ГЛАВА 3

Управление учетными записями пользователей

В главе 2 нами был изучена архитектура "модель–представление–контроллер" (MVC), позволяющая легко отделить выполняемые приложением операции от визуального представления результатов, и эта архитектура была реализована с использованием класса `Zend_Controller_Front`. Сейчас мы займемся расширением возможностей контроллера, реализуя аутентификацию и вход пользователя на сайт, а также управление учетными записями пользователей.

На этом этапе важно осознать разницу между понятиями *аутентификация* (*authentication*) и авторизация (*authorization*) пользователя.

- *Аутентификация* заключается в проверке, действительно ли пользователь — тот, за кого себя выдает. Обычно для этого проверяется его пользовательское имя-псевдоним и прилагающийся к нему пароль.
- *Авторизация* определяет, можно ли допустить пользователя к определенному ресурсу — при условии, что он успешно прошел процедуру аутентификации. В ходе авторизации также определяется, что разрешено делать на сайте пользователю, не прошедшему аутентификацию. В нашем приложении под ресурсом надо понимать определенную веб-страницу или операцию — например, операцию помещения в блог новой заметки.

В этой главе мы добавим в приложение средства аутентификации пользователей с помощью компонента `Zend_Auth` из библиотеки Zend Framework. В числе этих средств — таблицы базы данных для хранения информации о пользователях. Затем с помощью компонента `Zend_Acl` определим, каким пользователям предоставлять право доступа к тем или иным ресурсам. Кроме того, систему допусков нужно связать с классом `Zend_Controller_Front`.

Создание таблицы пользователей в базе данных

Поскольку в нашем приложении предусмотрено хранение учетных записей множества пользователей, все эти учетные записи надо как-то организовать. Для этого создадим таблицу базы данных под названием `users`. Каждая запись таблицы соответствует одному пользователю и содержит такую информацию, как имя, пароль и некоторые другие важные данные.

Доступ к нашему веб-приложению будут иметь три категории пользователей: *гости* (*guest*), *зарегистрированные пользователи* (*member*) и *администраторы* (*administrator*). Любой посетитель сайта автоматически квалифицируется как гость, пока не войдет на сайт с паролем как зарегистрированный пользователь. Чтобы отличать их от администраторов, в таблице `users` для каждого пользователя

предусмотрен столбец `role` (роль). Этот столбец будет использоваться при реализации списков управления доступом с помощью компонента `Zend_Acl`.

Примечание

В более сложной системе с одним пользователем можно ассоциировать несколько ролей, но мы для простоты позволим пользователям иметь только одну. Пользователь с правами администратора сможет также выполнять все функции обычного зарегистрированного пользователя. Категории пользователей можно было бы хранить в отдельной таблице, но опять-таки простоты ради мы пренебрежем этой возможностью и поместим статический список категорий прямо в код.

Для каждого пользователя в базе данных будут храниться следующие ключевые параметры:

- `user_id` — целое число, используемое для внутреннего представления пользователя, его идентификационный номер.
- `username` — уникальный буквенный идентификатор пользователя, используемый для входа на сайт. Это имя будет публичным — оно отображается в записях блога и других видах общедоступного контента сайта вместо настоящих имен, которые пользователи обычно скрывают.
- `password` — строка символов, по которой выполняется аутентификация пользователя. Пароли будут храниться в хешированном виде с использованием функции `md5()`. Это означает, что забытый пароль нельзя извлечь из базы и восстановить, а можно только заменить новым. Весь код, необходимый для этого, будет реализован.
- `user_type` — строка, обозначающая классификацию пользователя по системе типов или категорий (это будет либо `admin`, т.е. администратор, либо `member`, т.е. зарегистрированный пользователь; в будущем вы сможете легко добавить новые типы на основе полученных знаний).
- `ts_created` — время создания учетной записи.
- `ts_last_login` — время последнего входа пользователя на сайт под своим именем. Это поле может иметь значение `null`, поскольку только что зарегистрированный пользователь еще ни разу не входил на сайт.

В листинге 3.1 приведены команды SQL, необходимые для создания таблицы `users` в базе данных MySQL. Все SQL-определения структуры таблицы хранятся в файле `schema-mysql.sql`, помещенном в корневой каталог приложения. Если вместо MySQL используется PostgreSQL, соответствующая структура будет находиться в файле `schema-pgsql.sql`.

Примечание

Способ хранения структуры базы данных для вашего собственного веб-приложения — это целиком и полностью ваш выбор. Я выбрал такой способ только затем, чтобы легче было просматривать эту структуру, в том числе при загрузке кода к этой книге с веб-сайта.

Листинг 3.1. Код SQL для создания таблицы пользователей в базе данных MySQL (файл `schema-mysql.sql`)

```
create table users (
    user_id         serial          not null,
    username        varchar(255)    not null,
    password        varchar(32)     not null,
    user_type       varchar(20)     not null,
```

```
    ts_created      datetime        not null,
    ts_last_login   datetime,

    primary key (user_id),
    unique (username)
) type = InnoDB;
```

Тип столбца (поля) `user_id` определен как `serial`, что эквивалентно `bigint unsigned not null auto_increment`. Лично я предпочитаю `serial` — это короче, быстрее и совместимо с PostgreSQL.

Поле `username` может иметь до 255 символов в длину, хотя в нашем коде на длину будут наложены дополнительные ограничения. Пароль будет храниться в зашифрованном по методу MD5 виде, поэтому полю пароля достаточно иметь в длину 32 символа.

Далее идет поле `user_type`. Длина этого поля не играет особой роли; правда, название любого нового типа (категории), который вы захотите добавить, должно иметь в длину не более 20 символов (это внутреннее, невидимое пользователям имя, поэтому оно не обязано быть очень информативным). Это поле используется при проверках прав доступа.

Наконец, имеются два поля с временными метками. Собственно говоря, в MySQL имеется тип данных под названием `timestamp`, но я решил вместо него использовать тип `datetime`, поскольку СУБД автоматически обновляет столбцы, данные в которых имеют тип `timestamp`. В PostgreSQL вместо этого типа нужно использовать `timestamptz` (см. файл `schema-pgsql.sql` с определением структуры таблицы). Далее в разделе "Временные метки" имеется более подробная информация о работе с временем в PHP.

Совет

В листинге 3.1 при создании таблицы используется тип `InnoDB`, благодаря чему есть возможность выполнять транзакции SQL и разрешены реляционные связи по внешним ключам. По умолчанию использовался бы тип `MyISAM`, в котором этих возможностей нет.

Теперь необходимо создать в базе данных эту таблицу. Есть два способа сделать это. Во-первых, можно перенаправить весь файл `schema-mysql.sql` в базу данных следующей командой:

```
# mysql -u phpweb20 -p phpweb20 < schema-mysql.sql
```

После ввода этой команды вам будет предложено ввести пароль. База данных будет создана целиком с нуля.

Вместо этого можно подключиться напрямую к базе данных, скопировать и вставить структуру таблицы следующей командой:

```
# mysql -u phpweb20 -p phpweb20
```

Поскольку по мере работы над приложением мы будем дорабатывать базу данных, рекомендую второй способ, чтобы просто добавлять новые таблицы по мере надобности.

Временные метки

Способы работы с датой и временем в PHP, MySQL и PostgreSQL часто бывают малопонятны пользователю. Прежде чем двигаться дальше, необходимо прояснить некоторые важные моменты в работе с датой и временем в СУБД MySQL.

MySQL не хранит вместе с данными о дате и времени информацию о часовом поясе. Это означает, что сервер MySQL должен быть настроен на тот же часовой пояс, что и PHP, иначе временные метки будут вести себя странно. Например, если хотите воспользоваться функцией PHP `date()` для форматирования временной метки в таблице MySQL, будьте осторожны: извлечение отметки из базы с помощью функции MySQL `unix_timestamp()` даст неправильную дату, если часовые пояса отличаются.

В использовании полей типа "дата" в MySQL есть три основных недостатка.

- При необходимости переместить базу данных на новый сервер (например, при смене веб-хостинга) перемещаемые данные окажутся неправильными, если на новом месте используется другой часовой пояс. Потребуется изменить конфигурацию сервера, чего многие веб-серверы сами делать не станут.
- Возникнут различные проблемы с датами перехода на летнее и зимнее время (если в вашей зоне такой переход выполняется).
- Трудно хранить временные метки из разных часовых поясов. Прежде чем помещать их в базу, придется привести время к часовому поясу сервера.

Вы скажете, что такие проблемы возникают не так уж часто, — и будете правы. Хотя вот вам пример из практики. Одно веб-приложение, которое я разрабатывал, помещало на сайт полное расписание матчей одной спортивной лиги (кроме прочей информации). В течение нескольких недель матчи проходили в разных городах и, соответственно, в разных часовых поясах. Чтобы точно составить расписание, помещаемое на сайт (где указывалось, например, что до матча остается три часа), приходилось точно учитывать часовые пояса.

В СУБД PostgreSQL нет типа данных `datetime`. Вместо него я предпочитаю использовать поле `timestamptz`, в котором хранится дата, время и часовой пояс. Если не указать пояс при помещении значения в это поле, он будет автоматически взят по времени сервера (например, можно задавать как `2007-04-18 23:32:00`, так и `2007-04-18 23:32:00+9:30`; в первом выражении будет использован часовой пояс сервера, а во втором — `+9:30`).

В примере с расписанием матчей я использовал PostgreSQL, что позволило легко сохранять в базе часовой пояс матча. В базе данных PostgreSQL эквивалентом функции `unix_timestamp(ts_column)` является функция `extract(epoch from ts_column)`. С помощью типа `timestamptz` посредством нее можно получить точное значение, которое затем используется в функции PHP `date()`. Переход на зимнее и летнее время также учитывается этой функцией безо всяких проблем.

Профили пользователей

Может быть, вы заметили, что таблица `users` (листинг 3.1) не содержит никакой полезной информации о самих пользователях — ни имен, ни адресов электронной почты. Для хранения этих данных будет создана еще одна таблица — `users_profile`.

Вводя новую таблицу для хранения информации о пользователях, мы добиваемся того, что эту информацию можно будет неограниченно расширять, совсем не затрагивая таблицу `users`. Например, в эту таблицу можно поместить имя, адрес электронной почты, номера телефонов, место проживания, любимое блюдо и любую другую информацию, а также различные настройки.

Каждая запись в такой таблице будет соответствовать одному значению одного параметра пользовательского профиля. Другими словами, одна запись соответствует адресу почты, другая — имени и т.п. Извлечение этих данных в ходе работы

сайта несколько более затратно, чем в других случаях, но гибкость хранения информации компенсирует эти затраты. Таблица должна иметь три столбца:

- `user_id` — ссылка на пользователя, т.е. запись в таблице `users`;
- `profile_key` — имя (определение) параметра, который хранится в данной записи; например, для записей, в которых хранятся адреса электронной почты, логично использовать имя `email`;
- `profile_value` — собственно значение параметра; если значение поля `profile_key` равно `email`, текущее поле будет содержать сам адрес.

Совет

Полю `profile_value` у нас назначен тип `text`, поскольку это позволяет при необходимости сохранить в нем большое количество информации. В СУБД MySQL и PostgreSQL нет разницы в быстродействии при работе с типами `varchar` и `text`. На самом деле MySQL даже создает внутреннее поле `varchar` в качестве наименьшего возможного поля типа `text`, в зависимости от указанного размера.

В листинге 3.2 показано определение таблицы `users_profile`. Код по управлению профилями пользователей будет реализован несколько позже в этой главе.

Листинг 3.2. Запрос SQL для создания таблицы `users_profile` в MySQL (файл `schema-mysql.sql`)

```
create table users_profile (
    user_id         bigint unsigned not null,
    profile_key     varchar(255)    not null,
    profile_value   text            not null,

    primary key (user_id, profile_key),
    foreign key (user_id) references users (user_id)
) type = InnoDB;
```

Как говорилось ранее, тип поля `serial` (используемый в поле `user_id`, см. листинг 3.1) — это условное обозначение целочисленного типа `bigint` без знака, с автоприращением. Поскольку поле `user_id` в данной таблице ссылается на таблицу `users`, необходимо использовать тип `bigint unsigned` — автоприращение от этого поля здесь не требуется. Первичными ключами в таблице `users_profile` выбраны `user_id` и `profile_key`, поскольку параметры профиля не могут повторяться для одного пользователя. А вот несколько различных параметров пользователь вполне может иметь.

Примечание

При работе с PostgreSQL для поля `user_id` используется тип данных `int`, поскольку именно на нем основан тип `serial` в PostgreSQL. Напомним, что версия таблицы для PostgreSQL определена в файле `schema-pgsql.sql`.

Основы работы с компонентом Zend_Auth

Итак, таблица `users` создана, и появилась возможность применения компонента `Zend_Auth` для аутентификации пользователя. Но прежде чем заняться реализацией этой возможности, следует понять, как именно работает `Zend_Auth`.

Прежде всего разберемся с терминологией, используемой этим компонентом. Уникальную информацию, которая целиком определяет пользователя, назовем его *идентификационными данными* (*identity*). После успешной аутентификации и входа пользователя на сайт его идентификационные данные сохраняются в параметрах сеанса PHP для распознавания этого пользователя при обработке последующих запросов.

Примечание

Можно разработать и другие методы хранения этих данных, но все-таки, вероятно, наиболее популярным является хранение в переменной сеанса PHP. Компонент `Zend_Auth` для этой цели располагает классом `Zend_Auth_Storage_Session`. Этот класс, в свою очередь, использует компонент `Zend_Session`, который в принципе представляет собой оболочку для переменной `$_SESSION` языка PHP (хотя и предоставляет больше возможностей). Чтобы создать другой метод хранения, нужно реализовать интерфейс `Zend_Auth_Storage_Interface`. Например, при необходимости запоминать пользователя между его сеансами можно было бы создать класс, записывающий идентификационные данные в cookie-файлы. После этого следовало бы написать адаптер (об этом скоро будет сказано подробнее) для выполнения аутентификации на основе данных из cookie-файлов. Но с этим способом нужно соблюдать осторожность — он представляет потенциальную опасность, поскольку cookie-файлы можно сфальсифицировать. Одно из средств безопасности в этом случае — назначить пользователю ограниченные права, пока он не подтвердит свою личность вводом пароля, как это сделано на сайте Amazon.com. Там пользователя запоминают, но не разрешают вносить изменения в свою учетную запись, пока он не введет пароль. Еще один пример альтернативного хранения идентификационных данных можно найти в среде со сбалансированной нагрузкой (когда один сайт распределен между несколькими серверами). Сеансы, хранимые на диске, обычно не могут быть доступны всем серверам, и каждый следующий запрос пользователя может выполняться не тем сервером, который выполнял предыдущий. Помещение информации о сеансе в базу данных решает эту проблему.

Чтобы пользователь прошел аутентификацию, он должен предоставить *контрольную информацию* (*credentials*). В нашем приложении такой информацией будет служить поле `password` (пароль) таблицы `users`.

Проверка идентификационных данных и контрольной информации путем их сравнения с информацией из базы данных выполняется *адаптером*. В компоненте `Zend_Auth` адаптеры реализуют интерфейс `Zend_Auth_Adapter_Interface`. К счастью, в библиотеке Zend Framework имеется готовый адаптер, которым можно пользоваться для работы с базой данных MySQL. Если бы понадобилось при аутентификации извлекать данные пользователя из других источников (LDAP или файл пароля, сгенерированный функцией Apache `htpasswd`), то нужен был бы новый адаптер.

Мы будем пользоваться адаптером `Zend_Auth_Adapter_DbTable`, специально разработанным для компонента `Zend_Auth`. Если вы решите написать свой собственный, достаточно будет реализовать метод `authenticate()`, который возвращает объект `Zend_Auth_Result`. Этот объект содержит информацию о том, успешно ли прошла аутентификация пользователя, а также диагностические сообщения (например, о правильности контрольной информации, неудачной аутентификации из-за отсутствия учетной записи и т.п.)

По умолчанию адаптер `Zend_Auth_Adapter_DbTable` возвращает только представленное пользователем имя в объекте `Zend_Auth_Result`. Но нам-то нужно хранить больше информации: настоящие имена и, что самое важное, тип пользователя. Эта проблема будет решена в ходе реализации входа пользователей на сайт через компонент `Zend_Auth`.

Создание объекта Zend_Auth

Компонент Zend_Auth допускает создание только одного своего экземпляра (как и Zend_Controller_Front, который использовался в главе 2). Поэтому для получения этого экземпляра можно использовать метод getInstance(). Затем нужно задать способ хранения идентификационной информации (мы используем для этого переменные сеанса) с помощью метода setStorage(). Если используется сразу несколько способов хранения, этот метод понадобится вызывать всякий раз, когда необходимо обратиться к идентификационным данным в новом месте. Но все-таки, как правило, нужда в его вызове возникает один раз: в начале запроса.

Следующий код используется для создания экземпляра класса Zend_Auth. Как видите, все происходит довольно очевидным образом:

```
<?php
    $auth = Zend_Auth::getInstance();
    $auth->setStorage(new Zend_Auth_Storage_Session());
?>
```

Объект $auth будет использоваться в нескольких местах нашего приложения. Во-первых, это проверка допусков пользователя с помощью компонента Zend_Acl (см. раздел "Основы работы с компонентом Zend_Acl" позже в этой главе). Во-вторых, это методы входа на сайт и выхода с него, поскольку в каждом из них идентификационные данные нужно будет сохранить, а затем уничтожить.

Как мы уже поступали ранее с конфигурацией приложения и соединением с базой данных, объект $auth будет помещен в реестр приложения с помощью класса Zend_Registry. В листинге 3.3 показан загрузочный файл index.php после добавления в него работы с компонентом Zend_Auth.

Листинг 3.3. Загрузочный файл приложения с использованием компонента Zend_Auth (файл index.php)

```
<?php
    require_once('Zend/Loader.php');
    Zend_Loader::registerAutoload();

    // загрузка конфигурации приложения
    $config = new Zend_Config_Ini('../settings.ini', 'development');
    Zend_Registry::set('config', $config);

    // создание объекта системного журнала
    $logger = new Zend_Log(new Zend_Log_Writer_Stream($config-
>logging->file));
    Zend_Registry::set('logger', $logger);

    // соединение с базой данных
    $params = array('host'     => $config->database->hostname,
                    'username' => $config->database->username,
                    'password' => $config->database->password,
                    'dbname'   => $config->database->database);

    $db = Zend_Db::factory($config->database->type, $params);
    Zend_Registry::set('db', $db);

    // настройка аутентификации пользователей
    $auth = Zend_Auth::getInstance();
    $auth->setStorage(new Zend_Auth_Storage_Session());
```

```
    // обработка запроса пользователя
    $controller = Zend_Controller_Front::getInstance();
    $controller->setControllerDirectory($config->paths->base .
                                        '/include/Controllers');
    $controller->registerPlugin(new
CustomControllerAclManager($auth));

    // настройка визуализатора
    $vr = new Zend_Controller_Action_Helper_ViewRenderer();
    $vr->setView(new Templater());
    $vr->setViewSuffix('tpl');
    Zend_Controller_Action_HelperBroker::addHelper($vr);

    $controller->dispatch();
?>
```

Аутентификация пользователя компонентом Zend_Auth

В главе 4 мы реализуем формы входа на сайт и выхода с него для нашего приложения. Но перед этим необходимо разобраться, как вообще работают эти процедуры. Как уже говорилось, для этих целей мы будем использовать адаптер аутентификации `Zend_Auth_Adapter_DbTable`. Прежде чем пользоваться им, необходимо создать реально работающий объект `Zend_Db`.

Объект `Zend_Auth_Adapter_DbTable` обладает очень гибкими возможностями и приспособлен к работе с любыми конфигурациями баз данных, поэтому ему надо сообщить конкретный способ хранения идентификационной информации пользователей. При создании этого объекта нужно указать следующую информацию:

- имя используемой таблицы базы данных (наша таблица носит имя `users`);
- столбец (поле) с личным идентификатором пользователя (у нас для этого выделен столбец `username` в таблице `users`);
- столбец (поле) с контрольной информацией пользователя (у нас это столбец `password`);
- и наконец, какую операцию следует выполнять над контрольной информацией. Это фактически функция, которая применяется к этой информации (если она задана). Напомним, что в поле `password` хранится пароль, хешированный по методу MD5. Поэтому в качестве последнего аргумента передается `md5(?)`, а вопросительный знак указывает объекту `Zend_Db`, куда подставлять пароль.

После создания и инициализации объекта `Zend_Auth_Adapter_DbTable` (в виде переменной `$adapter`) можно задать значения идентификационных данных (имени пользователя) и контрольной информации (пароля). Для этого используются методы `setIdentity()` и `setCredentials()`.

После этого вызывается метод `authenticate()` из объекта `$auth` (экземпляра `Zend_Auth`). Единственный передаваемый в этот метод аргумент — объект-адаптер (`$adapter`). В результате возвращается экземпляр класса `Zend_Auth_Result`. К этому объекту можно применить метод `isValid()`, чтобы проверить, успешно ли прошла аутентификация пользователя. Если это не так, то можно либо вызвать метод `getMessages()` и посмотреть причину, либо сгенерировать собственное сообщение об ошибках на основе кода ошибки, возвращаемого методом `getCode()`.

Примечание

Хотя объект `Zend_Auth_Result` позволяет легко различать, когда пользователь ввел неверное имя, а когда — неверный пароль, эту информацию, как правило, не нужно показывать пользователю. Иначе злонамеренному посетителю дается неявный намек на то, существует ли вообще данное имя или нет, что может помочь ему незаконно проникнуть на сайт. Пример в листинге 3.4 выдает соответствующие сообщения просто для того, чтобы продемонстрировать, как диагностируются ошибки. А вот в коде, который добавляется в наше приложение, мы не будем информировать пользователя, что именно он ввел неправильно: пароль или имя.

В листинге 3.4 приведен код для инициализации объекта `Zend_Auth_Adapter_DbTable` и аутентификации пользователя по данным таблицы `users`. На этом этапе просто укажем фиктивное имя и пароль, поскольку таблица `users` еще не заполнена. Как видно из листинга, в программе также обрабатываются ошибки аутентификации и выдается сообщение о причине ошибки.

Листинг 3.4. Аутентификация пользователя по таблице базы данных компонентами `Zend_Auth` и `Zend_Db` (файл `listing-3-4.php`)

```
<?php
    require_once('Zend/Loader.php');
    Zend_Loader::registerAutoload();

    // соединение с базой данных
    $params = array('host' => 'localhost',
                    'username' => 'phpweb20',
                    'password' => 'myPassword',
                    'dbname' => 'phpweb20');

    $db = Zend_Db::factory('pdo_mysql', $params);

    // настройка системы аутентификации
    $auth = Zend_Auth::getInstance();
    $auth->setStorage(new Zend_Auth_Storage_Session());

    $adapter = new Zend_Auth_Adapter_DbTable($db,
                                             'users',
                                             'username',
                                             'password',
                                             'md5(?)');

    // попытка войти под именем "fakeUsername"
    $adapter->setIdentity('fakeUsername');
    $adapter->setCredential('fakePassword');
    $result = $auth->authenticate($adapter);

    if ($result->isValid()) {
        // успешная аутентификация пользователя
    }
    else {
        // пользователь не аутентифицирован

        switch ($result->getCode()) {
            case Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND:
                echo 'Identity not found';
                break;
            case Zend_Auth_Result::FAILURE_IDENTITY_AMBIGUOUS:
```

```
                echo 'Multiple users found with this identity!';
                break;
            case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
                echo 'Invalid password';
                break;
            default:
                var_dump($result->getMessages());
        }
    }
?>
```

Можно также проверить, прошла ли аутентификация пользователя успешно, посредством объекта `$auth`. Метод `hasIdentity()` определяет, произошла ли аутентификация или нет. Затем для определения, какой именно это пользователь, можно вызвать метод `getIdentity()`.

Аналогично, для выхода пользователя из системы используется метод `clearIdentity()`. Если данные о пользователе хранятся в переменных сеанса, то идентификационная информация при этом удаляется.

Как уже говорилось, в случае успешного выполнения вызова `$auth->authenticate()` с помощью `Zend_Auth_Adapter_DbTable` в качестве идентификационных данных сохраняется только имя пользователя. В главе 4, в ходе реализации формы для входа пользователя, мы добавим в сохраняемую идентификационную информацию и другие данные — например, тип (категорию) пользователя.

Основы работы с компонентом Zend_Acl

Компонент `Zend_Acl` входит в библиотеку Zend Framework и обеспечивает работу такого средства, как список управления доступом (*access control list — ACL*). Хотя для его работы не требуется `Zend_Auth`, мы объединим эти два компонента с целью контроля, на что пользователи имеют право и на что не имеют в нашем веб-приложении.

По сути компонент `Zend_Acl` определяет, достаточно ли определенная *роль* имеет полномочий (*привилегий*) для доступа к некоторому *ресурсу*.

- **Ресурс** — это некоторый объект веб-приложения (не в смысле объектно-ориентированного программирования, а информационный), доступ к которому можно регулировать. Примером ресурса может служить операция веб-приложения — например, утверждение статьи модераторами перед ее публикацией на сайте или удаление пользователя из системы. Привилегии доступа к ресурсам можно назначать и на более дробном уровне. Например, в случае утверждения статьи ресурсом может выступать система публикации статей (или одна конкретная заметка, в зависимости от контекста), тогда как операция утверждения этой статьи будет привилегией доступа.
- **Роль** — это некоторый объект, запрашивающий доступ к ресурсу. В нашем веб-приложении это будет пользователь с определенными привилегиями.

Может возникнуть путаница с терминами, поэтому поясним дополнительно. Каждый пользователь нашего приложения (т.е. каждая запись в таблице базы данных `users`) имеет определенный тип, или категорию. Вот этот тип и есть роль.

Доступ к тому или иному ресурсу (скажем, операциям публикации записи в блоге или смены пароля) будет основываться на ролях пользователя. Как уже говорилось при создании таблицы `users`, существуют три типа пользователей: *гости* (`guest`), *зарегистрированные пользователи* (`member`) и *администраторы* (`administrator`).

Примечание

Имеется возможность наследовать роли от других ролей и права доступа к одним ресурсам — от других ресурсов. Пусть роли А назначены определенные привилегии. Если сделать так, чтобы роль Б наследовала роли А, то она получит все привилегии роли А и в дополнение — свои собственные, специально назначенные ей. От этого система допусков становится запутанной (особенно если наследовать сразу от нескольких ролей или ресурсов), поэтому в нашем приложении мы попробуем ограничиться самыми простыми вариантами.

Ниже описан типичный алгоритм работы с компонентом `Zend_Acl` в веб-приложении.

1. Создать экземпляр класса `Zend_Acl` (пусть объект имеет имя `$acl`).
2. Добавить одну или несколько ролей в объект `$acl` с помощью метода `addRole()`.
3. Добавить список ресурсов в объект `$acl`, используя метод `add()`.
4. Добавить полный список привилегий для каждой роли (указать с помощью методов `allow()` и `deny()`, к каким ресурсам имеет доступ данная роль).
5. Применить метод `isAllowed()` объекта `$acl`, чтобы определить, имеет ли та или иная роль доступ к той или иной комбинации ресурса и привилегии.
6. Повторить шаг 5 столько раз, сколько нужно, в ходе выполнения сценария.

Пример работы с классом Zend_Acl

Приведем пример практического применения класса `Zend_Acl`. В этом примере будут использоваться те же имена ролей, что и в нашем будущем приложении. Распределение привилегий в примере должно дать вам представление о том, что мы будем делать при интегрировании компонента `Zend_Acl` в наше приложение.

Первым делом необходимо создать и инициализировать экземпляр класса `Zend_Acl`. Его конструктор не принимает никаких аргументов:

```
$acl = new Zend_Acl();
```

Затем создаются все роли, для которых должны проверяться допуски. Как уже говорилось, ролей будет три: гость (`guest`), зарегистрированный пользователь (`member`) и администратор (`administrator`).

```
$acl->addRole(new Zend_Acl_Role('guest'));
$acl->addRole(new Zend_Acl_Role('member'));
$acl->addRole(new Zend_Acl_Role('administrator'));
```

После создания ролей можно создать ресурсы. Вообще, порядок создания можно и изменить; важно, чтобы и роли, и ресурсы добавлялись до того, как определяются допуски.

Для этого примера в качестве ресурсов, требующих допуска, будут добавлены только `account` ("учетная запись") и `admin` ("администратор"). В нашем приложении будут и другие ресурсы, но нам здесь нужны только те, к которым будут назначаться допуски, поскольку при проверке допусков первым делом проверяется существование запрашиваемого ресурса. Способ проверки допусков к несуществующему ресурсу целиком зависит от вашей воли как разработчика. В данном случае я просто разрешу доступ к запрошенному ресурсу, если он не добавлен в объект `$acl`.

```
$acl->add(new Zend_Acl_Resource('account'));
$acl->add(new Zend_Acl_Resource('admin'));
```

Следующим шагом будет определение разных уровней допуска, требующихся для работы приложения. Это делается серией вызовов методов `allow()` и `deny()`

экземпляра Zend_Acl. Первый аргумент этой функции — роль, а второй — ресурс. Передавая третий параметр, название привилегии, можно обеспечить более тонкий контроль доступа.

В системе допусков, используемой в нашем приложении, имя контроллера (в контексте класса Zend_Controller) является ресурсом, а операция контроллера — видом допуска. Как показано в следующем примере, можно разрешить или запретить доступ ко всему контроллеру (например, роли guest к контроллеру admin), а можно разрешить одну или две конкретные операции в пределах контроллера (скажем, ввода имени login и передачи пароля fetchpassword для роли guest).

```
$acl->allow('guest');           // дать гостям доступ ко всему ...
$acl->deny('guest', 'admin');   // ... кроме администрирования ...
$acl->deny('guest', 'account'); // ... и управления учетными записями
$acl->allow('guest', 'account', // ... но позволить им входить в систему
            array('login', 'fetchpassword'));
```

Кроме прав гостей, нужно еще определить, какие права имеют зарегистрированные пользователи. Это более высокий статус, чем гости, и поэтому прав доступа им полагается больше:

```
$acl->allow('member');          // можно заходить куда угодно ...
$acl->deny('member', 'admin'); // ... кроме раздела администрирования
                                  сайта
```

Далее определим допуски администраторов, которые имеют еще больше привилегий, чем зарегистрированные пользователи:

```
$acl->allow('administrator'); // администраторы могут заходить куда угодно!
```

После определения допусков можно посылать запрос, каковы права доступа к тому или иному ресурсу у тех или иных категорий пользователей. Вот несколько примеров:

```
// проверка допусков
$acl->isAllowed('guest', 'account');            // возвращает "ложь"
$acl->isAllowed('guest', 'account', 'login');   // "истина"
$acl->isAllowed('member', 'account');           // "истина"
$acl->isAllowed('member', 'account', 'login');  // "истина"
$acl->isAllowed('member', 'admin');             // "ложь"
$acl->isAllowed('administrator', 'admin');      // "истина"
```

Отметим, что в нашем приложении имена ролей будут определяться автоматически в зависимости от вошедшего на сайт пользователя, а ресурсы и допуски будут определяться запрашиваемым контроллером и его операцией.

На практике вызов метода isAllowed() делается в операторе if, например:

```
<?php
    if ($acl->isAllowed('member', 'account')) {
        // отобразить область сайта,
        // соответствующую учетной записи
    }
?>
```

Совет

Если попытаться проверить допуски к неопределенному ресурсу, возникнет исключительная ситуация. Способ ее обработки выбирает разработчик приложения. Можно, например, автоматически отвергнуть запрос, а можно автоматически разрешить его. Еще один вариант — переключиться на другой ресурс, если запрашиваемый отсутствует. Для проверки существования ресурса используется функция has(). Тот же принцип применяется и к ролям. В нашем приложении пользователь переводится обратно в категорию гостей, если его роль не найдена (такое может случиться из-за ошибочного значения в столбце user_type таблицы users).

Наша система допусков будет практически идентичной той, которая описана в этом примере. Зарегистрированные пользователи точно так же смогут иметь доступ к ресурсу "учетная запись", а гости — нет. Администраторы же будут иметь доступ ко всем частям приложения и сайта.

Примечание

В коде используется как идентификатор `admin`, так и `administrator`. Тип пользовательской учетной записи (роль) называется `administrator`, тогда как контроллер (ресурс) называется `admin`. Другими словами, только пользователи типа `administrator` смогут иметь доступ к URL-адресу `http://phpweb20/admin`.

Совместная работа компонентов Zend_Auth, Zend_Acl и Zend_Controller_Front

Следующим шагом в разработке нашего веб-приложения будет интеграция компонентов `Zend_Auth` и `Zend_Acl`. В этом разделе мы изменим поведение контроллера приложения (экземпляра `Zend_Controller_Front`) так, чтобы он проверял допуски посредством компонента `Zend_Acl` до отправки пользовательского запроса. В ходе проверки допусков для определения роли текущего пользователя будет использоваться идентификационная информация, сохраненная с помощью компонента `Zend_Auth`.

Для целей управления допусками будем считать каждый контроллер ресурсом, а обработчики операций и событий в этих контроллерах — видами допусков, ассоциированными с ресурсами. Например, позже в этой главе мы создадим файл `AccountController.php`, с помощью которого будет контролироваться все связанное с учетными записями пользователей (вход на сайт, выход с него, передача паролей, модификация персональных данных). Ресурсом для `Zend_Acl` является контроллер `AccountController`, тогда как привилегии, ассоциированные с этим ресурсом, — это упомянутые выше операции (вход, выход, передача пароля, модификация данных).

Примечание

Существует много способов построения системы допусков. В этом приложении мы просто управляем доступом к обработчикам событий в файлах контроллеров. Это относительно простой подход, поскольку все проверки по спискам управления доступом можно автоматизировать в соответствии с именами операций/событий и контроллеров в запросе пользователя.

Способ, которым достигается такая структура управления допусками через имена контроллеров и операций, состоит в том, чтобы написать подключаемый модуль для `Zend_Controller` (расширив класс `Zend_Controller_Plugin_Abstract`). В этом модуле определяется метод `preDispatch()`, который принимает пользовательский запрос до того, как центральный контроллер отправляет запрос соответствующему обработчику. Фактически мы перехватываем запрос и проверяем, достаточно ли у пользователя привилегий на выполнение той или иной операции.

Для регистрации модуля в классе `Zend_Controller` вызывается метод `registerPlugin()` нашего экземпляра класса `Zend_Controller_Front`. Но прежде чем делать это, создадим подключаемый модуль, который назовем `CustomControllerAclManager`. В этом классе мы создадим все роли и ресурсы для `Zend_Acl`, а также организуем проверку допусков.

В листинге 3.5 показано содержимое файла `CustomControllerAclManager.php`, который помещен в каталог `/var/www/phpweb20/include`.

Листинг 3.5. Подключаемый модуль `CustomControllerAclManager`, проверяющий допуски перед отправкой запроса на обработку (файл `CustomControllerAclManager.php`)

```
<?php
    class CustomControllerAclManager extends
Zend_Controller_Plugin_Abstract
    {
        // роль по умолчанию, если не вошел (или роль не определена)
        private $_defaultRole = 'guest';

        // выполнять эту операцию, если у пользователя не хватает
        привилегий
        private $_authController = array('controller' => 'account',
                                         'action' => 'login');

        public function __construct(Zend_Auth $auth)
        {
            $this->auth = $auth;
            $this->acl = new Zend_Acl();

            // добавляем разные роли
            $this->acl->addRole(new Zend_Acl_Role
                  ($this->_defaultRole));
            $this->acl->addRole(new Zend_Acl_Role('member'));
            $this->acl->addRole(new Zend_Acl_Role('administrator'),
                  'member');

            // добавляем контролируемые ресурсы
            $this->acl->add(new Zend_Acl_Resource('account'));
            $this->acl->add(new Zend_Acl_Resource('admin'));

            // по умолчанию даем всем пользователям доступ ко всему,
            // кроме управления учетными записями и администрирования
            $this->acl->allow();
            $this->acl->deny(null, 'account');
            $this->acl->deny(null, 'admin');

            // добавляем исключение, чтобы гости могли войти или
            // зарегистрироваться для получения привилегий
            $this->acl->allow('guest', 'account', array('login',
                              'fetchpassword', 'register',
                              'registercomplete'));

            // позволяем зарегистрированным пользователям доступ
            // к управлению учетными записями
            $this->acl->allow('member', 'account');

            // даем администраторам доступ в область администрирования
            $this->acl->allow('administrator', 'admin');
        }

        /**
         * preDispatch
         *
         * Прежде чем отправлять запрос на обработку, проверяет,
```

```
         * есть ли у пользователя нужные привилегии. Если нет,
         * инициирует операцию по умолчанию
         *
         * @param Zend_Controller_Request_Abstract $request
         */
        public function preDispatch(Zend_Controller_Request_Abstract
        $request)
        {
            // проверка, вошел ли пользователь и имеет ли нужную роль;
            // если нет, ему назначается роль по умолчанию (гость)
            if ($this->auth->hasIdentity())
                $role = $this->auth->getIdentity()->user_type;
            else
                $role = $this->_defaultRole;

            if (!$this->acl->hasRole($role))
                $role = $this->_defaultRole;

            // контролируемый ресурс - имя запрашиваемого контроллера
            $resource = $request->controller;

            // привилегия - имя запрашиваемой операции
            $privilege = $request->action;

            // если ресурс не определен явно, проверить
            // глобальные допуски по умолчанию
            if (!$this->acl->has($resource))
                $resource = null;

            // в доступе отказано - выполняется операция по умолчанию
            if (!$this->acl->isAllowed($role, $resource, $privilege)) {
                $request->setControllerName(
                          $this->_authController['controller']);
                $request->setActionName($this->_authController
                         ['action']);
            }
        }
    }
?>
```

Роли, ресурсы и привилегии определяются в конструкторе класса. В листинге 3.5 роль `administrator` вначале наследует роли `member`. Это означает, что все права и допуски зарегистрированных пользователей сайта даются также и администраторам. Кроме них, мы можем дать администраторам и дополнительные привилегии их ролей для доступа к средствам администрирования сайта.

Далее мы устанавливаем допуски по умолчанию, которые автоматически даются всем ролям. Такой допуск позволяет доступ ко всем ресурсам, кроме `account` и `admin`. Очевидно, гостю нужно иметь возможность войти на сайт под своим именем и паролем и стать привилегированным пользователем, поэтому ему надо открыть доступ к привилегиям `login` (ввод пользовательского имени) и `fetchpassword` (передача пароля). Если же гость еще не зарегистрирован, ему надо дать возможность это сделать, открыв доступ к операциям `register` (регистрация) и `registercomplete` (вспомогательная операция подтверждения регистрации).

Как только гость вошел на сайт под именем и паролем, он стал либо зарегистрированным пользователем, либо администратором, и должен иметь право доступа к ресурсу `account` (учетная запись). Поскольку роль `administrator` наследует при-

вилегии от роли member, выдача допуска к этому ресурсу зарегистрированным пользователям автоматически дает такой же допуск и администраторам.

Наконец, мы даем администраторам доступ к административной части сайта. Кроме них, никто не сможет пользоваться соответствующими средствами — ни гости, ни обычные зарегистрированные пользователи.

Рассмотрим метод preDispatch(), который принимает в качестве аргумента пользовательский запрос. Прежде всего мы настраиваем роли и ресурсы, чтобы проверка по спискам управления доступом проходила нормально. Если ресурс не найден, переменная $resource устанавливается равной null, и, следовательно, данной роли будет автоматически дан допуск по умолчанию. Учитывая выбранный нами уровень допуска (т.е. по умолчанию ко всему), такая проверка фактически возвратит значение "истина". Если такая роль не найдена в списке, будет использоваться роль guest (гость).

Примечание

Здесь мы обращаемся к свойству user_type (тип пользователя) идентификационных данных, сохраненных с помощью компонента Zend_Auth. Вопрос сохранения этой информации в составе идентификационных данных при входе пользователя на сайт пока не рассматривался; об этом пойдет речь в главе 4, где операция входа будет реализована в нашем контроллере учетных записей.

Наконец, вызывается метод isAllowed(), который определяет, имеет ли роль $role доступ к привилегии $privilege на ресурсе $resource. Если он возвращает значение "истина", мы ничего не делаем и позволяем центральному контроллеру продолжать свой рабочий цикл распределения запросов. Если же возвращается значение "ложь", запрос перенаправляется так, что в итоге выполняется операция login (вход) контроллера учетных записей. Другими словами, если посетитель пытается сделать что-то, что ему как гостю не разрешено, его перенаправляют на страницу входа на сайт для ввода своих учетных данных — имени и пароля.

Примечание

У этого способа есть один побочный эффект: если зарегистрированный посетитель пытается попасть в административную зону сайта, его перенаправляют на страницу входа, хотя он уже вошел в систему. В качестве упражнения можно так изменить код, чтобы в объекте $auth разыскивались идентификационные данные посетителя, и если их там нет, то он перенаправлялся бы на страницу входа, а если есть (т.е. если пользователь уже вошел, но не имеет достаточного уровня допуска) — на какую-нибудь другую страницу.

Управление списком пользователей через класс DatabaseObject

Класс DatabaseObject был разработан мною несколько лет назад, и я активно пользуюсь им практически во всех своих PHP-проектах. Он служит дополнительным уровнем надстройки над соединением с базой данных, сильно упрощая задачи чтения, записи и удаления строк базы данных. Файл DatabaseObject.php можно найти в каталоге ./include исходного кода к книге, доступного в Интернете.

По сути я расширил абстрактный класс DatabaseObject для каждой из основных таблиц приложения. Так, для работы с учетными записями пользователей нашего веб-приложения, хранящимися в таблице users, создается класс DatabaseObject_User. После инициализации экземпляра этого класса можно извлекать записи из базы данных методом load(), добавлять или обновлять записи

в базе методом `save()` (в зависимости от того, были ли там уже такие записи), удалять записи методом `delete()`.

Примечание

Когда я занимался разработкой класса `DatabaseObject`, еще не вышли в свет ни PHP 5, ни библиотека Zend Framework. С тех пор я доработал его в расчете на использование PHP 5 и компонента `Zend_Db`. Если использовать `Zend_Db` не планируется, то придется внести соответствующие изменения.

Не вдаваясь в подробности реализации, рассмотрим имеющиеся в классе функции и практические приемы работы с объектами `DatabaseObject`.

- `load()`. Загружает запись из базы данных, выполняя запрос `select`. Возвращает значение `true` ("истина"), если запись загружена успешно.
- `isSaved()`. Возвращает значение "истина", если запись ранее загружалась методом `load()`.
- `save()`. Сохраняет текущую запись данных в базе. Если запись ранее не загружалась, используется оператор `insert`, в противном случае она обновляется с использованием оператора SQL `update`.
- `delete()`. Если запись была загружена ранее, эта функция выполняет запрос SQL `delete`.
- `getID()`. Извлекает из базы внутренний идентификатор сохраненной записи.

Есть также ряд функций уведомления (*callback*), которые можно определить самому. Они вызываются автоматически при возникновении определенных условий.

- `postLoad()`. Вызывается после успешной загрузки записи. При необходимости ее можно использовать для загрузки данных из других таблиц.
- `preInsert()`. Вызывается перед добавлением новой записи (в этом случае функция `save()` отличает добавление от обновления). Ее можно использовать для динамического задания значений, например вставки временной метки, фиксирующей дату добавления.
- `postInsert()`. Вызывается после сохранения новой записи. В случае нашей таблицы `users` эта функция будет использоваться для отправки почтового сообщения новому зарегистрированному пользователю.
- `preUpdate()`. Вызывается перед обновлением существующей записи. Ее можно использовать для динамического задания значений, например временной метки с датой обновления.
- `postUpdate()`. Вызывается после обновления существующей записи.
- `preDelete()`. Вызывается перед удалением существующей записи. Если с этой записью связаны данные в других таблицах, которые тоже нужно будет удалить, это можно сделать здесь.
- `postDelete()`. Вызывается после удаления существующей записи. Если необходимо удалить из файловой системы приложения файл, связанный с этой записью, то удобно сделать это здесь.

Все функции уведомления, кроме `postLoad()`, должны возвращать значение `true` (истина) или `false` (ложь). Если возвращается `false`, то вся транзакция отменяется. Например, если функция `postDelete()` возвращает `false`, то запись не удаляется и отменяются любые запросы, выполненные в функции `preDelete()`. При реализации всех этих функций очень важно правильно возвращать значение.

Примечание

Все таблицы, операции с которыми выполняются через объект `DatabaseObject`, должны иметь сходную структуру в силу некоторых особенностей этого объекта. Так, таблица должна иметь одно поле первичного ключа с автоприращением в нем. Таблица `users`, созданная ранее в этой главе, следует этому образцу, поскольку в ней поле `user_id` объявлено как `serial`. Таблица `users_profile` не следует этому образцу, поэтому и работать с ней придется несколько по-другому.

Класс DatabaseObject_User

Вкратце ознакомившись с объектом `DatabaseObject`, займемся созданием дочернего класса для управления записями в таблице `users`, а создав — изучим, как с ним работать.

Для создания класса на самом деле достаточно задать имя таблицы базы данных и имя ее поля первичного ключа, а затем определить список полей (столбцов) таблицы. При необходимости можно задать также типы столбцов, чтобы объект `DatabaseObject` работал с данными соответственно. На этом этапе единственным явно заданным типом будет `DatabaseObject::TYPE_TIMESTAMP`.

В листинге 3.6 приведено содержимое файла `User.php`, который следует поместить в каталог `DatabaseObject` (полный путь — `/var/www/phpweb20/include/-DatabaseObject`). Следует заметить, что файл назван так, чтобы автозагрузчик библиотеки Zend Framework автоматически подключил этот код в случае необходимости.

Листинг 3.6. Первоначальная версия класса `DatabaseObject_User` (файл `User.php`)

```
<?php
    class DatabaseObject_User extends DatabaseObject
    {
        public function __construct($db)
        {
            parent::__construct($db, 'users', 'user_id');
            $this->add('username');
            $this->add('password');
            $this->add('user_type', 'member');
            $this->add('ts_created', time(), self::TYPE_TIMESTAMP);
            $this->add('ts_last_login', null, self::TYPE_TIMESTAMP);
        }
    }
?>
```

В листинге 3.6 вначале вызывается конструктор родительского класса. Этот метод принимает в качестве первого аргумента соединение с базой данных (экземпляр класса `Zend_Db_Adapter`), второго — имя таблицы базы данных, третьего — имя столбца первичного ключа в этой таблице.

Затем мы добавляем список полей с помощью функции `add()`. Первый ее аргумент — имя поля, второй (если есть) — значение поля по умолчанию, третий — тип поля. Если тип не указан, значение воспринимается в том виде, в каком записано.

В листинге можно видеть, что поля `ts_created` и `ts_last_login` представляют собой временные метки. Полю `ts_created` присваивается текущее время, а полю `ts_last_login` — значение `null`, поскольку пользователь еще не входил в систему.

Примечание

Можно было бы сделать значение поля `ts_created` равным `null` по умолчанию, а затем динамически изменить его в функции уведомления `preInsert()`. Особой разницы тут нет, разве что за исключением случаев, когда между созданием объекта и вызовом его метода `save()` проходит очень много времени.

В поле `user_type` устанавливается значение по умолчанию `member`. Ранее в этой главе рассматривались три категории пользователей: гости, зарегистрированные пользователи и администраторы. По определению гость — это тот, у кого нет учетной записи на сайте (и соответствующей строки в таблице `users`). Поэтому значение по умолчанию — `member`, т.е. зарегистрированный пользователь.

Теперь самое время определить в коде список типов (категорий) пользователей. Код должен позволять в будущем добавлять новые типы, модифицируя всего один список (пока пренебрежем потенциальной необходимостью изменять допуски к ресурсам). Список типов пользователей можно было бы держать в таблице базы данных, но простоты ради поместим его в статический массив в классе `DatabaseObject_User`.

В дополнение к этому можно так расширить метод `__set()`, чтобы он перехватывал устанавливаемое значение и проверял его корректность.

Примечание

В PHP 5 разрешается применение *magic*-метода `__set()`, который автоматически вызывается (при условии, что он определен), когда код пытается модифицировать несуществующее свойство объекта. Класс `DatabaseObject` использует этот метод для задания значений, сохраняемых в таблице базы данных. Его можно также определить в дочернем классе `DatabaseObject_User` для изменения значений перед вызовом `__set()` родительского класса. В PHP 5 имеется также аналогичный метод `__get()`, который автоматически вызывается при попытке чтения несуществующего свойства. Этот метод также используется в классе `DatabaseObject`.

Прежде чем заняться подробным изучением кода, следует рассмотреть еще одно значение, которое мы должны перехватить и модифицировать до его записи в базу данных. Это пароль. Раньше уже упоминалось о том, что пароль хранится в базе в виде хеш-кода исходной строки по алгоритму MD5. Поэтому перед сохранением пароля в базе к нему нужно применить функцию `md5()`.

Примечание

Можно либо воспользоваться PHP-версией функции `md5()`, либо вызвать ее в SQL-запросе. Ради простоты и независимости от конкретного формата базы данных здесь используется вариант PHP.

В листинге 3.7 приведена новая версия файла `User.php`, в которой теперь определяется список типов пользователей и проверяется корректность присваиваемого типа. Там также модифицируется значение пароля, преобразуясь в хеш-код по методу MD5.

Листинг 3.7. Новая версия класса `DatabaseObject_User` с правильным заданием типа пользователя и пароля (файл `User.php`)

```
<?php
    class DatabaseObject_User extends DatabaseObject
    {
        static $userTypes = array('member' => 'Member',
                                  'administrator' => 'Administrator');
```

```
        public function __construct($db)
        {
            parent::__construct($db, 'users', 'user_id');

            $this->add('username');
            $this->add('password');
            $this->add('user_type', 'member');
            $this->add('ts_created', time(), self::TYPE_TIMESTAMP);
            $this->add('ts_last_login', null, self::TYPE_TIMESTAMP);
        }

        public function __set($name, $value)
        {
            switch ($name) {
                case 'password':
                    $value = md5($value);
                    break;

                case 'user_type':
                    if (!array_key_exists($value, self::$userTypes))
                        $value = 'member';
                    break;
            }

            return parent::__set($name, $value);
        }
    }
?>
```

Использование класса DatabaseObject_User

Итак, мы уже умеем создавать объект класса `DatabaseObject_User`; теперь займемся его использованием. В листинге 3.8 продемонстрировано типичное использование дочернего класса `DatabaseObject`: вначале задаются значения некоторых свойств, а затем вызывается метод `save()`, который выполняет операцию языка SQL по вставке записи в базу (`insert`). Далее модифицируются некоторые свойства того же объекта и снова вызывается метод `save()`, только в этот раз выполняется запрос SQL на обновление — `update`. Наконец, предпринимается попытка загрузить существующую запись и потом удалить ее из таблицы базы данных.

Листинг 3.8. Пример использования класса DatabaseObject_User (файл `listing-3-8.php`)

```
<?php
    require_once('Zend/Loader.php');
    Zend_Loader::registerAutoload();

    // соединение с базой данных
    $params = array('host'     => 'localhost',
                    'username' => 'phpweb20',
                    'password' => 'myPassword',
                    'dbname'   => 'phpweb20');

    $db = Zend_Db::factory('pdo_mysql', $params);

    // Создание нового пользователя
    $user = new DatabaseObject_User($db);
```

```
    $user->username = 'someUser';
    $user->password = 'myPassword';
    $user->save();

    // Обновление и сохранение новых данных
    $user->user_type = 'admin';
    $user->ts_last_login = time();
    $user->save();

    // Поиск пользователя по номеру (user_id) 5 и удаление
    $user2 = new DatabaseObject_User($db);
    if ($user2->load(5)) {
        $user2->delete();
    }
?>
```

Если посмотреть на таблицу `users` после выполнения этого сценария, она будет выглядеть примерно так:

```
mysql> select user_id, username, password from users;
+---------+----------+----------------------------------+
| user_id | username | password                         |
+---------+----------+----------------------------------+
|       7 | someUser | deb1536f480475f7d593219aa1afd74c |
+---------+----------+----------------------------------+
```

Управление профилями пользователей

Когда ранее в этой главе мы создали таблицу `users`, одновременно с ней была создана также таблица `users_profile`, предназначенная для хранения данных пользовательских профилей. Ввиду особенностей структуры этой таблицы в нее можно добавлять сколько угодно элементов данных, соответствующих одному профилю пользователя. Среди них могут быть как персональные данные, такие как имя или адрес электронной почты, так и любая другая информация — например, хочет ли пользователь получать ежемесячную рассылку новостей.

Такая система используется в большинстве систем, над которыми я работаю, поэтому я написал класс `Profile` для управления такого рода данными. Это абстрактный класс, который следует расширить для каждой конкретной таблицы, в которую нужно записывать данные. Мы создадим класс `Profile_User`, расширяющий класс `Profile`.

Работа с профилем обычно состоит из нескольких шагов.

1. Создать новый экземпляр класса `Profile_User`. Один экземпляр отвечает за данные профиля одного пользователя.
2. Выбрать идентификационный номер пользователя и загрузить существующие данные профиля этого пользователя.
3. Задать новые значения, модифицировать имеющиеся, удалить ненужные (по необходимости).
4. Сохранить данные профиля.

Чтобы автоматически загружать эти классы через `Zend_Loader`, файл `Profile.php` следует поместить в каталог `./include`, а файл `User.php` (содержащий класс `Profile_User.php`) — в каталог `./include/Profile`.

В классе Profile_User не нужно реализовать никаких методов. Достаточно указать таблицу базы данных, используемую для хранения данных профилей. Кроме того, необходимо добавить один вспомогательный метод для установки идентификационного номера пользователя.

Поскольку данные профилей всех пользователей помещаются в одной таблице, в родительский класс Profile необходимо добавить *фильтр* для корректного считывания и записи данных профиля.

В листинге 3.9 показано содержимое файла User.php, где определяется дочерний класс Profile_User.

Листинг 3.9. Дочерний класс Profile_User, используемый для инициализации системы управления профилями пользователей (файл User.php)

```
<?php
    class Profile_User extends Profile
    {
        public function __construct($db, $user_id = null)
        {
            parent::__construct($db, 'users_profile');

            if ($user_id > 0)
                $this->setUserId($user_id);
        }

        public function setUserId($user_id)
        {
            $filters = array('user_id' => (int) $user_id);
            $this->_filters = $filters;
        }
    }
?>
```

Для создания и инициализации экземпляра класса Profile_User необходимо передать соединение с базой данных, а также необязательный идентификационный номер пользователя. Если идентификатор сразу не указывается, можно затем вызвать метод setUserId(). Как только этот номер будет установлен, можно вызывать метод load() для загрузки данных профиля, имеющихся в базе.

Примечание

Вызов метода setUserId() необходимо выполнить раньше, чем вызов load() или save(). В противном случае данные могут сохраниться неверно или произойдет другая ошибка.

Использование класса Profile_User

Выше мы проанализировали код класса Profile_User, а теперь рассмотрим пример его практического использования. Для этого предположим, что в таблице users уже создана учетная запись пользователя с идентификационным номером 1234 (вспомним, что поле user_id в структуре таблицы users_profile является внешним ключом к таблице users, так что соответствующая запись должна существовать).

Прежде всего нужно создать экземпляр класса и загрузить данные:

```
<?php
    $profile = new Profile_User($db, 1234);
```

```
    $profile->load();
?>
```

Вместо передачи номера в конструктор можно вызвать метод `setUserId()`. Именно так мы будем поступать в ходе интегрирования класса `Profile_User` с `DatabaseObject_User`.

```
$profile = new Profile_User($db);
$profile->setUserId(1234);
$profile->load();
```

Теперь можно задать новое значение какого-нибудь параметра профиля (или модифицировать существующее) простым обращением к свойству объекта, например:

```
$profile->email = 'user@example.com';
```

Можно удалить параметр профиля, вызвав метод `unset()`:

```
unset($profile->email);
```

А с помощью метода `isset()` можно проверить, существует ли тот или иной параметр профиля:

```
if (isset($profile->email)) {
    // какие-то операции
}
```

И наконец, вызвав метод `save()`, сохраним любые изменения, которые мы вносим в базу данных:

```
$profile->save();
```

В листинге 3.10 приведен более полный пример использования класса `Profile_User`, в который включено создание базы данных.

Листинг 3.10. Пример с заполнением данных профиля и выводом простого сообщения (файл `listing-3-10.php`)

```
<?php
    require_once('Zend/Loader.php');
    Zend_Loader::registerAutoload();

    // соединение с базой данных
    $params = array('host'     => 'localhost',
                    'username' => 'phpweb20',
                    'password' => 'myPassword',
                    'dbname'   => 'phpweb20');

    $db = Zend_Db::factory('pdo_mysql', $params);

    $profile = new Profile_User($db);
    $profile->setUserId(1234);
    $profile->load();

    $profile->email = 'user@example.com';
    $profile->country = 'Australia';
    $profile->save();

    if (isset($profile->country))
        echo sprintf('Your country is %s', $profile->country);
?>
```

Если проверить данные в таблице users_profile после выполнения этого примера, получим примерно следующее:

```
mysql> select * from users_profile where user_id = 1234;
+---------+-------------+-------------------+
| user_id | profile_key | profile_value     |
+---------+-------------+-------------------+
| 1234    | country     | Australia         |
| 1234    | email       | user@example.com  |
+---------+-------------+-------------------+
2 rows in set (0.00 sec)
```

Интегрирование класса Profile_User с классом DatabaseObject_User

Итак, у нас есть средство управления профилями пользователей. Его необходимо интегрировать с классом DatabaseObject_User, чтобы можно было централизованно управлять всеми данными пользователей из одного места. Для этого нужно сделать следующее:

- создать экземпляр класса Profile_User в классе DatabaseObject_User;
- автоматически загрузить данные профиля, когда загружается учетная запись пользователя;
- автоматически сохранить данные профиля, когда записывается учетная запись пользователя;
- автоматически удалить данные профиля, когда удаляется учетная запись.

Кроме того, идентификационный номер пользователя еще неизвестен при создании новой учетной записи объектом DatabaseObject_User, и эту проблему надо как-то решать. Для этой цели годятся функции уведомления, имеющиеся в классе DatabaseObject. Как они будут использоваться, описано ниже.

- В функции уведомления о загрузке (postLoad()) установим идентификационный номер и загрузим данные профиля.
- Перед добавлением записи в базу (preInsert()) сгенерируем для пользователя пароль. Пока что для генерирования пароля будет применяться функция PHP uniqid(), но в главе 4 этот подход будет усовершенствован, когда понадобится рассылать новым пользователям почтовые сообщения.
- После добавления записи в базу данных (postInsert()) установим идентификационный номер и сохраним данные из профиля.
- После обновления записи (postUpdate()) сохраним данные профиля — в этот момент идентификационный номер пользователя уже известен.
- Перед удалением записи (preDelete()) удалим все данные профиля. Следует отметить, что это обязательно нужно проделать до удаления самой учетной записи (т.е. не в функции postDelete()), потому что иначе будут нарушены связи по внешнему ключу, — таблица users_profile содержит зависимость от таблицы users, поэтому нельзя удалять из таблицы users те данные, на которые есть ссылка в users_profile.

В листинге 3.11 приведена новая версия класса DatabaseObject_User, в которой определяются все функции уведомления. Важно, что функции postInsert() и postUpdate() тоже возвращают значение true, — это необходимо для завершения транзакции базы данных.

Листинг 3.11. Класс `DatabaseObject_User` со средствами управления профилями пользователей (файл `User.php`)

```
<?php
    class DatabaseObject_User extends DatabaseObject
    {
        static $userTypes = array('member'        => 'Member',
                                  'administrator' => 'Administrator');

        public $profile = null;

        public function __construct($db)
        {
            parent::__construct($db, 'users', 'user_id');

            $this->add('username');
            $this->add('password');
            $this->add('user_type', 'member');
            $this->add('ts_created', time(), self::TYPE_TIMESTAMP);
            $this->add('ts_last_login', null, self::TYPE_TIMESTAMP);

            $this->profile = new Profile_User($db);
        }

        protected function preInsert()
        {
            $this->password = uniqid();
            return true;
        }

        protected function postLoad()
        {
            $this->profile->setUserId($this->getId());
            $this->profile->load();
        }

        protected function postInsert()
        {
            $this->profile->setUserId($this->getId());
            $this->profile->save(false);
            return true;
        }

        protected function postUpdate()
        {
            $this->profile->save(false);
            return true;
        }

        protected function preDelete()
        {
            $this->profile->delete();
            return true;
        }

        public function __set($name, $value)
```

```
        {
            switch ($name) {
                case 'password':
                    $value = md5($value);
                    break;

                case 'user_type':
                    if (!array_key_exists($value, self::$userTypes))
                        $value = 'member';
                    break;
            }

            return parent::__set($name, $value);
        }
    }
?>
```

Кроме определений функций уведомления, в конструкторе еще создается объект `Profile_User`. Поскольку в `DatabaseObject` используются функции-перегрузчики PHP 5 `__set()` и `__get()`, необходимо включить также в определение класса свойство `$profile`.

Внимание!

При вызове метода `save()` из профиля ему в качестве аргумента передается `false`, что не позволяет классу `Profile` сохранить данные посредством транзакции. Так необходимо поступить, потому что `DatabaseObject` уже инициировал транзакцию и в нее включено сохранение данных профиля. Другими словами, если возвратить `false` из функции `postUpdate()`, то транзакция будет отменена, изменения в таблице данных пользователя не будут сохранены и данные профиля в базе данных останутся неизменными.

После добавления всех этих новых возможностей в класс `DatabaseObject_User` мы можем легко манипулировать всеми данными о пользователях. В листинге 3.12 показан пример создания новой учетной записи пользователя и заполнения данных профиля.

Листинг 3.12. Создание новой учетной записи пользователя и заполнение данных профиля (файл `listing-3-12.php`)

```
<?php
    require_once('Zend/Loader.php');
    Zend_Loader::registerAutoload();

    // соединение с базой данных
    $params = array('host'     => 'localhost',
                    'username' => 'phpweb20',
                    'password' => 'myPassword',
                    'dbname'   => 'phpweb20');
    $db = Zend_Db::factory('pdo_mysql', $params);

    // Создание учетной записи нового пользователя
    $user = new DatabaseObject_User($db);
    $user->username = 'someUser';
    $user->password = 'myPassword';

    // Заполнение данных профиля
    $user->profile->email = 'user@example.com';
    $user->profile->country = 'Australia';
```

```
    // Сохранение учетной записи и профиля
    $user->save();

    // Загрузка и удаление другой учетной записи
    $user2 = new DatabaseObject_User($db);
    if ($user2->load(1234))
        $user2->delete();
?>
```

Резюме

В этой главе была создана инфраструктура управления и учета пользователей в нашем веб-приложении. Сначала мы рассмотрели компоненты `Zend_Auth` и `Zend_Acl` из библиотеки Zend Framework. По ходу дела мы обсудили разницу между аутентификацией и авторизацией и выясним, чем эта разница важна для нашего приложения.

Далее оба этих компонента были интегрированы с классом `Zend_Controller_Front`, чтобы ограничить права доступа к нашему приложению на уровне контроллеров и их операций. После этого настала очередь манипулирования базой данных с помощью классов `DatabaseObject` и `Profile`, расширение которых позволило создать систему управления пользовательскими учетными записями.

В следующей главе система работы с пользователями нашего приложения будет дорабатываться и совершенствоваться. Пользователи получат возможность создавать новые учетные записи, входить на сайт под именем и паролем, модифицировать данные своих профилей, и все это благодаря коду, разработанному в этой главе.