест. Например, страница помощи может отображать статичную вспомогательную информацию наряду с именем авторизованного пользователя, отображаемого вверху.

Для решения этой проблемы мы можем варьировать содержимое кэша в зависимости от имени пользователя, но это было бы очень большой тратой драгоценного места (на диске) для кэширования практически одинакового содержимого, кроме имени пользователя. Мы также можем разделить страницу на несколько фрагментов и кэшировать их по отдельности, но это усложняет наше представление и делает код слишком сложным. Наилучший способ — использование возможности *динамического содержимого*, обеспечиваемой классом `CController`.

Динамическое содержимое означает фрагмент содержимого на выходе, который не должен кэшироваться, даже если он включен в кэшированный фрагмент. Для обеспечения динамичности содержимого, его нужно генерировать каждый раз, даже если окружающий контент взят из кэша. Для решения этой задачи, нам требуется, чтобы динамическое содержимое генерировалось некой функцией или методом.

Мы вызываем метод `CController::renderDynamic()` для вставки динамического содержимого в нужное место.

```
...другое HTML-содержимое...
<?php if($this->beginCache($id)) { ?>
...фрагмент кэшируемого содержимого...
    <?php $this->renderDynamic($callback); ?>
```

```
...фрагмент кэшируемого содержимого...
```

```
<?php $this->endCache(); } ?>
...другое HTML-содержимое...
```

В коде выше, `$callback` — это корректный обратный PHP-вызов. Это может быть строка с именем метода текущего контроллера или глобальной функции. Также это может быть массив, ссылающийся на метод класса. Любые дополнительные параметры в методе `renderDynamic()` должны быть переданы обратному вызову. Обратный вызов должен не отображать динамическое содержимое, а вернуть его.

Расширение Yii

Обзор

Дописывание Yii путем расширения — стандартная практика в процессе разработки. Например, для написания нового контроллера, вам необходимо расширить Yii путем наследования его класса `CController`; для написания виджета — класса `CWidget` или класса уже существующего виджета. Если дописанный код оформлен для использования сторонними разработчиками, мы называем его *расширением (extension)*. Как правило, каждое расширение служит только для одной цели. Используя терминологию, принятую в Yii, расширения можно классифицировать следующим образом:

- компонент приложения;
- поведение;
- виджет;
- контроллер;
- действие;
- фильтр;
- команда консоли;
- валидатор: компонент, наследующий класс `CValidator`;
- помощник: класс, содержащий только статические методы, схожий с глобальной функцией, использующей имя класса в качестве пространства имен;
- модуль: самодостаточная программная единица, состоящая из моделей, действий, контроллеров и необходимых компонентов. Модуль во многом схож с приложением. Основное отличие состоит в том, что модули входят в состав приложения. Например, у нас может быть модуль, реализующий функционал управления пользователями.

Впрочем, расширение может и не соответствовать ни одной из перечисленных категорий. Yii изначально был спроектирован таким образом, чтобы любую его часть можно было изменить и дополнить для любых нужд.

Использование расширений

Порядок использования расширений, как правило, включает три шага:

1. Скачать расширение из репозитория Yii;
2. Распаковать расширение в подпапку `extensions/xyz` базовой директории приложения, где `xyz` — имя расширения;
3. Подключить, настроить и использовать расширение.

Каждое расширение уникально идентифицируется по имени. Если расширение называется `xyz`, то, используя псевдоним пути `ext.xyz`, мы всегда можем определить папку, в которой хранятся файлы данного расширения.

Примечание: Псевдоним `ext` доступен с версии 1.0.8. Ранее мы должны были использовать псевдоним `application.extensions` для обращения к директории, содержащей все расширения. В дальнейшем, мы предполагаем, что определен псевдоним `ext`. Поэтому, если Вы используете версию 1.0.7 или ниже, Вы должны заменить `ext` на `application.extensions`.

Разные расширения предъявляют разные требования к импорту, настройке и порядку использования. Ниже, мы приведем несколько общих вариантов использования расширений согласно классификации, представленной в обзоре.

Расширения Zii

Перед тем, как рассказать о использовании сторонних расширений, стоит упомянуть библиотеку расширений Zii — набор расширений, разрабатываемый командой Yii и включаемой в каждую новую версию Yii начиная с 1.1.0. Библиотека Zii размещается на Google Code в отдельном проекте `zii`. При использовании расширения Zii, необходимо обращаться к соответствующим классам используя псевдоним пути вида `zii.path.to.ClassName`. Здесь `zii` — предопределённый в Yii маршрут, соответствующий корневой директории библиотеки Zii. К примеру, чтобы использовать `CGridView`, необходимо использовать в шаблоне представления следующий код:

```
$this->widget('zii.widgets.grid.CGridView', array(
    'dataProvider'=>$dataProvider,
));
```

Компонент приложения

Для использования компонента приложения в первую очередь необходимо изменить конфигурацию приложения, добавив новый элемент в свойство `components`:

```
return array(
    // 'preload'=>array('xyz',...),
    'components'=>array(
```

```

        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // прочие настройки компонентов
    ),
);

```

Теперь можно обращаться к компоненту в любом месте приложения через `Yii::app()->xyz`. Компонент будет создан отложено (т.е. в момент первого обращения), если, конечно, мы не укажем его в свойстве `preload`.

Поведение

Поведение может быть использовано в любом компоненте. Делается это в два шага: присоединение к компоненту и вызов метода поведения из компонента. К примеру:

```

// $name соответствует уникальному поведению компонента
$component->attachBehavior($name,$behavior);
// test() является методом $behavior
$component->test();

```

Чаще всего поведение присоединяется к компоненту с использованием конфигурации, а не вызовом метода `attachBehavior`. К примеру, чтобы присоединить поведение к компоненту приложения, мы можем использовать следующую конфигурацию:

```

return array(
    'components'=>array(
        'db'=>array(
            'class'=>'CDbConnection',
            'behaviors'=>array(

                'xyz'=>array(

                    'class'=>'ext.xyz.XyzBehavior',
                    'property1'=>'value1',
                    'property2'=>'value2',
                ),
            ),
        ),
    ),
    //...
),
);

```

Приведённый выше код присоединяет поведение `xyz` к компоненту приложения `db`. Это возможно так как `CApplicationComponent` определяет свойство `behaviors`. При инициализации компонент присоединит перечисленные в нём поведения.

Для классов `CController`, `CFormModel` и `CActiveRecord`, которые необходимо расширять, присоединение поведений происходит при помощи переопределения метода `behaviors()`. При инициализации, классы автоматически присоединят поведения, объявленные в данном методе. К примеру:

```

public function behaviors()

```

```
{
    return array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzBehavior',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
    );
}
```

Виджет

Виджеты в основном используются в представлениях. Виджетом класса `XyzClass` расширения `xyz`, можно воспользоваться в представлении следующим образом:

```
// виджет без внутреннего содержимого
<?php $this->widget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

// виджет, который может иметь внутреннее содержимое
<?php $this->beginWidget('ext.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

...содержимое виджета...

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

Действие

Действия используются в контроллерах для обработки запросов пользователя. Действие класса `XyzClass` расширения `xyz` можно использовать путем переопределения метода `CController::actions` класса нашего контроллера:

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



```
}
```

Теперь к действию можно обратиться по маршруту `test/xyz`.

Фильтры

Фильтры также используются в контроллерах. В основном они используются в действиях для осуществления пред- и пост-обработки пользовательского запроса. Фильтр класса `XYZClass` расширения `xyz` можно использовать путем переопределения метода `CController::filters` в нашем классе контроллера:

```
class TestController extends CController
{
    public function filters()
    {
        return array(
            array(
                'ext.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // прочие фильтры
        );
    }
}
```

Выше мы можем использовать операторы '+' и '-' в первом элементе массива для применения фильтра только к определенным действиям. Подробнее ознакомиться можно в документации к `CController`.

Контроллер

Контроллер предоставляет набор действий, которые могут быть запрошены пользователем. Для использования расширения контроллера необходимо настроить свойство `CWebApplication::controllerMap` в конфигурации приложения:

```
return array(
    'controllerMap'=>array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // прочие контроллеры
    ),
);
```

Теперь к действию `a` контроллера можно обратиться через маршрут `xyz/a`.

Валидатор

Валидатор применяется в классе модели (наследующего либо `CFormModel` или `CActiveRecord`). Класс валидатора `XYZClass` расширения `xyz` используется путем переопределения метода `CModel::rules` в нашем классе модели:

```
class MyModel extends CActiveRecord // or CFormModel
{
    public function rules()
```



```

{
    return array(
        array(
            'attr1', 'attr2',
            'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // прочие правила проверки
    );
}
}

```

Команда консоли

Расширение консольной команды, как правило, используется для добавления новой команды в утилите `yii`. Консольную команду `XyzClass` расширения `xyz` можно использовать, настроив конфигурацию консольного приложения:

```

return array(
    'commandMap'=>array(
        'xyz'=>array(
            'class'=>'ext.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // прочие команды
    ),
);

```

Теперь в утилите `yii` добавилась еще одна команда `xyz`.

Примечание: Консольное приложение, как правило, использует иной файл конфигурации нежели веб-приложение. Если приложение было создано командой консоли `yii webapp`, то конфигурационный файл для консоли `protected/yii` находится в `protected/config/console.php`, а конфигурация веб-приложения — в `protected/config/main.php`.

Модуль

Информация о порядке использования и создания модулей представлена в разделе Модуль.

Компонент общего вида

Чтобы использовать компонент общего вида, нужно для начала включить его класс:

```

Yii::import('ext.xyz.XyzClass');

```

Теперь мы можем создавать экземпляр этого класса, настроить его свойства, вызывать его методы. Кроме того, можно его расширить для создания дочерних классов.

Создание расширений

Поскольку создание расширений подразумевает их использование сторонними разработчиками, процесс создания требует дополнительных усилий. Ниже приведены основные правила, которые необходимо соблюдать при создании расширений:

- расширение должно быть самодостаточным, т.е. зависимость от внешних ресурсов должна быть минимальна. Очень неудобно, когда для работы расширения требуется устанавливать дополнительные пакеты, классы и иные файлы ресурсов;
- все файлы расширения должны быть собраны в одной папке, имя которой должно совпадать с именем расширения;
- классы расширения должны начинаться с префикса, чтобы избежать конфликтов имен с классами других расширений;
- расширение должно включать подробную документацию по API и порядку установки, чтобы сократить время, необходимое для изучения и работы с расширением;
- расширение должно использовать подходящую лицензию. Если вы хотите, чтобы ваше расширение могло быть использовано как открытыми, так и закрытыми проектами, вы можете воспользоваться лицензиями BSD, MIT и др., но не GPL, т.к. последняя требует, чтобы код, где было использовано ваше расширение, также был открыт.

Ниже мы расскажем о том, как создать новое расширение в соответствии с классификацией, приведенной в обзоре. Пояснения в равной степени распространяются и на расширения, используемые исключительно в собственных проектах.

Компонент приложения

Компонент приложения должен реализовывать интерфейс `IApplicationComponent` или расширять класс `CApplicationComponent`. Основной метод, который необходимо реализовать, — `IApplicationComponent::init`. В этом методе компонент инициализируется. Метод вызывается после того, как компонент создан и получены начальные значения, обозначенные в конфигурации приложения.

По умолчанию, компонент приложения создается и инициализируется только в момент первого обращения к нему в ходе обработки запроса. Если необходимо принудительно создавать компонент сразу после создания экземпляра приложения, то нужно добавить идентификатор этого компонента в свойство `CApplication::preload`.

Поведение

Для того, чтобы создать поведение необходимо реализовать интерфейс `IBehavior`. Для удобства в Yii имеется класс `CBehavior`, реализующий этот интерфейс и предоставляющий некоторые общие методы. Наследуемые классы могут реализовать дополнительные методы, которые будут доступны в компонентах, к которым прикреплено поведение.

При разработке поведений для `CModel` и `CActiveRecord` можно наследовать `CModelBehavior` и, соответственно, `CActiveRecordBehavior`. Эти базовые классы предоставляют дополнительные возможности, специально созданные для `CModel` и `CActiveRecord`. К примеру, класс `CActiveRecordBehavior` реализует набор методов для обработки событий жизненного цикла `ActiveRecord`. Наследуемый класс, таким образом, может перекрыть эти методы и выполнить код, участвующий в жизненном цикле AR.

Следующий код демонстрирует пример поведения `ActiveRecord`. Если это поведение назначено объекту AR и вызван метод `save()`, атрибутам `create_time` и `update_time` будет автоматически выставлено текущее время.

```
class TimestampBehavior extends ActiveRecordBehavior
{
    public function beforeSave($event)
    {
        if($this->owner->isNewRecord)
            $this->owner->create_time=time();
        else
            $this->owner->update_time=time();
    }
}
```

Виджет

Виджет должен расширять класс `CWidget` или производные от него.

Наиболее простой способ создать виджет — расширить существующий виджет и переопределить его методы или заменить значения по умолчанию. Например, если вы хотите заменить CSS-стиль для

CTabView, то, используя виджет, нужно сконфигурировать свойство CTabView::cssFile. Можно также расширить класс CTabView, чтобы при использовании виджета не требовалась постоянная конфигурация, следующим образом:

```
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();
    }
}
```

Выше мы переопределяем метод CWidget::init и, если свойство CTabView::cssFile не установлено, присваиваем ему значение URL нового CSS-стиля по умолчанию. Файл нового CSS-стиля необходимо поместить в одну папку с файлом класса **MyTabView**, чтобы их можно было упаковать как расширение. Поскольку CSS-стиль не доступен из веб, его необходимо опубликовать как ресурс.

Чтобы написать виджет с нуля, нужно, как правило, реализовать два метода: CWidget::init и CWidget::run. Первый метод вызывается, когда мы используем конструкцию `$this->beginWidget` для вставки виджета в представление, а второй — когда используется конструкция `$this->endWidget`. Если необходимо получить и обработать некоторый контент между вызовами этих двух функций можно запустить буферизацию вывода в CWidget::init и получать сохраненный вывод для дальнейшей обработки в методе CWidget::run.

На странице, где используется виджет, он обычно подключает CSS-стили, JavaScript файлы и прочие файлы ресурсов. Файлы такого рода называются *ресурсы*, поскольку хранятся с файлом класса виджета и, как правило, недоступны веб-пользователям. Для того, чтобы дать к ним доступ, их необходимо опубликовать, используя CWebApplication::assetManager, как показано в фрагменте кода выше. Помимо этого, если необходимо подключить файлы CSS-стиля или JavaScript на текущей странице, их необходимо зарегистрировать посредством CClientScript:

```
class MyWidget extends CWidget
{
    protected function registerClientScript()
    {
        // ...подключаем здесь файлы CSS или JavaScript...
        $cs=Yii::app()->clientScript;
        $cs->registerCssFile($cssFile);
        $cs->registerScriptFile($jsFile);
    }
}
```

Виджет также может иметь собственные файлы представлений. В этом случае необходимо создать папку **views** в папке с файлом класса виджета и поместить в нее все файлы представлений. Чтобы отрендерить представления виджета в его классе используется конструкция `$this->render('ViewName')`, аналогично использованию в контроллере.

Действие

Действие должно расширять класс CAction или производные от него. IAction::run — основной метод, который необходимо реализовать для действия.

Фильтр

Фильтр должен расширять класс `CFilter` или производные от него. Основными методами, которые необходимо реализовать, являются `CFilter::preFilter` и `CFilter::postFilter`. Первый вызывается до выполнения действия, второй — после.

```
class MyFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // применяется до выполнения действия
        return true; // значение false возвращается, если действие не должно выполняться
    }

    protected function postFilter($filterChain)
    {
        // применяется после завершения выполнения действия
    }
}
```

Параметр `$filterChain` — экземпляр класса `CFilterChain`, содержащий информацию о действии, к которому применяются фильтры в настоящий момент.

Контроллер

Контроллер, предлагаемый как расширение, должен наследовать класс `CExtController`, а не класс `CController`. Основной причиной этого является то, что в случае класса `CController` предполагается, что файлы представлений располагаются в `application.views.ControllerID`, а в случае класса `CExtController` считается, что файлы представлений находятся в папке `views`, расположенной в папке с файлом класса этого контроллера. Очевидно, что расширение-контроллер удобнее распространять, когда все файлы расширения собраны в одном месте.

Валидатор

Валидатор должен расширять класс `CValidator` и реализовывать его метод `CValidator::validateAttribute`.

```
class MyValidator extends CValidator
{
    protected function validateAttribute($model,$attribute)
    {
        $value=$model->$attribute;
        if($value has error)
            $model->addError($attribute,$errorMessage);
    }
}
```

Команда консоли

Консольная команда должна расширять класс `CConsoleCommand` и реализовывать его метод `CConsoleCommand::run`. При желании можно переопределить метод `CConsoleCommand::getHelp`, который отвечает за информационную справку по команде.

```
class MyCommand extends CConsoleCommand
{
    public function run($args)
    {
        // $args — массив аргументов, переданных с командой
    }
}
```

```
public function getHelp()
{
    return 'Usage: how to use this command';
}
}
```

Модуль

Информация о порядке использования и создания модулей представлена в разделе Модуль.

Если сформулировать требования в общем виде, то модуль должен быть самодостаточным, файлы ресурсов (CSS, JavaScript, изображения), используемые модулем, должны распространяться вместе с модулем, а сам модуль должен публиковать ресурсы, чтобы они были доступны для веб-пользователей.

Компонент общего вида

Разработка компонента общего вида аналогична написанию класса. Компонент, как и модуль, должен быть самодостаточен и удобен для использования разработчиками.

;

Обзор

Примечание: Функционал, описанный в данном разделе, реализован начиная с Yii 1.1. Тем не менее, это не означает, что вы не можете тестировать приложения, написанные с использованием Yii 1.0.x.

Существует множество фреймворков тестирования, таких как PHPUnit и SimpleTest, которые могут вам в этом помочь.

Тестирование — важная составляющая процесса разработки ПО. Вне зависимости от того, осознаём ли мы это или нет, мы проводим тестирование на протяжении всего процесса разработки приложения. К примеру, при написании PHP-класса мы используем `echo` или `die` для того, чтобы проверить корректность выполнения метода. При создании страницы, содержащей сложные HTML-формы, мы вводим некоторые тестовые данные, чтобы проверить её работу. Более опытные разработчики напишут код, автоматизирующий этот процесс и дающий возможность выполнить все тесты автоматически за один раз. Этот процесс называется *автоматизированное тестирование* и является главной темой данного раздела. В Yii поддерживается *модульное тестирование* и *функциональное тестирование*.

Модульный тест проверяет, что единица кода работает так, как должна. В ООП такой единицей является класс. Поэтому модульный тест должен проверить, что каждый открытый метод класса работает должным образом. То есть, имея входные тестовые данные, тест проверяет, что метод возвращает ожидаемый результат. Модульные тесты обычно пишутся теми же, кто разрабатывает сам класс.

Функциональный тест проверяет, что некоторая возможность (например, управление записями блога) работает как надо. Функциональный тест, по сравнению с модульным, относится к более высокому уровню так как чаще всего производится проверка работы нескольких классов. Функциональные тесты обычно разрабатываются теми, кто очень хорошо знает требования к системе (это могут быть как разработчики, так и инженеры QA).

Разработка через тестирование

Ниже приведён цикл разработки через тестирование (TDD):

1. Создаём новый тест, описывающий функционал планируемой возможности. Тест, конечно же, при первом запуске завершится неудачей, так как сама возможность ещё не реализована.
2. Запускаем все тесты. Проверяем, что все они завершились неудачно.
3. Пишем код, который проходит тесты.
4. Запускаем все тесты. Проверяем, что все они завершились удачно.
5. Рефакторим написанный код и проверяем, что он всё ещё проходит тесты.

Повторяем шаги 1 — 5 для нового функционала.

Настройка тестового окружения

Тестирование в Yii требует установленного PHPUnit 3.4+ и Selenium Remote Control 1.0+. Как устанавливать PHPUnit и Selenium Remote Control вы можете прочитать в их документации.

При использовании консольной команды `yii webapp` для создания нового приложения генерируются следующие директории, позволяющие писать и выполнять тесты:

testdrive/	
protected/	файлы приложения
tests/	тесты
fixtures/	фикстуры БД
functional/	функциональные тесты
unit/	модульные тесты
report/	отчёты по покрытию кода тестами
bootstrap.php	загрузчик
phpunit.xml	конфигурация для PHPUnit
WebTestCase.php	базовый класс для функциональных тестов страниц

Как показано выше, код тестов главным образом находится в трёх директориях: `fixtures`, `functional` и `unit`. Директория `report` используется для хранения генерируемых отчётов о покрытии кода тестами. Для того, чтобы запустить тесты (как модульные, так и функциональные), необходимо выполнить следующие команды в консоли:

```
% cd testdrive/protected/tests
% phpunit functional/PostTest.php // запускает отдельный тест
% phpunit --verbose functional // запускает все тесты в директории 'functional'
% phpunit --coverage-html ./report unit
```

Последняя команда выполнит все тесты в директории `unit` и создаст отчёт о покрытии кода в директории `report`. Стоит отметить, что для отчётов требуется установленное расширение `xdebug`.

Загрузчик тестов

Давайте посмотрим, что может находиться в файле `bootstrap.php`. Мы уделяем ему большое внимание, так как он играет такую же роль, как входной скрипт и является стартовой точкой при запуске набора тестов.

```
$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);
```

В приведённом коде мы сначала подключаем файл `yiit.php`, который инициализирует некоторые глобальные константы и подключает необходимые базовые классы тестов. Затем мы создаём экземпляр приложения, используя файл конфигурации `test.php`. Конфигурация в нём наследуется от `main.php` и добавляет компонент `fixture` (класс `CDbFixtureManager`). Фикстуры будут детально описаны в следующем разделе.

```
return CMap::mergeArray(
    require(dirname(__FILE__).'main.php'),
    array(
        'components'=>array(
            'fixture'=>array(
                'class'=>'system.test.CDbFixtureManager',
            ),
            /* раскомментируйте, если вам нужно подключение к тестовой БД
            'db'=>array(
```

```

        'connectionString'=>'DSN для БД',
    ),
    */
),
)
);

```

При запуске тестов, включающих работу с базой данных, необходимо переопределить БД для того, чтобы не испортить реальные данные или данные, используемые при разработке. Для этого необходимо раскомментировать конфигурацию `db` выше и указать DSN тестовой базы в свойстве `connectionString`. При помощи такого входного скрипта при запуске тестов мы получаем экземпляр приложения, максимально приближённый к реальному. Главное отличие в том, что в нём есть поддержка фикстур и используется тестовая БД.

Тестирование

Определение фикстур

Автоматические тесты необходимо выполнять неоднократно. Мы хотели бы выполнять тесты в некоторых известных состояниях для гарантии повторяемости процесса тестирования. Эти состояния называются *фикстуры*. Например, для тестирования функции создания записи в приложении блога, каждый раз, когда мы выполняем тесты, таблицы, хранящие соответствующие данные о записях (например, таблицы `Post`, `Comment`), должны быть восстановлены к некоторому фиксированному состоянию. Документация по PHPUnit хорошо описывает основную установку фикстур. В основном в этом разделе мы описываем установку фикстур базы данных так, как мы только что описали в примере.

Установка фикстур базы данных является, наверное, одной из наиболее длительных частей в тестировании основанных на БД веб-приложений. Yii вводит компонент приложения `CDbFixtureManager` для облегчения этой проблемы. В основном он делает следующие вещи при выполнении ряда тестов:

- Перед выполнением всех тестов сбрасывает все таблицы, относящиеся к тестам к некоторому известному состоянию.
- Перед выполнением отдельного тестового метода сбрасывает определенные таблицы к некоторому известному состоянию.
- Во время выполнения тестового метода обеспечивает доступ к строкам данных, которые вносятся в фикстуру.

Для использования компонента `CDbFixtureManager`, мы конфигурируем его конфигурации приложения следующим образом:

```

return array(
    'components'=>array(
        'fixture'=>array(
            'class'=>'system.test.CDbFixtureManager',
        ),
    ),
);

```

Далее мы сохраняем данные фикстуры в директории `protected/tests/fixtures`. Эта директория может быть настроена свойством `CDbFixtureManager::basePath` конфигурации приложения. Данные фикстур организованы как коллекция PHP-файлов, называемых файлами фикстур. Каждый файл фикстуры возвращает массив, представляющий начальные строки данных для конкретной таблицы. Имя файла - такое же, как название таблицы. Далее приведен пример данных фикстуры для таблицы `Post`, сохраненной в файле `Post.php`:

```

<?php
return array(

```



```

'sample1'=>array(
    'title'=>'Тестовая запись 1',
    'content'=>'Содержимое тестовой записи 1',
    'createTime'=>1230952187,
    'authorId'=>1,
),
'sample2'=>array(
    'title'=>'Тестовая запись 2',
    'content'=>'Содержимое тестовой записи 2',
    'createTime'=>1230952287,
    'authorId'=>1,
),
);

```

Как видим, в коде выше возвращаются 2 строки данных. Каждая строка представлена в виде ассоциативного массива, ключи которого — это имена столбцов, а значения - значения соответствующих столбцов. Кроме того, каждая строка индексирована строкой (например `sample1`, `sample2`), которую называют *псевдоним строки*. Позже, когда мы пишем тестовые скрипты, мы можем легко обращаться к строке по ее псевдониму. Мы опишем это подробно в следующем разделе.

Вы могли заметить, что мы не определяем значения столбца `id` в коде фикстуры выше. Это потому, что столбец `id` — автоинкрементный первичный ключ, значение которого будет заполнено при вставке новых строк.

При первом обращении к компоненту `CDbFixtureManager` он будет просматривать каждый файл фикстуры и использовать его для сброса соответствующей таблицы. Он сбрасывает таблицу, очищая её, сбрасывая значение первичного ключа, и затем вставляя строки данных из файла фикстуры в таблицу.

Иногда мы не хотим сбрасывать каждую таблицу, имеющую файл фикстуры, прежде, чем мы выполним ряд тестов, потому что сброс слишком многих файлов фикстур может занять длительное время. В этом случае, мы можем написать PHP-скрипт для возможности настройки работы инициализации. PHP-скрипт должен быть сохранен в файле `init.php` в той же директории, что и файлы фикстур. Когда компонент `CDbFixtureManager` обнаружит этот скрипт, он выполнит этот скрипт вместо того, чтобы сбрасывать каждую таблицу.

Также возможно, что нам не нравится способ сброса таблицы по умолчанию, то есть, очистка таблицы полностью и вставка данных фикстуры. Если дело обстоит так, мы можем написать скрипт инициализации для определенного файла фикстуры. Скрипт должен иметь имя, в начале которого идет имя таблицы, а далее - `.init.php`. Например, скрипт инициализации для таблицы `Post` назывался бы `Post.init.php`. Когда компонент `CDbFixtureManager` увидит этот скрипт, он выполнит скрипт вместо того, чтобы использовать значение сброса таблицы по умолчанию.

Подсказка: Наличие большого количества файлов фикстур может сильно увеличить время выполнения теста. Поэтому, Вы должны создавать файлы фикстур только для тех таблиц, содержание которых может измениться во время теста. Таблицы, которые служат для просмотра, не изменяются и, таким образом, не нуждаются в файлах фикстур.

В следующих двух разделах мы опишем, как использовать фикстуры, которыми управляет компонент `CDbFixtureManager`, в модульных и функциональных тестах.

Модульное тестирование

Поскольку тестировочная часть Yii построена на PHPUnit, рекомендуется сначала изучить документацию PHPUnit, чтобы получить общее представление о том, как писать модульные тесты. Далее мы приведём основные принципы написания модульных тестов в Yii:

- Модульный тест — это класс `xyzTest`, наследующий класс `CTestCase` или `CDbTestCase`, где `xyz` — название тестируемого класса. Например, для тестирования класса `Post` по соглашению мы называем соответствующий класс модульного теста `PostTest`. Базовый класс `CTestCase` предназначен для общего модульного тестирования, а класс `CDbTestCase` — для тестирования

классов моделей Active Record. Мы можем использовать все методы этих классов, унаследованные от класса `PHPUnit_Framework_TestCase`, поскольку он — предок обоих классов (`CTestCase` и `CDbTestCase`).

- Класс модульного теста хранится в PHP-файле с именем `XYZTest.php`. По соглашению файл модульного теста может быть сохранен в директории `protected/tests/unit`.
- Основное содержание тестового класса — набор тестовых методов с именами вида `testABC`, где `ABC` — часто имя тестируемого метода класса.
- Обычно тестовый метод содержит последовательность выражений утверждений (например, `assertTrue`, `assertEquals`), служащих контрольными точками при проверке поведения целевого класса.

Далее мы опишем, как писать модульные тесты для классов моделей Active Record. Мы расширяем наши тестовые классы, наследуя их от класса `CDbTestCase`, поскольку он обеспечивает поддержку фикстур базы данных, которые мы представили в предыдущем разделе.

Предположим, что мы хотим проверить класс модели `Comment` в демо-блоге. Начнем с создания класса `CommentTest` и сохраним его в файле `protected/tests/unit/CommentTest.php`:

```
class CommentTest extends CDbTestCase
{
    public $fixtures=array(
        'posts'=>'Post',
        'comments'=>'Comment',
    );
}
```

В этом классе мы определяем переменную-член класса `fixtures` массивом, содержащий список фикстур, используемых в данном тесте. Массив представляет собой отображение имен фикстур на имена классов моделей или имена таблиц фикстур (например, фикстуры с именем `posts` на класс модели `Post`). Заметим, что при отображении на имя таблицы фикстуры мы должны использовать имя таблицы с префиксом `:` (например, `:Post`), чтобы отличать его от имени класса модели. А при использовании имен классов моделей, соответствующие таблицы будут рассматриваться в качестве таблиц фикстур. Как описано выше, таблицы фикстур будут сброшены в некоторое известное состояние каждый раз при выполнении тестового метода.

Имя фикстуры позволяет нам получить удобный доступ к данным фикстуры в тестовых методах. Следующий код показывает типичное использование:

```
// возвращает все строки таблицы фикстур `Comment`
$comments = $this->comments;
// возвращает строку с псевдонимом 'sample1' в таблице фикстур `Post`
$post = $this->posts['sample1'];
// возвращает экземпляр класса AR, представляющего строку данных фикстуры 'sample1'
$post = $this->posts('sample1');
```

Примечание: Если фикстура объявлена с использованием имени её таблицы (например, `'posts'=>:Post`), то третий пример в коде выше не является допустимым, так как мы не имеем информации о том, какой класс модели ассоциирован с таблицей.

Далее мы пишем метод `testApprove` для тестирования метода `approve` в классе модели `Comment`. Код очень прямолинеен: сначала мы вставляем комментарий со статусом ожидания, затем проверяем, комментарий имеет статус ожидания или другой, извлекая его из базы данных, и, наконец, мы вызываем метод `approve` и проверяем, изменился ли статус, как ожидалось.

```

public function testApprove()
{
    // вставить комментарий в лист ожидания
    $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));

    // проверить наличие комментария в листе ожидания
    $comment=Comment::model()->findByPrimaryKey($comment->id);
    $this->assertTrue($comment instanceof Comment);

    $this->assertEquals(Comment::STATUS_PENDING,$comment->status);

    // вызвать метод approve() и проверить, что комментарий утвержден
    $comment->approve();
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
    $comment=Comment::model()->findByPrimaryKey($comment->id);
    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
}

```

Автоматическая генерация кода

Начиная с версии 1.1.2, в состав Yii входит веб-инструмент для генерации кода, называемый *Gii*. Он заменяет существовавший до этого консольный генератор `yii shell`. В данном разделе описано, как использовать Gii и как расширить его для ускорения разработки.

Использование Gii

Gii является модулем и должен быть использован в составе существующего приложения Yii. Для использования Gii необходимо отредактировать файл конфигурации приложения следующим образом:

```

return array(
    ...
    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',
            'password'=>'задайте свой пароль',
            // 'ipFilters'=>array(...список IP...),
            // 'newFileMode'=>0666,
            // 'newDirMode'=>0777,
        ),
    ),

```

```
),
);
```

Выше мы объявили модуль с именем `gii` и классом `GiiModule`. Также мы задали пароль, который будет использоваться для доступа к `Gii`.

По умолчанию, в целях безопасности, `Gii` доступен только для `localhost`. Если необходимо дать доступ к нему с других компьютеров, нужно задать свойство `GiiModule::ipFilters` как показано в коде выше.

Так как `Gii` будет генерировать и сохранять новые файлы с кодом в существующее приложение, необходимо убедиться в том, что процесс веб-сервера имеет на это права. Показанные выше свойства `GiiModule::newFileMode` и `GiiModule::newDirMode` содержат права, с которыми будут создаваться файлы и директории.

Примечание: `Gii` является инструментом разработчика. Поэтому он должен быть установлен исключительно на компьютере или сервере разработчика. Так как он может генерировать новые скрипты PHP, необходимо уделить особое внимание безопасности (пароль, IP фильтры).

Теперь можно запустить `Gii` по URL `http://hostname/path/to/index.php?r=gii`, где `http://hostname/path/to/index.php` — URL вашего приложения.

Если существующее приложение использует формат URL `path` (см. красивые адреса URL), мы можем запустить `Gii` по URL `http://hostnamepath/to/index.php/gii`. Может понадобится добавить следующие правила URL перед уже существующими:

```
'components'=>array(
    ...

    'urlManager'=>array(
        'urlFormat'=>'path',
        'rules'=>array(
            'gii'=>'gii',
            'gii/<controller:w+>'=>'gii/<controller>',
            'gii/<controller:w+>/<action:w+>'=>'gii/<controller>/<action>',
            ...существующие правила...
        ),
    ),
),
);
```

В составе `Gii` есть готовый набор генераторов кода. Каждый генератор отвечает за свой тип кода. К примеру, генератор контроллера создаёт класс контроллера вместе с несколькими шаблонами отображения; генератор модели создаёт класс `ActiveRecord` для определённой таблицы БД.

Последовательность работы с генератором следующая:

1. Зайти на страницу генератора;
2. Заполнить поля, которые задают параметры генерируемого кода. К примеру, для генерации модуля необходимо указать его ID;
3. Нажать кнопку `Preview` для предварительной оценки генерируемого кода. Вы увидите таблицу файлов, которые будут сгенерированы и сможете просмотреть их код;
4. Нажать кнопку `Generate` для создания файлов;
5. Просмотреть журнал генерации кода.

Расширение Gii

Несмотря на то, что включённые в состав `Gii` генераторы создают достаточно функциональный код, часто требуется его немного изменить или создать новый генератор по своему вкусу и потребностям. К примеру, нам может понадобиться изменить стиль генерируемого кода или добавить поддержку нескольких языков. Всё это может быть легко реализовано через `Gii`.

`Gii` можно расширять двумя способами: изменяя существующие шаблоны кодогенераторов и создавая свои генераторы.

Структура кодогенератора

Генератор кода размещается в директории, чьё имя является именем генератора. Директория обычно содержит:

<code>model/</code>	корневая директория генератора модели
<code>ModelCode.php</code>	модель, используемая для генерации кода
<code>ModelGenerator.php</code>	контроллер кодогенератора
<code>views/</code>	отображения генератора
<code>index.php</code>	шаблон по умолчанию
<code>templates/</code>	шаблоны кода
<code>default/</code>	набор шаблонов 'default'
<code>model.php</code>	шаблон для генерации класса модели

Путь поиска генераторов

Gii ищет генераторы в списке директорий, указанных в свойстве `GiiModule::generatorPaths`. В том случае, если необходимо добавить свои генераторы, следует настроить приложение следующим образом:

```
return array(
    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',

            'generatorPaths'=>array(

                'application.gii',    // псевдоним пути
            ),
        ),
    ),
);
```

Приведённые выше настройки заставляют Gii искать генераторы в директории с псевдонимом `application.gii` в дополнение к стандартному `system.gii.generators`.

Возможно иметь несколько одноимённых генераторов, если у них разные пути поиска. В этом случае будет использоваться генератор, путь поиска которого указан выше в `GiiModule::generatorPaths`.

Изменение шаблонов кода

Изменение шаблонов кода — самый простой и самый распространённый путь расширения Gii. Мы будем использовать примеры для того, чтобы описать, как изменить шаблоны кода. Допустим, нам необходимо изменить код, создаваемый генератором модели.

Сначала мы создаём директорию `protected/gii/model/templates/compact`. Здесь `model` означает, что мы собираемся *перекрыть* генератор модели по умолчанию. А `templates/compact` — что мы добавляем новый набор шаблонов кода `compact`.

После этого мы добавляем в настройки приложения в свойство `GiiModule::generatorPaths` значение `application.gii`, как показано в предыдущем подразделе.

Теперь открываем страницу генератора модели. Щёлкаем на поле `Code Template`. Вы должны увидеть выпадающий список, содержащий нашу только что созданную директорию шаблонов `compact`. Тем не менее, если мы выберем этот шаблон, будет выведена ошибка. Происходит это потому, что в наборе `compact` ещё нет самих шаблонов кода.

Скопируем файл `framework/gii/generators/model/templates/default/model.php` в `protected/gii/model/templates/compact`. Если попробовать сгенерировать код с набором `compact` ещё раз, генерация должна пройти успешно. Тем не менее, генерируемый код ничем не отличается от кода, получаемого из набора `default`.

Время сделать некоторые изменения. Откроем файл

`protected/gii/model/templates/compact/model.php`. Данный файл будет использован как шаблон отображения, что означает, что он может содержать выражения и код PHP. Изменим шаблон таким

образом, что метод `attributeLabels()` генерируемого кода будет использовать `Yii::t()` для перевода заголовков полей:

```
public function attributeLabels()
{
    return array(
        <?php foreach($labels as $name=>$label): ?>
            <?php echo "'$name' => Yii::t('application', '$label'),
        "; ?>
        <?php endforeach; ?>
    );
}
```

В каждом шаблоне кода у нас есть доступ к некоторым предопределённым переменным, таким как, например, `$labels`. Эти переменные задаются соответствующим генератором кода. Разные генераторы могут предоставлять шаблонам различные наборы переменных. Стоит внимательно изучить описание шаблонов кода по умолчанию.

Создание новых генераторов

В этом подразделе мы покажем, как реализовать новый генератор, который сможет создавать новые классы виджетов.

Сначала создадим директорию `protected/gii/widget`. В ней создадим следующие файлы:

- `WidgetGenerator.php`: содержит класс контроллера `WidgetGenerator`, который является входной точкой генератора виджетов.
- `WidgetCode.php`: содержит класс модели `WidgetCode`, который отвечает за логику генерации кода.
- `views/index.php`: отображение, содержащее форму ввода генератора.
- `templates/default/widget.php`: шаблон кода по умолчанию для генерации класса виджета.

Реализация `WidgetGenerator.php`

Файл `WidgetGenerator.php` предельно простой. Он содержит лишь следующий код:

```
class WidgetGenerator extends CCodeGenerator
{
    public $codeModel='application.gii.widget.WidgetCode';
}
```

Здесь мы описываем, что генератор будет использовать класс модели, чей псевдоним пути `application.gii.widget.WidgetCode`. Класс `WidgetGenerator` наследуется от `CCodeGenerator`, реализующего большое количество функций, включая действия контроллера, необходимые для координации процесса генерации кода.

Реализация `WidgetCode.php`

Файл `WidgetCode.php` содержит класс модели `WidgetCode`, в котором реализована логика генерации класса виджета на основе полученных от пользователя параметров. В данном примере будем считать, что единственное, что вводит пользователь — имя класса виджета. `WidgetCode` выглядит следующим образом:

```
class WidgetCode extends CCodeModel
{
    public $className;

    public function rules()
    {
        return array_merge(parent::rules(), array(
            array('className', 'required'),
        ));
    }
}
```

```

        array('className', 'match', 'pattern'=>'/^w+$/'),
    ));
}

public function attributeLabels()
{
    return array_merge(parent::attributeLabels(), array(
        'className'=>'Widget Class Name',
    ));
}

public function prepare()
{
    $path=Yii::getPathOfAlias('application.components.' . $this->className) . '.php';
    $code=$this->render($this->templatepath.'/widget.php');

    $this->files[]=new CCodeFile($path, $code);
}
}

```

Класс `WidgetCode` наследуется от `CCodeModel`. Как и в обычном классе модели, в данном классе мы реализуем методы `rules()` и `attributeLabels()` для валидации ввода и генерации подписей полей соответственно. Стоит отметить, что так как базовый класс `CCodeModel` уже описывает некоторое количество правил валидации и названий подписей, то мы должны объединить их с нашими правилами и подписями.

Метод `prepare()` подготавливает код к генерации. Главная задача метода — подготовить список объектов `CCodeFile`, каждый из которых представляет будущий файл с кодом. В нашем примере необходимо создать всего один объект `CCodeFile`, представляющий класс виджета, который будет сгенерирован в директории `protected/components`. Для непосредственной генерации кода используется метод `CCodeFile::render`. Данный метод содержит PHP-шаблон кода и возвращает сгенерированный код.

Реализация `views/index.php`

После реализации контроллера (`WidgetGenerator`) и модели (`WidgetCode`) самое время заняться отображением `views/index.php`:

```
<h1>Генератор виджета</h1>
```

```

<?php $form=$this->beginWidget('CCodeForm', array('model'=>$model)); ?>

<div class="row">
    <?php echo $form->labelEx($model,'className'); ?>
    <?php echo $form->textField($model,'className',array('size'=>65)); ?>
    <div class="tooltip">
        Класс виджета должен содержать только буквы.
    </div>
    <?php echo $form->error($model,'className'); ?>
</div>

```

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

В данном коде мы отображаем форму, используя виджет CCodeForm. В этой форме мы показываем поле для ввода атрибута `className` модели `WidgetCode`.

При создании формы мы можем использовать две замечательные возможности CCodeForm. Одна — подсказки для полей. Вторая — запоминание введённых значений.

Если вы использовали один из стандартных генераторов кода, вы могли заметить красивые всплывающие подсказки, появляющиеся рядом с полем при получении им фокуса. Использовать данную возможность очень легко: достаточно после поля вставить `div` с CSS классом `tooltip`.

Для некоторых полей полезно запомнить последнее верное значение и тем самым позволив пользователю не вводить значения повторно каждый раз, когда он использует генератор. Примером может служить поле ввода базового класса контроллера в стандартном генераторе. Такие поля изначально отображаются как подсвеченный статичный текст. При щелчке они превращаются в поля ввода.

Для того, чтобы сделать поле запоминаемым, необходимо сделать две вещи.

Во-первых, нужно описать правило валидации `sticky` для соответствующего атрибута модели. К примеру, для стандартного генератора контроллера используется приведённое ниже правило для запоминания атрибутов `baseClass` и `actions`:

```
public function rules()
{
    return array_merge(parent::rules(), array(
        ...
        array('baseClass', 'actions', 'sticky'),
    ));
}
```

Во-вторых, в отображении необходимо добавить CSS класс `sticky` контейнеру `div` поля ввода:

```
<div class="row sticky">
    ...поле ввода...
</div>
```

Реализация `templates/default/widget.php`

Наконец, мы создаём шаблон кода `templates/default/widget.php`. Как было описано ранее, он используется как РНР-шаблон отображения. В шаблоне кода мы всегда можем обратиться к переменной `$this`, которая содержит экземпляр модели кода. В нашем примере `$this` содержит объект `WidgetModel`. Таким образом, мы можем получить введённый пользователем класс виджета через `$this->className`.

```
<?php echo '<?php'; ?>

class <?php echo $this->className; ?> extends CWidget
{
    public function run()
    {
    }
}
```

На этом реализация генератора кода завершена. Обратиться к нему можно по URL `http://hostname/path/to/index.php?r=gii/widget`.

Специальные темы

Красивые адреса URL

Управление URL-адресами в веб-приложениях включает в себя два аспекта. Во-первых, приложению необходимо распарсить запрос пользователя, поступающий в виде URL, на удобоваримые параметры. Во-вторых, приложение должно предоставлять способ формирования адресов URL, с которыми оно сможет корректно работать. В Yii-приложениях эти задачи решаются с использованием класса `CUrlManager`.

Создание адресов URL

В принципе, адреса URL можно захардкодить прямо в представлениях контроллера, однако куда удобнее создавать их динамически:

```
$url=$this->createUrl($route,$params);
```

где `$this` относится к экземпляру контроллера; `$route` соответствует маршруту запроса, а `$params` является списком параметров `GET` для добавления к URL.

По умолчанию, адреса создаются посредством `createUrl` в `get`-формате. Например, при значениях параметров `$route='post/read'` и `$params=array('id'=>100)`, получим такой URL:

```
/index.php?r=post/read&id=100
```

где параметры указаны в виде набора пар `имя=значение`, соединенных знаком `&`, а параметр `r` указывает на маршрут. Однако, этот формат не очень дружелюбен по отношению к пользователю.

Мы можем сделать так, чтобы адрес, приведенный в качестве примера выше, выглядел более аккуратно и понятно за счет использования формата `path`, который исключает использование строки запроса и включает все `GET`-параметры в информационную часть адреса URL:

```
/index.php/post/read/id/100
```

Для изменения формата представления адреса URL, нужно настроить компонент приложения `urlManager` таким образом, чтобы метод `createUrl` мог автоматически переключиться на использование нового формата, а приложение могло корректно воспринимать новый формат адресов URL:

```
array(
    ...
    'components'=>array(
        ...
        'urlManager'=>array(
            'urlFormat'=>'path',
        ),
    ),
);
```

Обратите внимание, что указывать класс компонента `urlManager` не требуется, т.к. он уже объявлен как `CUrlManager` в `CWebApplication`.

Подсказка: Адрес URL, генерируемый методом `createUrl` является относительным. Для того, чтобы получить абсолютный адрес, нужно добавить префикс, используя `Yii::app()->request->hostInfo`, или вызвать метод `createAbsoluteUrl`.

Человекопонятные URL

Если в качестве формата адреса URL используется `path`, то мы можем определить правила формирования URL, чтобы сделать адреса более привлекательными и понятными с точки зрения пользователя. Например, мы можем использовать короткий адрес `/post/100` вместо длинного варианта

`/index.php/post/read/id/100`. `CUrlManager` использует правила формирования URL как для создания, так и для обработки адресов.

Правила формирования URL задаются путем конфигурации свойства `rules` компонента приложения `urlManager`:

```
array(
    ...
    'components'=>array(
        ...
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
                'pattern1'=>'route1',
                'pattern2'=>'route2',
                'pattern3'=>'route3',
            ),
        ),
    ),
);
```

Правила задаются в виде массива пар шаблон-путь, где каждая пара соответствует одному правилу. Шаблон правила — строка, которая должна совпадать с путём в URL. Путь правила должен указывать на существующий путь контроллера.

Кроме показанного выше способа задания правил, можно описать правило с указанием дополнительных параметров:

```
'pattern1'=>array('route1', 'urlSuffix'=>'.xml', 'caseSensitive'=>false)
```

Здесь массив содержит список дополнительных параметров. Начиная с версии 1.1.0, доступны следующие параметры:

- `urlSuffix`: суффикс URL, используемый исключительно для данного правила. По умолчанию равен `null`, что означает использование значения `CUrlManager::urlSuffix`.
- `caseSensitive`: учитывает ли правило регистр. По умолчанию параметр равен `null`, что означает использование значения `CUrlManager::caseSensitive`.
- `defaultParams`: GET-параметры по умолчанию (имя=>значение) для данного правила. При срабатывании правила параметры будут добавлены в `$_GET`.
- `matchValue`: должны ли значения GET-параметров при создании URL совпадать с соответствующими подвыражениями в основном правиле. По умолчанию параметр равен `null`, что означает использование значения `CUrlManager::matchValue`. При значении параметра `false` правило будет использовано для создания URL только если имена параметров совпадают с именами в правиле. При значении `true` значения параметров дополнительно должны совпадать с подвыражениями в правиле. Стоит отметить, что установка значения в `true` снижает производительность.

Использование именованных параметров

Правило может быть ассоциировано с несколькими GET-параметрами. Эти параметры указываются в шаблоне правила в виде маркеров следующим образом:

```
<ParamName:ParamPattern>
```

где `ParamName` соответствует имени GET-параметра, а необязательный `ParamPattern` — регулярному выражению, которое используется для проверки соответствия значению GET-параметра. Если `ParamPattern` не указан, то параметр должен соответствовать любым символам, кроме слэша `/`. В момент создания URL маркеры будут заменены на соответствующие значения параметров, а в момент обработки URL, соответствующим GET-параметрам будут присвоены результаты обработки.

Для наглядности приведем несколько примеров. Предположим, что наш набор правил состоит из трех правил:

```
array(
    'posts'=>'post/list',
    'post/<id:d+>'=>'post/read',
    'post/<year:d{4}>/<title>'=>'post/read',
)
```

- Вызов `$this->createUrl('post/list')` сгенерирует `/index.php/posts`. Здесь было применено первое правило.
- Вызов `$this->createUrl('post/read', array('id'=>100))` сгенерирует `/index.php/post/100`. Применено второе правило.
- Вызов `$this->createUrl('post/read', array('year'=>2008, 'title'=>'a sample post'))` сгенерирует `/index.php/post/2008/a%20sample%20post`. Использовано третье правило.
- Вызов `$this->createUrl('post/read')` сгенерирует `/index.php/post/read`. Ни одно из правил не было применено.

При использовании `createUrl` для генерации адреса URL, маршрут и GET-параметры, переданные методу, используются для определения правила, которое нужно применить. Правило применяется в том случае, когда все параметры, ассоциированные с правилом, присутствуют среди GET-параметров, а маршрут соответствует параметру маршрута.

Если же количество GET-параметров больше, чем требует правило, то лишние параметры будут включены в строку запроса. Например, если вызвать `$this->`

`createUrl('post/read', array('id'=>100, 'year'=>2008))`, мы получим `/index.php/post/100?year=2008`. Для того, чтобы лишние параметры были отражены в информационной части пути, необходимо добавить к правилу `/*`. Таким образом, используя правило `post/<id:d+>/*` получим URL вида `/index.php/post/100/year/2008`.

Как уже говорилось, вторая задача правил URL — разбирать URL-запросы. Этот процесс обратный процессу создания URL. Например, когда пользователь запрашивает `/index.php/post/100`, применяется второе правило из примера выше и запрос преобразовывается в маршрут `post/read` и GET-параметр `array('id'=>100)` (доступный через `$_GET`).

Примечание: Использование правил URL снижает производительность приложения. Это происходит по той причине, что в процессе парсинга запрошенного URL `CUrlManager` пытается найти соответствие каждому правилу до тех пор, пока какое-нибудь из правил не будет применено. Чем больше правил, тем больший урон производительности. Поэтому в случае высоконагруженных приложений использование правил URL стоит минимизировать.

Параметризация маршрутов

Начиная с версии 1.0.5, появилась возможность использовать именованные параметры в маршруте правила. Такое правило может быть применено к нескольким маршрутам, совпадающим с правилом. Это может помочь уменьшить число правил и, таким образом, повысить производительность приложения. Для того, чтобы показать параметризацию маршрутов, используем следующий набор правил:

```
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',
)
```

Мы использовали два именованных параметра в маршруте правил: `_c` и `_a`. Первый соответствует названию контроллера и может быть равен `post` или `comment`, второй — названию action-а и может принимать значения `create`, `update` или `delete`. Вы можете называть параметры по-другому, если их имена не конфликтуют с GET-параметрами, которые могут использоваться в URL.

При использовании правил, приведённых выше, URL `/index.php/post/123/create` будет обработано как маршрут `post/create` с GET-параметром `id=123`. По маршруту `comment/list` с GET-параметром `page=2`, мы можем создать URL `/index.php/comments?page=2`.

Параметризация имён хостов

Начиная с версии 1.0.11 возможно использовать имена хостов в правилах для разбора и создания URL. Можно выделять часть имени хоста в GET-параметр. Например, URL

`http://admin.example.com/en/profile` может быть разобран в GET-параметры `user=admin` и `lang=en`. С другой стороны, правила с именами хостов могут также использоваться для создания URL адресов.

Чтобы использовать параметризованные имена хостов, включите имя хоста в правила URL:

```
array(
    'http://<user:w+>.example.com/<lang:w+>/profile' => 'user/profile',
)
```

Пример выше говорит, что первый сегмент имени хоста должен стать параметром `user`, а первый сегмент пути — параметром `lang`. Правило соответствует маршруту `user/profile`.

Помните, что `CUrlManager::showScriptName` не работает при создании URL адреса с использованием правил с параметризованным именем хоста.

Стоит отметить, что правило с параметризованным именем хоста не должно содержать поддиректорий в том случае, если приложение находится в поддиректории корня вебсервера. К примеру, если приложение располагается по адресу `http://www.example.com/sandbox/blog`, мы должны использовать точно такое же правило URL, как описано выше. Без поддиректории: `sandbox/blog`.

Скрываем `index.php`

С целью сделать адрес URL еще более привлекательным можно спрятать имя входного скрипта `index.php`. Для этого необходимо настроить веб-сервер и компонент приложения `urlManager`.

Вначале сконфигурируем веб-сервер таким образом, чтобы адрес URL без указания имени входного скрипта все также передавался на обработку входному скрипту. Для сервера Apache HTTP server это достигается путем включения механизма преобразования URL и заданием нескольких правил. Для этого необходимо создать файл `/wwwroot/blog/.htaccess`, содержащий правила, приведённые ниже. Те же правила могут быть размещены в файле конфигурации Apache в секции `Directory` для `/wwwroot/blog`.

```
Options +FollowSymLinks
IndexIgnore */*
RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

Далее нужно установить свойство `showScriptName` компонента `urlManager` равным `false`.

Теперь, вызвав `$this->createUrl('post/read', array('id'=>100))`, мы получим URL `/post/100`. Что важно, этот адрес URL будет корректно распознан нашим веб-приложением.

Подмена окончания в адресе URL

В дополнение ко всему перечисленному выше, мы можем добавить к нашим адресам URL окончание.

Например, мы можем получить `/post/100.html` вместо `/post/100`, представив пользователю как будто бы статичную страничку. Для этого нужно просто настроить компонент `urlManager` путем назначения свойству `urlSuffix` любого желаемого окончания.

Аутентификация и авторизация

Аутентификация и авторизация необходимы на страницах, доступных лишь некоторым пользователям. Аутентификация — проверка, является ли некто тем, за кого себя выдаёт. Обычно она подразумевает ввод логина и пароля, но также могут быть использованы и другие средства, такие как использование смарт-карты, отпечатков пальцев и др. Авторизация — проверка, может ли аутентифицированный пользователь выполнять определённые действия (их часто обозначают как ресурсы). Чаще всего это определяется проверкой, назначена ли пользователю определённая роль, имеющая доступ к ресурсам.

В Yii встроен удобный фреймворк аутентификации и авторизации (auth), который, в случае необходимости, может быть настроен под ваши задачи.

Центральным компонентом auth-фреймворка является предопределённый *компонент приложения* «пользователь» — объект, реализующий интерфейс `IWebUser`. Данный компонент содержит постоянную информацию о текущем пользователе. Мы можем получить к ней доступ из любого места приложения, используя `Yii::app()->user`.

Используя этот компонент, мы можем проверить, аутентифицирован ли пользователь, используя `CWebUser::isGuest`. Мы можем произвести вход или выход. Для проверки прав на определённые действия удобно воспользоваться `CWebUser::checkAccess`. Также есть возможность получить уникальный идентификатор и другие постоянные данные пользователя.

Определение класса Identity

Для того, чтобы аутентифицировать пользователя, мы определяем класс `Identity`, описывающий процедуру его опознавания. Данный класс должен реализовывать интерфейс `IUserIdentity`. Для разных моделей аутентификации (таких как OpenID или LDAP) могут быть определены несколько классов. Начать можно с расширения `CUserIdentity`, являющегося базовым классом для парольной аутентификации.

Главная задача при создании класса `Identity` — реализация метода `IUserIdentity::authenticate`. Также данный класс может содержать дополнительную информацию о пользователе, которая необходима нам в процессе работы с его сессией.

В следующем примере мы проверим, совпадают ли введенные имя пользователя и пароль с теми, что мы получим из базы данных при помощи Active Record. Также мы переопределим метод `getId` так, чтобы он возвращал переменную `_id`, заданную во время аутентификации (реализация по умолчанию возвращает ID объекта). В процессе аутентификации мы, используя `CBaseUserIdentity::setState`, сохраняем полученное значение `title` в переменной состояния с тем же именем.

```
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;
    }
}
```

```
}
```

Информация, сохраняемая в переменные состояния, (через `CBaseUserIdentity::setState`) передаётся `CWebUser`, в котором она сохраняется в постоянном хранилище, таком как сессия. Данная информация становится доступной как свойства `CWebUser`. К примеру, мы можем получить значение `title` текущего пользователя при помощи `Yii::app()->user->title` (доступно с версии 1.0.3. В предыдущих версиях, использовался такой код: `Yii::app()->user->getState('title')`)

Инфо: По умолчанию `CWebUser` использует сессии для хранения данных. Если вы используете автоматический вход пользователя с помощью cookie (`CWebUser::allowAutoLogin` выставлен в `true`), данные пользователя будут также сохраняться в cookie. Убедитесь, что эти данные не содержат конфиденциальной информации, такой как пароли.

Вход и выход

Используя класс `Identity` и компонент `User`, мы можем с лёгкостью реализовать действия для входа и выхода:

```
// Аутентифицируем пользователя по имени и паролю
$identity=new UserIdentity($username,$password);
if($identity->authenticate())
    Yii::app()->user->login($identity);
```

```
else
```

```
    echo $identity->errorMessage;
```

```
...
```

```
// Выходим
```

```
Yii::app()->user->logout();
```

Вход на основе cookie

По умолчанию, после некоторого времени бездействия, зависящего от настроек сессии, будет произведён выход из системы. Для того, чтобы этого не происходило, необходимо выставить свойства компонента `User` `allowAutoLogin` в `true` и передать необходимое время жизни cookie в метод `CWebUser::login`. Пользователь будет автоматически аутентифицирован на сайте в течение указанного времени даже в том случае, если он закроет браузер. Данная возможность требует поддержки cookie в браузере пользователя.

```
// Автоматический вход в течение 7 дней.
// allowAutoLogin для компонента user должен быть выставлен в true.
Yii::app()->user->login($identity,3600*24*7);
```

Как уже упоминалось выше, когда включен вход на основе cookie, состояния, сохраняемые при помощи `CBaseUserIdentity::setState`, также будут сохраняться в cookie. При следующем входе состояния считываются из cookie и становятся доступными через `Yii::app()->user`.

Несмотря на то, что в Yii имеются средства для предотвращения подмены состояний в cookie на стороне клиента, не рекомендуется хранить в состояниях важную информацию. Гораздо более правильным решением будет хранение её в постоянном хранилище на стороне сервера (например, в БД).

Кроме того, для серьёзных приложений рекомендуется улучшить стратегию входа по cookie следующим образом:

- При успешном входе после заполнения формы генерируем и храним случайный ключ как в cookie состояния, так и в постоянном хранилище на сервере (т.е. в БД).
- При успешном входе через cookie сравниваем значения хранимых ключей и пускаем пользователя только если они равны.
- Если пользователь входит через форму ещё раз, ключ регенерируется.

Данная стратегия исключает возможность повторного использования старого состояния cookie, в котором может находиться устаревшая информация.

Для реализации нужно переопределить два метода:

- `CUserIdentity::authenticate()`. Здесь производится аутентификация. Если пользователь аутентифицирован, необходимо сгенерировать новый ключ и сохранить его в состояние (при помощи `CBaseUserIdentity::setState()` и БД).
- `CWebUser::beforeLogin()`. Вызывается перед входом. Необходимо проверить соответствие ключей в состоянии и базе данных.

Фильтр контроля доступа

Фильтр контроля доступа — схема авторизации, подразумевающая предварительную проверку прав текущего пользователя на вызываемое действие контроллера. Авторизация производится по имени пользователя, IP-адресу и типу запроса. Данный фильтр называется «accessControl».

Подсказка: Фильтр контроля доступа достаточен для реализации простых систем. Для более сложных вы можете использовать доступ на основе ролей (RBAC), который будет описан ниже.

Для управления доступом к действиям контроллера необходимо переопределить метод `CController::filters` (более подробно описано в разделе Фильтры).

```
class PostController extends CController
{
    ...
    public function filters()
    {
        return array(

            'accessControl',

        );
    }
}
```

Выше было описано, что фильтр access control применяется ко всем действиям контроллера `PostController`. Правила доступа, используемые фильтром определяются переопределением метода `CController::accessRules` контроллера.

```
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('*'),
            ),
        );
    }
}
```

```

        );
    }
}

```

Приведённый код описывает три правила, каждое из которых представлено в виде массива. Первый элемент массива — `'allow'` (разрешить) или `'deny'` (запретить). Остальные — пары имя-значение, задающие параметры правила. В данных правилах задано, что: действия `create` и `edit` не могут быть выполнены анонимными пользователями, а действие `delete` может быть выполнено только пользователями с ролью `admin`.

Правила доступа разбираются поочерёдно в порядке их описания. Первое правило, совпадающее с текущими данными (например, с именем пользователя, ролью или IP) определяет результат авторизации. Если это разрешающее правило, действие может быть выполнено, если запрещающее — не может. Если ни одно из правил не совпало — действие может быть выполнено.

Подсказка: Чтобы быть уверенным, что действие не будет выполнено, необходимо запретить все действия, которые не разрешены, определив соответствующее правило в конце списка:

```

return array(
    // ... разные правила ...
    // это правило полностью запрещает действие 'delete'
    array('deny',
        'actions'=>array('delete'),
    ),
);

```

Данное правило необходимо, так как если ни одно из правил не совпадёт, действие будет выполнено.

Правило доступа может включать параметры, по которым проверяется совпадение:

- `actions`: позволяет указать действия в виде массива их идентификаторов. Сравнение регистронезависимо;
- `controllers`: позволяет указать контроллеры в виде массива их идентификаторов. Сравнение регистронезависимо. Доступно начиная с версии 1.0.4;
- `users`: позволяет указать пользователей. Для сравнения используется `CWebUser::name`. Сравнение регистронезависимо. В параметре могут быть использованы следующие специальные символы:
 - `*`: любой пользователь, включая анонимного.
 - `?`: анонимный пользователь.
 - `@`: аутентифицированный пользователь.
- `roles`: позволяет указать роли, используя доступ на основе ролей, описанный в следующем разделе. В частном случае, правило применится, если `CWebUser::checkAccess` вернёт `true` для одной из ролей. Роли стоит использовать в разрешающих правилах так как роль ассоциируется с возможностью выполнения какого-либо действия. Также стоит отметить, что, несмотря на то, что мы используем термин «роль», значением может быть любой элемент `auth-фреймворка`, такой как роли, задачи или операции;
- `ips`: позволяет указать IP-адрес;
- `verbs`: позволяет указать тип запросов (например, `GET` или `POST`). Сравнение регистронезависимо;
- `expression`: позволяет указать выражение PHP, вычисление которого будет определять совпадение правила. Внутри выражения доступна переменная `$user`, указывающая на `Yii::app()->user`. Данный параметр доступен начиная с версии 1.0.3.

Обработка запроса авторизации

При неудачной авторизации, т.е. когда пользователю запрещено выполнять указанное действие, происходит следующее:

- Если пользователь не аутентифицирован и в свойстве `loginUrl` компонента `user` задан URL страницы входа, браузер будет перенаправлен на эту страницу. Заметим, что по умолчанию `loginUrl` перенаправляет к странице `site/login`;
- Иначе будет отображена ошибка HTTP с кодом 403.

При задании свойства `loginUrl` используется как относительный, так и абсолютный URL. Также можно передать массив, который будет использоваться `CWebApplication::createUrl` при формировании URL. Первый элемент массива задаёт маршрут до действия `login` вашего контроллера, а остальные пары имя-значение — GET-параметры. К примеру,

```
array(
    ...
    'components'=>array(
        'user'=>array(
            // это значение устанавливается по умолчанию
            'loginUrl'=>array('site/login'),
        ),
    ),
)
```

Если браузер был перенаправлен на страницу входа и вход удачный, вам может понадобиться перенаправить пользователя к той странице, на которой неудачно прошла авторизация. Как же узнать URL той страницы? Мы можем получить эту информацию из свойства `returnUrl` компонента `user`. Имея её, мы можем сделать перенаправление:

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

Контроль доступа на основе ролей

Контроль доступа на основе ролей (RBAC) — простой, но мощный способ централизованного контроля доступа. Для сравнения данного метода с другими обратитесь к статье в Википедии.

В Yii иерархический RBAC реализован через компонент `authManager`. Ниже мы сначала опишем основы данной схемы, затем то, как описывать данные, необходимые для авторизации. В завершение мы покажем, как использовать эти данные для контроля доступа.

Общие принципы

Основным понятием в RBAC Yii является *элемент авторизации*. Элемент авторизации — это права на выполнение какого-либо действия (создать новую запись в блоге, управление пользователями). В зависимости от структуры и цели, элементы авторизации могут быть разделены на *операции*, *задачи* и *роли*. Роль состоит из задач. Задача состоит из операций. Операция — разрешение на какое-либо действие (далее не делится). К примеру, в системе может быть роль *администратор*, состоящая из задач *управление записями* и *управление пользователями*. Задача *управление пользователями* может состоять из операций *создать пользователя*, *редактировать пользователя* и *удалить пользователя*. Для достижения большей гибкости, роль в Yii может состоять из других ролей и операций. Задача может состоять из других задач. Операция — из других операций.

Элемент авторизации однозначно идентифицируется его уникальным именем.

Элемент авторизации может быть ассоциирован с *бизнес-правилом* — PHP-кодом, который будет использоваться при проверке доступа. Пользователь получит доступ к элементу только если код вернёт `true`. К примеру, при определении операции `updatePost`, будет не лишним добавить бизнес-правило, проверяющее соответствие ID пользователя ID автора записи. То есть, доступ к редактированию записи имеет только её автор.

Используя элементы авторизации мы можем построить *иерархию авторизации*. Элемент *а* является родителем элемента *в* в иерархии, если *а* состоит из *в* (или *а* наследует права, представленные в *в*). Элемент может иметь несколько потомков и несколько предков. Поэтому иерархия авторизации является скорее частично упорядоченным графом, чем деревом. В ней роли находятся на верхних уровнях, операции — на нижних. Посередине расположены задачи.

После построения иерархии авторизации мы можем назначать роли из неё пользователям нашего приложения. Пользователь получает все права роли, которая ему назначена. К примеру, если назначить пользователю роль *администратор*, он получит административные полномочия, такие как *управление*

записями или управление пользователями (и соответствующие им операции, такие как *создать пользователя*).

А теперь самое приятное. В действии контроллера мы хотим проверить, может ли текущий пользователь удалить определённую запись. При использовании иерархии RBAC и назначенной пользователю роли, это делается очень просто:

```
if(Yii::app()->user->checkAccess('deletePost'))
{
    // удаляем запись
}
```

Настройка менеджера авторизации

Перед тем, как мы перейдём к построению иерархии авторизации и непосредственно проверке доступа, нам потребуется настроить компонент приложения `authManager`. В Yii есть два типа менеджеров авторизации: `CPhpAuthManager` и `CDbAuthManager`. Первый использует для хранения данных PHP, второй — базу данных. При настройке `authManager` необходимо указать, который из компонентов мы собираемся использовать и указать начальные значения свойств компонента. К примеру,

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'sqlite:path/to/file.db',
        ),
        'authManager'=>array(
            'class'=>'CDbAuthManager',

            'connectionID'=>'db',

        ),
    ),
);
```

После этого мы можем обращаться к компоненту `authManager` используя `Yii::app()->authManager`.

Построение иерархии авторизации

Построение иерархии авторизации состоит из трёх этапов: задания элементов авторизации, описания связей между ними и назначение ролей пользователям. Компонент `authManager` предоставляет полный набор API для выполнения поставленных задач.

Для определения элемента авторизации следует воспользоваться одним из приведённых ниже методов:

- `CAuthManager::createRole`
- `CAuthManager::createTask`
- `CAuthManager::createOperation`

После того, как мы определили набор элементов авторизации, мы можем воспользоваться следующими методами для установки связей:

- `CAuthManager::addItemChild`
- `CAuthManager::removeItemChild`
- `CAuthItem::addChild`
- `CAuthItem::removeChild`

После этого мы назначаем роли пользователям:

- `CAuthManager::assign`
- `CAuthManager::revoke`

Приведём пример построения иерархии авторизации с использованием данного API:

```
$auth=Yii::app()->authManager;
```

```

$auth->createOperation('createPost','создание записи');
$auth->createOperation('readPost','просмотр записи');
$auth->createOperation('updatePost','редактирование записи');
$auth->createOperation('deletePost','удаление записи');

$bizRule='return Yii::app()->user->id==$params["post"]->authID;';
$task=$auth->createTask('updateOwnPost','редактирование своей записи',$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');

```

После создания элементов авторизации, компонент `authManager` (или его наследники, например, `CPhpAuthManager`, `CDbAuthManager`) загружает их автоматически. НЕ требуется создавать элементы авторизации при каждом запросе, запуская код, приведённый выше.

Инфо: Довольно громоздкий пример выше предназначен скорее для демонстрации. Разработчикам обычно требуется создать интерфейс и дать возможность пользователям интуитивно построить иерархию авторизации.

Использование бизнес-правил

При построении иерархии авторизации мы можем назначить роль, задачу или операцию *бизнес-правилу*. Также мы можем указать его при назначении роли пользователю. Бизнес-правило — PHP-код, использующийся при проверке доступа. Возвращаемое данным кодом значение определяет, применять ли данную роль к текущему пользователю. В примере выше мы применили бизнес-правило для описания задачи `updateOwnPost`. В нём мы проверяем, совпадает ли ID текущего пользователя с ID автора записи. Информация о записи в массиве `$params` передаётся разработчиком при проверке доступа.

Проверка доступа

Для проверки доступа нам необходимо знать имя элемента авторизации. К примеру, чтобы проверить, может ли текущий пользователь создать запись, необходимо узнать, имеет ли он права, описанные операцией `createPost`. После этого мы можем вызвать `CWebUser::checkAccess`:

```
if(Yii::app()->user->checkAccess('createPost'))
{
    // создаём запись
}
```

Если правило авторизации использует бизнес-правило, требующее дополнительных параметров, необходимо их передать. К примеру, чтобы проверить, может ли пользователь редактировать запись, нужно сделать следующее:

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

Использование ролей по умолчанию

Примечание: данная возможность доступна начиная с версии 1.0.3

Некоторым веб-приложениям требуются очень специфичные роли, которые назначаются каждому или почти каждому пользователю. К примеру, нам необходимо наделить некоторыми правами всех аутентифицированных пользователей. Определять явно и хранить роли для каждого пользователя в этом случае явно неудобно. Для решения этой проблемы можно использовать *роли по умолчанию*. Роль по умолчанию автоматически назначается каждому пользователю, включая гостей. При вызове `CWebUser::checkAccess` сначала проверяются роли по умолчанию. Назначать их явно не требуется. Роли по умолчанию описываются в свойстве `CAuthManager::defaultRoles`. К примеру, приведённая ниже конфигурация описывает две роли по умолчанию: `authenticated` и `guest`.

```
return array(
    'components'=>array(
        'authManager'=>array(
            'class'=>'CDBAuthManager',
            'defaultRoles'=>array('authenticated', 'guest'),
        ),
    ),
);
```

Так как роль по умолчанию назначается каждому пользователю, обычно требуется использовать бизнес-правило, определяющее, к каким именно пользователям её применять. К примеру, следующий код определяет две роли: `authenticated` и `guest`, которые соответственно применяются к аутентифицированным пользователям и гостям.

```
$bizRule='return !Yii::app()->user->isGuest;';
$auth->createRole('authenticated', 'аутентифицированный пользователь', $bizRule);

$bizRule='return Yii::app()->user->isGuest;';
$auth->createRole('guest', 'гость', $bizRule);
```

Темы оформления

Темы оформления являются традиционным способом настроить внешний вид страниц веб-приложения. Применяв новую тему, мы можем изменить внешний вид всего приложения за считанные секунды. В Yii каждая тема представлена как папка, содержащая файлы представлений, макетов и прочих необходимых файлов, таких, как CSS, JavaScript и пр. Название папки соответственно определяет название темы. Все темы хранятся в папке `WebRoot/themes`, при этом быть активной, т.е. использоваться в текущий момент, может только одна из тем.

Подсказка: Папку, где по умолчанию хранятся темы — `WebRoot/themes` — можно легко изменить путем установки свойств `basePath` и `baseUrl` компонента `themeManager` на желаемые.

Использование темы

Для активации темы нужно установить значение `theme` равным имени соответствующей темы. Это можно проделать путем конфигурации приложения или прямо в ходе выполнения в действиях контроллера.

Примечание: Имя темы чувствительно к регистру, и, если попытаться активировать несуществующую тему, свойство `Yii::app()->theme` вернет `null`.

Создание темы

Содержимое папки с темами должно быть организовано точно также, как и содержимое базовой директории приложения, то есть, все файлы представлений должны находиться в папке `views`, макеты представлений в папке `views/layouts`, а файлы системных представлений в папке `views/system`. Например, если необходимо заменить представление `create` контроллера `PostController` на представление темы `classic`, нужно сохранить новый файл представления как `WebRoot/themes/classic/views/post/create.php`.

Для представлений контроллеров в модулях, соответствующие файлы оформленных представлений нужно также поместить в папку `views`. Например, если упомянутый выше контроллер `PostController` входит в модуль `forum`, необходимо сохранить файл представления `create` как `WebRoot/themes/classic/views/forum/post/create.php`. Если модуль `forum` является составной частью другого модуля `support`, то файл представления должен быть сохранен как `WebRoot/themes/classic/views/support/forum/post/create.php`.

Примечание: Папка `views` может содержать данные чувствительные с точки зрения безопасности, поэтому необходимо ограничить доступ к папке извне сервера.

В момент вызова метода `render` или `renderPartial` для отображения представления происходит обращение к соответствующим файлам представлений и макетов активной темы. Если файлы найдены, начнется формирование странички, в противном случае, будут использоваться файлы оформления по умолчанию, месторасположение которых устанавливается свойствами `viewPath` и `layoutPath`.

Подсказка: Часто в представлениях темы приходится ссылаться на прочие файлы темы, например, для отображения картинки, находящейся в подпапке темы `images`. Используя свойство `baseUrl` активной темы, можно сформировать корректную ссылку на картинку следующим образом:

```
Yii::app()->theme->baseUrl . '/images/FileName.gif'
```

Ниже приведён пример организации директорий приложения с двумя темами `basic` и `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

```

В настройках приложения, если мы будем использовать:

```

return array(
    'theme'=>'basic',
    ...
);

```

то будет применяться тема `basic`. То есть главный макет (layout) будет браться из `themes/basic/views/layouts`, а представление `index` — из `themes/basic/views/site`. Если файл представления не найден в теме, будет использован файл из `protected/views`.

Темизация виджетов

Начиная с версии 1.1.5, отображения, используемые в виджетах, можно темизировать. При вызове `CWidget::render()` для вывода отображения, Yii сделает попытку найти его в темах перед тем, как загрузить из директории виджета.

Для темизации отображения `xyz` виджета с именем класса `Foo`, необходимо создать директорию `Foo` (с тем же именем, что и у класса) внутри директории с отображениями активной темы. Если класс виджета находится в пространстве имён (начиная с PHP 5.3.0), таком как `appwidgetsFoo`, то необходимо создать директорию `app_widgets_Foo`. В имени мы заменяем разделители пространства имён на подчёркивание. После этого создаём файл отображения `xyz.php` в только что добавленной директории. К этому моменту мы имеем файл `themes/basic/views/Foo/xyz.php`, который и будет использоваться виджетом вместо его собственного отображения, если активная тема — `basic`.

Глобальная настройка виджетов

Примечание: данная возможность доступна с версии 1.1.3.

При использовании виджета, как стандартного, так и стороннего, часто требуется его настройка. К примеру, может понадобиться изменить значение `CLinkPager::maxButtonCount` с 10 (по умолчанию) на 5. Мы можем сделать это передав начальные значения при вызове `CBaseController::widget` для создания виджета. Тем не менее, делать это везде, где мы используем `CLinkPager` довольно неудобно.

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>5,
    'cssFile'=>false,
));
```

При использовании глобальной настройки, необходимо указать начальные значения лишь в одном месте — в файле конфигурации приложения. Для этого настраиваем `widgetFactory` следующим образом:

```
return array(
    'components'=>array(
        'widgetFactory'=>array(
            'widgets'=>array(
                'CLinkPager'=>array(
                    'maxButtonCount'=>5,
                    'cssFile'=>false,
                ),
                'CJuiDatePicker'=>array(
                    'language'=>'ru',
                ),
            ),
        ),
    ),
);
```

Выше мы указали глобальные настройки виджетов `CLinkPager` и `CJuiDatePicker` при помощи соответствующих свойств `CWidgetFactory::widgets`. Стоит отметить, что глобальные настройки указываются в виде пар ключ-массив значений, где ключ соответствует классу виджета, а массив значений задаёт начальные значения свойств этого класса.

Теперь всякий раз, когда мы используем виджет `CLinkPager` в отображении, его свойствам будут присвоены указанные выше начальные значения. Таким образом, чтобы использовать виджет будет достаточно следующего кода:

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
));
```

Мы можем переопределить начальные значения, если в этом есть необходимость. К примеру, если в каком-нибудь отображении мы хотим задать `maxButtonCount` равным 2, можно сделать следующее:

```
$this->widget('CLinkPager', array(
    'pages'=>$pagination,
    'maxButtonCount'=>2,
));
```

Скины

Примечание: Данная возможность доступна с версии 1.1.0.

В то время, как при использовании темы мы можем быстро менять вид представлений, мы также можем использовать скины для настройки вида виджетов, используемых в представлениях. Скин — это массив пар имя-значение, который может использоваться для инициализации свойств виджета. Скин принадлежит классу виджета, а класс виджета может иметь несколько скинов, идентифицируемых по имени. Например, у нас может быть скин `classic` для виджета `CLinkPager`. Для использования данной возможности нам, в первую очередь, необходимо изменить файл конфигурации приложения, выставив свойство `CWidgetFactory::enableSkin` компонента `widgetFactory` в `true`:

```
return array(
    'components'=>array(
        'widgetFactory'=>array(
            'enableSkin'=>true,
        ),
    ),
);
```

В версиях Yii до 1.1.3 необходимо использовать следующую конфигурацию:

```
return array(
    'components'=>array(
        'widgetFactory'=>array(

            'class'=>'CWidgetFactory',

        ),
    ),
);
```

Затем мы создаём необходимые скины. Скины, принадлежащие одному классу виджета, хранятся в одном файле PHP, имя которого совпадает с названием класса виджета. Все файлы скинов по умолчанию хранятся в директории `protected/views/skins`. Для изменения директории надо настроить свойство `skinPath` компонента `widgetFactory`. Например, мы можем создать в директории `protected/views/skins` файл `CLinkPager.php`, код которого представлен ниже:

```
<?php
return array(
    'default'=>array(
        'nextPageLabel'=>'&gt;&gt;',
        'prevPageLabel'=>'&lt;&lt;',
    ),
    'classic'=>array(
        'header'=>' ',
        'maxButtonCount'=>5,
    ),
);
```

В коде выше мы создаём для виджета `CLinkPager` два скина: `default` и `classic`. Первый скин будет применяться к любому виджету `CLinkPager`, в котором явно не указано свойство `skin`, а второй — к

виджету, свойство `skin` которого имеет значение `classic`. Например, в следующем коде представления первым виджет будет использовать скин `default`, а второй — скин `classic`:

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

<?php $this->widget('CLinkPager', array('skin'=>'classic')); ?>
```

Если мы создаём виджет с набором первоначальных значений, они будут иметь приоритет и будут объединены с любыми применяемыми скинами. Например, следующий код представления создаст постраничную разбивку, чьи первоначальные значения — это массив `array('header'=>' ', 'maxButtonCount'=>6, 'cssFile'=>false)`, который является результатом слияния первоначальных значений, указанных в представлении, и скина `classic`.

```
<?php $this->widget('CLinkPager', array(
    'skin'=>'classic',
    'maxButtonCount'=>6,
    'cssFile'=>false,
)); ?>
```

Заметим, что скинизация НЕ требует использования темы. Однако, если тема активна, Yii также будет искать скины в директории `skins` представлений темы (например, `WebRoot/themes/classic/views/skins`). В случае, если скин с таким же именем существует и в директории представления темы и в основной директории представления приложения, скин темы будет иметь приоритет.

Если виджет использует несуществующий скин, Yii по-прежнему будет создавать виджет как обычно, без каких-либо ошибок.

Информация: Использование скина может привести к снижению производительности, поскольку Yii должен найти файл скина, когда виджет создается впервые.

Использование скинов очень похоже на глобальную конфигурацию виджетов. Главные отличия следующие:

- Скины в большей степени относятся к изменению свойств, отвечающих за внешний вид виджета;
- У виджета может быть несколько скинов;
- Скин можно темизировать;
- Использование скинов более затратно, чем использование глобальной конфигурации.

Журналирование

Yii предоставляет гибкий и расширяемый функционал протоколирования. Сообщения можно классифицировать в соответствии с уровнем протоколирования и типом сообщений. Используя фильтры по уровню и категории, можно направить поток сообщений в файлы, электронную почту, окно браузера и т.д.

Протоколирование сообщений

Сообщение может быть запротоколировано путем вызова `Yii::log` или `Yii::trace`. Разница между ними заключается в том, что последний пишет в лог только тогда, когда приложение работает в режиме отладки.

```
Yii::log($message, $level, $category);
Yii::trace($message, $category);
```

Для внесения сообщения в лог, необходимо указать категорию и уровень сообщения. Категория представляет собой строку формата `xxx.yyy.zzz`, что очень схоже с форматом представления псевдонима пути. Например, если сообщение добавляется в лог в `CController`, мы можем использовать категорию `system.web.CController`. Уровень сообщения может иметь одно из следующих значений:

- `trace`: этот уровень используется методом `Yii::trace`. Он предназначен для отслеживания процесса выполнения приложения в ходе разработки;
- `info`: этот уровень предназначен для протоколирования информации общего характера;
- `profile`: данный уровень используется для профилирования (измерения) производительности;

- **warning**: этот уровень предназначен для сообщений-предупреждений;
- **error**: этот уровень используется для сообщений о критических ошибках.

Маршрутизация сообщений

Сообщения, протоколируемые с использованием `Yii::log` или `Yii::trace`, хранятся в памяти. Как правило, нам требуется либо отобразить их в окне браузера, либо сохранить в файле, отправить электронным письмом и пр. Направление сообщений в различные места назначения называется *маршрутизацией сообщений*.

В Yii за маршрутизацию сообщений отвечает компонент приложения `CLogRouter`. Этот компонент управляет множеством так называемых *маршрутов сообщений*. Каждый маршрут представляет одно место назначения потока сообщений. Сообщения, направляемые по тому или иному маршруту, можно отфильтровать в зависимости от их уровня и типа.

Для того, чтобы воспользоваться маршрутизацией сообщений, нам необходимо установить и подгрузить заранее компонент приложения `CLogRouter`. Кроме того, необходимо настроить свойство `routes` этого компонента, указав маршруты сообщений, которые предполагается использовать. Ниже приведен пример необходимой конфигурации приложения:

```
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',
                ),
            ),
        ),
    ),
),
),
)
```

В примере выше, у нас есть два маршрута сообщений. Первый - `CFileLogRoute` - сохраняет сообщения в папке приложения для временных файлов `runtime`. Сохраняются только сообщения с уровнем `trace` или `info` и чья категория начинается с `system.` Второй маршрут - `CEmailLogRoute` - отправляет сообщения на указанный электронный адрес. Отправляются только сообщения уровня **error** или **warning**.

В Yii доступны для использования следующие маршруты сообщений:

- `CDbLogRoute`: сохраняет сообщения в таблицу базы данных;
- `CEmailLogRoute`: отправляет сообщения на указанный адрес электронной почты;
- `CFileLogRoute`: сохраняет сообщения во временной папке приложения;
- `CWebLogRoute`: отображает сообщения в конце текущей страницы;
- `CProfileLogRoute`: отображает сообщения о производительности в конце текущей страницы.

Информация: Маршрутизация сообщения происходит в конце каждого текущего цикла обработки запроса, в момент, когда вызывается событие `onEndRequest`. Для прерывания процесса обработки текущего

запроса, используйте метод `CApplication::end()` вместо `die()` или `exit()`. `CApplication::end()` вызывает событие `onEndRequest`, что позволяет корректно запротоколировать сообщения.

Фильтрация сообщений

Как уже упоминалось выше, сообщения можно отфильтровать по их уровню и типу до того, как они будут направлены тем или иным маршрутом. Это осуществляется путем настройки свойств `levels` и `categories` соответствующего маршрута. Если необходимо указать несколько уровней или типов, значения должны быть разделены запятыми.

Поскольку типы сообщений указываются в формате `xxx.yyy.zzz`, мы можем воспринимать их как иерархию типов. В частности, мы говорим, что `xxx` является родителем `xxx.yyy`, а последний в свою очередь является родителем для `xxx.yyy.zzz`. Поэтому для указания типа `xxx`, а также всех его типов-потомков можно использовать выражение `xxx.*`.

Сохранение контекста сообщений

Начиная с версии 1.0.6, мы можем сохранять дополнительную информацию, такую как предопределённые переменные PHP (`$_GET`, `$_SERVER`), ID сессии, имя пользователя и т.д. Для этого необходимо задать необходимый фильтр в свойстве `CLogRoute::filter`.

В состав фреймворка входит удобный класс `CLogFilter`, который может быть использован в качестве фильтра в большинстве случаев. По умолчанию, `CLogFilter` будет записывать сообщение вместе с такими переменными, как `$_GET` и `$_SERVER`, которые обычно содержат ценную системную информацию. Можно настроить `CLogFilter` таким образом, чтобы перед каждым сообщением записывать ID сессии, имя пользователя и другие данные, которые могут облегчить поиск по большому количеству сообщений. Следующие настройки включают запись контекста сообщений. У каждого журнального маршрута может быть задан свой фильтр. По умолчанию никакого фильтра не задано.

```
array(
    ...
    'preload'=>array('log'),
    'components'=>array(
        ...
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error',
                    'filter'=>'CLogFilter',
                ),
                ...other log routes...
            ),
        ),
    ),
),
),
)
```

Начиная с версии 1.0.7, Yii поддерживает журналирование информации стека вызова в сообщениях, протоколируемых путем вызова `Yii::trace`. По умолчанию, данная особенность отключена, т.к. снижает производительность. Для ее использования, необходимо просто определить константу `YII_TRACE_LEVEL` в начале входного скрипта (до включения файла `yii.php`) целым числом большим нуля. Тогда Yii будет добавлять в каждое трассирующее сообщение имя файла и номер строки стека вызова, в которых был сделан вызов кода. Число `YII_TRACE_LEVEL` определяет количество слоев каждого стека вызова, которое должно быть записано. Эта информация особенно полезна на стадии разработки, так как может помочь нам определить места, в которых вызываются трассирующие сообщения.

Профилирование производительности

Для целей измерения производительности используется специальный тип сообщений. Его можно использовать для измерения времени исполнения некоторого блока кода и определения узких мест в производительности.

Для того, чтобы измерить производительность, необходимо указать ту часть кода, выполнение которой будет отслеживаться. Используя следующие методы, мы отмечаем начало и конец каждого измеряемого блока кода:

```
Yii::beginProfile('blockID');
...блок профилируемого кода...
Yii::endProfile('blockID');
```

где `blockID` — это уникальный идентификатор блока кода.

Обратите внимание, что блоки кода должны иметь корректную вложенность, т.е. они не могут пересекаться друг с другом. Они либо идут параллельно, либо один блок полностью включает другой.

Для того, чтобы увидеть результат профилирования, нам потребуется установить компонент приложения `CProfileLogRoute`, отвечающий за соответствующий маршрут протоколирования. Здесь все аналогично работе с простыми маршрутами сообщений. Маршрут `CProfileLogRoute` отобразит результаты измерения производительности в низу текущей страницы.

Профилирование SQL-запросов

Профилирование особенно полезно при работе с базой данных, так как SQL-запросы часто являются самым узким местом производительности приложения. Несмотря на то, что мы можем вставить в нужные места `beginProfile` и `endProfile` для того, чтобы замерить время, затраченное на каждый SQL-запрос, начиная с версии 1.0.6 Yii предоставляет более удобное решение данной проблемы.

Выставив в настройках приложения `CDbConnection::enableProfiling` в `true`, мы получим профилирование всех выполняемых SQL-запросов. Полученные результаты можно вывести при помощи вышеупомянутого `CProfileLogRoute`, показывающего, какой SQL-запрос сколько времени занял. Для вывода общего количества запросов и общего времени выполнения можно использовать `CDbConnection::getStats()`.

Обработка ошибок

Yii предоставляет полноценный функционал обработки ошибок на базе механизма обработки ошибок в PHP 5. В момент поступления пользовательского запроса создается экземпляр приложения, который регистрирует метод `handleError` для обработки предупреждений и уведомлений, а также метод `handleException` для обработки не пойманных исключений. Таким образом, если в процессе выполнения приложения возникают предупреждения, уведомления PHP или не пойманные исключения, один из обработчиков ошибок получит управление и запустит необходимую процедуру обработки ошибок.

Подсказка: Регистрация обработчиков ошибок осуществляется в конструкторе приложения путем вызова функций PHP `set_exception_handler` и `set_error_handler`. Если вы не хотите, чтобы Yii обрабатывал ошибки и исключения, во входном скрипте установите значение `false` константам `YII_ENABLE_ERROR_HANDLER` и `YII_ENABLE_EXCEPTION_HANDLER`.

По умолчанию, метод `errorHandler` (или `exceptionHandler`) вызывает событие `onError` (или `onException`). Если ошибка (или исключение) не обрабатывается обработчиком события, он обращается за помощью к компоненту приложения `errorHandler`.

Вызов исключений

Вызов исключений в Yii ничем не отличается от вызова обычного исключения PHP. В случае необходимости, вызов исключения осуществляется следующим образом:

```
throw new ExceptionClass('ExceptionMessage');
```

Yii определяет два класса для исключений: `CException` и `CHttpException`. Первый является типовым классом исключения, а второй отвечает за исключения, которые отображаются конечному пользователю. Кроме того, второй класс имеет свойство `statusCode`, соответствующее коду состояния HTTP. Класс исключения определяет также, каким образом отображается ошибка. Об этом будет рассказано ниже.

Подсказка: Вызов исключения `CHttpException` — это простой способ сообщить об ошибках, вызванных неверными действиями пользователя. Например, если пользователь указывает в адресе URL неверный

идентификатор записи, для отображения ошибки 404 (страница не найдена) мы можем выполнить следующее действие:

```
// если идентификатора записи не существует
throw new CHttpException(404, 'Указанная запись не найдена');
```

Отображение ошибок

В момент, когда компонент приложения `CErrorHandler` получает ошибку, выбирается соответствующее представление для её отображения. Если предполагается, что сообщение об ошибке должно отображаться конечным пользователям, например `CHttpException`, то используется представление с именем `errorXXX`, где `XXX` соответствует коду состояния HTTP (400, 404, 500 и т.д.). Если же это внутренняя ошибка и отображаться она должна только разработчикам, используется представление с именем `exception`. В последнем случае будет отображен весь стек вызовов, а также указание на строку возникновения ошибки.

Инфо: Если приложение запускается в производственном режиме, все ошибки, включая внутренние, отображаются с использованием представления `errorXXX`. Это сделано из соображений безопасности, поскольку стек вызова может содержать важную информацию. В этом случае для выявления причин возникновения ошибки необходимо использовать протокол ошибок.

`CErrorHandler` осуществляет поиск файла, соответствующего представлению, в следующем порядке:

1. `WebRoot/themes/ThemeName/views/system`: папка системных представлений текущей темы оформления;
2. `WebRoot/protected/views/system`: папка системных представлений приложения, используемая по умолчанию;
3. `yii/framework/views`: папка стандартных системных представлений, предоставляемых фреймворком.

Следовательно, если нам необходимо изменить внешний вид сообщений, мы можем просто создать файлы представлений ошибок в папке системных представлений приложения или темы. Каждый файл представления — это обычный PHP-скрипт, состоящий преимущественно из HTML-кода. Подробнее с этим можно разобраться, просто изучив используемые по умолчанию файлы, расположенные в папке фреймворка с именем `view`.

Управление отображением ошибок в действии контроллера

Начиная с версии 1.0.6 Yii позволяет использовать действие контроллера для отображения ошибок. Для этого необходимо задать обработчик ошибок в настройках приложения:

```
return array(
    ...
    'components'=>array(
        'errorHandler'=>array(
            'errorAction'=>'site/error',
        ),
    ),
);
```

Выше мы задали маршрут `site/error`, ведущий к действию `error` контроллера `SiteController`, свойству `CErrorHandler::errorAction`. Если необходимо, можно использовать другой маршрут. Код действия `error` должен выглядеть примерно так:

```
public function actionError()
{
    if($error=Yii::app()->errorHandler->error)
        $this->render('error', $error);
}
```

Сначала мы получаем подробную информацию об ошибке из `CErrorHandler::error`. Если она не пуста — отображаем её в представлении `error`. Информация, получаемая из `CErrorHandler::error` является массивом, содержащим следующие данные:

- `code`: код ответа HTTP (например, 403 или 500);
- `type`: тип ответа (например, `CHttpException` или `PHP Error`);
- `message`: текст сообщения;
- `file`: имя PHP-скрипта, в котором возникла ошибка;
- `line`: номер строки, на которой возникла ошибка;
- `trace`: стек вызовов ошибки;
- `source`: часть кода, где возникла ошибка.

Подсказка: Проверка `CErrorHandler::error` на пустое значение делается, т.к. действие `error` может быть вызвано пользователем напрямую. Так как мы передаём массив `$error` представлению, он будет автоматически развёрнут в отдельные переменные, поэтому мы можем обращаться к ним напрямую, как `$code` или `$type`.

Протоколирование сообщений

Если возникает ошибка, то соответствующее сообщение с уровнем `error` всегда вносится в лог. В случае, если ошибка — результат предупреждения или уведомления PHP, сообщению присваивается категория `php`, если же ошибка вызвана не пойманным исключением, сообщению присваивается категория `exception.ExceptionClassName` (в случае `CHttpException` к категории добавляется код состояния). Для отслеживания ошибок, возникающих в процессе выполнения приложения, можно использовать функционал журналирования.

Веб-сервисы

Веб-сервис — программная система, разработанная для обеспечения взаимодействия между несколькими компьютерами через сеть. В веб-приложении это обычно набор API, который можно использовать через интернет для выполнения действий на удалённом сервере, обслуживающем веб-сервис. К примеру, клиент, основанный на Flex, может вызывать функции, реализованные на сервере в PHP-приложении. В качестве базового уровня протокола используется SOAP.

Для того, чтобы упростить задачу создания веб-сервиса, в Yii включены `CWebService` и `CWebServiceAction`. API сгруппированы по классам, которые называются *провайдерами*. Для каждого класса Yii генерирует WSDL, описывающий функционал предоставляемого API и правила его использования клиентом. При обработке вызова клиента, Yii создаёт соответствующий ему экземпляр провайдера, вызывает метод API и отвечает на запрос.

Примечание: Для работы `CWebService` требуется расширение PHP SOAP. Убедитесь, что оно включено, прежде, чем пробовать примеры, описанные далее.

Создание провайдера

Как уже было описано, провайдер — это класс, реализующий методы, которые могут быть вызваны удалённо. Для того, чтобы определить, какие методы могут быть вызваны удалённо и какое значение возвращать, Yii использует специальные комментарии и reflection.

Попробуем реализовать простой сервис, отдающий информацию о котировках акций определённой компании. Для этого нам потребуется реализовать провайдер, как показано ниже. Стоит отметить, что наследуем класс провайдера `StockController` от `CController`. Наследование не является обязательным. Мы опишем, зачем оно нужно, ниже.

```
class StockController extends CController
{
    /**
     * @param string индекс предприятия
     * @return float цена
     * @soap
     */
```

```

public function getPrice($symbol)
{
    $prices=array('IBM'=>100, 'GOOGLE'=>350);
    return isset($prices[$symbol])?$prices[$symbol]:0;
    //...возвращаем цену для компании с индексом $symbol
}
}

```

Выше мы описали, что метод `getPrice` является частью API веб-сервиса, пометив его в комментарии тэгом `@soap`. Там же мы описали типы параметров и возвращаемого значения. Дополнительные методы API могут быть описаны точно таким же образом.

Реализация действия веб-сервиса

После создания провайдера необходимо сделать его доступным для клиентов. Для этого необходимо описать действие контроллера `CWebServiceAction`. В нашем примере мы используем `StockController`:

```

class StockController extends CController
{
    public function actions()

```

```
{
```

```

        return array(
            'quote'=>array(
                'class'=>'CWebServiceAction',
            ),
        );
    }

    /**
     * @param string индекс предприятия
     * @return float цена
     * @soap
     */
    public function getPrice($symbol)
    {
        //...возвращаем цену для компании с индексом $symbol
    }
}

```

Это всё, что требуется для создания веб-сервиса. Теперь при обращении к URL `http://hostname/path/to/index.php?r=stock/quote`, мы получим объёмистый XML, на самом деле являющийся WSDL описанного нами веб-сервиса.

Подсказка: По умолчанию, при использовании `CWebServiceAction` подразумевается, что текущий контроллер является провайдером. Именно поэтому мы определили метод `getPrice` в классе `StockController`.

Использование веб-сервиса

Для того, чтобы наш пример был полным, создадим клиент, использующий веб-сервис, который мы только что создали. В примере клиент будет написан на PHP, но для его реализации можно использовать и другие языки, такие как Java, C#, Flex и т.д.

```
$client=new SoapClient('http://hostname/path/to/index.php?r=stock/quote');
echo $client->getPrice('GOOGLE');
```

Запустив данный скрипт через браузер или в консоли, вы должны получить 350, что соответствует цене акций GOOGLE.

Типы данных

При описании методов и свойств класса, которые должны быть доступны через веб-сервис, нам необходимо определить типы параметров и возвращаемых значений. Для этого могут быть использованы следующие типы:

- str/string: соответствует `xsd:string`;
- int/integer: соответствует `xsd:int`;
- float/double: соответствует `xsd:float`;
- bool/boolean: соответствует `xsd:boolean`;
- date: соответствует `xsd:date`;
- time: соответствует `xsd:time`;
- datetime: соответствует `xsd:dateTime`;
- array: соответствует `xsd:string`;
- object: соответствует `xsd:struct`;
- mixed: соответствует `xsd:anyType`.

Если тип не является одним из приведённых выше, он воспринимается как составной тип, состоящий из свойств. Этот тип соответствует классу, а его свойства — public-переменным класса, отмеченных в комментариях `@soap`.

Также можно использовать массивы. Для этого необходимо дописать `[]` в конец примитивного или составного типа. Таким образом мы получим массив с элементами заданного типа.

Ниже приведён пример определения метода API `getPosts`, возвращающего массив объектов класса `Post`.

```
class PostController extends CController
{
    /**
     * @return Post[] список записей
     * @soap
     */
    public function getPosts()
    {
        return Post::model()->findAll();
    }
}

class Post extends CActiveRecord
{
    /**
     * @var integer ID записи
     * @soap
     */
    public $id;
    /**
     * @var string заголовок записи
     * @soap
     */
}
```



```

    */
    public $title;

    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }
}

```

Сопоставление классов

Для получения от клиента параметров составного типа, в приложении должны быть заданы соответствия типов WSDL классам PHP. Для этого необходимо настроить свойство `classMap` класса `CWebServiceAction`.

```

class PostController extends CController
{
    public function actions()
    {
        return array(
            'service'=>array(
                'class'=>'CWebServiceAction',
                'classMap'=>array(
                    'Post'=>'Post', // или просто 'Post'

```

```

                ),
            ),
        );
    }
    ...
}

```

Перехват удалённого вызова метода

Если реализован интерфейс `IWebServiceProvider`, провайдер может перехватывать удалённые вызовы методов. Используя `IWebServiceProvider::beforeWebMethod` можно получить текущий экземпляр `CWebService`. Через `CWebService::methodName` — название вызываемого метода. Если метод по каким либо причинам (например, отсутствие прав на его выполнение) не должен быть вызван, необходимо вернуть `false`.

Интернационализация

Интернационализация (сокращённо I18N) — процесс создания приложения, которое может работать на различных языках и с различными региональными особенностями без каких либо дополнительных изменений. Для веб-приложений это особенно важно, так как пользователь может быть из любой точки мира.

Yii поддерживает интернационализацию на нескольких уровнях:

- Предоставляет региональные данные для всех возможных языков и их вариаций.
- Сервис для перевода сообщений и файлов.
- Форматирование дат и чисел в зависимости от региональных настроек.

В последующих разделах мы рассмотрим перечисленные возможности более подробно.

Региональные настройки и язык

Региональные настройки — это набор параметров, задающих язык пользователя, страну и любые другие специальные настройки интерфейса. Обычно настройки определяются идентификатором, состоящим из языка и региона. К примеру, идентификатор `en_us` соответствует английскому языку и настройкам для США. В Yii все идентификаторы региональных настроек названы единообразно: `LanguageID` (язык) или `LanguageID_RegionID` (язык_регион) в нижнем регистре. Например: `ru`, `en_us`.

Региональные настройки хранятся в объекте класса `CLocale`, который можно использовать для получения зависимой от них информации, такой как символы и форматы валют и чисел, форматы дат и времени, название месяцев и дней недели. Так как информация о языке уже содержится в идентификаторе, в `CLocale` она не дублируется. По этой причине мы часто применяем в одном контексте термины «язык», «региональные настройки» и «локаль».

Имея идентификатор языка, мы можем получить соответствующий ему объект `CLocale`:

`CLocale::getInstance($localeID)` или `CApplication::getLocale($localeID)`.

Информация: в состав Yii включены региональные данные практически по всем языками и регионам. Данные получены из Common Locale Data Repository (CLDR). Доступны не все данные, предоставляемые CLDR, так как там содержится большое количество редко используемой информации. Начиная с версии 1.1.0 пользователи также могут предоставлять свои собственные специальные региональные данные. Для этого настройте свойство `CApplication::localeDataPath`, чтобы оно указывало на директорию, содержащую специальные региональные данные. Обратитесь к файлам региональных данных в директории `framework/i18n/data` для того, чтобы создать файлы специальных региональных данных.

В приложении Yii мы различаем язык приложения и исходный язык приложения. Язык приложения — это язык (локаль) пользователя, который работает с приложением. Исходный язык приложения — язык, который используется в исходном коде приложения. Интернационализация требуется только в том случае, если эти два языка различаются.

Вы можете установить язык приложения в настройках приложения или изменить его непосредственно перед использованием возможностей интернационализации.

Подсказка: Иногда нам требуется считать язык пользователя из браузера. Мы можем получить идентификатор локали используя `CHttpRequest::preferredLanguage`.

Перевод

Самой востребованной возможностью интернационализации, скорее всего, является перевод. Он включает в себя перевод сообщений и строк в представлениях. Первый используется для перевода отдельных сообщений, второй — для перевода файлов целиком.

В процессе перевода участвуют объект перевода, исходный язык и конечный язык. В Yii исходный язык по умолчанию приравнивается к исходному языку приложения, а язык к языку приложения. Если оба языка совпадают — перевод не производится.

Перевод сообщений

Перевод сообщений осуществляется при помощи метода `Yii::t()`. Метод переводит данное сообщение с исходного языка приложения на текущий язык приложения.

При переводе необходимо указать категорию сообщения, так как оно может иметь различные переводы в разных категориях (контекстах). Категория с именем `yii` является зарезервированной для использования ядром фреймворка.

Сообщения могут содержать параметры, которые при вызове `Yii::t()` заменяются соответствующими им значениями. К примеру, следующий вызов метода заменит `{alias}` на значение соответствующей переменной:

```
Yii::t('app', 'Path alias "{alias}" is redefined.',
    array('{alias}'=>$alias))
```

Примечание: переводимые сообщения не должны содержать переменных, изменяющих строку (`"Invalid { $message } content."`). Если есть необходимость подставлять в строку какие-либо значения — используйте параметры.

Переведённые сообщения хранятся в репозитории, называемом *источник сообщений*. Источник сообщений представляет собой экземпляр класса `CMessageSource` или его наследника. При выполнении

`Yii::t()` производится поиск сообщения в источнике сообщений и, если оно найдено — возвращается его переведённая версия.

Yii поддерживает несколько типов источников сообщений, перечисленных ниже. Также вы можете создать свой источник, унаследовав его от `CMessageSource`.

- `CPhpMessageSource`: переводы сообщений хранятся как пары ключ-значение в массиве PHP. Исходное сообщение при этом является ключом, а переведённое — значением. Каждый массив содержит переводы для определённой категории сообщений и находится в отдельном файле, имя которого совпадает с названием категории. Файлы с переводом для одного и того же языка хранятся в одной директории, имеющей такое же имя, как и идентификатор языка. Директории для всех языков располагаются в директории, указанной в `basePath`;
- `CGettextMessageSource`: переводы сообщений хранятся в формате GNU Gettext;
- `CDbMessageSource`: переводы сообщений хранятся в базе данных. Подробнее см. `CDbMessageSource`.

Источник сообщений загружается как компонент приложения. Сообщения, которые используются в приложении, хранятся в компоненте `messages`. По умолчанию тип данного источника сообщений — `CPhpMessageSource`. Путь, по которому хранятся файлы перевода — `protected/messages`.

Таким образом, для того, чтобы использовать механизм перевода сообщений, необходимо следующее:

1. Вызвать `Yii::t()` в нужных местах;
2. Создать файлы перевода `protected/messages/IdЯзыка/ИмяКатегории.php`. Каждый такой файл просто возвращает массив переведённых сообщений. Обратите внимание, что при этом используется `CPhpMessageSource`;
3. В файле конфигурации установите значения `CApplication::sourceLanguage` и `CApplication::language`.

Подсказка: Если в качестве источника сообщений используется `CPhpMessageSource`, для работы с сообщениями может использоваться утилита `yiic`. При помощи команды `message` возможно выбрать из исходного кода все сообщения, для которых необходим перевод и, при необходимости, объединить их с уже существующим переводом. Подробное описание команды `message` можно получить набрав в консоли `yiic help message`.

Начиная с версии 1.0.10, при использовании `CPhpMessageSource`, можно указывать сообщения специально для модулей. То есть, если сообщение принадлежит модулю с именем класса `Xyz`, то категория сообщений может быть указана в формате `Xyz.имяКатегории`. Соответствующий ей файл сообщений будет `ПутьДоМодуля/messages/IDЯзыка/имяКатегории.php`, где `ПутьДоМодуля` — директория, в которой находится класс модуля. При использовании `Yii::t()` для перевода сообщения модуля должен использоваться следующий код:

```
Yii::t('Xyz.имяКатегории', 'сообщение для перевода')
```

Начиная с версии 1.0.2, в Yii добавлена поддержка формата выбора. Формат выбора предназначен для выбора перевода в зависимости от заданного числа. К примеру, в английском языке слово 'book' может быть единственного или множественного числа в зависимости от количества книг. В других языках слово может не иметь специальной формы (как в китайском) или может подчиняться более сложным правилам для множественного числа (как в русском). Формат выбора решает данную проблему простым, но в то же время эффективным способом.

Для использования формата выбора перевод должен содержать последовательность пар выражение-сообщение, разделённых символом `|`:

```
'expr1#message1|expr2#message2|expr3#message3'
```

где `exprN` — выражение PHP, возвращающее логическое значение. Если выражение равно `true` — используется соответствующий ему перевод и дальнейшие выражения не вычисляются. Выражение может содержать специальную переменную `n` (не `$n!`), которая содержит число, переданное первым параметром. Допустим, если мы используем перевод

```
'n==1#one book|n>1#many books'
```

и передаём число 2 параметром `Yii::t()`, то получим `many books`.

Если проверяется соответствие определённому числу, можно использовать сокращённую запись, которая будет рассматриваться как `n==Number`:

```
'1#one book|n>1#many books'
```

Перевод файлов

Перевод файлов осуществляется вызовом `CApplication::findLocalizedFile()`. Параметром передаётся путь к файлу, который необходимо перевести. Метод ищет одноимённый файл в подпапке `localeID`. Если файл найден — возвращается его путь, иначе — путь к исходному файлу.

Перевод файла используется в основном при отображении представлений. При вызове одного из методов отображения контроллера или виджета, файлы представления будут переведены автоматически. К примеру, если язык приложения установлен как `zh_cn`, а исходный язык как `en_us` — при отображении представления `edit` будет произведён поиск представления

`protected/views/ControllerID/zh_cn/edit.php`. Если оно найдено — будет использована переведённая версия, если нет — исходная, расположенная в `protected/views/ControllerID/edit.php`.

Перевод файлом можно использовать и для других целей. Например, для того, чтобы отобразить переведённое изображение или загрузить локализованную версию файла.

Форматирование даты и времени

Дата и время в разных странах и регионах часто форматируются по-разному. Задача форматирования даты и времени таким образом сводится к генерации строки, подходящей для данной страны и региона. Для этого в Yii используется `CDateFormatter`.

Каждый экземпляр `CDateFormatter` соответствует некому языку приложения. Чтобы получить форматтер для выбранного языка, мы можем просто обратиться к свойству приложения `dateFormatter`.

Класс `CDateFormatter` содержит два метода, предназначенных для форматирования UNIX timestamp:

- `format`: форматирует переданную в формате UNIX timestamp дату согласно шаблону (например, `$dateFormatter->format('dd.MM.yyyy', $timestamp)`);
- `formatDateTime`: форматирует переданную в формате UNIX timestamp дату согласно шаблону, заданному для выбранного языка (например формат даты `short`, формат времени `long`).

Форматирование чисел

Также, как дата и время, числа могут писаться по-разному в разных странах и регионах. Форматирование чисел включает в себя форматирование десятичных дробей, валют и чисел с процентами. Для выполнения данных задач в Yii используется класс `CNumberFormatter`.

Для того, чтобы воспользоваться форматтером чисел, соответствующим выбранному языку, мы можем обратиться к свойству приложения `numberFormatter`.

Для форматирования целых чисел и дробей в классе `CNumberFormatter` есть следующие методы:

- `format`: форматирует число в соответствии с заданным форматом (например, `$numberFormatter->format('#,##0.00',$number)`);
- `formatDecimal`: форматирует число в соответствии с форматом, заданным для текущего языка приложения;
- `formatCurrency`: форматирует число и код валюты в соответствии с форматом, заданным для текущего языка приложения;
- `formatPercentage`: форматирует число в соответствии с форматом форматирования процентов, заданным для текущего языка приложения.

Альтернативный язык шаблонов

Yii позволяет разработчику использовать свой любимый язык шаблонов (например, Prado или Smarty) для описания представлений контроллера или виджета. Для этого требуется написать и установить свой компонент `viewRenderer`. Обработчик представления перехватывает вызовы `CBaseController::renderFile`, компилирует файл представления с альтернативным синтаксисом и отдаёт результат компиляции.

Информация: Не рекомендуется использовать альтернативный синтаксис шаблонов для описания представлений компонентов, выкладываемых в открытый доступ. Это приведёт к требованию использовать тот же синтаксис, что использован в представлении компонента.

Далее мы покажем, как использовать `CPradoViewRenderer` — обработчик представлений, позволяющий разработчику использовать синтаксис шаблонов, используемый в фреймворке Prado. Если вы хотите реализовать свои обработчики представлений, обязательно изучите `CPradoViewRenderer`.

Использование `CPradoViewRenderer`

Для использования `CPradoViewRenderer` необходимо настроить приложение следующим образом:

```
return array(
    'components'=>array(
        ...,
        'viewRenderer'=>array(
            'class'=>'CPradoViewRenderer',
        ),
    ),
);
```

По умолчанию `CPradoViewRenderer` будет компилировать исходные файлы представлений и сохранять получаемые файлы PHP в директорию `runtime`. PHP-файлы изменяются только в том случае, если изменено исходное представление. Поэтому, использование `CPradoViewRenderer` влечёт за собой очень незначительное падение производительности.

Подсказка: Несмотря на то, что `CPradoViewRenderer` добавляет новый синтаксис для более быстрого и удобного описания представлений, вы можете использовать код PHP также, как и в обычных представлениях.

Ниже будут описаны конструкции, поддерживаемые `CPradoViewRenderer`.

Сокращённые PHP-тэги

Сокращённые PHP-тэги — хороший способ сократить код, используемый в представлении. Выражение `<%= expression %>` преобразуется в `<?php echo expression ?>`. `<% statement %>` — в `<?php statement ?>`. К примеру:

```
<%= CHtml::textField($name, 'value'); %>
<% foreach($models as $model): %>
```

преобразуется в

```
<?php echo CHtml::textField($name, 'value'); ?>
<?php foreach($models as $model): ?>
```

Компонентные тэги

Компонентные тэги используются для того, чтобы вставить в представление виджет. Синтаксис следующий:

```
<com:WidgetClass property1=value1 property2=value2 ...>
    // содержимое виджета
</com:WidgetClass>

// виджет без содержимого
<com:WidgetClass property1=value1 property2=value2 .../>
```

Здесь `WidgetClass` определяет имя класса виджета или псевдоним пути. Начальные значения свойств могут быть как строками, заключёнными в кавычки, так и выражениями PHP, заключёнными в фигурные скобки. К примеру:

```
<com:CCaptcha captchaAction="captcha" showRefreshButton={false} />
```

преобразуется в

```
<?php $this->widget('CCaptcha', array(
    'captchaAction'=>'captcha',
    'showRefreshButton'=>false)); ?>
```

Примечание: Значение `showRefreshButton` задано как `{false}` вместо `"false"` так как последнее означает строку, а не логическое значение.

Кэширующие тэги

Кэширующие тэги — краткий способ использования кэширования фрагментов. Синтаксис следующий:

```
<cache:fragmentID property1=value1 property2=value2 ...>
    // содержимое, которое необходимо кэшировать
</cache:fragmentID >
```

Здесь `fragmentID` — уникальный идентификатор кэшируемого объекта. Пары имя-значение используются для настройки кэширования фрагментов. К примеру:

```
<cache:profile duration={3600}>
    // информация из профиля пользователя
</cache:profile >
```

будет преобразовано в

```
<?php if($this->cache('profile', array('duration'=>3600))): ?>
    // информация из профиля пользователя
<?php $this->endCache(); endif; ?>
```

Захватывающие тэги

Как и кэширующие тэги, захватывающие тэги — компактный способ использования `CBaseController::beginClip` и `CBaseController::endClip`. Синтаксис следующий:

```
<clip:clipID>
    // содержимое для захвата
</clip:clipID >
```

Здесь `clipID` — уникальный идентификатор захваченного содержимого. Захватывающие тэги преобразуются следующим образом:

```
<?php $this->beginClip('clipID'); ?>
    // содержимое для захвата
<?php $this->endClip(); ?>
```

Тэги комментариев

Тэги комментариев используются для написания комментариев, доступных исключительно разработчикам. Данные тэги будут удалены непосредственно перед отображением представления. Синтаксис следующий:

```
<!---
Этот комментарий будет вырезан... цензурой
--->
```

Одновременное использование шаблонов разного формата

Начиная с версии 1.1.2 возможно использовать одновременно как альтернативный, так и обычный PHP синтаксис шаблонов. Для этого необходимо задать свойству обработчика шаблонов `CViewRenderer::fileExtension` значение, отличное от `.php`. К примеру, если оно будет выставлено в `.tpl`, то все шаблоны с расширением `.tpl` будут обрабатываться выбранным обработчиком представлений. Шаблоны с расширением `.php`, как и ранее, будут использовать стандартный синтаксис PHP.

Консольные приложения

Консольные приложения главным образом используются для выполнения вторичных или фоновых задач, таких как генерация кода, компиляция поискового индекса, отправка сообщений электронной почты и т.д. Yii предоставляет инструмент для разработки консольных приложений, придерживаясь объектно-ориентированного подхода. Он позволяет консольному приложению получить доступ к ресурсам, которые использует основное веб-приложение (например, к базе данных).

Обзор

Каждая консольная задача представлена в Yii как команда. Консольная команда описывается в классе, наследуемом от `CConsoleCommand`.

Консольные команды управляются консольным приложением. Консольное приложение похоже на веб-приложение: оно также может быть сконфигурировано при помощи файла конфигурации и запускается при помощи входного скрипта.

Для запуска консольной команды следует использовать следующий формат:

```
php entryScript.php CommandName Param0 Param1 ...
```

Входной скрипт

Как уже было упомянуто ранее, входной скрипт требуется для запуска консольной команды. Если приложение создано при помощи инструмента `yiic webapp`, то в нём уже есть консольное приложение с необходимым входным скриптом, находящимся в `protected/yiic.php`. Также можно написать свой входной скрипт:

```
defined('YII_DEBUG') or define('YII_DEBUG',true);

// подключаем файл инициализации Yii
require_once('path/to/yii/framework/yii.php');

// создаем и запускаем экземпляр приложения
$configFile='path/to/config/file.php';

Yii::createConsoleApplication($configFile)->run();
```

Консольная команда

Консольные команды находятся в файлах с классами в папке, указанной в `CConsoleApplication::commandPath`. По умолчанию это `protected/commands`. Класс консольной команды должен быть унаследован от `CConsoleCommand`. Имя класса должно быть вида `XYZCommand`, где `XYZ` соответствует имени команды, первая буква которого приведена к верхнему регистру. К примеру, команда `sitemap` должна использовать класс `SitemapCommand`. По этой причине имена консольных команд регистрозависимы.

Подсказка: Конфигурируя `CConsoleApplication::commandMap`, можно при желании изменить порядок именования и расположения классов команд.

В классе консольной команды можно либо описать несколько действий (это будет описано в следующем подразделе) или перекрыть метод `CConsoleCommand::run()`:

```
public function run($args) { ... }
```

где `$args` — дополнительные параметры, переданные из командной строки.

Действие консольной команды

Примечание: Данная возможность доступна начиная с версии 1.1.5.

В консольной команде часто требуется обрабатывать различные параметры. К примеру, команда `sitemap` может принимать параметр, указывающий, какой тип карты сайта необходимо сгенерировать. Мы можем разбить команду на несколько действий, каждое из которых выполняет свою подзадачу, подобно действию `CController`.

Действие консольной команды описывается в методе её класса. Имя метода должно быть вида `actionXYZ`, где `XYZ` соответствует имени действия и первой буквой, приведённой к верхнему регистру. К примеру, метод `actionIndex` задаёт действие с именем `index`.

Для того, чтобы запустить определённое действие, используется следующий формат команды:

```
php entryScript.php CommandName ActionName --Option1=Value1 --Option2=Value2 ...
```

Дополнительные пары имя-значение передаются методу действия как параметры. Значение опции `xyz` соответствует параметру `$xyz` метода действия. К примеру, если мы определим следующий класс команды:

```
class SitemapCommand extends CConsoleCommand
{
```



```
public function actionIndex($type, $limit=5) { ... }
public function actionInit() { ... }
```

```
}
```

То все следующие консольные команды вызовут `actionIndex('News', 5)`:

```
php entryScript.php sitemap index --type=News --limit=5
```

```
// $limit принимает значение по умолчанию
```

```
php entryScript.php sitemap index --type=News
```

```
// $limit принимает значение по умолчанию.
```

```
// Так как 'index' — действие по умолчанию, мы можем опустить имя действия.
```

```
php entryScript.php sitemap --type=News
```

```
// порядок опций не важен
```

```
php entryScript.php sitemap index --limit=5 --type=News
```

Если значение опции не указано (то есть `--type` вместо `--type=News`), соответствующему параметру действия будет присвоено значение `true`.

Примечание: Альтернативные форматы указания опций, такие как `--type News` или `-t News` не поддерживаются.

Если объявить параметр как массив, он сможет принять массив значений:

```
public function actionIndex(array $types) { ... }
```

Чтобы передать массив значений необходимо указать одну и ту же опцию несколько раз:

```
php entryScript.php sitemap index --types=News --types=Article
```

Команда, приведённая выше, запустит `actionIndex(array('News', 'Article'))`.

Доступ к ресурсам

Из консольной команды через метод `Yii::app()` можно обратиться к экземпляру приложения. Консольное приложение также, как и веб-приложение, можно конфигурировать. Например, мы можем настроить компонент приложения `db` для доступа к базе данных. Конфигурация, как правило, оформляется в виде PHP-файла и передается конструктору класса консольного приложения (или методу `createConsoleApplication` во входном скрипте).

Использование утилиты `yiic`

Мы использовали утилиту `yiic` для создания первого Yii-приложения. Эта утилита на самом деле является консольным приложением с входным скриптом `framework/yiic.php`. Ее использование позволяет выполнить ряд задач: создание каркаса веб-приложения, генерация классов контроллеров и моделей, генерация кода, необходимого для операций CRUD, получение списка сообщений, которые необходимо перевести и т.д.

Можно совершенствовать утилиту, добавляя собственные команды. Для этого вначале необходимо создать каркас приложения с использованием команды `yiic webapp`, как описано в разделе Создание первого Yii-приложения. Эта команда создаст два файла в папке `protected/yiic` и `yiic.bat` — *локальные* копии утилиты `yiic`, сгенерированные специально для создаваемого веб-приложения.

Теперь мы можем создавать собственные команды в папке `protected/commands`. При запуске локальной версии утилиты мы увидим, что к стандартному набору команд добавились созданные нами. Кроме того,

можно создать команды, которые будут доступны при использовании `yiic shell`. Для этого файлы классов команд нужно просто поместить в папку `protected/commands/shell`.

Начиная с версии 1.1.1, можно создавать глобальные команды, общие для всех приложений Yii, установленных на одном сервере. Для этого необходимо задать переменную окружения `YII_CONSOLE_COMMANDS`, указывающую на существующую директорию, которая будет хранить классы команд. Команды будут доступны везде, где мы используем `yiic`.

Безопасность

Предотвращение межсайтового скриптинга

Межсайтовый скриптинг (также известный как XSS) — злонамеренный сбор информации пользователя через страницы веб-приложения. Чаще всего, производящий атаку, используя уязвимости приложения, включает в текст страницы JavaScript, VBScript, ActiveX, HTML или Flash. Делается это для получения информации других пользователей приложения и последующего её использования в нехороших целях. К примеру, плохо написанный форум может отображать сообщения пользователей без какой-либо проверки. Атакующий может вставить JavaScript-код в сообщение. Все, кто прочитает это сообщение, выполнит код на своём компьютере.

Чтобы не допустить XSS-атак, нужно всегда проверять то, что ввёл пользователь, прежде чем это отображать. Конечно, чтобы не допустить ввода скриптов, можно кодировать все HTML-сущности. В некоторых ситуациях такое поведение нежелательно, так как ввод HTML становится недоступен. Yii включает в себя библиотеку `HTMLPurifier` и предоставляет разработчику полезный компонент `CHtmlPurifier`, который может отфильтровать весь вредоносный код при помощи тщательно проверенного белого листа. Также компонент делает код совместимым со стандартами.

`CHtmlPurifier` может быть использован и как виджет и как фильтр. При использовании в качестве виджета `CHtmlPurifier` обрабатывает заключённое в него содержимое:

```
<?php $this->beginWidget('CHtmlPurifier'); ?>
```

...этот текст будет подвергнут деликатной санобработке...

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

Предотвращение подделки межсайтовых запросов

Подделка межсайтового запроса (CSRF) — атака, при которой сайт атакующего заставляет браузер пользователя выполнить какое-либо действие на другом сайте. К примеру, на сайте атакующего есть страница, содержащая тэг `img` с атрибутом `src`, указывающим на сайт банка:

`http://bank.example/перевод?сумма=10000&кому=кулхацкеру`. Если в браузере пользователя установлен cookie, позволяющий запомнить его на сайте, посещение такой страницы вызовет перевод 10000 тугриков нехорошему кулхацкеру. В CSRF, в отличие от межсайтового скриптинга, основанного на доверии пользователя к некоторому сайту, используется доверие сайта определённому пользователю. Для того, чтобы не допустить CSRF, важно придерживаться простого правила: `GET` — только для получения данных. Ничего менять при `GET`-запросах нельзя. Для `POST` необходимо использовать случайное значение, которое можно проверить на сервере и убедиться, что запрос идёт оттуда, откуда нужно.

В Yii реализована защита от CSRF-атаки, проводимой через `POST`. Защита основана на хранении случайного значения в cookie и сравнения его с значением в `POST`.

По умолчанию, защита от CSRF отключена. Для её включения необходимо настроить компонент `CHttpRequest` в файле конфигурации:

```
return array(
    'components'=>array(
        'request'=>array(
            'enableCsrfValidation'=>true,
        ),
    ),
);
```

Для отображения формы следует использовать `CHtml::form` вместо написания HTML-тэга. Данный метод позволяет автоматически включить случайное значение, используемое для проверки на CSRF, как скрытое поле формы.

Предотвращение атак через cookie

Защита cookie очень важна, так как именно в них чаще всего хранится ID сессии. Если злоумышленник получит ID сессии, он получит и всю информацию, которая в ней хранится.

Есть несколько способов предотвращения атак через cookie:

- Использовать SSL для создания защищённого соединения и передавать cookie только через него. Атакующий не сможет расшифровать содержимое передаваемых cookie.
- Вовремя объявлять сессию устаревшей, включая все cookie и маркеры сессии, для того, чтобы снизить возможность атаки.
- Предотвратить XSS, тем самым исключив захват cookie.
- Проверять данные cookie и определять, изменены ли они.

В Yii реализована проверка на изменения через подсчёт хэша HMAC от значений cookie.

По умолчанию проверка cookie отключена. Для её включения необходимо в конфигурации приложения настроить компонент CHttpRequest следующим образом:

```
return array(
    'components'=>array(
        'request'=>array(
            'enableCookieValidation'=>true,
        ),
    ),
);
```

При использовании проверки cookie, обращаться к ним необходимо через коллекцию cookies, а не напрямую через `$_COOKIE`:

```
// Получаем cookie с заданным именем
$cookie=Yii::app()->request->cookies[$name];
$value=$cookie->value;
...
// Отсылаем cookie
$cookie=new CHttpCookie($name,$value);
Yii::app()->request->cookies[$name]=$cookie;
```

Генерация кода при помощи консоли (устаревшее)

Примечание: Генераторы кода `yiic shell` считаются устаревшими начиная с версии 1.1.2. Пожалуйста, используйте более мощные расширяемые веб-генераторы Gii.

Откроем консоль и выполним следующие команды:

```
% cd WebRoot/testdrive
% protected/yiic shell
Yii Interactive Tool v1.1
Please type 'help' for help. Type 'exit' to quit.
>> model User tbl_user
    generate models/User.php
    generate fixtures/tbl_user.php
    generate unit/UserTest.php
```

The following model classes are successfully generated:

User

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

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

```
>> crud User
generate UserController.php
generate UserTest.php
mkdir D:/testdrive/protected/views/user
generate create.php
generate update.php
generate index.php
generate view.php
generate admin.php
generate _form.php
generate _view.php
```

Crud 'user' has been successfully created. You may access it via:

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

В примере выше мы использовали команду `shell` утилиты `yiic` для взаимодействия с созданным каркасом приложения. В командной строке мы вводим две команды: `model User tbl_user` и `crud User`. Команда `model` автоматически создает класс модели `User`, основываясь на структуре таблицы `tbl_user`, а команда `crud` генерирует класс контроллера и файлы представлений, которые обеспечивают выполнение соответствующих операций CRUD.

Примечание: Даже, если проверка соответствия требованиям показывает, что расширение PDO и драйвер PDO, соответствующий используемой базе данных, включены, могут возникать ошибки типа «...could not find driver». В этом случае необходимо запустить утилиту `yiic` следующим образом:

```
% php -c path/to/php.ini protected/yiic.php shell
```

где `path/to/php.ini` — путь до файла PHP ini

Давайте порадуемся нашим трудам, перейдя по следующему URL:

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

Мы увидим страницу со списком пользователей из таблицы `tbl_user`. Поскольку наша таблица пустая, то записей в ней не будет. Кликнем по кнопке `Create User` и, если мы еще не авторизованы, отобразится страница авторизации. Затем загрузится форма добавления нового пользователя. Заполним её и нажмем кнопку `Create`. Если при заполнении формы были допущены ошибки, мы увидим аккуратное сообщение об ошибке.

Возвращаясь к списку пользователей, мы должны увидеть в списке только что созданного пользователя. Добавим еще несколько пользователей. Обратите внимание, что при значительном количестве пользователей для их отображения на одной странице список будет автоматически разбиваться на страницы. Авторизовавшись в качестве администратора (`admin/admin`), можно увидеть страницу управления пользователями по адресу:

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

Появится аккуратная таблица пользователей. Можно кликнуть на заголовок таблицы, чтобы упорядочить записи по значениям соответствующего столбца. Для просмотра, редактирования или удаления соответствующей строки можно воспользоваться кнопками. Также можно переходить на разные страницы, фильтровать результаты и производить поиск по ним. Всё это не требует написания кода!

Страница управления пользователями

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	
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](#).

Страница добавления нового пользователя

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](#).