1. Документация для разработчиков

Содержание

Документация для разработчиков 1

От разработчика 1

Основные особенности и возможности 2

Пользователи, группы пользователей, права пользователей 3

ЧПУ, страница сайта 4

Файловая система 5

Структура СУБД 6

Как обрабатывается каждый запрос 6

Установка движка 7

Главный конфигурационный файл 8

Основные API-функции 8

MVC общедоступной части сайта 11

Контроллер 11

Представление 14

Модель 14

События (хуки) 15

Секции и виджеты 15

Админка 16

Типичный цикл обработки запроса администратора 17

Общая информация о правах доступа и организаци 18

Основные API-функции 19

Элементы управления админкой 19

Виджеты 20

Кнопки административного интерфейса (фронтенд) 20

Бакэнд 21

Меню 22

Особенности программирования админки 23

Шаблоны, информация для верстальщика 24

Класс HTML-формы (form) 25

Публичные атрибуты 26

Методы класса 26

Универсальная модель базы данных (model) 28

Методы класса 29

Атрибуты и методы для статического использования (перегрузка класса) 30

Правила валидации 31

Класс универсальной таблицы (table) 32

Методы класса 33

Работа с электронной почтой (класс email) 34

Методы класса 34

Работа с изображениями (класс picture) 35

Методы класса 35

1. От разработчика

Движок с условным названием «Ё» создавался сугубо для личных целей и поэтому его развитие велось и ведётся по мере необходимости. Однако со временем я понял, что эта CMS стала достаточно самостоятельной и может оказаться полезной не только мне...

«Ё» не является конструктором сайтов вроде Joomla или WordPress, он разрабатывался как платформа для действительно эффективной разработки сложных или нестандартных проектов. Однако, спустя время стало очевидно, что движок прекрасно подходит и для типичных сайтов и визиток. Но всё же без знания хотя бы основ PHP и вёрстки «выжать» что-то стоящее из этого движка сложно.

С другой стороны его использование оказалось весьма эффективным за счёт того, что в нём заложен практичный фреймворк, предоставляющий инструменты для скорейшего единообразного решения большинства рутинных задач. В то же время движок не накладывает значительных ограничений на свободу действий программиста, поэтому ситуации, когда сам фреймворк становится преградой для решения тех или иных задач, очень редки.

Например, тут нет «обёрток» для механизма сессий, cookie, глобальных массивов \_GET и \_POST: эти конструкции сами по себе работают хорошо, поэтому дополнительные классы только усложнят жизнь программиста, не привнеся ничего нового.

При проектировании движка не ставилось целью построить супер систему на все мыслимые случаи жизни — этот движок (можно воспринимать и как фреймворк), предназначен исключительно для конструирования сайтов. Поэтому в нём нет и не будет поддержки всевозможных СУБД — есть только MySQL и SQLite. И правда: хоть я и встречал такие нелепицы, но создавать PHP-сайт на основе СУБД MS SQL — это безумие!

Движок проектировался с таким расчётом, чтобы обеспечить максимальную простоту доработки базового сайта и создания новых модулей, но при этом не перегружать его излишними редко используемыми инструментами. Если придерживаться принятых правил, то можно утверждать, что программист может сосредоточиться исключительно на логике, уделяя минимальное внимание рутинным задачам.

Важно отметить уникальность реализации административного интерфейса — проектируя его в том виде, в котором он есть сейчас, я не был уверен (и сейчас не до конца уверен) в том, что будет достаточно удобно, в особенности тогда, когда на сайте накопится огромное количество данных. Поэтому проектирование заранее велось таким образом, чтобы оставалась возможность реализовать классический backend, не переписывая при этом добрую половину кода. К примеру все окна админки на самом деле открываются во фреймах, а значит возможна реализация страниц в классическом виде (даже предусмотрен шаблон — его элементы можно увидеть если исправить в ссылке «index2.php» на «index.php»

Высокая производительность — важное условие для жизни данного движка. Специальные замеры не проводились и есть только такие данные: опыт использования «обычного» хостинга из недорогих (без рекламы), на котором ежедневно проходит 25-30 тысяч запросов. С нагрузкой хостинг справляется на ура и сайт работает очень шустро.

1. Основные особенности и возможности

Требования: PHP 5.3 или выше.

Возможности:

- уникальный дружественный административный интерфейс;

- полностью открытый исходный код;

- ЧПУ (является неотъемлемой частью, отключить нельзя);

- полноценный фреймворк, имеющий MVC-архитектуру (имеет много особенностей);

- прозрачная поддержка СУБД MySQL и SQLite (других нет и не будет);

- уникальная система подмены пользователей (администратор может на время в глазах системы становиться любым другим пользователем - на практике это часто облегчает написание админки);

- широкая возможность модификации URL страниц (любой странице можно задать индивидуальный URL-адрес);

- метатеги для всех значимых для поисковых систем страниц сайта (например страница авторизации и восстановления пароля не являются значимыми);

- прозрачная поддержка LastModified;

- Поддержка XML-карты сайта, возможность настройки для дополнительных модулей;

- поддержка хуков (реакция системы на те или иные события, например, поиск по сайту);

- автоматизация большинства рутинных операций (в основном касается административного интерфейса);

- поддержка групп пользователей, разделение прав администраторов;

- кешированние, в том числе кеширование произвольных данных;

- отдельный шаблон для pda-версии;

- возможность использовать индивидуальные шаблоны для разных страниц сайта;

- модульность, расширяемость (используется главным образом для удаления неиспользуемых модулей);

- поддержка секций и виджетов (некоторый аналог модулей в joomla), тонкая настройка страниц, на которых должен быть опубликован виджет;

- прозрачное кеширование виджетов;

Внимание! Во многих аспектах различные переменные, напрямую влияющие на поведение движка, доступны для изменения. Это даёт программисту б**о**льшую свободу действий. Например в начальных стадиях обработки запроса можно изменить переменную $\_GET['corePath'] — она отвечает за то, какая страница будет отображена и какие виджеты будут опубликованы на странице.

Однако зачастую нужно быть предельно внимательным, например такой код:

$u=core::user();  
$u->group=255;

наделит текущего пользователя правами суперпользователя на протяжении всего времени пребывания на сайте (core::user() возвращает ссылку на объект, находящийся в сессии).

1. Пользователи, группы пользователей, права пользователей

Каждый пользователь, запросивший ту или иную страницу сайта, относится к одной из 256 групп. Смысл групп следующий:

0 - не авторизованный посетитель;

1-199 - авторизованный пользователь, не обладающий правами администрирования;

200-254 - авторизованный пользователь, наделённый правами администрирования;

255 - суперпользователь, ему можно всё.

Номера групп весьма условны, однако рекомендуется придерживаться практики, при которой чем больше номер группы пользователя, тем больше у него прав.

При регистрации пользователю назначается группа 1.

Другие группы (2-199) должны использоваться в тех случаях, когда на сайте необходимо организовать разграничение пользователей, например могут быть такие группы как "клиент", "оптовый клиент", "поставщик", "сбор статистики" и т.д.

Пользователи, относящиеся к группам 200-254 наделяются правами администрирования. Для каждой группы конкретные права назначаются индивидуально, однако, если на сайт вошёл пользователь с группой больше 199, то будут автоматически подключены модули, управляющие административным интерфейсом.

Особая группа 255 - это суперпользователи, которым доступны все функции сайта без исключения. Для этих пользователей права доступа к тем или иным модулям не проверяются вообще.

Следует заметить, что разделение прав администраторов не предполагает серьёзной защиты от несанкционированного доступа, это лишь инструмент, позволяющий облегчить интерфейс администратора, исключив лишние элементы управления и тем самым уменьшив вероятность человеческой ошибки (по отзывам клиентов это оказалось весьма удачным).

Движок также реализовывает механизм «подмены пользователей», при котором администратор может на время «переключиться» на любого другого пользователя и система будет воспринимать его как пользователя, на которого переключился администратор. На практике такой механизм часто позволяет существенно облегчить административный интерфейс за счёт того, что просто дополняется механизм личного кабинета, а не дублируется управление теми или иными элементами в админке.

1. ЧПУ, страница сайта

Вероятно любой движок должен иметь возможность для каждой страницы сайта где-то хранить сопутствующую информацию, такую, например, как идентификатор записи в базе данных, мета теги страницы, информацию о том, какие дополнительные блоки публиковать на этой странице, какой пункт меню является активным, какой шаблон использовать для данной страницы и т.д.

В данном движке вся дополнительная информация закладывается в ***адрес страницы***. Адрес страницы — это, грубо говоря, относительный URL страницы сайта.

Например, если есть такая страница: <http://example.com/article/blog/news?page=2>, то адресом страницы будет «article/blog/news». Приведённый пример — это типичная ссылка на одну из страниц сайта. ЧПУ является неотъемлемой частью движка и его выключение не предусмотрено.

***Адрес страницы*** должен состоять по крайней мере из двух элементов (artcle и blog) — это имя MVC-контроллера и имя *действия*. Если *действие* не задано (пример: <http://example.com/article>), то в качестве имени действия принимается «Index».

Имя контроллера — это файл, находящийся в директории /controller, а *действие* — это имя процедуры класса этого контроллера. Таким образом, в случае приведённой выше ссылки, движок загружает файл /controller/article.php, затем вызывает процедуру actionBlog(). Дальше эта процедура разбирает остальные части URL самостоятельно.

Именно ***адрес страницы*** (строка «article/blog/news») используется движком для того, чтобы определить какие дополнительные блоки (*виджеты* — аналог модулей в Joomla) должны быть опубликованы на данной странице. Таким образом никакого обособленного хранилища сопутствующей каждой странице информации нет.

Часть ссылки после знака вопроса не является ***адресом страницы*** и не должна влиять на то, какая страница будет отдана сервером. Это лишь дополнительные параметры отображения — их может быть сколько угодно много (в данном примере указывается вторая страница пагинации).

Для целей ajax-запросов используется особый формат адресов страниц: <http://example.com/index2.php?controller=article&action=blog&id=news&page=2> для них, разумеется, нет надобности тратить ресурсы сервера для преобразований ЧПУ ссылок. В корне сайта лежат два файла — index.php (все ЧПУ ссылки сводятся к нему) и index2.php. Их отличие состоит в том, что при обращении к index2.php, в отличии от index.php, шаблон (а также *виджеты*) обрабатываться не будет и сервером будет отдан только основной контент.

1. **Файловая система**

Структура директориев строго разграничена и файлы каждого модуля находятся в соответствующийх папках, единой папки для отдельного модуля нет.

/admin - административная часть сайта  
/admin/cache - содержи файлы кеша админки  
/admin/config - конфигурационные файлы, относящиеся исключительно к админке  
/admin/controller — MVC-контроллеры  
/admin/core - ядро админки  
/admin/data - данные, недоступные из веб  
/admin/hook - обработчики событий  
/admin/model - MVC-модели, а также другие библиотеки  
/admin/module - содержит информацию об устанолвенных модулях  
/admin/public - ресурсы админки - это единственный директорий, доступный из веб  
/admin/template - шаблоны админки  
/admin/view - MVC-представления

/cache - содержит файлы кеша  
/cache/custom - файлы пользовательского кеша  
/cache/template - кеш шаблонов  
/cache/widget - кеш виджетов  
/config - содержит конфигурационные файлы  
/config/\_core.php - основной конфигурационный файл  
/config/\_hook.php - список обработчиков событий  
/controller — MVC-контроллеры  
/core - файлы ядра движка  
/data - различные данные, недоступные из Веб  
/data/emial - содержит шаблоны электронных писем  
/data/widgetHTML - виджеты "произвольный текст"  
/hook - обработчики событиый  
/model - MVC-модели, а также другие библиотеки  
/public - ресурсы сайта (изображения, стили, видео и т.д.) - это единственный директорий, доступный из веб  
/public/css - директорий для каскадных таблиц стилей  
/public/js - директорий для JavaScript  
/public/template - ресурсы шаблонов, в том числе и .css-файлы  
/sitemap - содержит конструкторы xml-карт сайта для каждого из модулей  
/template - шаблоны сайта  
/template/pc.default.html - основной шаблон сайта  
/template/pda.default.html - шаблон сайта для pda-версий  
/tmp - директорий для временных файлов, из него производится установка и выгрузка модулей  
/view - MVC-представления, в том числе представления виджетов  
/widget — виджеты  
/captcha.php - генератор каптчи

1. **Структура СУБД**

Тут представлены только основные таблицы базы данных.

menu - список меню сайта

menuItem - пункты меню: menuId - menu.id, link - относительная ссылка на страницу, typeId - menuType.id

menuType - типы меню (фактически содержит список поддерживаемых движком типов страниц): controller — имя контроллера админки, action - имя действия админки, эти поля служат для указания файла и имени метода, который будет обрабатывать запросы на создание и изменение пунктов меню

modified - содержит дату последней модификации страниц, служит для поддержки тега LastModified

right - права пользователей (администрраторов): module - системное название модуля, group - группы пользователей, которым предоставлен доступ

section - определяет страницы, на которых должен выводиться тот или иной виджет: widgetId - widget.id

user - содержи список пользователей: group - группа пользователей

userGroup - список используемых на сайте групп пользователей

widget - настройки виджетов: data - данные виджета (параметры), cache - период кеширования (в минутах)

widgetType - список всех поддерживаемых движком типов

виджетов: controller - имя контроллера, action - имя действия. Эти поля идентифицируют файл и имя метода, который необходимо вызвать при созадании или изменении виджета.

1. **Как обрабатывается каждый запрос**

* В упрощенном виде цикл обработки запроса таков...
* Подключаются скрипты, которые перечислены в параметре «autoInclude» глобальной конфигурации (файл */config/\_core.php*).
* Обрабатывается полученный запрос, из него выделяется ***адрес страницы*** и он помещается в переменную $\_GET['corePath'].
* Если передан заголовок «Last Modified», то проверяется дата последнего изменения страницы, если страница не была изменена, то выполнение скрипта завершается с кодом 304.
* Инициализируется сессия.
* Если пользователь относится к группе с номером 200 или больше, то подключаются административные библиотеки.
* Создаётся экземпляр класса контроллера. Если такого файла нет, то генерируется ошибка 404.
* В конструкторе контроллера определяется правильное имя действия, а также проводятся дополнительные преобразования и проверки правильности заданного адреса, созадаётся переменная sController::$url (в ней хранится преобразованный, если в этом есть необходимость, адрес страницы).
* Если в $\_POST есть данные, относящиеся к этому контроллеру, то вызывается ***submit-действие*** этого контроллера (это процедура, предназначенная для обработки данных html-форм).
* ***Submit-действие*** может прервать дальнейшую обработку вызвав метод core::redirect (это редирект на другой адрес — если данные форм обработаны правильно, то всегда выполняется редирект).
* Далее происходит вызов ***действия*** контроллера (это процедура, подготавливающая данные для отображения страницы).
* ***Действие*** контроллера подготавливает необходимые данные, подключает каскадные таблицы стилей, скрипты javascript, устанавливает мета теги, а также заголовок страницы, затем возвращает **представление**.
* Генерируется событие «beforeRender» (вызывается перед началом вывода каких-либо данных).
* Если ***представление*** — это NULL или FALSE, то никаких данных не выводится, обработка запроса завершается.
* Если пользователь относится к группе 200 или выше, то подключаются файл стилей и файл javascript, необходимый для отображения административных диалоговых окон.
* Если мета теги не были установлены в контроллере, то сейчас они устанавливаются заданными по умолчанию.
* Если это не ajax-запрос, то:
  + Если включён режим отладки, или нет кеша шаблона, то запускается кеширование нужного шаблона.
  + Выводится первая часть html-шаблона (до того места, где должен быть вставлен контент). При этом также обрабатываются все виджеты.
  + Если в шаблоне предусмотрены «хлебные крошки», то проверяется наличие в контроллере процедуры, формирующей хлебные крошки для данного действия. Если такая процедура есть, то она вызывается и на основе полученных данных выводитс html-код хлебных крошек. Если такой процедуры нет, то хлебные крошки формируются по заданному по умолчанию алгоритму.
  + Если пользователь относится к группе пользователей с номером 200 или больше, то выводится html-код всех доступных кнопок административного интерфейса.
* Если полученное от контроллера ***представление*** — это строка, то подключается соответствующий файл из директория /view (в этом файле содержится html-код страницы).
* Если полученное от контроллера ***представление*** — это объект, то вызывается метод render() этого объекта, который должен вывести html-код страницы.
* Если это не ajax-запрос, то выводится вторая часть (после того места, где должен быть контент) шаблона.

1. Установка движка

Специальный установщик не предусмотрен, так как в нём нет никакой необходимости. Чтобы развернуть новый сайт нужно скопировать файлы сайта на сервер, загрузить дамп базы данных и внести необходимые изменения в конфигурационный файл (если используется MySQL). Также нужно удостовериться в правильно установленных правах на директории сайта — должна быть возможность писать в следующие директории:

/admin/cache  
/admin/config  
/cache  
/config  
/data  
/public  
/tmp

Конфигурационный файл находится тут: /config/\_core.php. В нём необходимо указать тип используемой базы данных (dbDriver), а также параметры подключения к MySQL (mysqlHost, mysqlUser, mysqlPassword, mysqlDatabase), если используется эта СУБД.

1. **Главный конфигурационный файл**

Он находится в файле /config/\_core.php и выглядит примерно так:

<?php return array(  
 'debug'=>true,  
 'adminEmailEmail'=>['admin@example.com](mailto:'admin@example.com)',  
 'adminEmailName'=>'CMS',  
 'dbDriver'=>'mysql',  
 'mysqlHost'=>'localhost',  
 'mysqlUser'=>'root',  
 'mysqlPassword'=>'pwd13pwd',  
 'mysqlDatabase'=>'cms0',  
 'autoInclude'=>'core/mysql;core/sqlite3;core/user.db',  
 'mainPath'=>'article/view/index',  
 'link'=>array()  
); ?>

Все параметры кроме dbDriver, mysqlHost, mysqlUser, mysqlPassword, mysqlDatabase, autoInclude настраиваются через веб-интерфейс административной части сайта. Их смысл следующий:  
dbDriver — тип базы данных, может быть «mysql» или «sqlite».  
mysqlHost, mysqlUser, mysqlPassword, mysqlDatabase — параметры подключения к базе данных MySQL.  
autoInclude — список скриптов, которые необходимо подключить в самом начале обработки запросов каких-либо страниц сайта.

Очевидно, что в большинстве случаев в параметре «autoInclude» должен быть указан только один из скриптов: «core/mysql» или «core/sqlite» - это драйверы соответствующих баз данных. Тем не менее ст**о**ит заметить, что использование сразу двух типов баз данных возможно.

1. **Основные API-функции**

string **core::path**()

Возвращает полный путь к корню сайта.

null **core::import**(string *fileName*)  
 Подключает указанный PHP-скрипт *fileName*, который является относительным адресом от корня сайта и не должен иметь ведущего слеша и расширения файла. Эта функция просто вызывает include\_once, но *fileName* преобразуется до полного адреса. Пример:  
core::import('core/email'); //подключит файл /core/email.php.

string **core::template**(string *template*);

Если *template* задан, то устанавливает используемый шаблон (соответствует файлу в директории «/template»). Тип шаблона (pc или pda) указывать не надо. Эта функция может быть вызвана в любое время до начала обработки шаблона.

Возвращает имя шаблона с типом шаблона (строка «(pc|pda).template»).

boolean **core:debug**();

Возвращает true, если сайт работает в отладочном режиме и false в противном случае.

array **core::config**([string *name*]); и array **core::configAdmin**([string *name*]);

Возвращает массив, содержащий конфигурацию с указанным именем, фактически возвращает массив из файла в директории «/config». Если name не указан, то возвращается *основная конфигурация* (\_core.php). Функция core::configAdmin() возвращает конфигурацию, относящуюся к админке (из директория /admin/config).

Каждая загруженная конфигурация остаётся в памяти до завершения отображения страницы, поэтому последующие загрузки той же конфигурации через эту функцию не приведут к паразитным нагрузкам.

object **core::user**(); и object **core::userCore**();

Возвращает экземпляр класса user (/core/user.db.php), который представляет из себя модель «пользователя». На самом деле этот экземпляр находится в сессии в переменной $\_SESSION['user']. Если пользователь авторизован с правами администратора и вошёл в режим подмены пользователя, то функция возвращает «пользователя», на которого переключился администратор.

Функция core::userCore() используется для тех случаев, когда нужно получить доступ к пользователю игнорируя режим подмены.

variant **core::cache**(*id*,*callback*,[*timeout*]);

Функция кеширует любые произвольные данные, он вызывает указанную функцию для обновления кеша, если в этом есть необходимость. Поля: id — уникальный идентификатор кешируемых данных, задаётся вызывающим скриптом; callback — функция, которая будет вызвана в случае, если кеш устарел, она должна сформировать и вернуть блок данных свежего кеша; timeout — время в секундах актуальности кешируемых данных.

integer **core::userId**();

Возвращает идентификатор текущего пользователя.

string **core::url**();

Возвращает относительный URL до корня сайта, обычно это «/».

object **core::sqlite**([newQuery]);

Возвращает экземпляр класса sqlite (/core/sqlite3.php) — это главный инструмент работы с базой данный SQLite 3. Если указан параметр newQuery, то будет открыто новое соединение (используется для тех случаев, когда нужно выполнять одновременно два запроса, например перебор записей и обновление записей).

object **core::mysql**([*newQuery*]);

Возвращает экземпляр класса mysql (/core/mysql.php) — это главный инструмент работы с базой данный MySQL. Если указан параметр newQuery, то будет открыто новое соединение (используется для тех случаев, когда нужно выполнять одновременно два запроса, например перебор записей и обновление записей).

object **core::db**([*newQuery*]);

Возвращает экземпляр класса mysql или sqlite, в зависимости от того, какая база данных используется в качестве основной (параметр dbDriver в файле /core/config/\_core.php).

integer **core::userGroup**();

Возвращает группу, к которой относится текущий пользователь.

null **core::controller**(name);

Подключает контроллер с именем name. Эта функция вызывается только один раз из файла /index.php или /index2.php.

string **core::link**(link);

Строит корректную ЧПУ-ссылку по переданной относительной ссылке link, которая должна иметь вид, подобный этой: «article/view/about». Если ссылка начинается с «http:» или «https:», то она не обрабатывается, а возвращается «как есть». Главное назначение этой функции — это выполнение преобразований ссылки, заданных пользователем (см. соответствующий раздел «user manual»). Поэтому нужно всегда «оборачивать» ссылки в эту функцию там, где может понадобиться механизм преобразования ЧПУ-ссылок.

object **core::form**([*namespace*]);

Возвращает экземпляр класса form (/core/formModel.php), позволяющего строить html-формы. Namespace — это имя контроллера, в котором будет проводиться обработка формы, если не указано, то принимается имя текущего контроллера.

object **core::model**([*namespace*],[*dbDriver*]);

Возвращает экземпляр класса model (/core/formModel.php), представляющего из себя универсальную модель. Эта модель позволяет выполнять валидацию и сохранение данных в базе данных. Namespace — это имя таблицы базы данных (не требуется, если выполнять запросы insert или update не предполагается); dbDriver — драйвер базы данных (sqlite или mysql), если не указано, то будет использоваться драйвер, заданный как основной.

null **core::redirect**(url,[*message*]);

Выполняет редирект на указанный в url адрес (дополнительно обрабатывается core::link(), не надо оборачивать самому). Если message задан, то в браузер будет выведено указанное сообщение — это поле в основном используется для сообщения посетителю сайта об успехе выполненного действия после обработки какой-либо html-формы.

После вызова этой функции дальнейшее выполнение скрипта прерывается.

null **core::error404**();

Генерирует 404-ю ошибку. После вызова этой функции дальнейшее выполнение скрипта прерывается.

boolean **core::error**([*message*]);

Выводит сообщение об ошибке, если оно (сообщение) заданно в переменной message. Используется самим движком и для разработчика представляет мало интереса.

boolean **core::success**([*message*]);

Выводи сообщение об успешно выполненной операции, если оно (сообщение) задано в параметре message или было ранее установлено при вызове core::redirect() с указанным параметром message. В основном используется сами движком и для разработчика представляет мало интереса.

null **core::widget**(*name*,[*options*],*cacheTime*,*title*,*link*);

Запускает рендер виджита с именем name (ему соответствует файл в директории /widget). Парметры: options (может быть любым типом) содержит все необходимые параметры; cacheTime — время в минутах актуальности кеша, если не указано или ноль, то виджет не кешируется; title — отображаемый на странице заголовок виджета (строка); link — в этом параметре движок передаёт шаблон ссылки (заканчивается символами «/», «.» или «\*»), согласно которой должен быть опубликован данный виджет.

Данная функция в основном вызывается самим движком при обработке секций. Однако, она вполне может быть использована в MVC-представлениях (файлы в директории /view), например, часто используется для организации пагинации (виджет pagination). Кроме того эта функция вызывается из шаблонов, при обработке конструкций в виде {{widget[name][options]}}.

null **core::section**(*name*);

Запускает рендер секции с именем name. Эта функция используется исключительно при обработке шаблонов.

string **core::script**(*script*); и string **core::scriptAdmin**(*script*);

Возвращает html-тег <script...></script>, параметр *script* — это имя файла (в основном без пути и расширения). Функция применяется для избежания подключения одного скрипта дважды: если функция для какого-либо скрипта уже вызывалась, то функция возвращает пустую строку. Функция core::scriptAdmin() подключает скрипт, относящийся к административной части сайта (директорий /admin/script).

boolean **core::hook**(*name*,[*data...*]);

Запускает обработку события с именем name. Кроме параметра name функция принимает и другие, их количество и тип индивидуальны для каждого конкретного события.

1. MVC общедоступной части сайта

В основе движка, как общедоступной части сайта, так и админки, лежит архитектура MVC, однако, она имеет много особенностей.

Модель в большинстве случаев выступает не в роли объекта, которым манипулирует контроллер, а в роли вполне самобытной библиотеки, предоставляющей инструментарий контроллеру, хотя в то же время модель может выступать даже в роле представления!

Функции контроллера, в свою очередь, не сводятся к манипуляции моделями, а в большинстве случаев реализуют всю основную логику обработки запросов. Дополнительное дробление кода на контроллер и модель представляется излишним.

В роли представления может выступать как php-скрипт, содержащий разметку HTML, так и экземпляр какого-либо класса. В последнем случае объект должен реализовывать метод render(), который будет вызван движком для генерации HTML-кода страницы. Если для общедоступной части сайта представление почти всегда имеет вид php-скрипта, то для обработки какого-либо запроса в админке очень часто достаточно вообще одного контроллера.

* 1. Контроллер

Для общедоступной части скрипты контроллера расположены в директории /controller. Имя контроллера для какого-либо HTTP-запроса определяется первой частью ЧПУ-ссылки. Например, если запрошена страница «exmaple.com/article/blog/news/131229», то контроллер — это «article», его реализация должна находится в файле /controller/article.php. Если такого файла нет, то движок сгенерирует ошибку 404.

Типичный вид контроллера:

<?php class sController extends controller {  
 function \_\_construct() {  
 parent::\_\_construct();  
 …  
 }

public function actionBlog() {  
 …  
 $this->pageTitle=$this->metaTitle='some text';  
 $this->metaKeyword=$this->metaDescription='some text';  
 $this->script('some\_script');  
 $this->style('some\_css');  
 return 'Blog';  
 }

public function actionBlogSubmit($data) { ... }

protected function breadcrumbBlog() {  
 return array( ... );  
 }

protected function adminBlogLink() {  
 return array( ... );  
 }

protected function adminBlogLink2($data) {  
 return array( ... );  
 }  
} ?>

Контроллер должен иметь имя sController и быть унаследован от controller.

Каждый HTTP-запрос в рамках контроллера проходит через четыре этапа, каждый этап — это отдельная процедура:  
- первичная обработка запроса в конструкторе контроллера;  
- обработка данных POST в ***submit-действии*** (если есть данные в POST);  
- подготовка в процедуре ***дейстия*** данных для отображения;  
- генерация хлебных крошек (если требуется);

После создания экземпляра класса контроллера движок вызовет ***действие*** — процедуру, осуществляющую обработку запроса. Имя действия берётся из второй части ЧПУ-ссылки (для приведённой выше ссылки — это «blog») с добавление слова «action» (actionBlog). Если второй части ссылки нет, то вместо неё будет подставлено действие «Index». Если процедуры с таким именем нет, то генерируется ошибка 404.

В конструкторе контроллера удобно проводить манипуляции с ЧПУ. Внутри конструктора необходимо первым делом вызывать parent::\_\_construct(), который инициализирует переменную controller::$url, записав в неё массив, состоящий из частей запрошенной ЧПУ-ссылки (это копия $\_GET['corePath'] — это переменную не рекомендуется изменять, т. к. её содержимое могут анализировать виджеты). Для ссылки «example.com/article/blog/news/131229» controller::$url будет содержать массив:  
 array('article','blog','news','131229').

После создания экземпляра класса контроллера движок вызовет действие, указанное в переменной $this->url[1]. Очевидно, что страница «example.com/article/blog/news» - это новостная лента, а «example.com/article/blog/news/131229» - это одна из новостей этой новостной ленты. Ленту новостей можно реализовать в процедуре actionBlog(), а одну новость в процедуре actionView(). Для такой реализации в конструкторе контроллера нужно посчитать количество элементов $this->url и, если их четыре, то выполнить «$this->url[1]='View';».

Действие контроллера должно подготовить данные для представления, подключить необходимые скрипты и стили, установить мета-теги и возвратить имя ***представления*** или объект, его реализующий.

controller::$pageTitle — отображаемый на сайте заголовок, если не задан, то заголовок не выводится.

controller::$metaTitle — содержимое тега <title></title>.

controller::$metaKeyword и controller::$metaDescription — соответствующие метатеги страницы.

controller::script(name) — подключает указанный JavaScript, указывается без директория и без расширения, например так: «jquery.min» (соответствует файлу /public/js/jquery.min.js). Эта функция также защищает от повторного подключения одного скрипта. Скрипт подключается в секции <head></head>.

controller::style(name) — подключает файл стилей, указывается без директория и без расширения, например так: «blog» (соответствует файлу /public/css/blog.css). CSS подключается в секции <head></head>.

В конце работы ***действие*** должно вернуть строку, содержащую имя действия, например «Blog» (обязательно соблюдать регистр!). Реализация этого представления находится в файле /view/articleBlog.php. Как уже было сказано, ***действие*** контроллера может вернуть объект, который будет расценен как ***представление*** и движок вызовет процедуру render() для генерации HTML-кода страницы.

Для генерации хлебных крошек необходимо использовать процедуру с именем ***breadcrumbИмяДействия***. Если имя действия — это «blog», то хлебные крошки для этой страницы будут генерироваться в процедуре «breadcrumbBlog». Когда именно будет вызвана эта процедура зависит от того, в каком месте шаблона должны быть выведены хлебные крошки, однако можно с уверенностью сказать, что ***действие*** всегда выполняется раньше. Это позволяет в этой процедуре использовать данные, которые были подготовлены в действии.

Процедура, генерирующая хлебные крошки, должна возвратить массив, содержащий элементы «пути» к странице — это могут быть либо ссылки либо просто текст. В этот массив не должна входить главная страница и последняя (т. е. сама просматриваемая страница). Вот пример:

protected function breadcrumbBlog() {  
 return array('<a href="'.core::link('article/blog').'">Статьи</a>');  
}

Данный код сформирует примерно такую строку:  
Главная » Статьи » Новости

Тут «новости» — это название страница (controller::$pageTitle) и «главная» добавляются автоматически. Таким образом в самой процедуре нужно добавить только недостающие элементы. Разумеется, процедуры, генерирующей хлебные крошки, может и не быть, тогда движок создаст их самостоятельно и они будут содержать только два элемента: «главная» и название раздела.

Также следует заметить, что если в шаблоне сайта хлебные крошки не предусмотрены, то эта процедура вообще не будет вызвана. Движок вообще построен таким образом, чтобы подключать какие-либо файлы и другие ресурсы только в том случае, если в них есть необходимость.

Разумеется, страницы сайта могут содержать HTML-формы и для обработки POST-запросов движок имеет внутренний простой, но достаточно эффективный механизм. Кроме ***действия*** контроллера есть ещё ***submit-действие*** — это процедура контроллера, которая вызывается в том случае, если в POST есть соответствующие данные. Имя процедуры ***submit-действия*** такое же как имя действия, но с добавлением слова «Submit», например:  
public function actionBlogSubmit($data);

Для того, чтобы было вызвано ***submit-действие,*** данные в POST должны быть «обёрнуты» в массив, ключ которого совпадает с именем контроллера. Тоесть чтобы была вызвана процедура actionBlogSubmit() контроллера article, нужно чтобы данные в POST имели вид:

array('article'=>array(...));

Поле HTML-формы должно иметь вид:

<input type="text" name="article[someName]">.

Это нужно для того, чтобы контроллер обрабатывал только те данные, которые действительно ему принадлежат и игнорировал какие-либо другие данные (например, если данные отправляет и обрабатывает какой-либо виджет, хотя это неправильно: данные должен получать контроллер).

В большинстве случаев удобно пользоваться конструктором HTML-форм, который берёт на себя заботу о правильности именования полей.

***Submit-действие*** получает один параметр — это блок данных ($\_POST['controllerName']). В этом блоке данных также присутствуют все загруженные файлы (информация из $\_FILES).

Предполагается, что после обработки данных и в случае успешно выполненной операции, ***submit-действие*** в конце работы должно вызвать core::redirect() для редиректа на какую-либо страницу (выполнение редиректа после обработки формы полезно выполнять всегда). Если же операция успешно не выполнена, то нужно установить сообщение об ошибке в controller::$error, затем завершить процедуру выполнив «return» (возвращаемое значение игнорируется).

Если после submit-действия не был выполнен редирект, то движок запустит обычное ***действие***, что приведёт к отображению соответствующей страницы (как правило она содержит HTML-форму, заполнение которой и привело к выполнению submit-действия). При этом, если ранее было установлено сообщение об ошибке в переменной core::$error, то на экран будет выведено это сообщение.

* 1. Представление

Представления находятся в директории «/view». Имя файла состоит из двух частей: имени контроллера и имени представления (его возвращает процедура ***действия*** контроллера), например так: articleBlog.php. Тут article — это имя контроллера, а Blog — произвольное имя представления, которое часто совпадает с именем действия.

Данные из контроллера в представление передаются самым простым способом: в представлении доступна переменная $this, которая указывает на контроллер. Таким образом в контроллере можно объявить, например, $this->data=... А в представлении точно также обращаться к «$this->data». Разумеется таким же образом можно обращаться и к публичным процедурам контроллера и к моделям (через переменные контроллера).

Другой вариант представления — это объект. Движок сам по себе предлагает только один вариант такого представления — это модель, генерирующая HTML-форму, однако этот инструмент можно как угодно расширить. Работает это так: в ***действии*** контроллера создаётся экземпляр класса, ему задаются все необходимые атрибуты, после чего он возвращается при помощи оператора *return*.

* 1. Модель

Как уже было сказано ранее, обычно модель представляет из себя вспомогательную библиотеку, нежели модель в привычном для MVC ракурсе. Специального механизма для связи контроллера и модели нет, если какая-то модель необходима, то нужно просто подключить необходимый файл и создать экземпляр нужного класса.

В качестве модели движок предлагает использовать класс, реализующий некое подобие ActiveRecord (хотя таковым он не является). Это универсальная модель, она работает так: прямо в контроллере задаётся описание полей какой-либо таблицы базы данных, ей передаются данные прямо от HTML-формы, затем запускается механизм валидации и сохранения данных в базе данных.

Организация полноценной ActiveRecord возможна — для этого нужно создать класс, унаследованный от универсальной модели (класс *model*) и использовать специально заложенные для этого механизмы.

1. События (хуки)

Обработчики событий позволяют заставить систему реагировать на те или иные события.

Обработка происходит по следующему сценарию:

- инициализация того или иного события начинается с вызова процедуры core::hook(*имя\_события*,[*аргумент1, аргумент2...*]);

Количество аргументов сугубо индивидуально для каждого типа события. Движок загружает конфигурацию из файла /config/\_hook.php. Он имеет следующий формат:

return array(  
 'имя\_события'=>array('обработчик1','обработчик2'...)  
);

«Ё» выбирает из этого массива список обработчиков, который соответствует заданному *имя\_события* и последовательно подключает файлы с соответствующим именем, находящиеся в директории /hook. Каждый из этих скриптов должен возвратить булевое значение: **true** свидетельствует об успехе и движок вызывает следующий обработчик. Если возвращаемое значение — **false**, то следующий обработчик вызван не будет, а процедура core::hook() вернёт значение **false**. Это значение используется, например, при удалении пункта меню или или виджета для того, чтобы соответствующие модули могли удалить не используемые более данные. Если по каким-то причинам пункт меню удалить нельзя, то соответствующий обработчик события возвращает **false** и пункт меню удалён не будет.

В общедоступной части сайта используются только два события:

* beforeRender — генерируется перед началом вывода HTML-кода;
* search — генерируется при выполнении поиска.

Поиск по сайту реализован таким образом, что каждый модуль может в результаты поиска добавлять свои данные, для этого он должен реагировать на событие *search*. Процедура «search» получает один параметр — фраза, по который выполняется поиск. Эта процедура всегда должна возвращать **true**. В самой процедуре результаты должны выводить оператором echo, а каждый из элементов должен быть «обёрнут» в тег <li>. Пример реализации можно увидеть, например, в файле /hook/article.php.

При необходимости как в общедоступной части сайта, так и в админке можно добавлять свои обработчики событий, добавляя вызов core::hook().

1. Секции и виджеты

О том, что такое секции и виджеты, а также каким способом виджеты привязываются к страницам сайта можно почитать в руководстве пользователя. Здесь даётся информация об устройстве виджетов.

В базе данных есть две таблицы: «widget» и «section». В первой хранится информация о самих виджетах, а вторая определяет страницы, на которых должен публиковаться каждый из виджетов. Система сама заботится вопросами нужно ли публиковать вирус на той или иной странице, поэтому программист может сосредоточить внимание на логике.

Каждый виджет реализуется в отдельном скрипте, которые находятся в директории /widget. Имя файла определяется именем виджета (разумеется нужно учитывать регистр для UNIX-серверов), оно соответствует полю «name» таблицы «widget».

Типичный вид виджета таков:

<?php **class** widgetBlog extends widget {

function \_\_construct($data,$link) {  
 $this->options=$data;  
 $this->link=$link;  
 }

public function action() {  
 ...  
 **return** true;  
 }

public function render() {}

public function title($title) {}

} ?>

Имя класса, реализующего виджет, должно состоять из слова «widget», после которого следует имя виджета. Класс должен быть унаследован от «widget».

В конструктор этого класса передаются два атрибута: первый (массив или строка) — это параметры (настройки) виджета; второй атрибут — это ссылка, заканчивающаяся символом «.», «/» или «\*» - это та ссылка, по которой движок определил что виджет должен быть опубликован на данной странице. Это одна из ссылок, перечисленных в базе данных в таблице «section».

В большинстве случаев второй атрибут не используется. В виджет не обязательно перегружать конструктор: по умолчанию он ведёт себя так, как показано в псевдокоде выше, тоесть первый параметр помещает в $this->options, а второй — в $this->link.

Метод *action* вызывается движком после создания экземпляра класса и предназначен для подготовки данных, из которых формируется HTML-представление, а также решает нужно ли вообще публиковать виджет (например, виджет может выводится на экран только для зарегистрированных пользователей). Если метод *action* вернёт **null** или **false**, то движок прекратит обработку виджета и не будет генерировать никакого HTML-кода.

Внимание! Null и false также означает, что на экране не появятся также и кнопки административного интерфейса.

Если метод *action* вернёт **true**, то будет вызван метод *render()*, который должен вывести HTML код представления. Не рекомендуется выводить что-либо в самом методе *action()*.

Метод *action()* также может вернуть строку, например «Blog», тогда для генерации HTML-кода движок подключит файл с именем widgetBlog.php, находящемся в директории /view. Тут «Blog» - это строка, которую вернул метод *action()*. Таким образом виджет может иметь своё ***представление***. В нём доступ к каким-либо данным, также как в MVC основного контента, возможен через переменную $this.

Последний вариант — если метод вернёт объект. Тогда движок воспримет этот объект как ***представление*** и вызовет метод *render()* этого объекта.

Процедура *title()* предназначена для вывода заголовка виджета. Она вызывается только тогда, когда в админке в настройках указано «публиковать название виджета». Если этой процедуры нет, то по умолчанию будет выведен такой заголовок:

<p class="title">ТЕКСТ\_ЗАГОЛОВКА</p>

Очень часто в заголовок виджета требуется вставить ссылку на какую-либо страницу сайта. Именно для таких случаев и предусмотрена процедура *title()*.

1. Админка

Программирование админки во многом похоже на программирование общедоступной части сайта. Существенное отличие — это отсутствие ЧПУ. Если в общедоступной части сайта первый элемент ЧПУ-ссылки — это имя контроллера, а второй — имя действия, то в админке они указываются явно:

example.com/admin/index.php?controller=article&action=blog.

Файлы админки находятся в директории /admin и по структуре схожи с общедоступной частью, но элементы управления, открывающие диалоговые окна, задаются в контроллерах общей части сайта.

Административная часть сайта — это диалоговые окна, содержащие элементы управления, относящиеся к каким-либо конкретным элементам сайта. Каждая кнопка (или пиктограмма) — это, по сути, ссылка на административную часть сайта, которая расположена в директории /admin.

Диалоговые окна представляют из себя фреймы, в которых подгружается контент, поэтому стили сайта не могут повлиять на стили админки (за исключением кнопок). Если нажать правой кнопкой мыши на одной из «кнопок» админки и выбрать в меню «открыть в новой вкладке», то можно видеть тот же контент, который открывается во всплывающих окнах. Этим удобно пользоваться при отладке админки.

На самом деле в движке заложен потенциал, при помощи которого возможно реализовать классический бакэнд, но пока в этом нет необходимости.

Типичный вид ссылки, которая открывает контент в диалоговом окне (в данном примере — редактирование статьи):

example.com/index2.php?controller=article&action=article&id=2&\_front

Тут index2.php (как и в общедоступной части сайта) означает, что в отличии от index.php, не нужно выводить шаблон админки. Параметр \_front сообщает движку, что запрос выполнен из диалогового окна с общедоступной части сайта — это несколько меняет поведение фреймворка (функции core::link() и core::redirect() ).

В большинстве случаев пиктограммы административного интерфейса располагаются в правильном месте страницы и выглядят адекватно, однако всегда найдутся случаи, когда кнопки накладываются друг на друга, закрывают собой жизненно важный контент или конфликтуют со стилями сайта. Специально для этого каждая кнопка, точнее каждый тег <img>, имеет уникальный идентификатор (правда, уникальный в рамках открытой страницы). Используя этот идентификатор в совокупностями с теми блоками, в которых находится тег <img>, можно корректировать стили кнопки. Корректировать положение кнопки нужно используя стиль «margin». Стили лучше всего писать в файл /public/css/admin.php — этот файл подгружается автоматически, если авторизованный пользователь обладает правами администратора.

* 1. Типичный цикл обработки запроса администратора

В большинстве случаев админка работает по такому принципу:

Пользователь, наделённый правами администрирования сайта, нажимает ту или иную кнопку административного интерфейса, что приводит к загрузке во всплывающий фрейм контента по такой ссылке:

/index2.php?controller=ИМЯ\_КОНТРОЛЛЕРА&action=ИМЯ\_ДЕЙСТВИЯ&ДРУГИЕ\_ПАРАМЕТРЫ.

***Действие*** контроллера, в свою очередь, создаёт экземпляр класса form (вызывая функцию core::form() ) и динамически создаёт все необходимые поля HTML-формы. При этом в форму подставляются данные соответствующего редактируемого элемента используя ДРУГИЕ\_ПАРАМЕТРЫ (например это может быть идентификатор редактируемой записи). Далее процедура контроллера возвращает экземпляр класса этой формы. Движок запускает рендеринг HTML-формы, которую пользователь видит в диалоговом окне.

После внесения изменений в эту форму и нажатия кнопки submit, запускается ***submit-действие*** (специальный метод контроллера), в котором создаётся экземпляр класса *model* (через вызов core::model('имя\_таблицы'); ). В эту модель передаются данные формы (из $\_POST), затем динамически задаются правила валидации данных формы и вызывается метод *save()*, который сохраняет запись в базе данных, выполняя SQL-запрос INSERT или UPDATE.

Если валидация данных была не успешной, то выполнение процедуры прерывается (модель *model* самостоятельно устанавливает описание ошибки используя заданные правила валидации). Если же валидация и сохранение данных прошли успешно, то в ***submit-действии*** выполняется редирект и выводится сообщение об успешно выполненной операции.

Намного реже вместо динамически создаваемых моделей (*form* и *model*) используются статические модели, унаследованные от этих двух классов.

* 1. Общая информация о правах доступа и организаци

Как уже было сказано ранее, каждый администратор относится к одной из групп пользователей (с номерами от 200 до 255). Каждая из этих групп пользователей, в свою очередь, может иметь разные права управления сайтом (за исключением группы 255 — им можно всё). Права имеют булевой тип, тоесть «можно» или «нельзя» для какого-либо «права». Каждый модуль CMS может дополнять перечень возможных прав доступа.

Права привязываются к модулям CMS и записываются так: «ИМЯ\_МОДУЛЯ.ИМЯ\_ДЕЙСТВИЯ», например так: «article.category» (управление категориями статей), «article.article» (управление отдельными статьями), «html.\*» (изменение текста в виджите «произвольный текст»).

Тут ИМЯ\_МОДУЛЯ часто соответствует названию контроллера, но вообще модуль может иметь несколько контроллеров. Звёздочка обозначает «любые действия, связанные с этим модулем», но это условное обозначение и оно вовсе не означает, что движок кому-то разрешит любые запрошенные действия для данного модуля.

Права сохраняются в базе данных в таблице «right», которая имеет следующие поля:

* module — системное название модуля;
* description — текстовое описание «права»;
* group — группы пользователей (через запятую);
* picture — имя файла (без расширения png) — пиктограмма, используемая для обозначения «кнопки», открывающей диалоговое окно.

Поле «group» определяет каким группам разрешён доступ к данному модулю и действию. Всем, кто не перечислен в этом поле действие запрещено (за исключением группы 255).

Записи в таблицу «right» можно заносить вручную, а также их могут создавать модули при установке (при удалении модуля эти записи также удаляются).

При авторизации любого пользователя, определяется к какой группе он относится. Если это группа с номером больше 199, то движок автоматически подгружает необходимые библиотеки, стили и JavaScript, реализующие административный диалог. Если группа пользователя меньше 200, то излишние скрипты не загружаются, ровно как и не происходит никакого анализа прав доступа.

Набор прав доступа для конкретного пользователя определяется в момент авторизации и в дальнейшем хранится в сессии, поэтому при изменении этих прав, они вступают в силу только после повторной авторизации.

На любой странице сайта в левом верхнем углу всегда расположен ряд «кнопок», открывающих диалоговые окна админки (их перечень, разумеется, зависит от назначенных прав), - это те элементы админки, которые справедливы для всех или большого количества страниц сайта (например «общие настройки», «группы пользователей» или «пользователи»). Именно права в таблице «right» определяют какие кнопки должны быть выведены в этой области: выводятся те, у которых поле «picture» не пустое. Текст всплывающей подсказки — это поле «description», содержимое диалогового окна определяется полем «module» — будет подгружен контент по такой ссылке:

example.com/admin/index2.php?controller=ИМЯ\_МОДУЛЯ&action=ИМЯ\_ДЕЙСТВИя&\_front

МОДУЛЬ и ДЕЙСТВИЕ — это содержимое поля «model», звёздочка заменяется словом «Index».

* 1. Основные API-функции

Набор функций полностью аналогичен приведённому выше для общедоступной части. Но есть некоторые замечания...

Функции core::path(), core::import() и core::config() относятся уже к админке, а не к общедоступной части сайта. Если нужно обратиться именно к общей части сайта, то нужно использовать core::pathPublic(), core::importPublic() и core::configPublic().

Функции core::mysql(), core::sqlite() и core::db() возвращают не тот же объект, что в общедоступной части сайта, а его расширенную версию. Расширение заключается в том, что эти драйверы СУБД реализуют единый интерфейс для добавления и удаления отдельных полей таблиц базы данных (это в общем-то нужно только в админке).

* 1. Элементы управления админкой

В контроллере предусмотрены две процедуры, предназначенные для формирования кнопок административного интерфейса:

controller::adminACTIONLink() { return array(); }  
controller::adminACTIONLink2($data) { return array(); }

Тут ACTION — это имя ***действия***, а имя действия, как было сказано, определяет какая страница будет отображена в браузере.

Первая из этих процедур предназначена для формирования «кнопок» админки, относящейся ко всей странице в целом, вторая — для кнопок, относящихся к одному из элементов страницы. Например, для какого-либо блога первая процедура может формировать кнопки «настройки блога, метатеги», «изменение вступительного текста», а вторая — кнопки «изменить статью», «удалить статью», тоесть кнопки, относящиеся к одной из записей блога, отображаемых на странице.

Каждая из этих функций должна вернуть массив массивов, каждый из которых описывает одну из «кнопок». Поля такого массива следующие:

0 => Модуль.Действие, по которому определяется имеет ли авторизованный пользователь доступ к выполнению данной операции. Это поле сравнивается с полем «module» в описанной выше таблице «right».

1 => условное имя файла изображения без директория и расширения. Например, если указать «edit», то это будет соответствовать файлу /admin/public/edit16.png.

2 => текст, всплывающая подсказка, появляющаяся при наведении курсора мыши.

3 => текст для параметра ALT тега <img>. Через этот параметр косвенно можно «запихнуть» любой html-код в тег <img>.

Обязательными являются только первые 2 параметра.

Важно помнить, что эти две процедуры всегда вызываются после того, как будет вызвано ***действие*** контроллера, поэтому необходимые данные (например идентификатор записи) скорее всего уже загружены в те или иные атрибуты класса контроллера.

Вот пример первой процедуры:

protected function adminBlogLink() {  
 return array(  
 array('article.category','?controller=article&action=category&id='.$this->category['id'],'edit','Править заголовок, мета-теги блога'),  
 array('article.article','?controller=article&action=article&categoryId='.$this->category['id'].'&blog','new','Добавить новую статью в блог')  
 );  
}

Из всех перечисленных в этом массиве кнопок будут созданы только те, на действие которых пользователь имеет право (за это отвечает первый параметр).

Вторая процедура должна вернуть точно такой же массив как и первая. Для этого ей передаётся один параметр, который содержит данные одного элемента из множества (одной записи блога, например).

Данные для этой процедуры «поставляются» из ***представления*** вызовом $this->admin($item), где $item - блок данных, который должен быть передан в процедуру построения кнопок. Обычно эта часть представления выглядит так:

foreach($this->data as $item) {  
 ...  
 $this->admin($item);  
 ...  
}

Виджеты также могут иметь кнопки административного интерфейса. Для этой цели в них предусмотрены две процедуры:

widget::adminLink() { return array(); }  
widget::adminLink2($data) { return array(); }

Смысл этих процедур точно такой же, как и в случае контроллера. Из ***представления*** виджета данные передаются точно таким же образом:

$this->admin($item);

* 1. Виджеты
     1. Кнопки административного интерфейса (фронтенд)

В общедоступной части сайта, в скриптах, реализующих виджеты (/widget), предусмотрены два метода:

public adminLink() { return array(...); }

public adminLink2($data) { return array(...); }

Оба этих метода возвращают массивы, каждый из которых описывает одну кнопку административного интерфейса. Работают они также, как и в случае с основным контентом, об этом было сказано ранее в разделе «Элементы управления админкой».

Эти процедуры реализуют кнопки управления, каждая из которых, фактически, просто размещает ссылку на ту или иную страницу админки, которая открывается во всплывающем окне. Но кроме них есть ещё одна ссылка на админку — соответствующая станица открывается при создании или изменении виджета через управление секциями, тоесть при нажатии на кнопку административного интерфейса со всплывающей подсказкой «управление виджетами в этой области». Об этом ещё будет сказано дальше.

* 1. Бакэнд

В базе данных есть таблица *widgetType*, которая содержит описание всех типов виджетов. Её поля таковы:

* *name* – системное имя виджета, также в различных местах определяет имена файлов, название классов и процедур;
* *title* – текстовое описание виджета, отображаемое при создании нового виджета (в админке);
* *controller* – имя контроллера (бакэнд), который обрабатывает запросы административного интерфейса (при нажатии на одну из кнопок);
* *action* – имя ***действия*** контроллера, которое обрабатывает запросы создания нового виджета и изменения существующего.

Записи в эту таблицу можно добавлять вручную, если разрабатывается виджет, индивидуальный для одного проекта. Или же их может добавить скрипт установки модуля.

Запросы админки обрабатывает контроллер, указанный в поле «*controller»* таблицы «widgetType». Как правило этот же контроллер обрабатывает запросы, связанные с обслуживанием основного контента. В контроллере для каждого виджета должны быть предусмотрены два метода:

public function actionWidget**NAME**($data=null) {  
 if(!$data) $data=array(...);  
 ...  
}

public function actionWidget**NAME**Submit($data) {  
 ...  
 return $data;  
}

Тут **NAME** – это имя виджета.

Первый метод вызывается при нажатии кнопки «добавить виджет» в диалоговом окне «управление виджетами в этой области», при выборе типа виджета из списка. А также при редактировании виджета из того же диалогового окна. Он должен сформировать дополнительные элементы управления, при помощи которых осуществляется настройка виджета. Метод принимает один параметр, содержащий настройки виджета, если он уже существует, если же виджет только создаётся, то этот параметр будет **null**.

Ст**о**ит обратить внимание на то, что данный метод является ***действием*** контроллера, поэтому он может возвращать ***MVC представление*** также, как и любое другое ***действие***, кроме того также можно устанавливать подсказки, использую $this->cite, а вот заголовок страницы и метатеги не имеют никакого смысла.

Второй метод — это ***submit-действие***, ему передаются данные формы (из $\_POST). Он должен вернуть сформированные настройки виджета — это может быть либо строка либо массив. Если форма была заполнена неправильно, то метод должен установить сообщение об ошибке в атрибут $this->error. При этом настройки виджета не будут сохранены, а метод ***действия*** будет вызван повторно, возвращаемое ***submit-действием*** значение игнорируется.

Довольно часто в ***submit-действии*** виджета можно сразу же вернуть полученный атрибут $data, однко это может привести к тому, что в возвращаемое значение попадут излишние данные — этого следует избегать, т. к. настройки виджета хранятся в базе данных и подгружаются при каждом его показе на фронтенде.

Движок может проводить кеширование виджетов автоматически, для этого нужно лишь задать время жизни кеша. Вопросы кеширования — это задачи программиста, а не администратора, поэтому время кеширования задаётся программистом. Для этого возвращаемое ***submit-действием*** значение должно быть массивом и содержать элемент «cacheTime», в котором указывается время актуальности кеша (в минутах). Ст**о**ит заметить, что кеширование не работает, если сайт работает в отладочном режиме.

Виджет может иметь дополнительные кнопки административного интерфейса и они работают точно также как и в случае кнопок основного контента, о чём было сказано ранее.

Так как описанные выше процедуры являются действиями контроллера, то для них также нужно настраивать права доступа при помощи описанного выше метода controller::right().

Очень часто виджеты создают какие-либо файлы или записи в базе данных, поэтому нужно всегда удалять созданные данные при удалении виджета. Например, виджет «произвольный текст», создаёт файл, который этот текст содержит. Очевидно, что при удалении виджета нужно также удалить этот файл. Для этого нужно установить обработчик для события «widgetDelete». Внутри этого обработчика нужно проверить, что удаляется именно виджет «html» (произвольный текст) и если это так, то удалить соответствующий файл. События в административной части сайта работают также, как и в общедоступной. О событиях было написано в разделе «События (хуки)».

* 1. Меню

В административной части сайта устройство меню довольно схоже с устройством виджетов.

В базе данных есть таблица «menuType», в которой хранится описание всех типов меню. Поля этой таблицы таковы:

* *title* – название типа меню;
* *controller* – имя контроллера, который обрабатывает запросы на создание и изменение пункта меню;
* *action* – имя действия, которое обрабатывает запросы на создание и изменение пункта меню.

При нажатии на кнопку «добавить пункт меню» и выборе из списка типа меню, в диалоговое окно подгружается форма, содержащая дополнительные элементы управления. Суть этой формы состоит в том, чтобы сформировать ссылку на какую-либо страницу сайта (при этом также могут вноситься какие-либо данные в базу данных или создаваться какие-либо файлы).

В контроллере, указанном в поле «controller» таблицы «menuType» должны быть предусмотрено два метода для каждого типа меню:

public function actionMenu**NAME**() {  
 if($\_GET['link']) $link=$\_GET['link']; else $link='';  
 ...  
 $f=core::form();  
 $f->submit('Продолжить','submit);  
 return $f  
}

public function actionMenu**NAME**Submit($data) {  
 ...  
 return $link;  
}

Тут **NAME** — это название типа виджета.

Первый метод должен сформировать HTML-форму, параметры которой определяют тип ссылки. Если метод вызван для редактирования существующей ссылки, то эта ссылка будет доступна через переменную $\_GET['link'].

Иногда никаких полей в форме не требуется, тогда нужно просто вернуть «пустую» форму так, как это сделано в примере. В данном примере для кнопки «продолжить» задано имя «submit» - это нужно для того, чтобы в $\_POST был хотя бы один параметр, иначе метод actionMenu**Name**Submit() вызван не будет.

Эта процедура является ***действием*** контроллера, поэтому она может возвращать ***MVC представление*** также, как это делается для основного контента (описано в разделе про MVC). Кроме того можно устновить текст подсказки через $this->cite, а вот метатеги и заголовок страницы не используется.

Второй метод является ***submit-действием***. Он получает данные HTML-формы из первой процедуры и должен вернуть строку, содержащую ссылку на страницу, например «article/blog/news».

Очевидно, что при создании пункта меню могут также создаваться сопутствующие данные (в базе данных или в файловой системе). Поэтому при удалении пункта меню нужно также обязательно удалить эти сопутствующие данные. Для этого нужно установить обработчик на событие «menuItemDelete». Обработчику передаётся один параметр — удаляемая ссылка.

Важно! Меню может содержать несколько ссылок на одну и ту же страницу, поэтому перед удалением данных нужно обязательно выполнить примерно такую проверку:

if($db->fetchValue('SELECT COUNT(id) FROM menuItem WHERE link='.$db->escape($link))!='1') return true;

Подробнее про обработку событий написано в разделе «События (хуки)».

* 1. Особенности программирования админки

1. Несмотря на очевидное отсутствие ЧПУ и подмены ссылок, всё же лучше использовать функцию core::link(), за исключением выполнения ajax-запросов. Эта функцция обеспечит правильное построение ссылки. В настоящий момент это используется очень слабо, т. к. движок не имеет классического вида бакэнда — это лишь залог на будущее. Ссылки в эту функцию должны передаваться в таком виде:

?ПАРАМЕТРЫ&ПАРАМЕТРЫ...

Тоесть без указания директория /admin и без index2.php. Параметр \_front также добавляется автоматически. Пример:

$link=core::link('?controller=artice&action=category&id='.$this->categoryId);

2. Если страница загружена в сплывающем диалоговом окне, то поведение функции core::redirect() несколько отличается: если передан второй параметр, то во всплывающем окне будет выведен текст сообщения и через три секунды страница (не админки, а открытая в браузере страница фронтенда) будет перезагружена и реального редиректа не произойдёт. Если же сообщение не указано, то контент по новому адресу будет загружен немедленно.

Кроме того есть ещё специальная процедура core::redirectPublic(), которая выполняет редирект страницы фронтенда. Эта процедура выполнит загрузку контента по указанному URL вне зависимости от того во всплывающем окне загружена страница или нет.

Также не ст**о**ит забывать, что эти процедуры дополнительно обрабатывают запрос функцией core::link(), поэтому переданная ей ссылка должна начинаться с «?».

3. Контроллер имеет дополнительный атрибут *cite* — это поясняющий текст, который должен быть напечатан под формой. Суть его — дать подсказку по назначению тех или иных элементов управления. Использовать его можно так:

public function actionCategory() {  
 ...  
 $this->cite='Содержимое поля <b>Вступительный текст</b> публикуется над списком статей.';   
 ...  
}

4. Движок автоматически формирует HTTP-заголовок LastModified, но для его работы необходимо, чтобы модули генерировали событие «modify» каждый раз, когда содержимое страницы изменяется. Обработчику события нужно передать один параметр — ссылку на страницу, содержимое которой изменилось. Этот механизм имеет существенный недостаток: при удалении каких-либо страниц ссылки на них по прежнему остаются в базе данных. Эта проблема в настоящей версии движка никак не решена.

1. Шаблоны, информация для верстальщика

Информация в этом разделе может быть полезна не только программисту, но и верстальщику.

Шаблоны хранятся в директории /template и представляют из себя HTML-документы, содержащие специальные теги для вставки контента. Имена файлов формируются по следующему принципу:

(pc|pda).ИМЯ\_ШАБЛОНА.html

Тут pc – шаблон дня настольных ПК, pda – шаблон для мобильных устройств. Версия для мобильных устройств доступна на поддомене *pda*. Переключение или редирект с одной версии на другую должен определяться в шаблоне с помощью явного указания ссылок или при помощи JavaScript.

ИМЯ\_ШАБЛОНА определяется программистом и устанавливается при помощи модуля «управление шаблонами» или же непосредственно в коде вызовом функции core::template(). В большинстве случаев есть только один шаблон и имя соответствующего файла «pc.default.html».

Все ресурсы (CSS, JavaScript, картинки и т. д.) должны находиться в пределах директория /public, рекомендуется складывать «всё в кучу» в директорий /public/template, но это не является обязательным требованием.

В шаблоне могут быть использованы следующие специальные теги:

* <?=core::url()?> - относительный URL адрес до корня сайта. Ссылки на ресурсы должны начинаться с этой php-вставки.
* {{metaTitle}} – заголовок, должен находиться внутри тега <title>. Рядом с этим тегом может быть и другой текст, например название сайта, если оно брендируется.
* {{metaKeyword}}, {{metaDescription}} – соответствующие метаданные, содержат только текст, поэтому должны быть «обёрнуты» в тег <meta>.
* {{head}} — должен быть добавлен в секцию <head> и в это место будут подставлены JavaScript и CSS для конкретных страниц.
* {{widget[admin]}} — на это место будут вставлены общие кнопки административного интерфейса. Обычно он вставляется сразу после <body>.
* {{widget[search]}} — виджет, содержащий текстовую строку для поиска по сайту.
* {{widget[menu][1]}} — меню сайта. Тут «1» - это идендификатор меню.
* {{widget[user]}} — форма авторизации пользователей.
* {{breadcrumb}} — хлебные крошки
* {{section[ИМЯ]}} — вставляет секцию с именем ИМЯ.
* {{pageTitle}} — отображаемый на сайте заголовок страницы. Он дополнительно «обёрнут» в тег <h1 class=”pageTitle”>.
* {{content}} — основной контент сайта.

В шаблоне можно вывести любой виджет при помощи тега {{widget}}, однако для этого нужно знать какие он должен принимать параметры. Формат этого тега следующий:

{{widget[ИМЯ][ДАННЫЕ][ВРЕМЯ\_КЕШИРОВАНИЯ]}}

Имя — это имя виджета (соответствует имени файла в директории /widget.  
ДАННЫЕ — параметры, необходимые для отображения виджета. Это может быть просто строка или число или массив, тогда он должен быть задан в таком формате: параметр\_1:"значение",параметр\_2:"значение"...

В шаблоне сайта может быть задано любое количество секций при помощи тега {{section}} — это, по сути, контейнер для публикации виджетов. Но всё же если виджет должен быть опубликован на всех страницах сайта, то лучше вставить его явно при помощи тега {{widget}}.

Внимание! Виджеты кешируются, поэтому, если в шаблон вносятся изменения, то нужно очистить кеш, либо перевести сайт в отладочный режим для автоматической очистки кеша.

Содержимое тега {{content}} зависит от отображаемой страницы сайта. HTML-разметку контента можно найти в файлах в директории /view. HTML-разметка виджетов может находится также в директории /view или в файлах в директории /widget.

1. Класс HTML-формы (form)

Представляет из себя генератор HTML-форм. Его использование помогает привести HTML-код к единому виду, кроме того, подставляет нужные данные в поле формы. Особо полезно использовать генератор в админке.

Данный класс можно использовать в роли *MVC-представления*, для этого *MVC-действие* должно вернуть экземпляр класса. В результате браузеру будет отдан HTML-код формы.

Типичный пример кода (файл /controller/example.com):

public function actionTest() { *//действие контроллера*  
 $f=core::form(); *//получение экземпляра класса*  
 $f->text('name','Введите что-нибудь');  
 $f->submit();  
 return $f; *//использовать форму в роли представления*  
}

public function actionTestSubmit($data) {  
 *// В $data будут находиться данные формы*  
}

Создание экземпляра класса form: class **core::form**(*namespace*);  
Тут *namespace* — область для POST-данных формы. Атрибут name всех полей формы имеет вид:  
[namespace][fieldName], например:

<input type="text" aname="namespace[имя поля]" />

*namespace* определяет для кого предназначаются данные. В *submit-действие* контроллера попадают только те данные, имя namespace для которых такое же, как имя контроллера. Тоесть: $\_POST['namespace']. В приведённом выше примере в функцию actionTestSubmit() будет передано содержимое $\_POST['example']. Такая группировка данных нужна чтобы, например, позволить виджету обрабатывать данные форм - контроллер при этом никак не должен реагировать на эти данные.

Если при вызове core::form() *namespace* не задан, то будет подставлено имя текущего контроллера.

* 1. Публичные атрибуты

form::$action — URL-адрес страницы, которая будет обрабатывать форму, строка задаётся в виде «controller/action/etc». Использовать core::link() при этом не нужно. Если этот атрибут не задан, то будет использоваться адрес текущей страницы.

form::$method — метод отправки данных («get» или «post»), по умолчанию «post».

* 1. Методы класса

null **form::hidden**(*name*, *value*, [*html]*);  
 Добавляет к форме скрытое поле с именем *name* и значением *value*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::label**(*label*, *value*);  
 Добавляет к форме надпись: «label: value».

null **form::text**(*name*, *label*, [*value]*, [*html]*);  
 Добавляет к форме текстовое поле с именем *name*, заголовком *label* и значением по умолчанию *value*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::select**(*name*, *label*, *items*, [*value]*, [*nullTitle]*, [*html]*);  
 Добавляет к форме поле с выпадающим списком с именем *name*, заголовком *label* и значением по умолчанию *value*. Список элементов задаётся параметром *items*: это может быть массив массивов:  
*array(array([0]=>'значение',[1]=>'отображаемый текст'))*или строка, содержащая SQL-запрос, который должен выбирать из базы данных два поля: первый — значение, второй — отображаемый текст.

*NullTitle* — отображаемый заголовок для «пустого» пункта списка. Если не задан, то в этом выпадающем списке нужно будет обязательно выбрать какое-то значение. Необязательный параметр *html* добавляет к тегу <select> произвольный код (например, «onclick="..."»).

null **form::listBox**(*name*, *label*, *items*, [*value]*, [*nullTitle]*, [*html]*);  
 Тоже что и form::select, но генерирует не поле с выпадающим списком, а многостраничный список.

null **form::radio**(*name*, *label*, *items*, [*value]*);  
 Добавляет в форму список переключателей с именем *name*, заголовком *label* и значением по умолчанию *value*. Смысл параметра *items* такой же, как и у form::select().

null **form::checkbox**(*name*, *label*, [*value*], [*html*]);  
 Добавляет в форму поле «checkbox» с именем *name*, заголовком *label* и значением по умолчанию *value*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::password**(*name*, *label*, [*html]*);  
 Добавляет к форме текстовое поле (для ввода пароля) с именем *name* и заголовком *label*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::textarea**(*name*, *label*, [*value]*, [*html]*);  
 Добавляет к форме поле многострочного текста с именем *name*, заголовком *label* и значением по умолчанию *value*. Необязательный параметр *html* добавляет к тегу <textarea> произвольный код (например, «onclick="..."»).

null **form::editor**(*name*, *label*, [*value]*, [*html]*);  
 Добавляет к форме поле многострочного текста (с визуальным редактором) с именем *name*, заголовком *label* и значением по умолчанию *value*. Необязательный параметр *html* добавляет к тегу <textarea> произвольный код (например, «onclick="..."»).

null **form::date**(*name*, *label*, [*value]*, [*html]*);  
 Добавляет к форме текстовое поле (со всплывающим календарём) с именем *name*, заголовком *label* и значением по умолчанию *value*. Value может быть задан в виде строки «дд.мм.ГГГГ» или целого числа (unix timestamp). Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::file**(*name*, *label*, multiple=false, [*html]*);  
 Добавляет к форме поле для загрузки файла. Имя поля - *name*, заголовок — *label*. Параметр *multiple* указывает можно ли выбирать сразу несколько файлов. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::captcha**(*name*, *label*, [*html]*);  
 Добавляет к форме каптчу. Поле с именем *name*, заголовком *label*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::reset**(*label*, [*html]*);  
 Добавляет к форме кнопку «сбросить» с заголовком *label*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::submit**(*label='Продолжить'*, [*html]*);  
 Добавляет к форме кнопку «послать» с заголовком *label*. Необязательный параметр *html* добавляет к тегу <input> произвольный код (например, «onclick="..."»).

null **form::field**(*type, f1*, *f2*, [*f3]*, [*f4]*);  
 Добавляет к форме одно из перечисленных выше полей, *type* указывает тип поля (hidden, text, label и так далее). Параметры f1, f2, f3 и f4 зависят от типа поля. Главным образом это поле предусмотрено для модуля «form» (настраиваемые формы). Но может быть полезно в любой другой ситуации, когда поля должны быть настроены через админку.

null **form::html**(*html*);  
 Добавляет к форме произвольный HTML-код.

string form::**getHidden**(), string **form::getText**(), string **form::getSelect**(), string **form::getListBox**, string **form::getRadio**(), string **form::getCheckbox**(), string **form::getPassword**(), string **form::getTextarea**(), string **form::getEditor**(), string **form::getDate**(), string **form::getFile**(), string **form::getSubmit**()  
 Возвращают HTML-код соответствующего поля, содержит только тег, без какой-либо HTML «обвески». Полезно использовать в *MVC-представлении*, когда необходимо реализовать уникальный дизайн. Это лучше, чем писать HTML-код самостоятельно, кроме того, могут быть случаи, когда заранее неизвестно количество и типы полей.

null **form::render**([*action*], [*html*]);  
 Генерирует HTML-форму и выводит её на экран. Параметр *action* задаёт адрес, на который будет отправлена форма (смотрите выше form::$action — это тоже самое). Необязательный параметр *html* добавляет к тегу <form> произвольный код (например, «onclick="..."»).

Именно этот метод будет вызван автоматически, если класс формы используется в роли представления.

1. Универсальная модель базы данных (model)

Класс *model* является упрощенным вариантом привычного ActiveRecord, реализуя возможность сохранять данные в таблицу БД, а также предварительно проводить фильтрацию и валидацию. Реализацию класса можно найти в файле */core/model.php*. Предусмотрены два варианта использования этого класса: динамический, когда перечень полей и их тип задаются непосредственно в коде и статический, когда класс наследуется от базового *model* и описывает параметры полей таблицы базы данных.

Общий алгоритм работы с классом такой:

1. получить экземпляр класса;
2. передать данные HTML-формы вызвав метод *set()* или поочерёдно установить все поля сеттерами;
3. вызвать метод *validate()* для проверки валидности и фильтрации данных;
4. записать данные в базу данных вызвав метод *save()*.

Вот типичный пример динамического использования:

public function actionTestSubmit($data) { //submit-действие контролле  
 $m=core::model('myTable'); //получение экземпляра класса, myTable — имя таблицы БД  
 $m->set($data); //передача \_POST-данных  
 $m->myField='some text';  
 if(!$m->save(array( //сохранить данные, проведя предварительную фильтрацию и валидацию  
 'id'=>array('primary'),  
 'name'=>array('string','имя',true,'min'=>2,'max'=>40),  
 'myField'=>array('string')  
 ))) return false;  
 core::redirect('myController/myAction','Изменения сохранены');  
}

Вот пример статического класса, содержащего минимум необходимой информации:

class myTable extends model {  
 protected $validate=array( //правила фильтрации и валидации  
 'id'=>array('primary'),  
 'name'=>array('string','имя',true,'min'=>2,'max'=>40),  
 'myField'=>array('string')  
 );  
 protected $fields='\*'; //перечень полей, которые нужно загружать, если метод *load* вызван без параметров.

public function \_\_construct() {  
 parent::\_\_construct('myTable');  
 }  
}

Использование этого класса:

public function actionTestSubmit($data) { //submit-действие контроллера  
 core::import('model/myTable');  
 $m=new myTable();  
 $m->set($data);  
 if(!$m->save()) return false;  
 core::redirect('myController/myAction','Изменения сохранены');  
}

Для того, чтобы передать данные полей в модель, необходимо воспользоваться методом *set()* или передать их по отдельности при помощи сеттеров. Таким же образом задаётся значение первичного ключа. Если таблица содержит первичный ключ и его значение было передано (не ноль, не null, не false и не пустая строка) в класс *model*, то при вызове метода *save()* будет выполнен SQL-запрос «UPDATE». Если же значение первичного ключа не было передано, или оно ложно, то будет выполнен SQL-запрос «INSERT». При этом значение первичного ключа можно получить через сеттер:

$id=$m->id;

Проведение валидации данных может привести к искажению данных за счёт того, что будут вырезаны HTML-теги, вставки скриптов, строки преобразованы к числу (если требуется, конечно) и т. д.

* 1. Методы класса

object **model::\_\_construct**(string *namespace*[,string *db*='db']);  
 тут *namespace* — имя таблицы базы данных, *db* — используемая СУБД (может принимать следующие значения: db — указанная в общих настройках (по умолчанию), mysql или sqlite).

null **model::set**(array *data*);  
 Передаёт модели данные полей таблицы, параметр *data* должен быть ассоциативным массивом, где ключ — имя поля, а значение — содержимое поля.

array **model::get**();  
 Возвращает ассоциативный массив, содержащий все поля записи БД.

&array **model::load**(string *where*[, string *field*]);  
 Загружает данные из базы данных, выбирая запись (одну первую) по указанному в параметре *where* условию (часть SQL-запроса, после «WHERE». В параметре *field* указывается перечень (через запятую) полей, которые необходимо выбрать. Если параметр *field* не указан, то список полей будет взят из атрибута model::$field. Если этот атрибут не был задан, то будут загружены все поля.

Функция возвращает ссылку на ассоциативный массив, поэтому можно менять значения полей «на лету» и записывать данные обратно в базу данных.

&array **model::loadById**(int *id*[,string *fields*]);  
 Тоже, что model::load(), но загружает данные не по условию, а по уникальному идентификатору.

bool **model::validate**([array *validate*]);  
 Выполняет фильтрацию и валидацию данных. Возвращает **true**, если проверка прошла успешно. Параметр *validate* должен быть ассоциативным массивом, содержащим правила валидации (описано далее). В случае статического использования класса этот параметр может быть опущен.

bool **model::save**([mixed *validate*],[string *fields*],[string *id*]);  
 Сохраняет данные записи в базе данных. Если в таблице содержится первичный ключ и он не false, то выполняется SQL-запрос «UPDATE», если в таблице нет первичного ключа или он не был задан или он false, то выполняется SQL-запрос «INSERT».

Если параметр *validate* является массивом, то он воспринимается как правила валидации и перед сохранением будет проведена фильтрация и валидация согласно этим правилам. Если параметр равен **true**, то валидация будет проведена согласна правилам, заданным в атрибуде model::$validate (только для статического использования). Если параметр *validate* равен **false**, то валидация проводиться не будет.

Если задан параметр *fields*, то в запросе INSERT или UPDATE будут учавствовать только поля, перечисленные в этом параметре. Этот параметр следует использовать только тогда, когда не задан первый параметр (*validate*), иначе информацию о перечне полей движок сможет взять из правил валидации.

Параметр *id* может содержать имя первичного ключа — параметр следует использовать в том случае, если не проводится валидация (в противном случае движок найдёт имя первичного ключа в правилах валидации).

bool **model::delete**([int *id*],[bool *affected*]);  
 Удаляет запись с идентификатором *id*. Если идентификатор не задан, то будет удалена запись, загруженная ранее. Если параметр *affected* равен **true**, то метод возвращает **true** только в том случае, если запись существовала и действительно была удалена, если её не существовало, то вернёт **false**.

* 1. ***Атрибуты и методы для статического использования (перегрузка класса)***

array **model::$\_data**  
 Ассоциативный массив, содержащий данные полей записи базы данных.

string **model::$\_primary**  
 Содержит имя первичного ключа.

object **model::$db**  
 Содержит экземпляр класса mysql или sqlite. SQL-запросы нужно выполнять через этот атрибут, т. к. возможна ситуация, когда модель используется не с основной СУБД.

array **model::$validate**  
 Должен содержать ассоциативный массив правил валидации (описано ниже).

string **model::$field**  
 Должна содержать строку, содержащую список полей (через запятую), которые необходимо выбрать из базы данных при вызове метода *model::load()*.

bool protected function **afterInsert**([id]);  
 Триггер, который будет выполнен после выполнения SQL-запроса «INSERT». В единственном параметре передаётся значение первичного ключа созданной записи или *null*, если первичнго ключа нет. Возвращаемое этой процедурой значение будет возвращено методом model::save().

bool protected function **afterUpdate**(id);  
 Тоже, что и model::afterInsert(), но выполняется после выполнения SQL-запроса «UPDATE».

* 1. Правила валидации

Правила валидации представляют из себя ассоциативный массив, каждый ключ которого — это имя поля, а значение — массив, описывающий это поле. Вот пример правил валидации:

Пример массива с правилами валидации:

array(  
 'id'=>array('primary'),  
 'title'=>array('string','заголовок',true,'min'=>6,'max'=>60),  
 'status'=>array('boolean'),  
 'description'=>array('html','описание',true)  
)

Этот массив описывает четыре поля с именами id, title, status и description.

В каждом вложенном массиве первый элемент (с индексом «0»)— это тип поля (или, можно сказать, имя валидатора). Второй элемент (не обязательный) — заголовок поля, который используется в случае, если в поле содержится недопустимое сообщение. Сообщение строится таким образом, что заголовок не склоняется по падежам, например: «Поле «заголовок» не может быть пустым» или «Поле «заголовок» должно содержать от 6 до 60 знаков». Если заведомо известно, что поле не может содержать недопустимое значение, то указывать заголовок не нужно. Третий элемент массива показывает может ли поле содержать пустое значение. В зависимости от типа поля могут быть заданы дополнительные параметры — они имеют не числовой, а буквенный индекс.

Ниже представлена подробная информация для каждого типа. Обратите внимание, что только первые три элемента имеют числовой индекс.

**primary**: array('primary', [string *title*], [bool *canNull*=true])  
 Первичный ключ таблицы базы данных. Если он не false, то будет выполнен SQL-запрос «UPDATE», в противном случае - «INSERT». Если *canNull* равен **false**, то поле не может быть пустым. Это означает, что будет возможно только обновить существующую запись, но не добавить новую.

**integer**: array('integer', [string *title*], [bool *canNull*=true], ['*min'*=>int], ['*max'*=>int])  
 Целое число. *min* — минимально возможное значение, *max* — максимально возможное значение. *canNull* — может ли поле быть пустым (false или пустая строка, но не 0). *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**float**: array('float', [string *title*], [bool *canNull*=true], ['*min'*=>int], ['*max'*=>int])  
 Дробное число. *min* — минимально возможное значение, *max* — максимально возможное значение. *canNull* — может ли поле быть пустым (false или пустая строка, но не 0). *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**boolean**: array('boolean')  
 true/false.

**date**: array('date', [string *title*], [bool *canNull*=true])  
 Дата. Значение даты может быть задано в виде unix timestamp (целое число) или строкой вида «дд.мм.гггг». *canNull* — может ли поле быть пустым. *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**string**: array('string', [string *title*], [bool *canNull*], ['*min'*=>int], ['*max'*=>int], ['*trim*'=>boolean])  
 Строка (html-теги вырезаются автоматически). *min* — минимально допустимая длина строки; *max* — максимально допустимая длина строки; *trim* — надо ли обрезать ведущие и завершающие пробелы (по умолчанию false). *canNull* — может ли поле быть пустым. *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**html**: array('html', [string *title*], [bool *canNull*])  
 Произвольный текст или HTML-код. *canNull* — может ли поле быть пустым. *title* — имя поля, используется для формирования текста ошибки, если поле пустое (имеет смысл только если canNull=true).

**latin**: array('latin', [string *title*], [bool *canNull*])  
 Строка, содержащая только латинские буквы, цифры, дефис и знак подчёркивания («\_»). *canNull* — может ли поле быть пустым. *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**email**: array('email', [string *title*], [bool *canNull*])  
 Адрес электронной почты. *canNull* — может ли поле быть пустым. *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**regular**: array('regular', [string *title*], [bool *canNull*], [string *expression*])  
 Любое значение, удовлетворяющее заданному регулярному выражению *expression*. *canNull* — может ли поле быть пустым; *title* — имя поля, используется для формирования текста ошибки, если поле содержит недопустимое значение.

**captcha**: array('captcha')  
 Каптча. Не требует дополнительных параметров.

**callback**: array('callback', [string *title*], [bool *canNull*], [callback *function*])  
 Валидация при помощи внешней функции *function*. *canNull* — может ли поле быть пустым; *title* — имя поля, используется для формирования текста ошибки, если поле пустое. Функция должна принимать два значения: имя поля и содержимое этого поля. Функция должна вернуть новое содержимое поле (экранированное или преобразованное для хранения в базе данных). Если значение не удовлетворяет требованиям, то функция должа установить текст ошибки в controller::$error.

1. Класс универсальной таблицы (table)

Представляет из себя генератор HTML-таблиц, может быть использован в роли представления. Этот класс используется только в административном интерфейсе.

Работа с классом начинается с получения его экземпляра:

$table=core::table();

Реализацию класса можно найти в файле */admin/core/html.php*.

Общий алгоритм работы с классом такой:

1. получить экземпляр класса
2. установить заголовки столбцов

в цикле добавлять ячейки таблицы (они будут авторматически разбиты на строки в зависимости от количества столбцов, заданных во втором этапе).

Типичный пример использования класса:

public function actionTest() { //действие контроллера  
 $t=core::table(); //получение экземпляра класса  
 $t->rowTh('Группа|Описание|'); //заголовки таблицы  
 ...  
 while($item=$db->fetch()) { //в цикле добавляет столбцы таблицы  
 $t->text($item[0]);  
 $t->link($item[1],'?controller=user&action=groupItem&id='.$item[0]);  
 $t->delete('?controller=user&action=groupDelete&id='.$item[0]);  
 }  
 return $t;   
}

* 1. Методы класса

**table::\_\_construct**([string *html*]);  
 Конструктор класса. Если задан атрибут *html*, то этот текст будет добавлен к HTML-тегу <table>.

**table::th**(string *title*,[string *html*]);  
 Добавляет к таблице один столбец (тег <th>). *title* – содержимое тега <th> (заголовк); html – произвольный текст, который должен быть добавлен к тегу <th>. Серия вызовов table::th() должна быть произведена ДО начала заполнения таблицы.

**table::rowTh**(mixed *data*);  
 Добавляет все заголовки столбцов таблицы (равносильно вызову table::th() несколько раз). Параметр *data* может быть массивом строк (каждый элемент – текст заголовока) или строкой, содержащей текст всех заголовков, разделённой символом «|» например: «ИД|Название|Статус|Действие».

**table::trHtml**(string *html*);  
 Задаёт html-код, который должен быть дописан внутрь HTML-тега <tr> (например: «class="myClass"». Заданный текст будет добавляться к каждой строке.

**table::row**(string *text*);  
 Добавляет одну строку во всю ширину таблицы, *text* — текст (или HTML), отображаемый внутри <td>.

**table::table**(string *text*,[integer *colSpan*],[string *html*]);  
 Добавляет ячейку с произвольным текстом *text*. Параметр *colSpan* соответствует атрибуту colspan тега <td>. *Html* — произвольный код, который необходимо дописать к теку <td> (например класс CSS).

**table::link**(string *text*, string *link*, [integer *colSpan*],[string *html*]);  
 Добавляет ячейку, содержащую ссылку на какую либо страницу админки. Параметр *text* — заголовок ссылки, *link* — адрес ссылки; *colSpan* соответствует атрибуту colspan тега <td>. *Html* — произвольный код, который необходимо дописать к теку <td> (например класс CSS).

**table::upDown**(string *link*, integer *index*, integer *count*);  
 Добавляет ячейку, содержащую стрелки «вверх» и «вниз» для управления порядком отображения элементов. Параметр *link* — ссылка на страницу админки, к этому адресу будет добавлено «up» (кнопка «вверх») или «down» (кнопка «вниз»), пример ссылки: «?controller=shop&id=7&action=category» (в конце будет приписано «up» или «down»). *index* — порядковый номер строки; *count* — кол-во строк. Последние два параметра позволяют скрыть кнопку «вверх» у первой строки и кнопку «вниз» у последней.

**table::delete**(string *link*,[string *confirm*]);  
 Добавляет ячейку с кнопкой «удалить». *link* — ссылка на страницу админки; *confirm* — текст, содержащий предупреждение удаления, по умолчанию «Подтвердите удаление».

**table::editDelete**(string *link*,[string *confirm*]);  
 Добавляет ячейку с кнопками «редактировать» и «удалить». *link* — ссылка на страницу админки, если в ссылке содержится слово «action», то к ней будет добавлено ключевое слово «Edit» или «Delete», в противном случае будет приписано «&action=edit» или «&action=delete»; *confirm* — текст, содержащий предупреждение удаления, по умолчанию «Подтвердите удаление».

**table::itemDelete**();  
 Тоже, что и table::editDelete(), но отличается ссылкой для кнопки «редактировать»: к указанной ссылке добавляется «Item» или «&action=item», в зависимости от того, присутствует ли в ссылке слово «action».

**table::render**();  
 Выводит HTML-представление таблицы.

1. Работа с электронной почтой (класс email)

Задачи этого класса тривиальны: отправка электронной почты при помощи PHP-функции mail() или методом SMTP. Метод отправки, а также параметры SMTP задаются в «общих настройках».

Текст сообщения может быть задан шаблоном: указывается имя файла шаблона, а также перечень данных для подстановки. Шаблон письма — это HTML-файл, который содержит специальную разметку — теги вида «{{метка}}», вместо которых будут подставлены реальные данные.

Сам файл шаблона — это файл с расширением «.html», который должен быть расположен в директории */data/email*, если класс используется в общедоступной части сайта или в директории */admin/data/email*, если класс используется в админке.

Реализацию класса можно найти в файле */core/email.php*.

Вот типичный пример использования класса:

public function actionTestSubmit($data) { //submit-действие  
 core::import('core/email');  
 $e=new email(); //создание экземпляра класса  
 $cfg=core::config();  
 $e->from($cfg['adminEmailEmail'],$cfg['adminEmailName']); //«от кого»  
 $e->subject('тема письма');  
 $e->messageTemplate('templateName',array(  
 'name'=>$data['name'],  
 'email'=>$data['email'],  
 'message'=>$data['message']  
 ));  
 $e->send($data['email']); //отправить письмо адресату  
}

* 1. Методы класса

**email::from**(string *email*,[string *name*]);  
 Устанавливает поле «от кого»: *email* — адрес электронной почты, *name* (не обязательно) — имя отправителя.

**email::subject**(string *value*);  
 Задаёт тему письма.

**email::message**(string *message*);  
 Устанавливает текст сообщения (в формате HTML), альтернатива email::messageTemplate().

**email::messageTemplate**(string *fileName*, array *data*);  
 Устанавливает текст сообщения из шаблона, альтернатива email::message(). Имя файла шаблона задаётся в параметре *fileName* (путь и расширение файла не указывать). Параметр *data* — это ассоциативный массив, содержащий данные для подстановки в шаблон, где ключи соответствуют именам меток в шаблоне.

**email::returnPath**(string *email*);  
 Задаёт адрес электронной почты, на который должно вернуться письмо в случае, если его доставить не удалось.

**email::replyTo**(string *email*, [string *name*]);  
 Задаёт адрес электронной почты (а также имя получателя), на который необходимо пересылать ответы на письмо.

**email::attach**(string *filename*, string *id*, [string *type*]);  
 Добавляет к письму вложенный файл *filename*. Имя файла должно содержать полный путь. Параметр *id* — это имя файла внутри письма, если MIME-тип в параметре *type* не задан, то будет установлен «application/octet-stream».

**email::attachImage**(string *filename*, [string *type*]);  
 Добавляет к письму вложенное изображение. Имя файла должно содержать полный путь. Параметр *type* — это MIME-тип изображения (допустимо также указывать расширение имени файла рисунка), если не задан, то он будет определён автоматически исходя из расширения файла (параметр нужен для тех случаев, когда имя файла не содержит расширение, например, если файл находится в $\_FILES).

Функция возвращает идентификатор, по которому можно вставить изображение в текст сообщения.

**email::getImg**(string *id*);  
 Возвращает строку, содержащую HTML-тег <img>. *id* — идентификатор, полученный функцией email::attachImage().

**email::send**(string *email*);  
 Отправляет письмо на указанный адрес электронной почты. Возможно последовательно вызвать эту функцию несколько раз, чтобы отослать несколько копий писем.

1. Работа с изображениями (класс picture)

Предназначен для обработки изображений. В «арсенал» класса входит всего два полезных инструмента: масштабирование и наложение водного знака.

Реализацию класса можно найти в файле */core/picture.php*.

Работа происходит следующим образом: сначала загружается исходное изображение, затем задаются параметры обработки и затем вызывается процедура *save()*, которая выполняет все необходимые действия и сохраняет новый файл. После этого можно назначить новые правила обработки и вновь вызвать метод *save()*, которых сохранит другую копию обраотанного изображения.

Типичный алгоритм использования класса:

core::import('picture');  
$picture=new picture('myfile.jpg'); //загрузить исходный файл  
if(controller::$error) return false; //если файл не является рисунком, то будет ошибка  
$picture->resize(400,'<800'); //изменение размеров  
if(!$picture->save(core::path().'public/myfile')) return false; //сохранить изображение в файл.

* 1. Методы класса

**picture::\_\_controller**(string *fileName*, [string *type*]);  
 Конструктор класса. *fileName* — имя файла (должно включать полный путь); *type* — MIME-тип изображения (допустимо указывать расширение имя файла). MIME-тип необходимо указывать в тех случаях, когда *fileName* не содержит расширения файла и нет возможности определить его тип.

Если файл не является корректным изображением, то переменная controller::$error будет содержать текст сообщения об ошибке.

**picture::resize**([mixed *width*], [mixed *height*]);  
 Изменяет размеры исходного изображения. Если один из параметров (ширина или высота) не задан, то он будет рассчитан по второму параметру пропорционально исходному размеру. *width* и *height* могут быть целым числом (точный размер в пикселях), или строкой, состоящей из символа «>» (больше) или «<» (меньше) и следующего за ним числа. Во втором случае размеры будут пропорциональны исходным, но «не более» или «не менее» заданного.

Для наглядности примеры размеров:

|  |  |  |
| --- | --- | --- |
| **Width** | **Height** | **Результат** |
| 400 | null | Ширина — 400 пикселей, высота — пропорционально исходной |
| 400 | 300 | Ширина — 400 пикселей, высота — 300 пикселей |
| <600 | null | Ширина соответствует исходной, если она меньше 600 пикселей или равна 600 пикселей, если исходная ширина больше. Высота пропорциональна исходной. |
| <800 | <600 | Изображением «вписывается» (с сохранением пропорций) в размер 800x600, если по одной или обоим граням оно больше заданного. |
| null | >300 | Высота будет растянута до 300 пикселей, если исходная меньше. Ширина пропорциональна исходной. |

**picture::watermark**(string *fileName*, mixed *x*, mixed *y*);  
 Накладывает водный знак на изображение. *fiileName* — имя файла изображения, содержащего водный знак. Параметры *x* и *y* задают координаты расположения водного знака: целые положительные числа — отсутп в пикселях от левого и верхнего края, отрицательные целые числа — отсуп в пикселях от правого и нижнего края изображения, строка, состоящая из числа и знака «%» - задаёт отступ в процентах от размеров изображений (например picture::watermark('myfile.png','50%','50%') расположит водный знак точно по центру изображения).

**picture::save**(string *fileName*, [integer *quality*], [string *type*]);  
 Сохраняет обработанное изображение в файл. Параметр *fileName* (имя файла) необходимо указывать БЕЗ РАСШИРЕНИЯ. *quality* — качество изображения для формата JPEG (число от 1 до 100). *type* — MIME-тип (допускается указывать расширение файла), если не указан, то будет сохранён в том же формате, что и исходный файл.

Функция picture::save() возвращает расширение сохранённого файла или **false** в случае неудачи.

Реализация picture::save() и необходимость дополнительно указывать MIME-тип необходима для того, чтобы упростить работу с изображениями, находящимися в $\_FILES, когда имя файла не содержит расширения. Однако реализация получилась не слишком удачной и, вероятно, этот класс будет доработан в будущем.