***Реализация MVC паттерна на примере создания сайта-визитки на PHP***

Как вы уже догадались из названия статьи, сегодня речь пойдет о самом популярном, разве что после Singleton, шаблоне проектирования MVC, хотя такое сравнение не совсем уместно. Понимание концепции MVC может помочь вам в рефакторинге и разрешении неприятных ситуаций в которые, возможно попал ваш проект. Дабы восполнить пробел, мы реализуем шаблон MVC на примере простого сайта-визитки.

**Оглавление**

Введение
1. Теория
1.1. Front Controller и Page Controller
1.2. Маршрутизация URL
2. Практика
2.1. Реализация маршрутизатора URL
2.2. Возвращаемся к реализации MVC
2.3. Реализация классов потомков Model и Controller, создание View's
2.3.1. Создадаем главную страницу
2.3.2. Создадаем страницу «Портфолио»
2.3.3. Создаем остальные страницы
3. Результат
4. Заключение
5. Подборка полезных ссылок по сабжу

Введение
Многие начинают писать проект для работы с единственной задачей, не подразумевая, что это может вырасти в многопользовательскую систему управления, ну допустим, контентом или упаси бог, производством. И всё вроде здорово и классно, всё работает, пока не начинаешь понимать, что тот код, который написан — состоит целиком и полностью из костылей и хардкода. Код перемешанный с версткой, запросами и костылями, неподдающийся иногда даже прочтению. Возникает насущная проблема: при добавлении новых фич, приходится с этим кодом очень долго и долго возиться, вспоминая «а что же там такое написано то было?» и проклинать себя в прошлом.

Вы можеть быть даже слышали о шаблонах проектирования и даже листали эти прекрасные книги:

- Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидесс «Приемы объектно ориентированного проектирования. Паттерны проектирования»;
- М. Фаулер «Архитектура корпоративных программных приложений».

А многие, не испугавшись огромных руководств и документаций, пытались изучить какой-либо из современных фреймворков и столкнувшись со сложностью понимания (в силу наличия множества архитектруных концепций хитро увязанных между собой) отложили изучение и применение современных интсрументов в «долгий ящик».

Представленная статья будет полезна в первую очередь новичкам. Во всяком случае, я надеюсь что за пару часов вы сможете получить представление о реализации MVC паттерна, который лежит в основе всех современных веб-фреймворков, а также получить «пищу» для дальнейших размышлений над тем — «как стоит делать». В конце статьи приводится подборка полезных ссылок, которые также помогут разобраться из чего состоят веб-

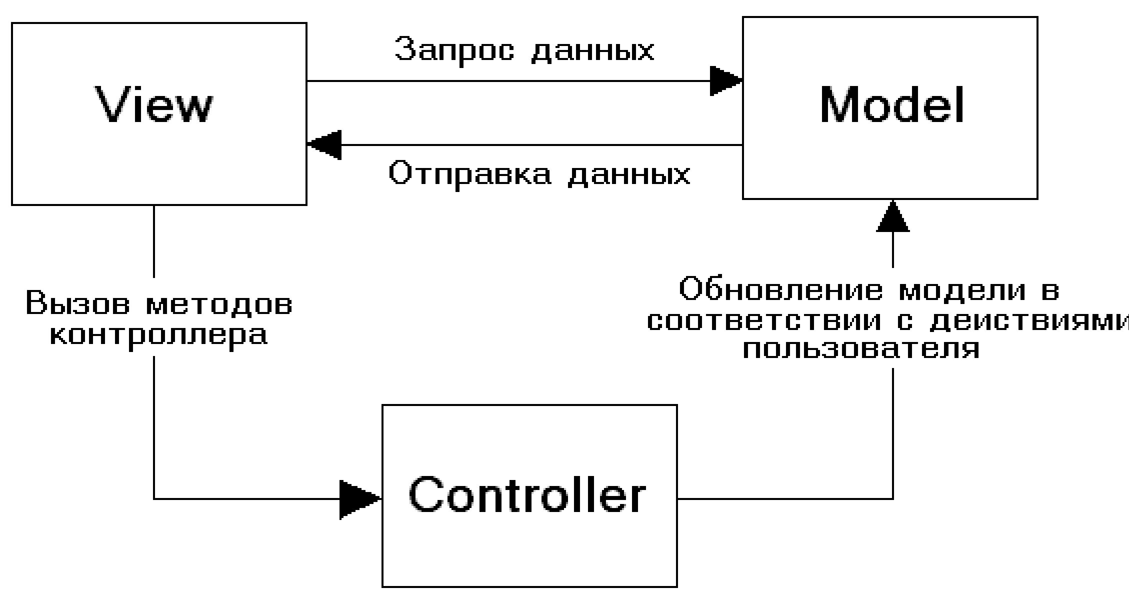
фреймворки (помимо MVC) и как они работают.

Прожженные PHP-программисты вряд ли найдут в данной статье что-то новое для себя, но их замечания и комментарии к основному тексту были бы очень кстати! Т.к. без теории практика невозможна, а без практики теория бесполезна, то сначала будет чуть-чуть теории, а потом перейдем к практике. Если вы уже знакомы с концепцией MVC, можете пропустить раздел с теорией и сразу перейти к практике.

## 1. 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 содержат 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.

## 1.1. 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
2. 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
2. www.example.com/index.php?user=4

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

Подход с множеством точек взаимодействия вы можете наблюдать на форумах с движком phpBB. Просмотр форума происходит через сценарий **viewforum.php**, просмотр топика через **viewtopic.php** и т.д. Второй подход, с доступом через один физический файл сценария, можно наблюдать в моей любимой CMS MODX, где все обращения проходят через **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).

### 1.2. Маршрутизация URL

Маршрутизация URL позволяет настроить приложение на прием запросов с URL, которые не соответствуют реальным файлам приложения, а также использовать ЧПУ, 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.

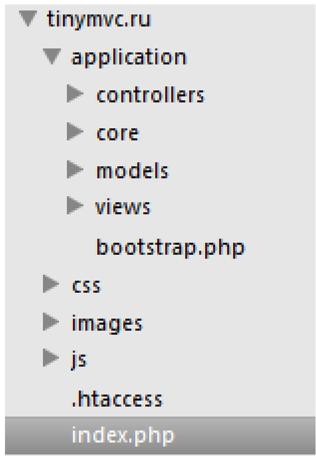
С использованием движка маршрутизации URL вы сможете для отображения той же информации настроить приложение на прием таких запросов:
http://www.example.com/contacts/feedback

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.

## 2. 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.

## 2.1. 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

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](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
или
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
то его перебросит на страницу «404»:
example.com/404
То же самое произойдет если пользователь обратится к действию, которое не описано в контроллере.

### 2.2. Возвращаемся к реализации MVC

Перейдем в папку core и добавим к файлу route.php еще три файла: **model.php, view.php и controller.php**

▼ core
controller.php
model.php
route.php
view.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_file — виды отображающие контент страниц;
2. $template_file — 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.

## 2.3. 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.

```
▼ controllers
    controller_404.php
    controller_contacts.php
    controller_main.php
    controller_portfolio.php
    controller_services.php
▼ models
    model_portfolio.php
▼ views
    404_view.php        ┐
    contacts_view.php   │
    main_view.php       ├─ виды с контентом для каждой страницы
    portfolio_view.php  │
    services_view.php   ┘
    template_view.php   ── шаблон, содержащий разметку, общую для всех страниц
```

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 сверстаем CSS шаблон и интегририруем его в наш сайт путем изменения структуры HTML-разметки и подключения CSS и JavaScript файлов:

```
<link rel="stylesheet" type="text/css" href="/css/style.css" />
<script src="/js/jquery-1.6.2.js" type="text/javascript"></script>
```

В конце статьи, в разделе «Результат», приводится ссылка на GitHub-репозиторий с проектом, в котором проделаны действия по интеграции простенького шаблона.

### 2.3.1. Создаем 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 с 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
- example.com/main
- example.com/main/index

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

### 2.3.2. Создаем 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 "САН 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 с 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.

### 2.3.3. Создаем остальные страницы

Остальные страницы создаются аналогично. Их код досутпен в репозитории на GitHub, ссылка на который приводится в конце статьи, в разделе «Результат».

## 3. Результат

А вот что получилось в итоге:

Скриншот получившегося сайта-визитки

Ссылка на GitHub: https://github.com/vitalyswipe/tinymvc/zipball/v0.1

А вот в этой версии я набросал следующие классы (и соответствующие им виды):

- Controller_Login в котором генерируется вид с формой для ввода логина и пароля, после заполнения которой производится процедура аутентификации и в случае успеха пользователь перенаправляется в админку.
- Contorller_Admin с индексным действием, в котором проверяется был ли пользователь ранее авторизован на сайте как администратор (если был, то отображается вид админки) и действием logout для разлогинивания.

Аутентификация и авторизация — это другая тема, поэтому здесь она не рассматривается, а лишь приводится ссылка указанная выше, чтобы было от чего оттолкнуться.

**4. Заключение**

Шаблон MVC используется в качестве архитектурной основы во многих фреймворках и CMS, которые создавались для того, чтобы иметь возможность разрабатывать качественно более сложные решения за более короткий срок. Это стало возможным благодаря повышению уровня абстракции, поскольку есть предел сложности конструкций, которыми может оперировать человеческий мозг.

Но, использование веб-фреймворков, типа Yii или Kohana, состоящих из нескольких сотен файлов, при разработке простых веб-приложений (например, сайтов-визиток) не всегда целесообразно. Теперь мы умеем создавать красивую MVC модель, чтобы не перемешивать Php, Html, CSS и JavaScript код в одном файле.

Данная статья является скорее отправной точкой для изучения CMF, чем примером чего-то истинно правильного, что можно взять за основу своего веб-приложения. Возможно она даже вдохновила Вас и вы уже подумываете написать свой микрофреймворк или CMS, основанные на MVC. Но, прежде чем изобретать очередной велосипед с «блекджеком и шлюхами», еще раз подумайте, может ваши усилия разумнее направить на развитие и в помощь

сообществу уже существующего проекта?!

P.S.: Статья была переписана с учетом некоторых замечаний, оставленных в комментариях. Критика оказалась очень полезной. Судя по отклику: комментариям, обращениям в личку и количеству юзеров добавивших пост в избранное затея написать этот пост оказалось не такой уж плохой. К сожалению, не возможно учесть все пожелания и написать больше и подробнее по причине нехватки времени… но возможно это сделают те таинственные личности, кто минусовал первоначальный вариант. Удачи в проектах!

**5. Подборка полезных ссылок по сабжу**

В статье очень часто затрагивается тема веб-фреймворков — это очень обширная тема, потому что даже микрофреймворки состоят из многих компонентов хитро увязанных между собой и потребовалась бы не одна статья, чтобы рассказать об этих компонентах. Тем не менее, я решил привести здесь небольшую подборку ссылок (по которым я ходил при написаниие этой статьи), которые так или иначе касаются темы фреймворков.