# Программирование на стороне сервера. Часть 1. Реализация MVC паттерна.

Оглавление

[Реализация MVC паттерна на примере создания сайта-визитки на PHP 1](#_Toc496957072)

[Теория 2](#_Toc496957073)

[Front Controller и Page Controller 3](#_Toc496957074)

[Маршрутизация URL 4](#_Toc496957075)

[Практика 6](#_Toc496957076)

[Реализация маршрутизатора URL 6](#_Toc496957077)

[Возвращаемся к реализации MVC 9](#_Toc496957078)

[Реализация классов потомков Model и Controller, создание View's 10](#_Toc496957079)

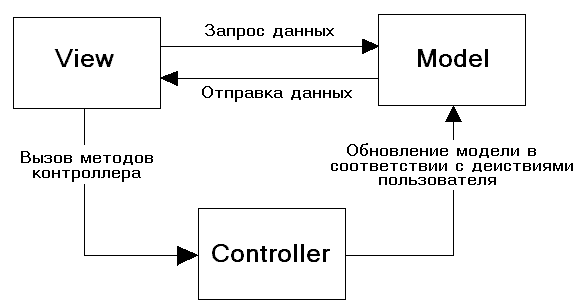
Создаем [главную страницу 11](#_Toc496957080)

Создаем [страницу «Портфолио» 12](#_Toc496957081)

[Создаем остальные страницы 13](#_Toc496957082)

[Задания по дням: 14](#_Toc496957083)

## Теория

Шаблон MVC описывает простой способ построения структуры приложения, целью которого является отделение бизнес-логики от пользовательского интерфейса. В результате, приложение легче масштабируется, тестируется, сопровождается и конечно же реализуется.  
  
Рассмотрим концептуальную схему шаблона MVC (на мой взгляд — это наиболее удачная схема из тех, что я видел):  
  
  
  
В архитектуре MVC модель предоставляет данные и правила бизнес-логики, представление отвечает за пользовательский интерфейс, а контроллер обеспечивает взаимодействие между моделью и представлением.  
  
Типичную последовательность работы MVC-приложения можно описать следующим образом:

1. При заходе пользователя на веб-ресурс, скрипт инициализации создает экземпляр приложения и запускает его на выполнение.  
   При этом отображается вид, скажем главной страницы сайта.
2. Приложение получает запрос от пользователя и определяет запрошенные контроллер и действие. В случае главной страницы, выполняется действие по умолчанию (*index*).
3. Приложение создает экземпляр контроллера и запускает метод действия,  
   в котором, к примеру, содержаться вызовы модели, считывающие информацию из базы данных.
4. После этого, действие формирует представление с данными, полученными из модели и выводит результат пользователю.

**Модель** — содержит бизнес-логику приложения и включает методы выборки (это могут быть методы ORM), обработки (например, правила валидации) и предоставления конкретных данных, что зачастую делает ее очень толстой, что вполне нормально.  
Модель не должна напрямую взаимодействовать с пользователем. Все переменные, относящиеся к запросу пользователя должны обрабатываться в контроллере.  
Модель не должна генерировать HTML или другой код отображения, который может изменяться в зависимости от нужд пользователя. Такой код должен обрабатываться в видах.  
Одна и та же модель, например: модель аутентификации пользователей может использоваться как в пользовательской, так и в административной части приложения. В таком случае можно вынести общий код в отдельный класс и наследоваться от него, определяя в наследниках специфичные для подприложений методы.  
  
**Вид** — используется для задания внешнего отображения данных, полученных из контроллера и модели.  
Виды cодержат HTML-разметку и небольшие вставки PHP-кода для обхода, форматирования и отображения данных.  
Не должны напрямую обращаться к базе данных. Этим должны заниматься модели.  
Не должны работать с данными, полученными из запроса пользователя. Эту задачу должен выполнять контроллер.  
Может напрямую обращаться к свойствам и методам контроллера или моделей, для получения готовых к выводу данных.  
Виды обычно разделяют на общий шаблон, содержащий разметку, общую для всех страниц (например, шапку и подвал) и части шаблона, которые используют для отображения данных выводимых из модели или отображения форм ввода данных.  
  
**Контроллер** — связующее звено, соединяющее модели, виды и другие компоненты в рабочее приложение. Контроллер отвечает за обработку запросов пользователя. Контроллер не должен содержать SQL-запросов. Их лучше держать в моделях. Контроллер не должен содержать HTML и другой разметки. Её стоит выносить в виды.  
В хорошо спроектированном MVC-приложении контроллеры обычно очень тонкие и содержат только несколько десятков строк кода. Чего, не скажешь о Stupid Fat Controllers (SFC) в CMS Joomla. Логика контроллера довольно типична и большая ее часть выносится в базовые классы.  
Модели, наоборот, очень толстые и содержат большую часть кода, связанную с обработкой данных, т.к. структура данных и бизнес-логика, содержащаяся в них, обычно довольно специфична для конкретного приложения.

### Front Controller и Page Controller

В большинстве случае, взаимодействие пользователя с web-приложением проходит посредством переходов по ссылкам. Посмотрите сейчас на адресную строку браузера — по этой ссылке вы получили данный текст. По другим ссылкам, например, находящимся справа на этой странице, вы получите другое содержимое. Таким образом, ссылка представляет конкретную команду web-приложению.  
  
Надеюсь, вы уже успели заметить, что у разных сайтов могут быть совершенные разные форматы построения адресной строки. Каждый формат может отображать архитектуру web-приложения. Хотя это и не всегда так, но в большинстве случаев это явный факт.  
  
Рассмотрим два варианта адресной строки, по которым показывается какой-то текст и профиль пользователя.  
  
Первый вариант:

1. [www.example.com/article.php?id=3](http://www.example.com/article.php?id=3)
2. [www.example.com/user.php?id=4](http://www.example.com/user.php?id=4)

Здесь каждый сценарий отвечает за выполнение определённой команды.  
  
Второй вариант:

1. [www.example.com/index.php?article=3](http://www.example.com/index.php?article=3)
2. [www.example.com/index.php?user=4](http://www.example.com/index.php?user=4)

А здесь все обращения происходят в одном сценарии **index.php**.

Эти два подхода совершенно различны. Первый — характерен для шаблона контроллер страниц (Page Controller), а второй подход реализуется паттерном контроллер запросов (Front Controller). Контроллер страниц хорошо применять для сайтов с достаточно простой логикой. В свою очередь, контроллер запросов объединяет все действия по обработке запросов в одном месте, что даёт ему дополнительные возможности, благодаря которым можно реализовать более трудные задачи, чем обычно решаются контроллером страниц. Я не буду вдаваться в подробности реализации контроллера страниц, а скажу лишь, что в практической части будет разработан именно контроллер запросов (некоторое подобие).

### Маршрутизация URL

Маршрутизация URL позволяет настроить приложение на прием запросов с URL, которые не соответствуют реальным файлам приложения, а также использовать [ЧПУ](http://ru.wikipedia.org/wiki/ЧПУ_(Интернет)), которые семантически значимы для пользователей и предпочтительны для поисковой оптимизации.  
  
К примеру, для обычной страницы, отображающей форму обратной связи, URL мог бы выглядеть так:  
<http://www.example.com/contacts.php?action=feedback>  
  
Приблизительный код обработки в таком случае:

switch($\_GET['action'])

{

case "about" :

require\_once("about.php"); // страница "О Нас"

break;

case "contacts" :

require\_once("contacts.php"); // страница "Контакты"

break;

case "feedback" :

require\_once("feedback.php"); // страница "Обратная связь"

break;

default :

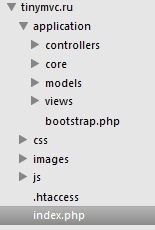
require\_once("page404.php"); // страница "404"

break;

}

Думаю, почти все так раньше делали.  
  
С использованием движка маршрутизации URL вы сможете для отображения той же информации настроить приложение на прием таких запросов:  
<http://www.example.com/contacts/feedback>  
  
Здесь contacts представляет собой контроллер, а feedback — это метод контроллера contacts, отображающий форму обратной связи и т.д. Мы еще вернемся к этому вопросу в практической части.  
  
Также стоит знать, что маршрутизаторы многих веб-фреймворков позволяют создавать произвольные маршруты URL (указать, что означает каждая часть URL) и правила их обработки.  
Теперь мы обладаем достаточными теоретическими знаниями, чтобы перейти к практике.

## Практика

Для начала создадим следующую структуру файлов и папок:  
  
  
Забегая вперед, скажу, что в папке core будут храниться базовые классы Model, View и Controller.  
Их потомки будут храниться в директориях controllers, models и views. Файл **index.php** это точка в хода в приложение. Файл **bootstrap.php** инициирует загрузку приложения, подключая все необходимые модули и пр.  
  
Будем идти последовательно; откроем файл index.php и наполним его следующим кодом:

ini\_set('display\_errors', 1);

require\_once 'application/bootstrap.php';

Тут вопросов возникнуть не должно.  
  
Следом, сразу же перейдем к фалу **bootstrap.php**:

require\_once 'core/model.php';

require\_once 'core/view.php';

require\_once 'core/controller.php';

require\_once 'core/route.php';

Route::start(); // запускаем маршрутизатор

Первые три строки будут подключать пока что несуществующие файлы ядра. Последние строки подключают файл с классом маршрутизатора и запускают его на выполнение вызовом статического метода start.

### Реализация маршрутизатора URL

Пока что отклонимся от реализации паттерна MVC и займемся мрашрутизацией. Первый ш﻿аг, который нам нужно сделать, записать следующий код в **.htaccess**:

RewriteEngine On

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule .\* index.php [L]

Этот код перенаправит обработку всех страниц на **index.php**, что нам и нужно. Помните в первой части мы говорили о Front Controller?!  
  
Маршрутизацию мы поместим в отдельный файл **route.php** в директорию core. В этом файле опишем класс Route, который будет запускать методы контроллеров, которые в свою очередь будут генерировать вид страниц.

**Содержимое файла route.php**

class Route

{

static function start()

{

// контроллер и действие по умолчанию

$controller\_name = 'Main';

$action\_name = 'index';

$routes = explode('/', $\_SERVER['REQUEST\_URI']);

// получаем имя контроллера

if ( !empty($routes[1]) )

{

$controller\_name = $routes[1];

}

// получаем имя экшена

if ( !empty($routes[2]) )

{

$action\_name = $routes[2];

}

// добавляем префиксы

$model\_name = 'Model\_'.$controller\_name;

$controller\_name = 'Controller\_'.$controller\_name;

$action\_name = 'action\_'.$action\_name;

// подцепляем файл с классом модели (файла модели может и не быть)

$model\_file = strtolower($model\_name).'.php';

$model\_path = "application/models/".$model\_file;

if(file\_exists($model\_path))

{

include "application/models/".$model\_file;

}

// подцепляем файл с классом контроллера

$controller\_file = strtolower($controller\_name).'.php';

$controller\_path = "application/controllers/".$controller\_file;

if(file\_exists($controller\_path))

{

include "application/controllers/".$controller\_file;

}

else

{

/\*

правильно было бы кинуть здесь исключение,

но для упрощения сразу сделаем редирект на страницу 404

\*/

Route::ErrorPage404();

}

// создаем контроллер

$controller = new $controller\_name;

$action = $action\_name;

if(method\_exists($controller, $action))

{

// вызываем действие контроллера

$controller->$action();

}

else

{

// здесь также разумнее было бы кинуть исключение

Route::ErrorPage404();

}

}

function ErrorPage404()

{

$host = 'http://'.$\_SERVER['HTTP\_HOST'].'/';

header('HTTP/1.1 404 Not Found');

header("Status: 404 Not Found");

header('Location:'.$host.'404');

}

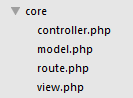
}

Замечу, что в классе реализована очень упрощенная логика (несмотря на объемный код) и возможно даже имеет проблемы безопасности. Это было сделано намерено, т.к. написание полноценного класса маршрутизации заслуживает как минимум отдельной статьи. Рассмотрим основные моменты…  
  
В элементе глобального массива $\_SERVER['REQUEST\_URI'] содержится полный адрес по которому обратился пользователь.  
Например: [example.ru/contacts/feedback](http://example.ru/contacts/feedback)  
  
С помощью функции *explode* производится разделение адреса на составлющие. В результате мы получаем имя контроллера, для приведенного примера, это контроллер *contacts* и имя действия, в нашем случае — *feedback*.  
  
Далее подключается файл модели (модель может отсутствовать) и файл контроллера, если таковые имеются и наконец, создается экземпляр контроллера и вызывается действие, опять же, если оно было описано в классе контроллера.  
  
Таким образом, при переходе, к примеру, по адресу:  
[example.com/portfolio](http://example.com/portfolio)  
или  
[example.com/portfolio/index](http://example.com/portfolio/index)  
роутер выполнит следующие действия:

1. подключит файл model\_portfolio.php из папки models, содержащий класс Model\_Portfolio;
2. подключит файл controller\_portfolio.php из папки controllers, содержащий класс Controller\_Portfolio;
3. создаст экземпляр класса Controller\_Portfolio и вызовет действие по умолчанию — action\_index, описанное в нем.

Если пользователь попытается обратиться по адресу несуществующего контроллера, к примеру:  
[example.com/ufo](http://example.com/ufo)  
то его перебросит на страницу «404»:  
[example.com/404](http://example.com/404)  
То же самое произойдет если пользователь обратится к действию, которое не описано в контроллере.

### Возвращаемся к реализации MVC

Перейдем в папку core и добавим к файлу route.php еще три файла: **model.php, view.php и controller.php**  
  
  
Напомню, что они будут содержать базовые классы, к написанию которых мы сейчас и приступим.  
  
Содержимое файла **model.php**

class Model

{

public function get\_data()

{

}

}

Класс модели содержит единственный пустой метод выборки данных, который будет перекрываться в классах потомках. Когда мы будем создавать классы потомки все станет понятней.  
  
Содержимое файла **view.php**

class View

{

//public $template\_view; // здесь можно указать общий вид по умолчанию.

function generate($content\_view, $template\_view, $data = null)

{

/\*

if(is\_array($data)) {

// преобразуем элементы массива в переменные

extract($data);

}

\*/

include 'application/views/'.$template\_view;

}

}

Не трудно догадаться, что метод *generate* предназначен для формирования вида. В него передаются следующие параметры:

1. $content\_view — виды отображающие контент страниц;
2. $template\_view — общий для всех страниц шаблон;
3. $data — массив, содержащий элементы контента страницы. Обычно заполняется в модели.

Функцией include динамически подключается общий шаблон (вид), внутри которого будет встраиваться вид  
для отображения контента конкретной страницы.  
  
В нашем случае общий шаблон будет содержать header, menu, sidebar и footer, а контент страниц будет содержаться в отдельном виде. Опять же это сделано для упрощения.

Содержимое файла **controller.php**

class Controller {

public $model;

public $view;

function \_\_construct()

{

$this->view = new View();

}

function action\_index()

{

}

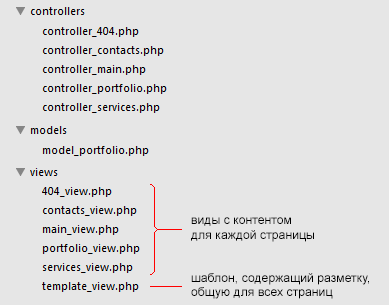
}

Метод *action\_index* — это действие, вызываемое по умолчанию, его мы перекроем при реализации классов потомков.

### Реализация классов потомков Model и Controller, создание View's

Теперь начинается самое интересное! Наш сайт-визитка будет состоять из следущих страниц:

1. Главная
2. Услуги
3. Портфолио
4. Контакты
5. А также — страница «404»

Для каждой из страниц имеется свой контроллер из папки controllers и вид из папки views. Некоторые страницы могут использовать модель или модели из папки models.  
  


На предыдущем рисунке отдельно выделен файл **template\_view.php** — это шаблон, содержащий общую для всех страниц разметку. В простейшем случае он мог бы выглядеть так:

<!DOCTYPE html>

<html lang="ru">

<head>

<meta charset="utf-8">

<title>Главная</title>

</head>

<body>

<?php include 'application/views/'.$content\_view; ?>

</body>

</html>

**Для придания сайту презентабельного вида сверстайте CSS шаблон и интегририруйте его в сайт путем изменения структуры HTML-разметки и подключения CSS и JavaScript файлов.**

### Создадаем главную страницу

Начнем с контроллера **controller\_main.php**, вот его код:

class Controller\_Main extends Controller

{

function action\_index()

{

$this->view->generate('main\_view.php', 'template\_view.php');

}

}

В метод *generate* экземпляра класса View передаются имена файлов общего шаблона и вида c контентом страницы.  
Помимо индексного действия в контроллере конечно же могут содержаться и другие действия.  
  
Файл с общим видом мы рассмотрели ранее. Рассмотрим файл контента **main\_view.php**:

<h1>Добро пожаловать!</h1>

<p>

<img src="/images/office-small.jpg" align="left" >

<a href="/">ОЛОЛОША TEAM</a> - команда первоклассных специалистов в области разработки веб-сайтов с многолетним опытом коллекционирования мексиканских масок, бронзовых и каменных статуй из Индии и Цейлона, барельефов и изваяний, созданных мастерами Экваториальной Африки пять-шесть веков назад...

</p>

Здесь содержиться простая разметка без каких либо PHP-вызовов.  
Для отображения главной странички можно воспользоваться одним из следующих адресов:

* [example.com](http://example.com/)
* [example.com/main](http://example.com/main)
* [example.com/main/index](http://example.com/main/index)

Пример с использованием вида, отображающего данные полученные из модели мы рассмотрим далее.

### Создадаем страницу «Портфолио»

В нашем случае, страница «Портфолио» — это единственная страница использующая модель.  
Модель обычно включает методы выборки данных, например:

1. методы нативных библиотек pgsql или mysql;
2. методы библиотек, реализующих абстракицю данных. Например, методы библиотеки PEAR MDB2;
3. методы ORM;
4. методы для работы с NoSQL;
5. и др.

Для простоты, здесь мы не будем использовать SQL-запросы или ORM-операторы. Вместо этого мы сэмулируем реальные данные и сразу возвратим массив результатов.  
Файл модели **model\_portfolio.php** поместим в папку models. Вот его содержимое:

class Model\_Portfolio extends Model

{

public function get\_data()

{

return array(

array(

'Year' => '2012',

'Site' => 'http://DunkelBeer.ru',

'Description' => 'Промо-сайт темного пива Dunkel от немецкого производителя Löwenbraü выпускаемого в России пивоваренной компанией "CАН ИнБев".'

),

array(

'Year' => '2012',

'Site' => 'http://ZopoMobile.ru',

'Description' => 'Русскоязычный каталог китайских телефонов компании Zopo на базе Android OS и аксессуаров к ним.'

),

// todo

);

}

}

Класс контроллера модели содержится в файле **controller\_portfolio.php**, вот его код:

class Controller\_Portfolio extends Controller

{

function \_\_construct()

{

$this->model = new Model\_Portfolio();

$this->view = new View();

}

function action\_index()

{

$data = $this->model->get\_data();

$this->view->generate('portfolio\_view.php', 'template\_view.php', $data);

}

}

В переменную *data* записывается массив, возвращаемый методом *get\_data*, который мы рассматривали ранее.  
Далее эта переменная передается в качестве параметра метода *generate*, в который также передаются: имя файла с общим шаблон и имя файла, содержащего вид c контентом страницы.  
  
Вид содержащий контент страницы находится в файле **portfolio\_view.php**.

<h1>Портфолио</h1>

<p>

<table>

Все проекты в следующей таблице являются вымышленными, поэтому даже не пытайтесь перейти по приведенным ссылкам.

<tr><td>Год</td><td>Проект</td><td>Описание</td></tr>

<?php

foreach($data as $row)

{

echo '<tr><td>'.$row['Year'].'</td><td>'.$row['Site'].'</td><td>'.$row['Description'].'</td></tr>';

}

?>

</table>

</p>

Здесь все просто, вид отображает данные полученные из модели.

### Создаем остальные страницы

**Остальные страницы создаются аналогично. Реализуйте их самостоятельно**

## Задания по дням:

## Инструкция к выполнению:

К каждой работе предоставляется отчет, состоящий:

1. Титульный лист
2. Лист с целью и описанием лабораторной работы
3. Дополнительное задание
4. Листинг

## Лабораторные работы:

1. Создание сайта с помощью MVC модели. Верстка простого шаблона и его интеграция. Добавление необходимых страниц.

**Дополнительное задание:** Представьте схему работы вашей MVC модели.

1. Изменение архитектуры для работы с базами данных. Модификация модели Портфолио для работы с СУБД.

<http://php.net/manual/ru/mysqli.examples-basic.php>

<http://www.php.su/articles/?cat=phpdb&page=010>

**Дополнительное задание:** Разработайте ERD схему вашей базы данных.

1. Добавление возможности авторизации Администратора с дальнейшим доступом к администраторскому разделу сайта. В этом разделе реализовать возможность добавления, редактирования и удаления записей для раздела Портфолио.

<http://blog.ox2.ru/php/avtorizaciya-i-rabota-sessii/>

<http://php.net/manual/ru/session.examples.basic.php>

**Дополнительное задание:** Составьте блок-схемы реализованных функций.

1. Добавить на сайт раздел Новости. В администраторском разделе предусмотреть возможность управления контентом этого раздела.

**Дополнительное задание:** ПредставьтеERD схему разработанного раздела и блок-схемы реализованных функций.

1. Реализовать возможность регистрации пользователей. Добавить возможность оставлять комментарии к новостям зарегистрированным пользователям. Предусмотреть возможность удаления своих комментариев.

**Дополнительное задание:** Опишите необходимые для валидации регулярные выражения. Представьте блок-схему функции регистрации.

1. Реализовать возможность администратору модерировать комментарии и учетные записи пользователей.

**Дополнительное задание:** Составьте руководство пользователя для вашего сайта со стороны программиста, администратора и пользователя.