



Создание web-приложений,
исполняемых на стороне сервера при помощи
языка программирования PHP
и технологии AJAX

Урок №5

Паттерн MVC. Фреймворк Codeigniter

Содержание

Введение	4
Паттерн MVC	5
Контроллер	6
Модель	6
Представление	7
Назначение паттерна MVC	8
Фреймворк Codeigniter.....	10
Создание приложения в Codeigniter.....	11
Конфигурирование приложения.....	14
Создание контроллера	17

Материалы к уроку прикреплены к данному PDF-файлу.

Для доступа к материалам необходимо открыть урок в программе
[Adobe Acrobat Reader](#).

Dependency Injection (инъекция зависимости) . . .	25
Создание модели	29
Организация представлений	33
Codeigniter и Bootstrap	35
Работа с формами	38
Улучшенная форма	46
Upload в Codeigniter	50
Сессии в Codeigniter	64
Валидация данных форм в Codeigniter	67
Использование Query Builder	76
Изменение адресной строки	82
Использование сторонних библиотек	84
Домашнее задание	87

Введение

Нам предстоит ознакомиться с новым, более эффективным способом использования языка РНР. То, что мы делали до сих пор, можно назвать использованием РНР в чистом виде. Вы уже понимаете возможности этого языка и рассмотрели большинство способов его применения. С другой стороны, для вас не секрет, что от разработчиков сегодня требуется создавать все более сложные приложения и, при этом, за более короткие сроки. Как можно удовлетворять постоянно возрастающие требования к разработке приложений? В ходе изучения этого урока вы узнаете ответ на этот вопрос.

Паттерн MVC

Безусловно, каждый из вас уже привык к тому, что во многих приложениях часто надо реализовывать одни и те же действия – возможно, с некоторой поправкой на каждый конкретный случай. Однако бывают подобные сходства более высокого уровня, имеющие отношение не к конкретным действиям того или иного приложения, а к поведению всего приложения в целом.

Давайте рассмотрим поведение типичного веб-приложения, использующего БД. Как выглядит работа с ним? Ее можно описать таким алгоритмом:

- пользователь активирует требуемое действие;
- для выполнения этого действия приложение отправляет запрос к БД;
- приложение получает ответ от БД и показывает пользователю ответ.

Такой цикл повторяется много раз, для выполнения каждого действия пользователя. Для унификации этого цикла существует архитектурный паттерн MVC (Model-View-Controller). Не сегодняшний день этот паттерн фактически является стандартом в веб-разработке, поэтому вы должны научиться свободно работать с ним. Возможно, вам интересно будет узнать, что этот паттерн появился и получил теоретическое обоснование еще в 1979 году, когда самого веба еще не существовало. Однако «звездный час» MVC наступил только во времена распространения веб-разработки.

Рассмотрим составные части паттерна. Компоненты Model и Controller в нем являются классами. Компонента View является веб-страницей, которую пользователь получает в клиентской области своего браузера. В одном приложении может быть произвольное количество контроллеров, моделей и представлений. Поговорим подробнее об этих трех компонентах.

Контроллер

Как мы уже отметили, контроллер является классом. Этот класс содержит в себе набор всех действий, которые пользователь может выполнять в веб-приложении. Такие действия в классе контроллера представляются методами. Другими словами, каждому пункту меню, каждой ссылке и кнопке, которые активируют какое-либо действие, соответствует свой метод в контроллере. Поэтому методы контроллера часто называют действиями (actions).

Задача контроллера заключается в том, чтобы принимать команды пользователя, определенным образом выполнять их и возвращать пользователю в браузер результат выполнения этих команд. Отметьте на этом этапе, что контроллер выполняет команды пользователя неким специальным способом, который мы рассмотрим позже.

Модель

Для выполнения команды пользователя контроллер обращается к модели. Модель тоже является классом. Этот класс инкапсулирует работу с источником дан-

ных, чаще всего – с БД. В паттерне MVC доступ к базе данных может иметь только модель. Именно модель отвечает за доступ к данным и за способы обработки этих данных. Ни контроллер, ни представление не должны иметь доступа к БД.

Для выполнения работы с базой данных класс модели содержит в себе набор методов. Каждый из таких методов модели выполняет какое-либо фиксированное действие с БД. Когда контроллер получает от пользователя очередную команду, активируется соответствующий метод контроллера. Этот метод контроллера, в свою очередь, активирует метод модели, который сможет выполнить команду пользователя.

Задача модели заключается в том, чтобы выполнять запросы к БД, получать результаты выполнения этих запросов и возвращать полученные результаты контроллеру. Отметим, что модель взаимодействует только с БД и с контроллером. Модель не имеет никакого доступа к представлениям.

Представление

Представление является веб-страницей. Задача представления заключается в том, чтобы отобразить в браузере пользователя данные, являющиеся результатом обработки команды пользователя. Можно рассматривать представление как некий шаблон. Одно представление выведет данные в виде таблицы, другое – в виде списка, третье просто отобразит текстовое сообщение. Представление может общаться только с контроллером и ничего не знает о модели и БД. Запомните,

что представление не является «привязанным» к одним и тем же данным. Еще раз повторю – представление это шаблон. В какой-то момент представление может отображать, например, список книг, поступивших в библиотеку за последние полгода. А затем это же представление покажет все книги выбранного писателя. А потом это представление сможет показать еще какой-то набор данных. Во всех этих случаях разные данные будут отображаться в этом представлении одинаково – например, в виде списка.

Назначение паттерна MVC

После начального знакомства с паттерном MVC вам должно быть понятно схематичное изображение взаимодействия между компонентами этого паттерна.

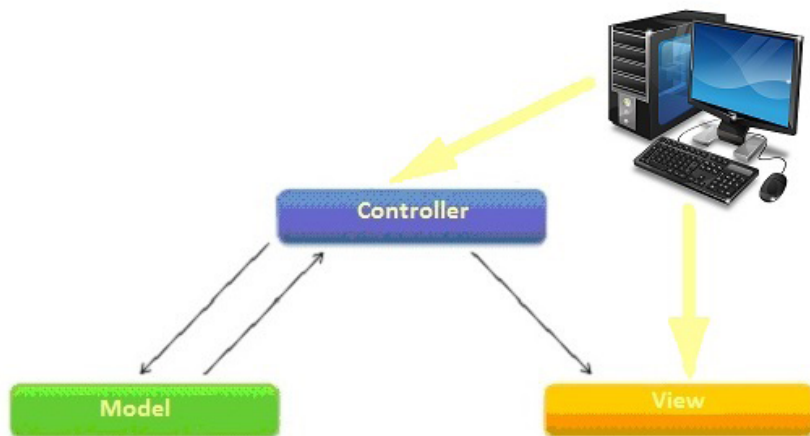


Рис. 1. Схема взаимодействия паттерна MVC

Какие задачи решает этот паттерн, и почему он завоевал всеобщее признание? Применение архитектуры MVC в приложении позволяет отделить бизнес-логику приложения от ее визуализации. Это особенно важно в тех случаях, когда вам необходимо представлять одни и те же данные в разном виде. Применение MVC повышает повторное использование кода, исключая дублирование и сокращая общий объем кода, и делает приложение более масштабируемым.

Отделение визуализации данных особенно актуально сегодня, когда одно приложение должно уметь обслуживать клиентов, использующих настольные компьютеры и разные мобильные устройства (Android, iOS или Windows Phone). И в этом случае MVC оказывается очень полезным. Вы увидите все достоинства паттерна MVC в ходе дальнейшего изучения этого урока.

Фреймворк Codeigniter

Сегодня очень широко распространено применение в разработке программных продуктов различных фреймворков. Фреймворк – это программный продукт, который используется для формализации и частичной автоматизации процесса разработки приложений. Использование фреймворка накладывает определенные ограничения на архитектуру разрабатываемого приложения, задавая некий его каркас и предлагая пользователю придерживаться ряда договоренностей. Взамен фреймворк избавляет разработчика от ряда рутинных действий и создает приложение с унифицированной структурой, упрощая его сопровождение. Особенно широко фреймворки используются при разработке веб-приложений. Существует много фреймворков для РНР. Мы с вами начнем учиться пользоваться этими инструментами на примере фреймворка Codeigniter. Сначала мы рассмотрим создание нового приложения с помощью Codeigniter. Потом познакомимся с компонентами, из которых будет состоять приложение: с контроллерами, моделями и представлениями. И, наконец, рассмотрим разные аспекты использования Codeigniter.

Почему мы выбрали для изучения именно этот фреймворк? Часто можно слышать, что он уже устарел, поскольку вышел на рынок еще в 2006. Однако, Codeigniter постоянно обновляется и совершенствуется, поэтому его возраст это не недостаток, а достоинство. Расмус Лерддорф, создатель РНР, однажды сказал, что он

любит Codeigniter, потому что он быстрый и легкий – because it is faster, lighter and the least like a framework. Еще надо упомянуть о том, что Codeigniter очень простой в изучении и в использовании. Вряд ли, найдется другой фреймворк, способный сравниться с ним в простоте. Codeigniter очень хорошо описан на сайте разработчика https://www.codeigniter.com/user_guide/, что тоже является важным плюсом. И, наконец, этот фреймворк является отличным ключом к пониманию MVC.

Создание приложения в Codeigniter

Мы уже говорили о том, что каждый фреймворк накладывает на архитектуру создаваемого приложения ряд своих ограничений. Одно из таких ограничений – это структура папок, в которых должно размещаться приложение. Codeigniter предлагает свою структуру папок.

Создание каждого нового приложения под управлением Codeigniter начинается с извлечения из архива фреймворка заготовки нового приложения. Сам архив можно скачать со страницы разработчика https://www.codeigniter.com/user_guide/installation/downloads.html. Мы будем использовать последнюю на сегодня версию – v3.1.2. Созданное нами первое приложение не будет содержать какую-либо пользовательскую логику. Оно будет служить макетом для создания контроллеров, моделей и представлений, будет демонстрировать работу с БД, использование сессий, валидацию и другие технические приемы. В качестве тестовой БД для этого приложения будем использовать базу данных нашего интернет-магазина.

Распакуем архив CodeIgniter-3.1.2.zip в корневую папку OpenServer и переименуем полученную папку CodeIgniter-3.1.2 в appсi1. В папке appсi1 вы должны увидеть такое содержимое:

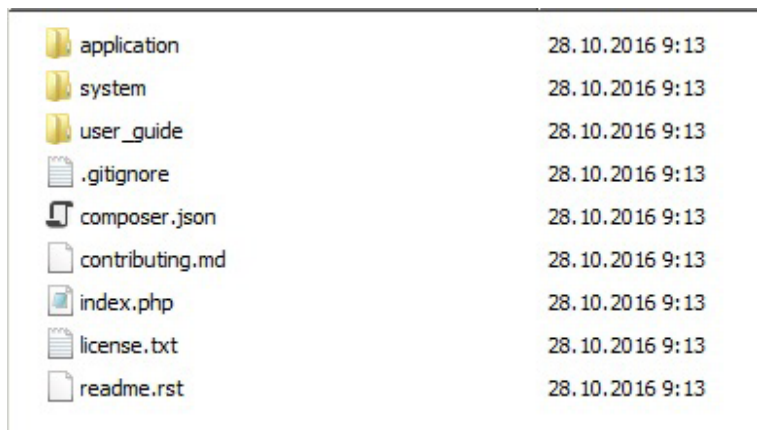


Рис. 2. Структура приложения Codeigniter

В папке system располагается сам фреймворк: его классы, библиотеки, модули и прочее. В этой папке мы работать не будем, не будем ничего здесь изменять, добавлять или удалять. В папке user_guide находится руководство пользователя. И, наконец, в папке application располагается наше приложение. Именно в этой папке мы и будем работать. Откройте эту папку:



cache	28.10.2016 9:13
config	28.10.2016 9:13
controllers	28.10.2016 9:13
core	28.10.2016 9:13
helpers	28.10.2016 9:13
hooks	28.10.2016 9:13
language	28.10.2016 9:13
libraries	28.10.2016 9:13
logs	28.10.2016 9:13
models	28.10.2016 9:13
third_party	28.10.2016 9:13
views	28.10.2016 9:13
.htaccess	28.10.2016 9:13
index.html	28.10.2016 9:13

Рис. 3. Папка application

Вы сразу должны заметить три папки с уже знакомыми вам именами: `controllers`, `models` и `views`. Фреймворк Codeigniter создает приложения в соответствии с паттерном MVC, поэтому в этих папках мы будем создавать для нашего приложения контроллеры, модели и представления. С назначением остальных папок познакомимся далее.

После распаковки архива необходимо выполнить начальную настройку приложения. Конфигурационные файлы располагаются в папке `config`. Каждый конфигурационный файл является текстовым файлом, в котором выполняется инициализация специального конфигурационного массива. Рассмотрим обязательную начальную конфигурацию приложения.

Конфигурирование приложения

Перейдем в папку `config`. В этой папке собраны файлы, которые позволяют настроить конфигурацию создаваемого приложения. Другими словами, изменяя файлы в этой папке, мы можем указать, какие ресурсы Codeigniter мы хотим загружать при запуске приложения. Дело в том, что разработчики Codeigniter постарались сделать фреймворк максимально облегченным. Поэтому по умолчанию он активирует только самые необходимые из своих ресурсов. Загрузкой остальных должны управлять мы. Рассмотрим этот процесс.

В данном случае словом «ресурс» названы такие вещи, как библиотеки, хелперы, пакеты и др. Каждый конфигурационный файл в этой папке отвечает за настройку какой-либо из различных опций: за работу с БД, за обработку адресной строки и т.п. Настройка в каждом файле сводится к инициализации элементов определенного массива.

Откройте в блокноте файл `autoload.php`. В этом файле мы работаем с массивом с именем `$autoload`. В строке 42 файла `autoload.php` расположено выражение, отвечающее за загрузку пакетов:

```
$autoload['packages'] = array();
```

Если в правой стороне выражения стоит пустой массив (`array()`), значит, никакие пакеты не подключаются. Если в `array()` указаны какие-то имена, значит, подгружаются пакеты с указанными именами. В данном случае никакие пакеты не загружаются. Нам в этом

приложении пакеты не будут нужны, поэтому оставляем эту строку без изменения.

В строке 55 файла `autoload.php` можно выполнить загрузку библиотек. Нам потребуются библиотеки для работы с БД и с сессиями, поэтому строку 55 надо привести к такому виду:

```
$autoload['libraries'] = array('database', 'session');
```

В строке 67 подключаются хелперы – это специальные классы, инкапсулирующие работу определенного направления (создание форм, обработка адресной строки и др.). Хелперы являются очень полезными инструментами, и мы познакомимся с ними в ходе работы с этим уроком. Строку 67 надо привести к такому виду:

```
$autoload['helper'] = array('form', 'url');
```

Мы подключили в наше приложение хелперы для работы с формами и с адресными строками. На этом настройка в файле `autoload.php` завершена. Перейдем к следующему файлу.

Откройте файл `routes.php`. В нем выполняется инициализация массива `$route`, отвечающего за обработку путей в приложении. В строке 41 этого файла надо указать имя контроллера по умолчанию в нашем приложении.

```
$route['default_controller'] = 'home';
```

Мы указали, что контроллером по умолчанию в нашем приложении будет контроллер с именем `Home`. В каждом приложении можно объявить один из кон-

троллеров контроллером по умолчанию. Имя такого контроллера можно не указывать в адресной строке. Мы поговорим об этом подробнее позже. Больше в этом файле изменять ничего не надо.

Теперь откроем файл `config.php` и приведем в нем строку 26 к такому виду:

```
$config['base_url'] = 'http://localhost/appci1/';
```

Здесь мы указали базовый адрес нашего приложения. Этот адрес будет использоваться функциями, выполняющими обращения к разным частям приложения.

Следующим открываем файл `database.php`. Вы уже догадались, что в этом файле содержится настройка для подключения к БД, которая содержится в массиве `$db`. Измените строки 51-54, указав в них реквизиты для подключения к своей БД:

```
$db['default']['hostname'] = 'localhost';  
$db['default']['username'] = 'root';  
$db['default']['password'] = '123456';  
$db['default']['database'] = 'shop';
```

Обратите внимание, что значение элемента `$db['default']['dbdriver']` по умолчанию равно `'mysqli'`. Это означает, что приложение настроено на работу с СУБД MySQL. Если надо будет использовать другую СУБД, то это значение надо будет изменить.

На этом начальные настройки приложения закончены. Если понадобится изменить какую-либо из настроек, это можно будет сделать позже.

Создание контроллера

Контроллер является центральным элементом MVC приложения, связывающим в единое целое все три компоненты. Поэтому мы начнем разработку нашего приложения с создания контроллера. Выполняя конфигурацию приложения, мы указали в файле `routes.php`, что наш контроллер будет называться `Home`. Вот его мы сейчас и создадим. Контроллеры располагаются в папке `application/controllers`, поэтому перейдем в эту папку. Там уже есть два файла, мы пока не будем их трогать. Создайте рядом с ними файл с именем `Home.php`. Обратите внимание: в имени файла обязательно первый символ должен быть в верхнем регистре, а все остальные – в нижнем. Больше ни одного символа в верхнем регистре! Это требование Codeigniter, начиная с версии 3. Раньше (для версии 2 и ниже) имя файла обязательно должно было состоять только из символов в нижнем регистре. Запомните эту особенность. Это – одно из требований фреймворка. В дальнейшем мы встретим еще ряд подобных требований, и будем их все соблюдать. Это плата за использование фреймворка. Однако, надо понимать, что такие требования могут изменяться от версии к версии и относиться к этому спокойно.

Откроем созданный файл и начнем создавать в нем наш контроллер. Добавьте в `Home.php` такой код:

```
<?php

class Home extends CI_Controller
{
    public function index()
    {
        echo 'Hello World!';
    }
}
```

Мы создали класс с именем `Home`, в точности (до регистра) совпадающим с именем файла. Это имя нашего контроллера. Обратите внимание, что наш класс наследует классу с именем `CI_Controller`. Этот класс создан разработчиками Codeigniter и будет базовым для всех наших контроллеров, передавая по наследству в них свою функциональность. Мы создали в своем классе метод `index()`. Запомните, что `index()` – это имя метода по умолчанию в любом контроллере. Создавать метод с именем `index()` в каждом контроллере совсем не обязательно, но присутствие такого метода – это хорошая практика. Наш метод `index()` пока очень прост, но постепенно мы сделаем его более интересным.

Проверим созданный контроллер в действии, но перед этим поговорим об адресных строках, используемых в Codeigniter. Вы уже много раз активировали созданные ресурсы в браузере. При этом всякий раз вы вводили такой адрес:

```
localhost/directory/file.php
```

Здесь:

localhost – адрес веб-сервера;

directory/ – одна или несколько папок в корневой папке веб-сервера;

file.php – имя активируемого файла.

Теперь вам надо привыкать к другой структуре адресной строки. Codeigniter рассматривает адресную строку так:

```
domain/[controller]/[controller-method]/[arguments]
```

Здесь:

domain – домен приложения;

controller/ – имя активируемого контроллера;

controller-method – имя метода контроллера;

arguments – параметры для метода.

Обратите внимание на прямоугольные скобки вокруг имени контроллера, метода и параметров. Эти скобки означают, что атрибут не обязательный и может отсутствовать в адресной строке. Для нашего примера доменом приложения будет такая конструкция – <http://localhost/CI/index.php>. Именем контроллера будет Home, именем метода – index, параметров не будет. Собрав все это вместе, мы получим такую адресную строку, которую надо ввести в браузере: <http://localhost/CI/index.php/Home/index>

Вы получите в браузере вывод нашего метода index() – Hello World! Такой же вывод вы получите, если не укажете имя метода, или имя метода и контроллера, по-

сколько у нас Home – имя по умолчанию, и index() – имя по умолчанию.

Конечно же, такая адресная строка является непривычной, но скоро мы ее оптимизируем и уберем из нее имя главной страницы index.php. А потом вы привыкнете и убедитесь, что такой способ адресации очень удобен. Забегая наперед, скажу, что такая система адресации присуща не только Codeigniter, и на сегодня она более распространена, чем «классическая», с которой мы работали до этого. Кроме того, на такой адресации построена важная технология REST, с которой мы познакомимся позже.

Вернемся к нашему контроллеру. Что мы увидели? Мы увидели, что активируемый метод контроллера вывел в браузер строку с помощью конструкции echo. Однако в реальных приложениях методы контроллера не будут выводить в браузер что-либо с помощью конструкции echo. Одна из задач контроллера – выводить в браузер представление. Посмотрим, как это можно сделать. Для начала нам надо создать представление. Представления нашего приложения должны находиться в папке application/views. Создайте в этой папке файл с именем page1.php. Это и будет наше первое представление. Занесите в этот файл такую разметку:

```
<h2>Page1</h2>
<p>This is view page1 brought to you by Home controller</p>
```

Теперь нам надо изменить метод index() в нашем контроллере, чтобы он выводил нам это представление. Приведите этот метод к такому виду:

```
public function index()  
{  
    $this->load->view('page1');  
}
```

Разберем, что выполняет приведенная строка. Вы, конечно же, узнали указатель `$this`. С помощью этого указателя мы обращаемся в классах к членам класса. В этом случае – к свойству нашего класса с именем `load`. Откуда это свойство появилось в нашем классе? Мы наследовали его из базового класса `CI_Controller`. У свойства `load`, которое в свою очередь является объектом, есть методы, позволяющие активировать разные объекты. Сейчас мы вызвали метод `view()`, который, как не сложно догадаться, предназначен для вызова представлений. Обратите внимание, что метод `view()` требует при вызове, чтобы ему передали в строковом виде имя того представления, которое надо вывести в браузер. Мы указываем только имя файла, в котором находится представление, и не указываем расширение и путь к файлу. Метод `view()` знает, что представления находятся в папке `application/views` и имеют расширения `.php`. Активируйте метод `index()` снова, и вы увидите в браузере наше представление. Итак, запомните – для вызова представления в методе контроллера надо обратиться к методу `view()` свойства `load`. Мы часто будем использовать свойство `load` и будем вызывать с его помощью не только представления.

Я не думаю, что на вас произвел впечатление такой вызов представления. Мы просто вызвали статическую страницу. Однако контроллер может не только вызы-

вать представления, но и при вызове передавать им данные. Посмотрим, как это происходит.

Прежде чем мы перейдем к кодированию, отметьте, что из контроллера в представление можно передавать произвольное количество данных самой разной природы (строки, массивы, объекты). При передаче все такие данные должны упаковываться в один ассоциативный массив. Названия ключей этого массива будут использоваться в представлении для извлечения упакованных в контроллере данных. Теперь посмотрим на новый вид метода `index()`:

```
public function index()
{
    $data['title']='Page1';
    $data['text']='This text was send from Home
                                controller';
    $data['countries']=array('Argentina',
                            'Belgium','Canada','Great Britain');
    $this->load->view('page1', $data);
}
```

Мы создали массив с именем `$data` и добавили в него три произвольных элемента – две строки и массив. Имя самого массива `$data` и имена его индексов тоже произвольны. Обратите внимание, что теперь при вызове метода `view()` мы передаем наши данные во втором параметре. Если вам кажется, что метод `view()` перегруженный – это не так. Просто у его параметров есть значения по умолчанию, и мы можем не указывать некоторые из них при вызове.

Теперь надо изменить и представление, чтобы оно

могло обработать присланные данные. Приведите представление `page1` к такому виду:

```
<?php
echo '<h2>'.$title.'</h2>';
echo '<p>'.$text.'</p>';
echo '<ul>';
    foreach($countries as $c)
    {
        echo '<li>'.$c.'</li>';
    }
echo '</ul>';
```

Вы должны узнать в этом коде обращение к индексам, которые мы создали в контроллере. Самого массива `$data` в представлении нет, но его элементы превращаются в самостоятельные переменные, имена которых совпадают с именами их индексов в массиве `$data`.

Активируйте метод `index()` снова и вы увидите в браузере такой вывод:

Page1

This text was send from Home controller

- Argentina
- Belgium
- Canada
- Great Britain

Снова запомните: для передачи данных из метода контроллера в представление эти данные надо упаковать в ассоциативный массив и передать этот массив методу `view()` свойства `load` во втором параметре, после имени представления. Даже если вы хотите передать в

представление одну переменную, ее тоже надо упаковать в массив.

Помните, мы говорили о том, что представление – это шаблон для вывода данных? Чтобы было понятнее, что это значит, используем наше представление для вывода каких-нибудь других данных. Приведите код метода `index()` к такому виду:

```
public function index()
{
    $data['title']='Page1';
    $data['text']='The books of Clifford Donald
                                     Simak';
    $data['countries']=array('Ring Around the Sun',
    'Time is the Simplest Thing', 'Way Station', 'All
                                     Flesh Is Grass',
    'The Goblin Reservation', "The Fellowship of the
                                     Talisman");
    $this->load->view('page1', $data);
}
```

Активируйте метод `index()` снова и вы увидите в браузере такой вывод:

Page1

The books of Clifford Donald Simak

- Ring Around the Sun
- Time is the Simplest Thing
- Way Station
- All Flesh Is Grass
- The Goblin Reservation
- The Fellowship of the Talisman

Обратите внимание, что в представлении мы ничего не изменили, оно по-прежнему выводит один заголовок, текстовую строку и список, но содержание данных теперь другое.

Конечно же, в приложении мы не будем вводить данные в методе контроллера так, как мы сделали в этом примере. Данные, как правило, будут выбираться из БД. А для того, чтобы работать с БД, паттерн MVC рекомендует использовать модель. Перейдем к знакомству с моделями.

Dependency Injection (инъекция зависимости)

Прежде чем двигаться дальше, нам надо поговорить еще об одном паттерне, который называется Инъекция Зависимости (Dependency Injection или просто DI). Это очень простой и очень распространенный паттерн, но часто его используют неправильно. Поэтому будет полезно уделить его рассмотрению несколько минут. Скоро вы увидите, что этот паттерн используется в Codeigniter при взаимодействии контроллера с моделью.

Мы уже говорили о том, что методы контроллера должны вызывать методы модели. Чтобы вызвать метод какого-либо класса, надо иметь в своем распоряжении объект этого класса. Значит, чтобы в классе контроллера была возможность вызывать методы модели, надо иметь в контроллере объект модели. За передачу объекта одного класса в другой класс и отвечает паттерн DI.

Рассмотрим такой пример. Для вашего сайта важно подробно расписать работу с пользователями. Пользователи распределяются по ролям. У каждой роли своя

функциональность: разные правила начисления скидок, доступ к разным способам доставки и т.п. Вы создали класс `Role`, описывающий роли и их функциональность:

```
class Role
{
    function __construct($role='Customer') {...}
    function getInfo() {...}
    function setDiscount(...) {...}
}
```

Такое схематичное обозначение класса должно быть понятным.

Предположим, мы решили, что будет полезно хранить информацию о роли в описании пользователя, чтобы было проще использовать функциональность класса `Role`:

```
class User
{
    private $role;
}
```

Свойство `$role` является объектом класса `Role`. Работая с пользователем, мы сможем легко получать доступ к методам класса `Role`. Давайте позаботимся об инициализации свойства `$role` в классе `User`. Это можно сделать разными способами.

Например, так:

```
class User
{
    private $role;
    function __construct()
    {
        $this->role=new Role();
    }
}
```

Или так:

```
class User
{
    private $role;
    function __construct($rolename)
    {
        $this->role=new Role($rolename);
    }
}
```

Или так:

```
class User
{
    private $role;
    function __construct()
    {
        $this->role=new Role('Newcomer');
    }
}
```

Все приведенные выше способы являются плохими. Использование литерала является плохим способом потому, что лишает код гибкости. Использование параметра конструктора класса User также является плохой практикой – потому, что привлекает в класс

несвойственные ему данные, нарушая правила ООП. И вообще, создание объекта класса Role внутри класса User может привести к значительным неудобствам. Например, при необходимости изменения конструктора класса Role надо будет изменять и код класса User. Как инициализировать свойство \$role правильно?

Вот здесь на помощь приходит паттерн DI. Вместо того, чтобы создавать объект класса Role внутри класса User, этот паттерн предлагает внедрить объект извне – например, через параметр конструктора:

```
private $role;  
function __construct($role)  
{  
    $this->role=$role;  
}
```

Вот это и есть инъекция зависимости! Такой код является более гибким с точки зрения сопровождения. При необходимости изменения класса Role не требуется изменять код класса User. Создание объекта пользователя теперь может выглядеть так:

```
$role=new Role('Admin');  
$user=new User($role);
```

Внедрять объект одного класса в другой можно не только через конструктор. Можно создать сеттер, можно использовать public свойство или как-нибудь еще. Все это определяется в каждом конкретном случае.

Почему мы отвлеклись на рассмотрение этого паттерна? Потому что он используется в Codeigniter для доступа из контроллера к методам модели.

Возможно, кто-то из вас давно использует такой прием в своих приложениях и даже не подозревал о том, что это – паттерн. Теперь вы будете знать, что в таких ситуациях пользуетесь паттерном Dependency Injection. И при этом будете пользоваться этим паттерном правильно – не создавая объект внедряемого класса в своем классе. Остальным надо запомнить этот пример и использовать его, когда в этом возникнет необходимость. Теперь возвращаемся к нашей модели.

Создание модели

Повторим, что мы уже знаем о модели. Модель – это класс, предназначенный для работы с БД. В этом классе собраны методы, выполняющие запросы к БД. Эти методы будут вызываться методами контроллера. И еще вспомним, что в этом примере мы планируем использовать БД shop, созданную для интернет-магазина. Перейдите в папку application/models, создайте там файл с именем Home_model.php и добавьте в него такой код:

```
<?php
class Home_model extends CI_Model
{
    public function __construct()
    {
        $this->load->database();
    }
}
```

Мы видим, что наш класс Home_model наследует системному классу CI_Model. Кроме этого, в модели мы явным образом прописали конструктор. Обратите внимание – в конструкторе мы обращаемся к уже известно-

му нам свойству `load` (правда, сейчас это свойство класса `CI_Model`, а в контроллере мы работали с таким свойством класса `CI_Controller`) и вызываем метод `database()`. Вызов этого метода делает возможным доступ к БД, при условии, что выполнены настройки в файле конфигурации `routes.php`. У нас настройки выполнены. Поскольку мы вызвали метод `database()` в конструкторе модели, то доступ к БД будет возможен из любого метода модели. Если бы мы не сделали этот вызов в конструкторе, нам надо было бы писать строку `$this->load->database()` в каждом методе модели, где нам надо обращаться к БД. Вызов метода `database()` делает возможным доступ в классе модели к свойству модели `$this->db`, через которое мы будем работать с базой данных.

Добавим в наш класс модели еще один метод, задача которого заключается в получении из БД списка всех товаров из таблицы `'Items'`:

```
public function getItems()
{
    $res = $this->db->get('items');
    return $res->result _array();
}
```

В этом методе мы использовали специальный модуль `Codeigniter` для работы с базами данных, называемый `Query Builder`. Подробнее мы поговорим о нем позже, а сейчас отметьте, что метод `get()` выполняет простой запрос `'SELECT * FROM'` из той таблицы, имя которой передается ему при вызове. В нашем случае это таблица `Items`. Запросы, созданные с помощью `Query Builder`, не привязаны к какой-то определенной СУБД.

При изменении в конфигурации настройки на подключение к другой СУБД, этот код в модели не надо будет редактировать. Конечно же, при условии, что в другой БД будут таблицы с такими же именами и полями. Кроме того, все данные используемые в запросах автоматически проходят валидацию.

Итак, мы забираем результат выполнения запроса `'SELECT * FROM Items'` в ресурс с именем `$res`. Затем от имени этого ресурса вызываем метод `result_array()`. Помните, раньше мы после получения ресурса выполняли цикл `while`, в котором либо вызывали функцию `mysql_fetch_array()`, либо, при работе с PDO – метод `fetch()`, для извлечения из ресурса строк данных? В Codeigniter такой цикл не нужен, всю работу цикла выполняет метод `result_array()`, который возвращает массив с извлеченными из ресурса строками. Каждый элемент массива, возвращенного методом `result_array()`, сам является массивом, а его элементы – полями прочитанной таблицы. Можно сказать, что `result_array()` возвращает массив массивов. Есть еще подобный метод `result()`, который извлекает из ресурса и возвращает массив объектов. В этом случае каждая прочитанная строка является объектом, а поля таблицы – свойствами такого объекта.

Итак, что мы сделали? Мы создали модель, открыли в модели возможность для работы с БД и написали метод `getItems()`, возвращающий описание всех товаров из таблицы `Items`. Кто должен вызывать метод `getItems()`? В архитектуре MVC методы модели должны вызываться методами контроллера. Вернемся в наш контроллер и создадим в нем новый метод, чтобы он вызывал метод

getItems() модели и передавал полученные из модели данные в представление.

Нам надо в методе контроллера выполнить вызов метода модели. Сейчас вы увидите, как реализуется DI в классе контроллера. Добавьте в наш контроллер такой конструктор:

```
public function __construct()
{
    parent::__construct();
    $this->load->model('home _ model');
}
```

Выделенная строка как раз и реализует в контроллере внедрение зависимости, делая доступным объект модели в виде свойства с именем конкретной модели, в нашем случае – свойства с именем \$home_model. От имени этого свойства мы будем вызывать в контроллере все необходимые нам методы модели. Теперь добавьте в контроллер метод, который будет получать данные от модели и передавать их в представление:

```
public function ItemsList()
{
    $data['title']='List Of Items';
    $data['items']=$this->home _ model->getItems();
    $this->load->view('items',$data);
}
```

В этом методе вам все должно быть понятно. Мы создаем заголовок страницы в элементе с индексом title, а в элемент с индексом items мы заносим данные, полученные из модели. Обратите внимание, как мы обращаемся к методу модели getItems() – от имени свойства

`$home_model`, инициализированного в конструкторе контроллера.

Добавьте в папку `application/views` новый файл `items.php` с таким содержанием:

```
<?php
echo '<h2>'.$title.'</h2>';
echo '<table>';
foreach($items as $c)
{
    echo '<tr>';
    echo '<td>'.$c['itemname'].'</td>';
    echo '<td>'.$c['pricein'].'</td>';
    echo '<td>'.$c['pricesale'].'</td>';
    echo '<td>'.$c['info'].'</td>';
    echo '</tr>';
}
echo '</table>';
```

Занесите в браузер адрес <http://localhost/appcil/index.php/home/ItemsList>, чтобы проверить работу с моделью. Если вы все сделали правильно, то в браузере увидите таблицу со своими товарами из таблицы `Items`. Согласитесь, что выглядит результат работы нашего метода не очень привлекательно. Это объясняется тем, что мы пока не уделяли внимания оформлению и в наших представлениях совсем не используются стили. Исправим эту ситуацию.

Организация представлений

Рассмотрим способ эффективного использования представлений. До этого момента мы использовали наши представления разрозненно, независимо друг от друга и совсем без оформления. Однако в реальном

приложении нам надо будет применять оформление ко всем нашим страницам и сделать использование этого оформления удобным. Надо будет создать меню и повторять его на многих страницах. Сейчас мы выполним эту работу.

Добавьте в папку `application/views` два файла представлений с именами `header.php` и `footer.php`. В первый файл занесите такую разметку:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Codeigniter 3</title>
</head>
<body>
```

Во второй – такую:

```
</body>
</html>
```

Вы догадались, что мы создали заготовки для верхней и нижней частей наших страниц, которые теперь можно вызывать в каждом представлении, где это необходимо. Давайте рассмотрим применение этих двух представлений на странице `items.php`, которую мы создали для последнего метода:

```

<?php
$this->load->view('header');
echo '<h2>'.$title.'</h2>';
echo '<table class="table table-striped" >';
foreach($items as $c)
{
    echo '<tr>';
    echo '<td>'.$c['itemname'].'</td>';
    echo '<td>'.$c['pricein'].'</td>';
    echo '<td>'.$c['pricesale'].'</td>';
    echo '<td>'.$c['info'].'</td>';
    echo '</tr>';
}
echo '</table>';
$this->load->view('footer');

```

Таким образом мы можем унифицировать страницы нашего приложения. При этом все необходимые ссылки на внешние ресурсы будут храниться в одном месте. Наверное, уже пришло время добавить в наше приложение любимый Bootstrap.

Codeigniter и Bootstrap

Рассмотрим, каким образом можно применять Bootstrap в приложениях, созданных с помощью Codeigniter. Для начала создайте рядом с папками application и system папку с именем assets. Именно в этой папке мы и будем хранить необходимые ресурсы, такие как Bootstrap, jQuery и другие. Имя этой папки может быть произвольным. Скопируйте в assets три папки, содержащие Bootstrap с именами css, js и fonts из предыдущего приложения. Теперь добавьте в header.php необходимые ссылки на эти ресурсы и затем – простое меню:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Codeigniter 3</title>
    <link rel="stylesheet"
        href="<?php echo base _url("assets/css/
                                bootstrap.css"); ?>" />
</head>
<body>
//menu
<nav class="navbar navbar-inverse">
    <div class="container-fluid">
        <div class="collapse navbar-collapse" id="myNavbar">
            <ul class="nav navbar-nav">
                <li><a href="<?php echo site _url('home/
                                index'); ?>">
                    Index</a> Index</a></li>
                <li><a href="<?php echo site _url('home/
                                itemslist'); ?>">
                    Get All Items</a></li>
                <li><a href="<?php echo site _url('home/
                                getItemInfo'); ?>">
                    Get Item</a></li>
                <li><a href="<?php echo site _url('home/
                                getItemInfo2'); ?>">
                    Select Item</a></li>
            </ul>
        </div>
    </div>
</nav>

```

Обратите внимание, что для ссылки на методы контроллера мы используем функцию `site_url()`. Это функция Codeigniter, и мы можем использовать ее потому, что при конфигурировании нашего приложения мы за-

грузили хелпер url. Обратите внимание, что некоторых методов, на которые ссылаются наши пункты меню, еще нет. Это нормально, мы скоро создадим их. Аналогичную работу выполним в представлении footer.php:

```
<script type="text/javascript"
    src="<?php echo base_url("assets/js/jquery-
        2.0.0.min.js"); ?>"></script>
<script type="text/javascript"
    src="<?php echo base_url("assets/js/bootstrap.
        min.js"); ?>"></script>
</body>
</html>
```

Здесь используется функция `base_url()` из хелпера `url`, которая возвращает адрес корневой папки приложения. Снова вызовем в браузере наш метод `ItemList()`, чтобы посмотреть, подключился ли Bootstrap. У меня это выглядит так:

Index Get All Items Get Item Get Item Select			
List Of Items			
ASUS PC	780	1350	Reliable and powerful PC that can be used for work and for playing games. It is very good choice for home and for office
Dell PC	1150	1650	Powerful Dell computer
ASUS PC mini	1000	1480	Mini PC
HP notebook	1100	1550	2.2GHz AMD Athlon II Dual-Core P340 3GB (1x2GB, 1x1GB) RAM 320GB 5400rpm Hard Drive SuperMulti DVD Burner with LightScribe
ASUS notebook	1200	1700	1.6GHz Intel Core i7-720QM Quad-Core 4GB (2x2GB) RAM 500GB 7200rpm Hard Drive SuperMulti DVD Burner

Рис. 4. Применение Bootstrap

Можем констатировать, что Bootstrap успешно подключен.

Работа с формами

Рассмотрим, каким образом выполняется наш метод `itemsList()`, который активируется во втором пункте меню `Get All Items`. Последовательность действий такая:

- активируется метод контроллера `itemsList()`;
- в методе `itemsList()` вызывается метод модели `getItems()`;
- метод модели `getItems()` отправляет запрос в БД и получает результат выполнения этого запроса;
- метод модели `getItems()` возвращает результат выполнения запроса методу контроллера `itemsList()`;
- метод контроллера `itemsList()` активирует представление `item` и передает ему полученные данные.

Давайте напишем код, который позволит нам получить из БД информацию о каком-то конкретном товаре. В этом случае нам надо будет сначала указать, какой товар мы хотим увидеть. Для этого надо будет создать форму, в которой мы сможем указать, какой товар нас интересует. Для реализации такого сценария мы напишем в нашем контроллере новый метод. Жизненный цикл этого метода будет отличаться от предыдущего. Сначала наш метод должен будет отобразить форму, в которой мы сможем указать информацию об интересующем нас товаре. Затем метод контроллера должен будет получить из формы данные и передать в модель информацию о выбранном товаре, чтобы модель получила из БД требуемое описание. Затем метод контроллера получит данные из модели и передаст их в итоговое представление, которое отобразит результат в браузере.

Чтобы выполнить задуманную работу нам надо будет создать:

- новый метод в контроллере, назовем его `getItemInfo()`;
- представление с формой для выбора товара, назовем его `form_item_id.php`;
- метод модели для получения информации о выбранном товаре, назовем его `getItemById()`;
- представление для вывода конечного результата, назовем его `item_info.php`.

Начнем с создания представления `form_item_id.php`. Сейчас мы создадим первый вариант такой формы, в которой пользователю надо будет вводить в текстовое поле идентификатор выбранного товара. Будет полезно создать сначала именно такую – простую, хотя и не дружелюбную к пользователю форму, потому что в ходе ее создания мы познакомимся с еще одним полезным инструментом Codeigniter – хелпером `form`. Затем мы заменим эту форму другой, в которой товар надо будет выбирать из списка.

Для создания формы мы будем использовать хелпер `form`. Рассмотрим те из его методов, которые мы будем использовать в нашей форме. Этот хелпер содержит набор методов, имена которых начинаются с префикса `form_`. Каждый такой метод возвращает разметку какого-либо из элементов форм.

Метод **`form_open()`**

```
echo form_open('home/getItemInfo');
```

вернет такую разметку:

```
<form method="post" accept-charset="utf-8"
      action="http://apccil/index.php/'home/getItemInfo'">
```

Если в форму надо добавить атрибуты, это можно сделать так:

```
$ar=array('style'=>'form-horizontal');
echo form_open('home/getItemInfo',$ar);
```

Метод **form_label()**

```
echo form_label('Item id','itemid',array('class'=>
                                         'control-label col-sm-4'));
```

создаст разметку метки для элемента формы с именем 'itemid':

```
<Label for='itemid' class='control-label col-sm-4'>
                                         Item id </label>
```

Метод **form_input()**

```
echo form_input('itemid');
```

создаст разметку текстового поля с именем 'itemid':

```
<input type="text" name="itemid">
```

Но этот метод гораздо удобнее вызывать, передавая ему массив с требуемыми атрибутами:

```
$ar=array('name'=>'itemid', 'class'=>'form-control',
          'size'=>'20', 'style'=>'color:green;', 'placeholder'=>'select
id', 'type'=>'text'); echo form_input($ar);
```


Метод `form_submit()`

```
$ar=array('style'=>'form-horizontal');
echo form_opea$ar=array('name'=>'send', 'value'=>'OK',
'class'=>'btn btn-success col-sm-2')
echo form_submit($ar);n('home/getItemInfo',$ar);
```

конечно же, создаст в форме кнопку submit.

Этот хелпер включает в себя и другие методы, с некоторыми из которых мы познакомимся позже. Для создания нашей формы достаточно рассмотренных методов. Есть ли преимущества в использовании этого хелпера по сравнению с обычным созданием форм в HTML? Да. Формы, созданные с помощью хелпера, автоматически выполняют экранирование опасных символов в данных формы и делают валидацию данных формы совместно с библиотекой `form_validation` намного проще.

Вооружившись этой информацией, создайте в папке `application/views` файл с именем `form_item_id.php` и добавьте в него такой код:

```
<?php
$this->load->view('header');

$data['class']='form-horizontal';
$data['accept-charset']='utf8';
echo form_open('home/getItemInfo',$data);
echo "<div class='col-md-offset-4'>";
$inp=array('name'=>'itemid','class'=>'col-md-2',
'style'=>'color:green;margin:5px;',
'placeholder'=>'select id','type'=>'text');
echo form_input($inp);
echo form_submit(array('name'=>'send','value'=>'OK',
'class'=>'btn btn-sm btn-success col-sm-2'));
```

```
echo "</div>";
echo form _ close();

$this->load->view('footer');
```

Теперь добавим в контроллер метод getItemInfo(), который будет выполнять наш сценарий:

```
public function getItemInfo()
{
    $send=$this->input->post('send');
    if(!$send)
        $this->load->view('form _ item _ id');
    else
    {
        $id=$this->input->post('itemid');
        $item=$this->home _ model->getItemById($id);
        $data['item']=$item;
        $data['title']='Description Of Items '.$id;
        $this->load->view('item _ info',$data);
    }
}
```

В этом коде мы встречаемся с новым свойством класса контроллера с именем \$input. Это свойство предназначено для доступа к данным, введенным в форму. Мы используем для доступа к данным нашей формы метод post(), указывая ему в строковом виде имя элемента формы, данные которого хотим получить. Если вы внимательно присмотритесь к этому коду, то увидите аналогию с обработчиками, которые мы писали до этого: сначала выполняется проверка, была ли нажата кнопка submit, а затем либо в браузер выводится форма, либо выполняется обработчик. В обработчике строка

`$id=$this->input->post('itemid')` извлекает из текстового поля с именем 'itemid' занесенное туда значение (идентификатор товара) и сохраняет его в переменной `$id`. Затем вызывается метод модели `getItemById()`, в который передается полученный из формы идентификатор товара. Далее выполняются уже известные вам действия, которые завершаются выводом полученной из модели информации о выбранном товаре в представление 'item_info'.

Добавьте в класс модели `Home_model` такой метод:

```
public function getItemById($id)
{
    $query = $this->db->get _ where('items',
                                     array('id' => $id));
    return $query->result _ array();
}
```

В отличие от метода `get()`, использованного нами ранее, метод `get_where()` выполняет запрос `SELECT` с условием `WHERE`. Как видно, условие задается во втором параметре этого метода в виде массива. Когда вы выполняете запросы к БД с условием по уникальному полю, вы должны всегда помнить о том, что такой запрос не может вернуть более одной строки. Именно таким является запрос в этом методе модели. Помните, мы говорили о том, что функция `result_array()` извлекает данные из ресурса и возвращает их в виде массива, каждый элемент которого является в свою очередь массивом? С этой точки зрения важно понимать, что будет представлять собой результат функции `result_array()` в случае запроса, возвращающего одну строку. В этом случае это

тоже будет массив массивов, т.е. двумерный массив, но заполненной в этом массиве будет только строка с индексом 0. Запомните это замечание.

Нам осталось создать представление `item_info.php`, в котором контроллер выведет описание выбранного товара. Создайте файл с таким именем в папке `application/views` и занесите в него такой код:

```
<?php
$this->load->view('header');
echo "<div class='col-md-offset-1'>";
    echo '<h2>'.$title.'</h2>';

    echo '<h3>'.$item[0]['itemname'].'</h3>';
    echo '<p>'.$item[0]['pricein'].'</p>';
    echo '<p style="color:red;font-size:14pt;
                                font-family:Verdana;">'.
        $item[0]['pricesale'].'</p>';
    echo '<p>'.$item[0]['info'].'</p>';
echo "</div>";
```

Обратите внимание, как мы обращаемся к требуемым полям в массиве `$item`. Мы указываем первый индекс, равный 0. В предыдущих примерах мы не делали этого явно, потому что перебирали элементы массива в цикле, который извлекал их для нас из массива по одному. Сейчас цикла нет, и мы указываем индекс строки явно. А поскольку строка у нас только одна, то ее индекс равен 0.

У нас метод контроллера `getItemInfo()` активируется в третьем пункте меню. Повторим еще раз алгоритм работы этого метода:

- в браузер выводится форма для ввода идентификатора товара;

- после нажатия в этой форме кнопки submit из формы извлекается введенное значение идентификатора;
- вызывается метод модели, в которой передается идентификатор товара;
- принимаются данные, полученные от модели;
- в браузер выводится представление, в котором описывается выбранный товар.

Для проверки работы метода getItemInfo() запустите наше приложение и активируйте третий пункт меню. Вы должны увидеть нашу форму. Занесите в нее идентификатор какого-либо из своих товаров и нажмите кнопку submit. Если вы все сделали правильно, то в браузере получите примерно такое:

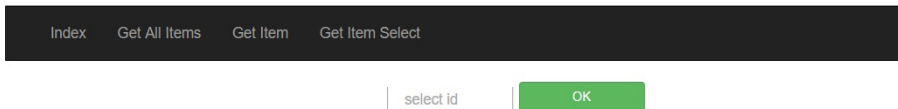


Рис. 5. Форма для выбора товара

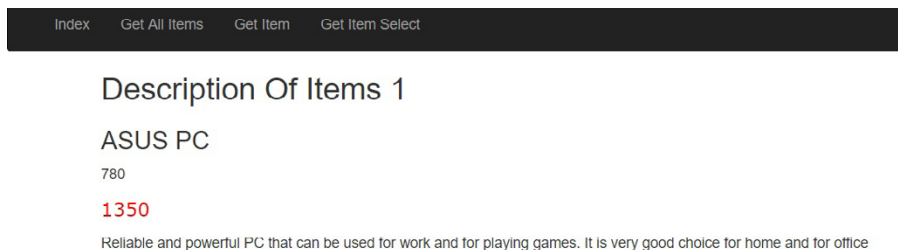


Рис. 6. Описание выбранного товара

Улучшенная форма

Вы должны понимать, что метод `getItemInfo()` в том виде, в котором он у нас сейчас создан, нельзя использовать в реальном приложении. Главный недостаток этого метода – форма для ввода идентификатора товара. Посмотрите, как поведет себя наш метод, если вы занесете в эту форму идентификатор несуществующего товара. Следующий недостаток этого метода – отсутствие валидации. Давайте исправлять эти недостатки. Начнем с формы. А потом рассмотрим валидацию.

Создадим форму, в которой вместо текстового поля будет находиться список с названиями товаров. Пользователь будет выбирать требуемый товар по названию, а список будет возвращать идентификатор выбранного товара. В этом случае пользователь не сможет нечаянно или преднамеренно ввести что-то, не являющееся идентификатором существующего товара.

Давайте подумаем, что нам надо, чтобы создать список с именами существующих товаров, который бы возвращал идентификатор выбранного товара. Для этого нам нужна информация из БД. А такую информацию мы можем получить только из модели. Поэтому нам надо будет обратиться к модели, получить необходимые данные для списка, создать представление с формой со списком и показать это представление пользователю.

Добавим в наш контроллер еще один метод, который будет выполнять описанную работу. Поскольку этот метод очень похож на метод `getItemInfo()`, назовем его `getItemInfo2()`:

```

function getItemInfo2()
{
    if (!$this->input->post('send'))
    {
        $data['list']=$this->Home _model->getItems();
        $this->load->view('form _ item _ id2',$data);
    }
    else
    {
        $id=$this->input->post('itemid');
        $item=$this->home _model->getItemById($id);
        $data['item']=$item;
        $data['title']='Description Of Items '.$id;
        $this->load->view('item _ info',$data);
    }
}

```

Если кнопка submit еще не нажималась, этот метод обращается к модели за данными, из которых можно создать список товаров. В нашей модели уже есть метод, который полностью подходит для этого случая – это метод `getItems()`, возвращающий список всех товаров из таблицы `Items`. Получаем из модели список товаров и передаем его в представление с именем `form_item_id2`. Это представление должно заменить форму с текстовым полем из предыдущего примера. В нем вместо текстового поля мы создадим список. Добавьте новое представление с указанным именем и вставьте в него такой код:

```

$this->load->view('header');
$st['class']='form-horizontal';
echo form_open('home/getItemInfo2',$st);
echo '<div class="">';
echo form_label('Select item',array
                                ('class'=>'control-label'));
echo '<select name="itemid">';
foreach ($list as $l){
    echo '<option value='.$l['id'].'>';
    echo $l['itemname'];
    echo '</option>';
}
echo '</select>';
echo form_submit(array('name'=>'send','value'=>'Send',
                        'class'=>'btn btn-success'));
echo '</div>';
echo form_close();
$this->load->view('footer');

```

В хелпере form есть метод `form_dropdown()`, позволяющий создавать списки, но мы сейчас не использовали его, создав список вручную. Это сделано специально, с той целью, чтобы показать вам, что методы хелпера можно использовать совместно с обычным созданием элементов управления в одной форме. Метод `form_dropdown()` мы рассмотрим далее.

Вы видите, что вторая часть метода `getItemInfo2()` полностью совпадает с методом `getItemInfo()`, а это значит, что нам не надо создавать никакие новые объекты и мы можем перейти к тестированию созданного метода.



Select item

Рис. 7. Улучшенная форма для выбора товара

Такая улучшенная форма предоставляет пользователю лучший способ выбора товара, чем предыдущая форма с текстовым полем. Во-первых, пользователю не надо помнить, какой идентификатор имеет тот или другой товар. Во-вторых, форма не позволяет пользователю ничего заносить в список самостоятельно, а позволяет только выбирать одно из предоставленных ему значений, исключая, таким образом, возможность занести опасные данные намеренно или по ошибке.

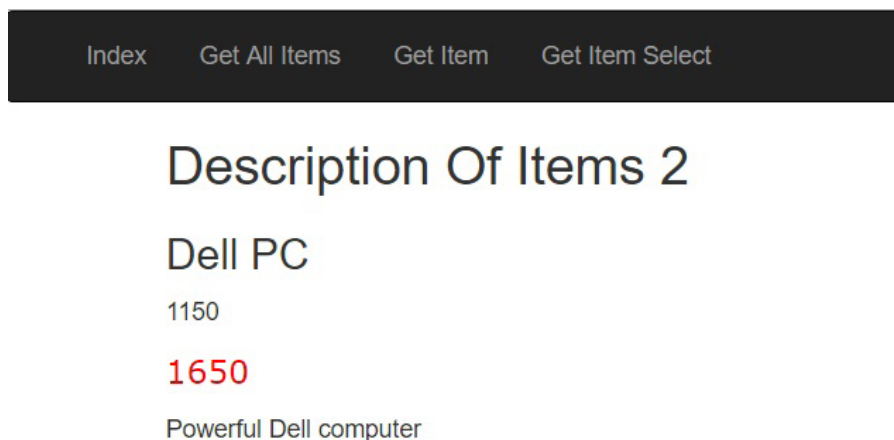


Рис. 8. Описание выбранного товара

Вы помните, что в качестве примера в этом уроке мы используем БД нашего интернет-магазина. В той БД есть таблица Images, предназначенная для хранения изображений. Давайте рассмотрим, как Codeigniter позволяет выполнять upload, создав форму добавления данных в таблицу Images.

Подробнее можете почитать о хелпере form по адресу: https://www.codeigniter.com/userguide3/helpers/form_helper.html

Upload в Codeigniter

Прежде всего, нам надо создать форму, которая будет содержать в себе элемент управления для выбора картинок. Затем надо будет создать метод контроллера, который будет выводить эту форму пользователю и извлекать введенные в форму данные. Наконец, надо будет создать метод модели, который будет получать данные из метода контроллера и заносить их в БД. Мы так подробно описываем эти шаги для того, чтобы вы постоянно помнили, что работаете в архитектуре MVC, которая диктует свои правила взаимодействия разных частей приложения. Назовем новую форму `form_upload.php`, метод в контроллере – `selectImages()`, и метод в модели – `addImages()`.

Но перед написанием кода надо сделать еще кое-что. Добавьте новый пункт меню в наше представление `header.php`:

```
<li>
    <a href="<?php echo site_url('home/
        selectImages');?>">Upload Image</a>
</li>
```

Создайте в папке `assets` еще одну папку с именем `images`. Именно в ней мы будем хранить выгруженные файлы.

Теперь перейдем к написанию нового кода. Создайте новый файл представления `form_upload.php` в папке `application/views` и добавьте в него такой код:

```

<?php
$this->load->view('header');
if(isset($error))
{
    echo '<div style="color:red;">'.$error.'</div>';
}
else if(isset($result))
{
    echo '<div style="color:green;">'.$result.'</div>';
}

$st['class']='form-horizontal';
echo form_open_multipart('home/selectImages',$st);
echo '<div class="col-md-offset-3">';
echo form_label('Select image ',
    'image',array('class'=>'control-label'));
echo '&nbsp;';
echo form_upload('image',array('class'=>'control-
    label'));
echo '&nbsp;';
echo form_submit(array('name'=>'send', 'value'=>'Send',
    'class'=>'btn btn-success'));
echo '</div>';
echo form_close();

$this->load->view('footer');

```

Для создания этого представления мы применили два новых метода хелпера form: form_open_multipart() и form_upload(). Первый метод, form_open_multipart(), создает открывающий тег <form> с атрибутом enctype="multipart/form-data". Этот атрибут требуется в тех формах, где предполагается использование upload. Это как раз наш случай. Метод form_upload() создает input с type="file". Все остальное в этой форме вам уже

знакомо. Обратите внимание, что в начале формы мы проверяем существование переменных `$error` и `$result`. Переменная `$error` будет содержать в себе описание ошибки, если таковая возникнет при работе с формой. В случае ошибки мы будем переадресовывать пользователя снова на эту форму, но при этом будем снабжать его описанием ошибки. Если же выгрузка завершится успешно, мы также сообщим пользователю об этом, указав идентификаторы добавленных в БД строк, которые мы заносим в переменную `$result`. Теперь добавим в контроллер новый метод `selectImages()`:

```
public function selectImages()
{
    $send=$this->input->post('send');
    if(!$send)
        $this->load->view('form_upload');
    else
    {
        $config['upload_path'] = './assets/images/';
        $config['allowed_types'] =
            'gif|jpg|png';
        $config['max_size'] = 2048;
        $config['max_width'] = 1024;
        $config['max_height'] = 768;

        $this->load->library('upload', $config);

        if ( ! $this->upload->do_
            upload('image'))
        {
            $data = array('error' =>
                $this->upload->display_errors());
            $this->load->view('form_upload',
                $data);
        }
    }
}
```

Рассмотрим подробно код в обработчике. Проверяем, нажималась ли кнопка submit. Если нет – выводим пользователю в браузер форму. Если да – начинаем обработку. Сначала создаем и инициализируем массив с именем `$config`. Этот массив нужен для загрузки библиотеки `upload`, которая выполняет выгрузку файлов на сервер. Мы не выполняли глобальную загрузку этой библиотеки при начальной конфигурации приложения, поэтому будем загружать ее локально в тех методах, где это необходимо. При загрузке этой библиотеки надо выполнить ряд настроек. Рассмотрим их.

Элемент `$config['upload_path']` указывает путь на сервере, где будут храниться выгруженные файлы. В нашем случае мы используем папку `images` внутри папки `assets`. Создайте эту папку, если еще не сделали это.

Элемент `$config['allowed_types']` указывает расширения файлов, которые можно будет выгружать на сервер.

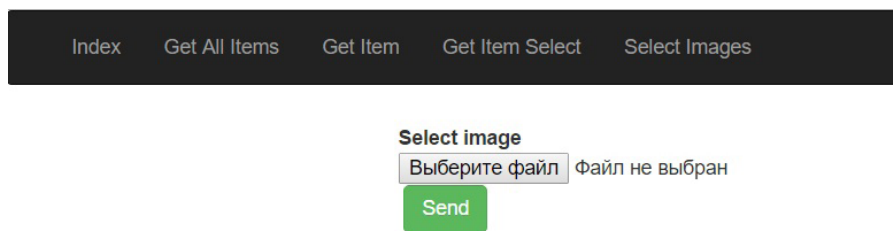
Элемент `$config['max_size']` задает максимальный размер выгружаемого файла в килобайтах.

Элементы `$config['max_width']` и `$config['max_height']` задают максимальные габариты выгружаемого файла в пикселях.

Выполнение процедуры выгрузки происходит при вызове метода `do_upload()` библиотеки `upload`. Этот метод требует при вызове, чтобы ему передали в строковом формате имя элемента `input` с `type="file"`. В нашей форме имя этого элемента `'image'`. Этот метод возвращает `true` при успешном выполнении `upload`, и `false` – при возникновении ошибки. Поэтому `do_upload()` удобно

вызывать в `if`, как в нашем коде. Именно в этом `if` мы формируем сообщение об ошибке, получая описание ошибки методом `display_errors()` и переадресовывая пользователя снова на нашу форму.

Хотя мы еще не выполнили весь свой план, будет полезно сделать промежуточное тестирование и проверить, выгружаются ли файлы в папку `assets/images`. А затем мы допишем код, добавляющий картинки в таблицу `Images`. Активируйте наше приложение, а затем – добавленный пункт меню `Select Images`. Вы увидите новую форму. Выберите изображение, отвечающее нашим ограничениям по настройке (расширение, размер и т.д.) и нажмите кнопку `submit`.



The screenshot shows a dark navigation bar with five menu items: 'Index', 'Get All Items', 'Get Item', 'Get Item Select', and 'Select Images'. Below the navigation bar, the 'Select image' form is displayed. It includes a text input field with the placeholder 'Выберите файл' (Select file), a status message 'Файл не выбран' (File not selected), and a green 'Send' button.

Рис. 9. Форма выгрузки файлов

У меня выбранные картинки успешно загружаются в папку `assets/images`.

Теперь нам осталось в методе контроллера добавить вызов метода модели, который занесет информацию о выбранном файле в таблицу `Images`. Откройте нашу модель и добавьте в нее такой метод:

```
public function addImages($data)
{
    $insert = $this->db->insert("Images",$data);
    if($insert)
    {
        return $this->db->insert _id();
    }
    else
    {
        return false;
    }
}
```

Этот метод добавляет запись в таблицу Images и возвращает идентификатор вставленной записи или false в случае ошибки. Метод insert() выполняет SQL запрос INSERT. В первом параметре ему надо передавать имя таблицы, а во втором – данные для вставки. Данные должны быть упакованы в ассоциативный массив, в котором названия индексов должны совпадать с именами полей в таблице. Такой массив мы будем готовить в методе контроллера.

Наша таблица Images содержит три поля: id, itemid и imagepath. Поскольку мы не указываем в запросе INSERT значение для поля id, нам надо создать массив с двумя ключами 'itemid' и 'imagepath'. В первый элемент надо заносить идентификатор товара, для которого добавляется картинка, а во второй – путь к картинке. Поскольку в нашей форме не предусмотрен выбор товара, мы будем заносить в элемент 'itemid' фиксированное значение идентификатора какого-либо существующего товара.

Изменим наш метод контроллера `selectImages()`, добавив в него код для создания массива с данными для записи в БД и для вызова метода модели:

```
public function selectImages()
{
    $send=$this->input->post('send');
    if(!$send)
        $this->load->view('form_upload');
    else
    {
        $config['upload_path'] = './assets/images/';
        $config['allowed_types'] =
            'gif|jpg|png';
        $config['max_size'] = 2048;
        $config['max_width'] = 1024;
        $config['max_height'] = 768;

        $this->load->library('upload', $config);

        if ( ! $this->upload->do_
            upload('image'))
        {
            //receive data about upload
            $info = array('upload_data'
                => $this->upload->data());
            //path to the uploaded images folder
            $path='assets/images/';
            //create array $data for model method
            $data=array('itemid'=>1,
                'imagepath'=>$path.$info
                ['upload_data']['file_name']);
            $itemid=$this->home_model-
                >addImages($data);
        }
    }
}
```



```

//create array for upload form
//with success message
If($itemid != false)
{
    $info = array('result' =>
        'Successfully Inserted New
        Image With Id='.$itemid);
        $this->load->view('form_
        upload', $info);
    }
}
}

```

Сначала мы создаем описанный выше ассоциативный массив для передачи в метод модели. В этом массиве создаем два элемента с ключами 'itemid' и 'imagepath'. Обратите внимание, что значение идентификатора товара для ключа 'itemid' мы прописали в коде. Если у вас нет товара с id=1, то вам надо указать здесь значение своего идентификатора. Для ключа 'imagepath' мы формируем путь к нашей папке assets/images. Вызываем метод модели addImages(), передавая ему созданный массив. Метод addImages(), при успешном завершении, возвращает идентификатор вставленной в таблицу Images строки. Мы получаем этот идентификатор и выводим его в форме.

Проверьте работу созданного метода. В результате успешной выгрузки картинки вы увидите такое сообщение:

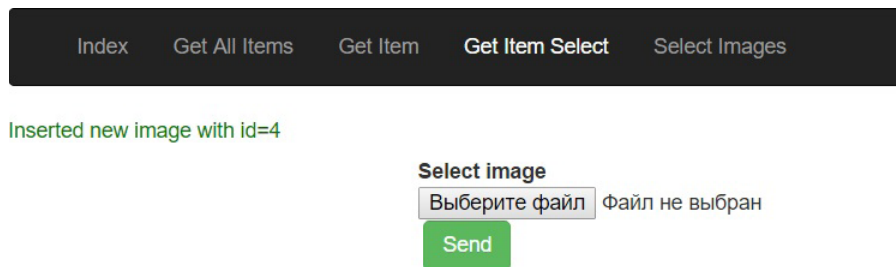


Рис. 10. Успешная выгрузка файла

Подводя промежуточный итог, можно сказать, что мы разобрали выполнение `upload` в Codeigniter. Вы, конечно же, помните, что `upload` бывает таким, при котором мы можем в одной форме выбрать сразу несколько файлов и все их отправить на сервер одним запросом. Получится ли сделать такое в нашем случае? Нет. Написанный код не позволяет выполнять множественную выгрузку (`multiple upload`). Более того, Codeigniter не поддерживает `multiple upload` по умолчанию, но если вам надо использовать множественную выгрузку в вашем приложении – выход есть.

Сейчас мы сделаем еще один пример, в котором продемонстрируем множественную выгрузку в Codeigniter. При этом будут использованы некоторые недокументированные приемы. Для реализации задуманного нам надо будет создать новый метод в контроллере и новое представление. Метод модели, добавляющий картинки в БД, будем использовать тот же, что и в предыдущем примере.

Сначала создадим форму, подобную форме в представлении `form_upload`, но с тем отличием, что в ней

элемент `input` с `type='file'` будет позволять выбирать несколько файлов за один раз. Назовем это новое представление `form_upload_multiple` и занесем в него такой код:

```
<?php
$this->load->view('header');
if(isset($error))
{
    echo '<div style="color:red;">';
    echo '<ul>';
    foreach ($error as $item => $value)
    {
        echo '<li>'.$item.':'.$value.'</li>';
    }
    echo '</div>';
    echo '</ul>';
}
if(isset($success))
{
    echo '<div style="color:green;">';
    echo '<ul>';
    foreach ($success as $item => $value)
    {
        echo '<li>'.$item.':'.$value.'</li>';
    }
    echo '</div>';
    echo '</ul>';
}

$st['class']='form-horizontal';
echo form_open_multipart('home/
                                selectMultipleImages',$st);
echo '<div class="col-md-offset-3">';
echo form_label('Select image
','image',array('class'=>'control-label'));
echo '&nbsp;';
echo form_upload('upfile[]','multiple');
```

```
echo '&nbsp;';  
echo form_submit(array('name'=>'send','value'=>'Send',  
    'class'=>'btn btn-success'));  
echo '</div>';  
echo form_close();  
$this->load->view('footer');
```

Мы снова создаем форму с помощью метода `form_open_multipart()`, поскольку нам нужна форма с атрибутом `enctype`. Следующее отличие касается использования метода `form_upload()` – здесь мы указываем имя элемента как массив и передаем ему атрибут `'multiple'`. Кроме того, обратите внимание, что в случае ошибки и в случае успешного завершения выгрузки мы в этой форме выводим не одно сообщение, как в предыдущем примере, а массив сообщений. Ведь сейчас нам надо описать выполнение выгрузки не одного единственного файла, а сразу нескольких.

Теперь перейдем к созданию нового метода в контроллере. Добавьте в наш контроллер еще один метод с именем `selectMultipleImages()`:

```

public function selectMultipleImages()
{
    $send=$this->input->post('send');
    if(!$send)
        $this->load->view('form_upload_multiple');
    else
    {
        $config['upload_path'] = './assets/images/';
        $config['allowed_types'] =
            'gif|jpg|png|jpeg';
        $config['max_size'] = 2048;
        $config['max_width'] = 1024;
        $config['max_height'] = 768;

        //now we initialize the upload library
        $this->load->library('upload', $config);
        //we retrieve the number of files that were
        //uploaded
        $number_of_files = sizeof($_FILES['upfile']
                                   ['tmp_name']);

        //we create array $files out of uploaded files
        $files = $_FILES['upfile'];
        $error = array();
        $success = array();

        //now, taking into account that there can
        //be more than one file,
        //we use loop to process all the files
        for ($i = 0; $i < $number_of_files; $i++)
        {
            $_FILES['upfile']['name'] =
                $files['name'][$i];
            $_FILES['upfile']['type'] =
                $files['type'][$i];
            $_FILES['upfile']['tmp_name'] =
                $files['tmp_name'][$i];
            $_FILES['upfile']['error'] =
                $files['error'][$i];
            $_FILES['upfile']['size'] =
                $files['size'][$i];
        }
    }
}

```

```

        if($_FILES['upfile']['error'][$i] != 0)
        {
            $error['msg'][$i]='Not uploaded file '.
            $_FILES['upfile']['name'][$i];
            continue;
        }
        if ( ! $this->upload->do_upload('upfile'))
        {
            $error['msg'][$i]= 'Not uploaded file'.
            $_FILES['upfile']['name'][$i];
        }
        else
        {
            $final_files_data[] = $this->upload->data();
            //Continue processing the
            //uploaded data
            //receive data about upload
            $info = array('upload_data' =>
                $this->upload->data());
            //path to the uploaded images
            //folder
            $path='assets/images/';
            //create array for model method
            $data=array('itemid'=>2,
                'imagepath'=>$path.$info['upload_
                data']['file_name']);
            $itemid=$this->home_model->addImages($data);
            //create array for upload form with
            //success message
            $success['msg'][$i]= 'Successfully
                Inserted
                New Image With Id='.$itemid;
        }
    }
    var_dump($success);
    echo '<br/>';
    var_dump($error);
    $result['error']=$error;
    $result['success']=$success;
    $this->load->view('form_upload_multiple',
$result);
    }
}

```

Мы говорили о том, что для реализации множественной выгрузки используются некие «хитрые приемы». Один из них – строка `$files = $_FILES['upfile']`. В этой строке мы создаем массив на основе глобального массива `$_FILES`. В созданном таким образом массиве `$files`, мы сохраняем информацию о файлах, выбранных в форме. Затем мы перебираем этот массив в цикле и возвращаем поочередно информацию о каждом таком файле снова в массив `$_FILES`, выполняя необходимую для выполнения upload «штатную» обработку массива `$_FILES`. По ходу обработки мы выполняем проверку на наличие ошибок. Если при извлечении из `$_FILES` какого-либо файла возникла ошибка, мы прерываем только текущую итерацию и переходим к обработке следующего файла с помощью инструкции `continue`. При этом информацию об ошибке мы заносим в массив `$error`. Если же в массиве `$_FILES` находится корректная информация о текущем из выбранных файлов, мы выполняем выгрузку этого файла. И снова проверяем, не возникла ли ошибка при выгрузке. Если возникла – снова заносим информацию об ошибке в массив `$error`. Если ошибок выгрузки нет – переходим к вызову метода модели `addImages()`, который занесет файл в БД. Для вызова метода модели `addImages()` мы сначала создаем ассоциативный массив, в котором находятся данные для записи в БД. Имена индексов этого массива должны совпадать с именами полей в таблице, в которую мы хотим заносить информацию. В нашем случае это таблица `Images`. Выполняя обработку, мы собирали в массив не только информацию об ошибках. В массив `$success`

мы параллельно собирали информацию об успешно завершенных выгрузках. При завершении обработки мы передаем оба массива \$error и \$success в представление, которое выведет пользователю информацию о выполненной выгрузке.

Добавим в наше меню еще один пункт, демонстрирующий Multiple Upload:

```
<li>
    <a href="?php echo site_url('home/
        selectMultipleImages'); ?>">
        Multiple Upload</a>
</li>
```

В приведенном рисунке вы видите отчет о выполнении множественной выгрузки файлов:

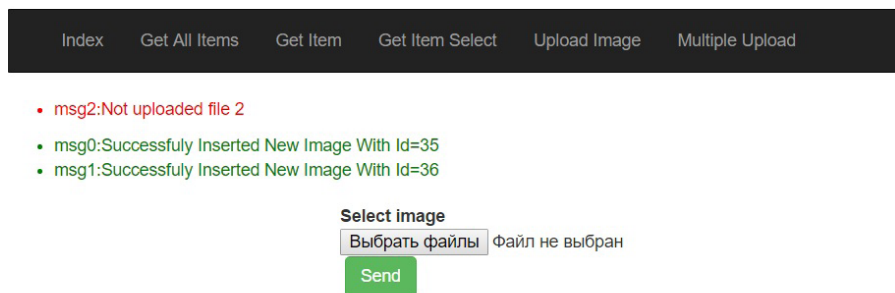


Рис. 11. Отчет о выгрузке файлов

Сессии в Codeigniter

Для того чтобы работать с сессиями в Codeigniter, сначала необходимо выполнить одну настройку в конфигурационном файле config.php, который находится в

папке `application/config`. Откройте этот файл, найдите в нем строку:

```
$config['encryption_key'] = '';
```

Занесите в этот элемент любое строковое значение. Это значение будет использоваться при хешировании ваших данных. Работа с сессиями реализована в Codeigniter в виде библиотеки. Эту библиотеку можно загружать двумя способами: глобально и локально.

Глобальная загрузка выполняется в файле `autoload.php` в папке `application/config`, где в строку инициализации элемента `$autoload['libraries']` надо добавить имя библиотеки `'session'`:

```
$autoload['libraries'] = array('session');
```

Мы так и сделали при начальной настройке нашего приложения.

Если глобальная загрузка этой библиотеки не выполнена, то для работы с сессиями надо загружать эту библиотеку в каждом методе или конструкторе контроллера, где нужна сессия:

```
$this->load->library('session');
```

Такую загрузку иногда называют локальной загрузкой библиотеки. Если библиотека загружена, можно заносить пользовательские данные в сессию и извлекать их из сессии. Для добавления данных в сессию используется метод `set_userdata()`:

```
$this->session->set_userdata('reguser', $name);
```

где:

'reguser' – имя сессии;

\$name – переменная, значение которой будет занесено в сессию;

Чтобы извлечь данные из сессии, используется метод `userdata()`:

```
$user = $this->session->userdata('reguser');
```

Отметьте, что методы `set_userdata()` и `userdata()` вызываются от имени свойства `session`, которое становится доступным после загрузки библиотеки `session`.

Метод `set_userdata()` позволяет за один вызов занести в сессию несколько значений. Для этого все требуемые значения надо упаковать в массив и этот массив передать в метод `set_userdata()`:

```
$data = array(
    'reguser' => $name,
    'roleid' => $roleid
);
$this->session->set_userdata($data);
```

Если вы попытаете извлечь из сессии данные, которых там нет, метод `userdata()` вернет `NULL`. Чтобы избежать такой ситуации, можно использовать булевский метод `has_userdata()` для проверки существования требуемого значения, перед тем как извлекать его из сессии:

```
if($this->session->has_userdata('reguser'))
{
    $user = $this->session->userdata('reguser');
}
```

Чтобы удалить из сессии какие-либо значения, надо использовать метод **unset_userdata()**:

```
$this->session->unset_userdata('reguser');
```

Чтобы удалить все сессию целиком, надо использовать метод **sess_destroy()**:

```
$this->session->sess_destroy();
```

Подробнее можете почитать об использовании сессий по адресу:

<https://codeigniter.com/userguide3/libraries/sessions.html>

Валидация данных форм в Codeigniter

Мы уже создали несколько форм в Codeigniter и рассмотрели принципы их использования. К этому моменту вы должны понять, что формы лучше создавать с помощью хелпера `form`, некоторые методы которого мы применили при создании форм в наших примерах. С другой стороны, вы должны также помнить о необходимости валидации тех данных, которые пользователь вводит через форму в приложение. Сейчас мы рассмотрим, каким образом Codeigniter выполняет валидацию данных форм. Для этого создадим стандартную форму регистрации. Такая форма является хорошим примером

для рассмотрения валидации, поскольку в ней собраны поля, для которых надо задействовать разные способы валидации.

Создайте еще одно представление в файле с именем `form_validation.php` и добавьте в него такой код:

```
<?php
$this->load->view('header');

$st['class']='form-horizontal';
echo form_open('home/registration',$st);

echo '<div class="row " style="margin:2px;">';
    echo form_label('Enter your login: ','login',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'login','size' => '50',
        'class' => '' );
    echo form_input($data);
echo '</div>';

echo '<div class="row " style="margin:2px;">';
    echo form_label('Enter password: ','pass1',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'pass1','size' => '50',
        'class' => '' );
    echo form_password($data);
echo '</div>';

echo '<div class="row " style="margin:2px;">';
    echo form_label('Confirm password: ','pass2',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'pass2', 'size' =>
        '50', 'class' => '' );
    echo form_password($data);
echo '</div>';

echo '<div class="row 1" style="margin:2px;">';
    echo form_label('Enter email: ','email',
        array('class'=>'control-label col-md-3'));
```

```

        $data = array('name' => 'email', 'type'    =>
                        'email', 'size' => '50', 'class' => '' );
        echo form_input($data);
    echo '</div>';

    echo '<div class="row" style="margin:2px;">';
        echo form_
            submit(array('name'=>'send', 'value'=>'Send',
                        'class'=>'btn btn-success col-md-
                                offset-4'));

        echo form_
            reset(array('name'=>'reset', 'value'=>'Reset',
                        'class'=>'btn btn-info'));
    echo '</div>';
    echo form_close();

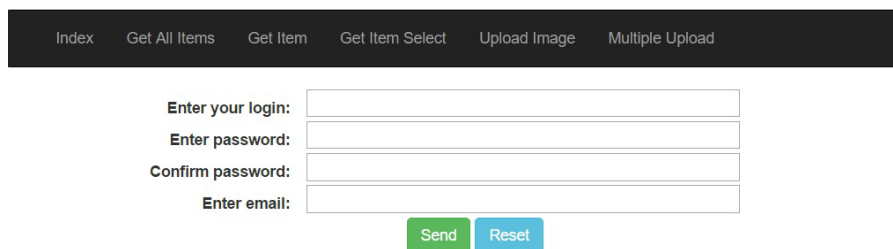
    $this->load->view('footer');

```

При создании этой формы мы использовали еще один метод хелпера form с именем form_reset(), который предназначен для создания стандартной кнопки очистки формы. Кроме этого, мы использовали форму вызова метода form_input(), в которой методу передается только один ассоциативный массив. В зависимости от значения элемента type этого массива, метод form_input() создает тот или иной элемент управления формы. В нашей форме мы создаем таким образом элемент input со значением type='email'.

В контроллер добавьте еще один метод, который будет работать с этой формой. В дальнейшем именно в этом методе мы рассмотрим выполнение валидации, а сейчас просто проверим, как выглядит созданная форма.

```
function registration()  
{  
    $this->load->view('form_validation');  
}
```



Index Get All Items Get Item Get Item Select Upload Image Multiple Upload

Enter your login:

Enter password:

Confirm password:

Enter email:

Рис. 12. Форма регистрации

Теперь создадим для нашей формы механизм валидации. В чем он заключается? В том, что для каждого поля формы надо создать свои правила проверки данных, введенных в него. Затем, когда пользователь занесет данные в поле, они будут проверяться на предмет соответствия этим правилам. Для выполнения валидации в Codeigniter существует библиотека `form_validation()`. А для создания набора правил валидации для каждого поля формы используется функция этой библиотеки `set_rules()`. Эта функция имеет три обязательных параметра и один необязательный. Все четыре параметра – строковые. Первый параметр – имя элемента формы, для которого создаются правила валидации. Функция `set_rules()` должна вызываться для каждого элемента управления, для которого надо создать правила валидации. Вторым параметром этой функции позволяет задать для элемента управления понятный пользователю псевдоним. Если данные не пройдут валида-

цию, пользователь получит сообщение об этом, и в нем, вместо непонятного пользователю технического имени элемента управления, будет отображаться информативный псевдоним. В третьем параметре передаются правила валидации. В последнем, необязательном параметре, можно указать свои сообщения об ошибках. При отсутствии четвертого параметра будут отображаться стандартные сообщения.

Валидация происходит при вызове функции `run()`. Если все данные формы прошли валидацию, эта функция возвращает `true`. Если при валидации произошли ошибки – функция возвращает `false`. Узнать о происшедших ошибках можно, вызвав функцию `validation_errors()`.

Познакомившись с этой информацией, добавим для нашей формы валидацию. Приведите метод `registration()` к такому виду:

```
function registration()
{
    $this->load->library('form_validation');
    $this->form_validation->set_rules('login',
                                     'User name',
                                     'trim|required|min_length[5]|max_length[12]|is_
                                     unique[customers.login]',
                                     array('required' => 'You have not filled %s.',
                                             'is_unique' => 'Value %s already
                                                         exists.'))
    );
    $this->form_validation->set_rules('pass1',
                                     'Password',
                                     'trim|required|min_length[5]|max_
                                     length[12]');
```

```

$this->form_validation->set_rules('pass2', '
                                Password Confirmation',
                                'required|matches[pass1]');
$this->form_validation->set_rules('email',
                                'Email',
                                'required|valid_email');

if ($this->form_validation->run() == FALSE)
{
    $this->load->view('form_validation');
}
else
{
    $data['success']='Form data passed the
                                validation';
    $this->load->view('form_validation',$data);
}
}

```

Сначала мы загружаем библиотеку `form_validation`, затем четыре раза вызываем функцию `set_rules()` для установки правил валидации четырьмя элементами управления нашей формы. Правила валидации интуитивно понятны. В качестве примера, в первом вызове функции `set_rules()` мы используем необязательный четвертый параметр – массив с собственными сообщениями об ошибках при нарушении разных правил валидации.

Для вывода пользователю информации об ошибках или об успешном завершении валидации в самое начало представления `form_validation` добавьте еще такой блок:


```

echo '<span style="color:red;margin-left:20px;">';
echo validation_errors();
echo '</span>';
if(isset($success))
{
    echo '<span style="color:green;margin-left:20px;">';
    echo $success;
    echo '</span>';
}

```

Активируйте нашу форму и попробуйте ввести данные с нарушениями правил валидации. Вы получите сообщения о допущенных нарушениях:

[Index](#)
[Get All Items](#)
[Get Item](#)
[Get Item Select](#)
[Upload Image](#)
[Multiple Upload](#)

The User name field must be at least 5 characters in length.

The Password Confirmation field does not match the Password field.

Enter your login:

Enter password:

Confirm password:

Enter email:

Рис. 13. Нарушение правил валидации

При вводе правильных данных во все поля формы получите что-то такое:

[Index](#)
[Get All Items](#)
[Get Item](#)
[Get Item Select](#)
[Upload Image](#)
[Multiple Upload](#)

Form data passed the validation

Enter your login:

Enter password:

Confirm password:

Enter email:

Рис. 14. Данные прошли валидацию

В ходе тестирования этой формы вы должны были заметить один неприятный момент: если в форму занести неверные данные только в одно поле, все правильно введенные данные сбрасываются при повторном выводе формы, и их надо заносить снова. Давайте сделаем так, чтобы правильные данные в элементах управления формы в таком случае оставались в форме. В этом нам поможет функция `set_value('field name')`, которую надо применить в представлении в атрибутах `value` элементов управления. Добавьте нашим элементам управления такие атрибуты `value`:

```

echo '<div class="row " style="margin:2px;">';
    echo form_label('Enter your login: ','login',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'login','size' => '50',
        'value'=> set_value('login') );
    echo form_input($data);
echo '</div>';

```

```

echo '<div class="row " style="margin:2px;">';
    echo form_label('Enter password: ','pass1',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'pass1','size' => '50',
        'value'=> set_value('pass1') );
    echo form_password($data);
echo '</div>';

echo '<div class="row " style="margin:2px;">';
    echo form_label('Confirm password: ','pass2',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'pass2', 'size' => '50',
        'value'=> set_value('pass2') );
    echo form_password($data);
echo '</div>';

echo '<div class="row 1" style="margin:2px;">';
    echo form_label('Enter email: ','email',
        array('class'=>'control-label col-md-3'));
    $data = array('name' => 'email', 'type' => 'email',
        'size' => '50',
        'value'=> set_value('email') );
    echo form_input($data);
echo '</div>';

```

Такое изменение формы приведет к тому, что при нарушении валидации в каком-либо поле формы правильные данные в других полях останутся на своем месте после повторного вывода формы пользователю.

Подробнее можете почитать о валидации по адресу: https://www.codeigniter.com/userguide3/libraries/form_validation.html

Использование Query Builder

Мы уже создали несколько методов в нашей модели, и вы видели, каким образом Codeigniter выполняет запросы с СУБД. Давайте подробнее рассмотрим использование специального класса Query Builder, предназначенного именно для работы с СУБД. Еще раз повторим, что использование этого класса позволяет писать код, независимый от конкретной СУБД. Другими словами, если вы подключитесь к другой СУБД, код модели не надо будет изменять, потому что синтаксис Query Builder автоматически транслируется в запросы каждой конкретной СУБД. Кроме этого, при использовании Query Builder выполняется автоматическая проверка данных, передаваемых в SQL запросы, на наличие опасных символов. Вы уже могли заметить, что при использовании запросов не приходится выполнять работу по внедрению переменных в тело запроса, требующую супертерпения и супервнимательности при расстановке кавычек и точек.

Рассмотрим основные методы Query Builder и способы их использования. Предположим, что в модели уже доступно свойство `$this->db`.

`$res = $this->db->get($table)` транслируется в запрос `SELECT * FROM` для таблицы, имя которой передается в параметре `$table`. Результат выполнения запроса заносится в переменную типа ресурс (`$res` в нашем примере). Для извлечения результата из ресурса можно использовать методы `result_array()` или `result()`. Первый возвращает массив массивов, второй – массив объектов.

```

$res = $this->db->get('Countries');
foreach ($res ->result_array() as $row)
{
    echo $row['id'] + ' ' + $row['country'] +
    '<br/>';
}

$res = $this->db->get('Countries');
foreach ($res ->result() as $row)
{
    echo $row->id+ ' ' + $row->country +
    '<br/>';
}

```

`$res = $this->db->get_where ($table,$cond)` подобно предыдущему методу, транслируется в запрос `SELECT * FROM $table WHERE $cond` для таблицы, имя которой передается в параметре `$table` с условием, заданным в массиве `$cond`. Результат выполнения запроса заносится в переменную типа ресурс (`$res` в нашем примере). Для извлечения результата из ресурса можно использовать методы `result_array()` или `result()`.

```

$res = $this->db->get_where('Countries',
                        array('id' => 1));
$row = $res ->result_array() ;
echo $row['id'] + ' ' + $row['country'] + '<br/>';

```

`$this->db->select()` позволяет указать, какие поля вы хотите получить при выполнении запроса `SELECT`. Предыдущие методы возвращают все поля таблицы.

```

$this->db->select('itemname, pricein, pricesale');
$query = $this->db->get('Items');

```

В результате вызова этих методов будет выполнен запрос:

```
'SELECT itemname, pricein, pricesale FROM Items'
```

Обратите внимание, что список полей передается в одной строке.

`$this->db->select_max()` позволяет выполнить запрос с агрегационной функцией для какого-либо поля какой-либо таблицы:

```
$this->db->select_max('pricesale');  
$query = $this->db->get('Items');
```

В результате вызова этих методов будет выполнен запрос:

```
'SELECT max(pricesale) FROM Items'
```

Конечно же, существуют методы для других агрегационных функций:

- **`$this->db->select_min()`**
- **`$this->db->select_avg()`**
- **`$this->db->select_sum()`**

`$this->db->from()` позволяет задавать данные для FROM части SQL запроса:

```
$this->db->select('itemname, pricein, pricesale');  
$this->db->from('Items');  
$query = $this->db->get();
```

Обратите внимание, что в этом случае метод **`get()`** используется без параметров, ведь имя таблицы уже указано в методе **`from()`**.

\$this->db->join() позволяет задавать данные для JOIN части SQL запроса:

```
$this->db->select('*');
$this->db->from('Items');
$this->db->join('categories', 'categories.id =
Items.catid');
$query = $this->db->get();
```

\$this->db->where() позволяет задавать данные для WHERE части SQL запроса:

```
$this->db->where('itemname', $name);
$query = $this->db->get('Items');
```

Для формирования требуемых условий можно вызывать метод **where()** несколько раз:

```
$this->db->where('catid', $catid);
$this->db->where('pricesale <', $maxprice);
$query = $this->db->get('Items');
```

При использовании метода **where()** параметры для него можно задавать в ассоциативном массиве:

```
$ar = array('catid' => $catid, 'pricesale <' =>
$maxprice);
$this->db->where($ar);
$query = $this->db->get('Items');
```

Кроме этого, параметры для метода **where()** можно задавать в обычной строке:

```
$str = 'catid = 2 OR catid = 5';
$this->db->where($str);
$query = $this->db->get('Items');
```

В предыдущем примере, когда мы вызывали метод `where()` несколько раз, все условия объединялись логическим оператором **AND**. Набор таких вызовов:

```
$this->db->where('catid', $catid);
$this->db->where('pricesale <', $maxprice);
$query = $this->db->get('Items');
```

приводит к созданию такого запроса:

```
SELECT * FROM Items WHERE catid = $catid AND
                                pricesale < $maxprice
```

`$this->db->or_where()` позволяет объединять условия для **WHERE** логическим оператором **OR**. Набор таких вызовов:

```
$this->db->where('catid', $catid);
$this->db->or_where('pricesale <', $maxprice);
$query = $this->db->get('Items');
```

приводит к созданию такого запроса:

```
SELECT * FROM Items WHERE catid = $catid OR pricesale
                                < $maxprice
```

Есть еще набор подобных методов: `where_in()`, `or_where_in()`, `where_not_in()`, `or_where_not_in()`, позволяющих формировать условия с SQL функцией **IN**.

`$this->db->like()` позволяет сформировать условие для поиска с использованием SQL инструкции **LIKE**. Набор таких вызовов:


```
$this->db->like('itemname', 'Lenovo');
$query = $this->db->get('Items');
```

приводит к созданию такого запроса:

```
SELECT * FROM Items WHERE itemname LIKE '%Lenovo%'
```

При использовании метода `like()` параметры для него можно задавать в ассоциативном массиве. Набор таких вызовов:

```
$ar = array('itemname' => 'Lenovo', 'info' =>
                                                    'notebook');
$this->db->like($ar);
$query = $this->db->get('Items');
```

приводит к созданию такого запроса:

```
SELECT * FROM Items
WHERE itemname LIKE '%Lenovo%' AND info LIKE
                                                    '%notebook%'
```

А если использовать метод `or_like()`:

```
$this->db->like('itemname', 'Lenovo');
$this->db->or_like('info', 'notebook');
$query = $this->db->get('Items');
```

то это приведет к созданию такого запроса:

```
SELECT * FROM Items
WHERE itemname LIKE '%Lenovo%' OR info LIKE
                                                    '%notebook%'
```

Наряду с выполнением стандартных SQL запросов, методы Query Builder предоставляют разработчику еще

много полезных опций. Если вы только что выполнили запрос INSERT и вам надо узнать идентификатор вставленной строки, вы можете воспользоваться методом `$this->db->insert_id()`:

```
$this->db->insert("Images",$data);  
$lastid=$this->db->insert_id();
```

Мы использовали этот метод в нашем методе модели `addImages()`.

Если вы только что выполнили запрос INSERT, UPDATE или DELETE и хотите узнать, сколько строк в БД было обработано, вы можете воспользоваться методом `$this->db->affected_rows()`.

Если вам надо узнать, сколько строк находится в какой-либо из таблиц вашей БД, вы можете воспользоваться методом `$this->db->count_all()`, передав ему имя таблицы:

```
$rows=$this->db->count_all('Items');
```

Класс Query Builder содержит еще много других методов, позволяющих создать и выполнить SQL запрос любого вида. Вы можете посмотреть более подробное описание этого класса по адресу: https://www.codeigniter.com/userguide3/database/query_builder.html

Изменение адресной строки

Вы уже познакомились с новой структурой адресной строки, которая основывается сейчас на именах контроллеров и методов. По умолчанию в структуру строки включается имя главной страницы сайта –

index.php. Ссылку на это имя в адресной строке можно убрать, и тогда строка станет короче и выразительнее. Посмотрим, как это можно сделать.

Сначала надо убедиться, что в настройках Apache в файле httpd.conf активирована строка:

```
LoadModule setenvif_module modules/mod_setenvif.so.
```

Найдите в своем конфигурационном файле эту строку и убедитесь, что перед ней удален символ комментария "#".

Затем надо в корневой папке приложения создать файл с именем .htaccess. Обратите внимание, имя файла начинается с символа точка '.'. Это обычный текстовый файл, и создать его можно блокнотом. Затем в созданный файл надо занести такое содержимое:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1 [L]
```

Наконец, надо сделать небольшие изменения в конфигурационном файле config.php в папке application/config. Найдите в файле config.php строку:

```
$config['index_page'] = "index.php"
```

и приведите ее к такому виду:

```
$config['index_page'] = ""
```

Кроме того, желательно в строке инициализации элемента \$config['uri_protocol'] выставить такое значение:

```
$config['uri_protocol'] = "REQUEST_URI"
```

Если вы проделали все эти изменения, перезапустите Open Server, и убедитесь, что имя `index.php` больше не требуется в адресной строке.

Использование сторонних библиотек

Codeigniter позволяет использовать в приложениях сторонние ресурсы и библиотеки, что часто бывает необходимо при разработке веб-приложения. Рассмотрим, например, как можно использовать в Codeigniter приложении функциональность `googlemaps`.

Перейдите по ссылке

<http://biostall.com/codeigniter-google-maps-v3-api-library/> на страницу разработчиков библиотеки `googlemaps.php` и загрузите эту библиотеку. Достаньте из архива два файла: `Googlemaps.php` и `Jsmin.php` и скопируйте их в папку `application/libraries` нашего приложения. Запомните, что эта папка предназначена для хранения сторонних библиотек, которые можно загружать в приложения. Эта библиотека позволит нам выполнять в нашем приложении стандартные действия с картами: отображать карту, отображать маркеры на карте, сделать маркеры кликабельными и т.п.

Для демонстрации этих возможностей создадим еще один метод контроллера и еще одно представление. Модель в этом случае задействовать не будем. Создайте в контроллере метод `showMap()` и занесите в него такой код:

```

function showMap()
{
    $this->load->library('googlemaps');

    $config['center'] = '51.5137415,-0.5514789';
    $config['zoom'] = 'auto';
    $this->googlemaps->initialize($config);

    $marker = array();
    $marker['position'] = '51.5137415,-0.5514789';
    $this->googlemaps->add_marker($marker);
    $data['map'] = $this->googlemaps->create_map();

    $this->load->view('view_map', $data);
}

```

Затем создайте представление `view_map.php` и в него занесите такой код:

```

<?php
$this->load->view('header');
echo '<script src="//maps.googleapis.com/maps/api/
                                js?key=YOUR_KEY"
        async="" defer="defer" type="text/
                                javascript"></script>';
echo "<div class='col-md-offset-2 col-md-10'>";
    echo $map['js'];
    echo $map['html'];
echo "</div>";
$this->load->view('footer');

```

Обратите внимание на строку подключения к `maps.googleapis.com`. Еще совсем недавно можно было обходиться без нее, но этим летом корпорация Google снова изменила правила пользования своим ресурсом `googlemaps`.

Почитать об этом можно здесь:

<https://maps-apis.googleblog.com/2016/06/building-for-scale-updates-to-google.html>

Теперь для отображения карт надо зарегистрироваться и получать ключ, это можно сделать здесь:

<https://developers.google.com/maps/documentation/javascript/get-api-key#key>

Регистрация бесплатна и занимает пару минут. Полученный ключ позволит вам бесплатно за один день использовать 25000 обращений к google API.

Получив ключ, вы сможете работать с картами в своем приложении. У меня это выглядит так:

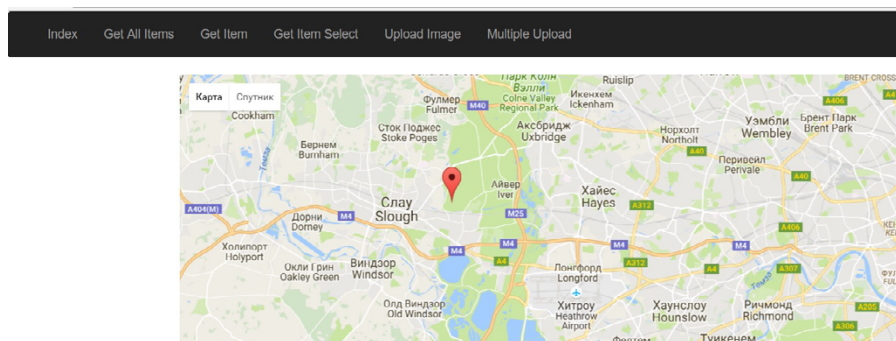


Рис. 15. Отображение карты

Безусловно, вы можете использовать другие библиотеки сторонних разработчиков. В том числе, и для работы с картами. Данная библиотека была выбрана просто в качестве примера, в силу своей известности и наглядности.

Домашнее задание

В этом уроке вы рассмотрели использование фреймворка Codeigniter для разработки веб приложений. Чтобы закрепить полученные навыки, добавьте в приложение еще одну страницу, подобную странице Catalog в нашем интернет-магазине. Отобразите на этой странице все товары из таблицы Items базы данных shop. Также добавьте на эту страницу два фильтра: по группам товаров и по ценам. Каждый товар отображайте в контейнере, подобно тому, как это делал наш метод draw().



Урок №5

Паттерн MVC.

Фреймворк Codeigniter

© Александр Геворкян

© Компьютерная Академия «Шаг»

www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.