Урок 5

Парадигма MVC. Обновления движка.

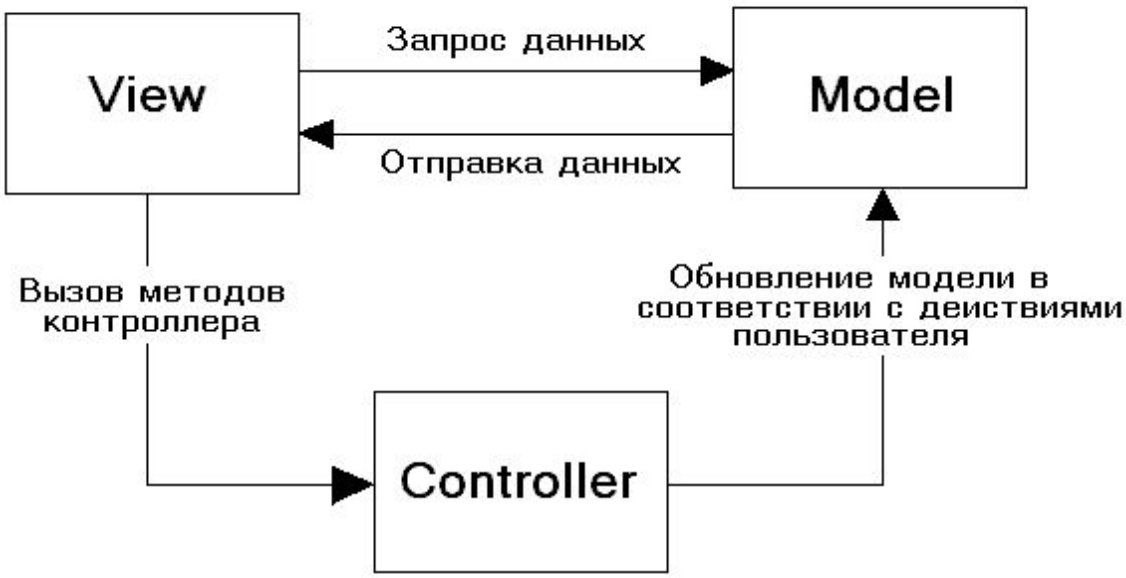
Знакомство с парадигмой-паттерном “Model-View-Controller”. Обновление архитектуры системы. Стандартизация кода.

Многие начинают писать проект для работы с единственной задачей, не подразумевая, что это может вырасти в многопользовательскую систему управления, ну допустим, контентом или 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 вёрсткой, запросами и костылями, неподдающийся иногда даже прочтению. Возникает насущная проблема: при добавлении новых фич, приходится с этим кодом очень много и долго возиться, вспоминая «а что же там такое написано-то было?» и проклинать себя в прошлом.
Если посмотреть на движок, написанный на курсе PHP Уровень 1 с текущей высоты знаний, то приходит понимание, что он хорош для решения относительно простых задач, но плохо пригоден для использования при решении сложных бизнес-задач. В нём сложно настраивать поведение страниц, он не гибок в отношении создания новых шаблонных модулей.
Вы может быть даже слышали о шаблонах проектирования. А многие, не испугавшись огромных руководств и документаций, пытались изучить какой-либо из современных фреймворков и столкнувшись со сложностью понимания (в силу наличия множества архитектурных концепций, хитро увязанных между собой) отложили изучение и применение современных инструментов в «долгий ящик».
На этом уроке мы поговорим о том, как же правильно структурировать код и познакомимся с большим апгрейдом нашего движка, версией 2.0, на базе которой будем развивать проект.

Парадигма MVC

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.



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 создаёт 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 создаёт экземпляр контроллера и запускает метод действия, в котором, к примеру, содержатся 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 её 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 (представление) — 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 её 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.

Одной из важных вещей в MVC является **единая точка входа** в приложение вместо множества PHP-файлов, делающих примерно следующее:

```
<?php
include ('global.php');
// Здесь код страницы
```

```
?>
```

У нас будет один файл, обрабатывающий все запросы. Это значит, что нам не придётся мучиться с подключением global.php каждый раз, когда нам нужно создать новую страницу. Эта «одна точка входа» будет называться index.php и на данный момент будет такой:

```
<?php
// Здесь что-нибудь делаем
?>
```

Очень похожее поведение реализовано в первой версии движка, но, если там оно имеет слабую структуру кода за счёт отсутствия использования преимуществ ООП, то в данном случае мы стремимся сделать прозрачную, но в то же время чёткую и удобную структуру кода.

Как вы можете заметить, этот скрипт пока ещё ничего не делает. Чтобы направить все запросы на главную страницу, мы воспользуемся mod_rewrite и установим в .htaccess директиву RewriteRule. Вставим следующий код в файл .htaccess и сохраним его в той же директории, что и index.php:

```
AddDefaultCharset UTF-8
DirectoryIndex index.php index.html
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} -f [NC,OR]
RewriteCond %{REQUEST_FILENAME} -d [NC]
RewriteRule .* - [L]
RewriteRule ^(.*)/$ ?path=$1 [QSA,L]
```

Сперва мы проверяем, существует ли запрашиваемый файл, используя директиву RewriteCond, и, если нет, то перенаправляем запрос на index.php. Такая проверка на существование файла необходима, так как иначе index.php будет пытаться обрабатывать все запросы к сайту, включая запросы на изображения. А это нам как раз и не надо.

Если у вас нет возможности использовать .htaccess или mod_rewrite, то вам придётся вручную адресовать все запросы к index.php. Другими словами, все ссылки должны будут иметь вид «index.php?path=[здесь-идёт-запрос]». Например, «index.php?path=chat/index».

Архитектура системы

Немаловажной частью реализации является расположение файлов в директориях сайта. Обычно применяется следующее наименование:

- configuration – директория файлов конфигурации;
- controller – директория прикладных контроллеров;
- data – директория хранения дампов;
- lib – подключаемые библиотеки и основные контроллеры;

- logs – директория логов;
- public – директория, на которую смотрит веб-сервер;
- templates – директория Twig-шаблонов;
- tests – директория с тестами.

URL у нас формируется из 3 частей по шаблону:

1. Контроллер.
2. Действие.
3. Параметр.

Таким образом адрес /catalog/index/ вызовет контроллер CatalogController.class.php, после чего произойдёт выполнение метода index в этом контроллере.

Итак, точкой входа нам служит файл index.php. Он не выполняет ничего, кроме безопасного подключения файла app.php.

Уже в файле app.php мы пытаемся создать экземпляр класса App, используя паттерн Одиночка. Но перед этим мы описываем логику автозагрузки классов из файлов. Это необходимо сделать для того, чтобы каждый раз не задумываться о том, какие файлы надо включить в require. Также обратите внимание, что мы используем функцию spl_autoload_register, которая позволяет нам создавать много раздельных автозагрузчиков. Но не забывайте о том, что правила автозагрузки могут переопределять друг друга, что может привести к конфликтам. Именно поэтому автозагрузчик Twig указан до системного автозагрузчика нашего движка.

В методе init мы разбираем url-адрес на сегменты, передавая системе информацию о том, что пользователь хочет получить. Затем, согласно парадигме MVC, мы создаём экземпляр контроллера, который готовит систему к генерации представления, используя данные из моделей.

Структура БД

Поскольку мы сильно ограничены во времени и наша задача ознакомиться с современными методиками разработки, мы ограничим возможности магазина до минимума. Что это за минимум? Пусть посетитель сайта видит каталог товаров, имеет возможность собрать несколько товаров в корзину, и оформить заказ. Стандартные функции по регистрации пользователя мы портируем из движка V1.

Для реализации такого функционала нам необходимо создать следующую структуру базы данных:

1. Каталог товаров.
2. Каталог категорий товаров.
3. Список заказов.
4. Связка заказов и выбранных товаров (для корзины).

Структура каталога товаров:

- идентификатор
- дата создания;
- дата обновления;
- цена;
- название;
- описание;
- статус (Активен-не активен. Флаг, при котором товар доступен на сайте);
- категория.

Структура категорий:

- идентификатор;
- дата создания;
- дата обновления;
- название;
- родительский элемент (идентификатор);
- статус (активен, неактивен, больше не в продаже).

Структура заказа:

- идентификатор;
- телефон;
- идентификатор клиента;
- адрес доставки;
- дата создания;
- дата обновления;
- статус (активен, неактивен, оплачен, доставлен).

Структура корзины:

- идентификатор;
- идентификатор заказа;
- идентификатор товара;
- дата создания;
- дата обновления;
- количество товара;
- статус (активен, неактивен, удалён, подтверждён).

Для чего нам нужно хранить удалённые и неактивные записи? Это может быть необходимо для аналитики. Есть и чисто технический аспект: операция удаления из БД гораздо дороже, чем редактирование.

Немного о стандартах

PHP достаточно поздно обзавёлся стандартами написания кода, однако на свет уже успело появиться 8 описаний различных аспектов данного языка программирования.

Сегодня мы рассмотрим два из них: PSR-0 и PSR-1

PSR-0 – Стандарт автозагрузки

Ниже представлены требования, обязательные к исполнению в целях обеспечения совместимости механизмов автозагрузки.

Обязательные требования:

- Полностью определённое пространство имён и имя класса должны иметь следующую структуру: \<Vendor Name>\(<Namespace>\)*<Class Name>.
- Каждое пространство имён должно начинаться с пространства имён высшего уровня, указывающего на разработчика кода («имя производителя»).

- Каждое пространство имён может включать в себя неограниченное количество вложенных подпространств имён.
- Каждый разделитель пространства имён при обращении к файловой системе преобразуется в РАЗДЕЛИТЕЛЬ_ИМЁН_КАТАЛОГОВ.
- Каждый символ _ («знак подчёркивания») в ИМЕНИ_КЛАССА преобразуется в РАЗДЕЛИТЕЛЬ_ИМЁН_КАТАЛОГОВ. При этом символ _ («знак подчёркивания») не обладает никаким особенным значением в имени пространства имён (и не претерпевает преобразований).
- При обращении к файловой системе полностью определённое пространство имён и имя класса дополняются суффиксом .php.
- В имени производителя, имени пространства имён и имени класса допускается использование буквенных символов в любых комбинациях нижнего и верхнего регистров.

PSR-1 – Базовый стандарт оформления кода

Данный раздел описывает стандартные элементы, являющиеся существенными для обеспечения высокой технической совместимости кода, созданного и/или поддерживаемого различными разработчиками.

1. Общие положения

- В файлах НЕОБХОДИМО использовать только теги <?php и <?=.
- Файлы НЕОБХОДИМО представлять только в кодировке UTF-8 без BOM-байта.
- В файлах СЛЕДУЕТ либо объявлять структуры (классы, функции, константы и т.п.), либо генерировать побочные эффекты (выполнять действия) (например: передавать данные в выходной поток, модифицировать настройки и т.п.), но НЕ СЛЕДУЕТ делать одновременно и то, и другое.
- Имена пространств имён и имена классов ДОЛЖНЫ следовать стандарту PSR-0.
- Имена классов ДОЛЖНЫ быть объявлены с использованием т.н. «StudlyCaps» (каждое слово начинается с большой буквы, между словами нет разделителей).
- Константы классов ДОЛЖНЫ быть объявлены исключительно в верхнем регистре с использованием символа подчёркивания для разделения слов.
- Имена методов ДОЛЖНЫ быть объявлены с использованием т.н. «camelCase» (первое слово пишется в нижнем регистре, далее каждое слово начинается с большой буквы, а между словами нет разделителей).

2. Файлы

2.1. PHP-теги

PHP-код ОБЯЗАТЕЛЬНО следует заключать в полную версию (<?php ?>) тегов или укороченную (сокращённую запись echo) версию (<?= ?>) тегов и НЕДОПУСТИМО заключать ни в какие иные разновидности тегов.

2.2. Кодировка символов

PHP-код ДОЛЖЕН быть представлен только в кодировке UTF-8 без BOM-байта.

2.3. Побочные эффекты

В файлах СЛЕДУЕТ либо объявлять структуры (классы, функции, константы и т.п.) и не создавать побочных эффектов (например: передавать данные в выходной поток, модифицировать настройки и т.п.), либо реализовывать логику, порождающую побочные эффекты, но НЕ СЛЕДУЕТ делать одновременно и то, и другое.

Под «побочными эффектами» понимается реализация логики, не связанной с объявлением классов, функций, констант и т.п. – даже подключение внешнего файла уже является «побочным эффектом».

«Побочные эффекты» включают (но не ограничиваются этим перечнем): передачу данных в выходной поток, явное использование require или include, изменение настроек, генерирование ошибочных ситуаций или порождение исключений, изменение глобальных или локальных переменных, чтение из файла или запись в файл и т.п.

3. Имена пространств имён и имена классов

Имена пространств имён и имена классов ДОЛЖНЫ следовать стандарту PSR-0. В конечном итоге это означает, что каждый класс должен располагаться в отдельном файле и в пространстве имён с хотя бы одним верхним уровнем (именем производителя).

Имена классов ДОЛЖНЫ быть объявлены с использованием т.н. «StudlyCaps» (каждое слово начинается с большой буквы, между словами нет разделителей).

Код, написанный для PHP 5.3 и более новых версий, ДОЛЖЕН использовать формальные пространства имён, например:

```
<?php
// PHP 5.3 и новее:
namespace Vendor\Model;
class Foo
{
}
```

В коде, написанном для PHP 5.2.x и ниже, СЛЕДУЕТ при именовании классов соблюдать соглашение о псевдопространствах имён с префиксом в виде имени производителя (Vendor_):

```
<?php
// PHP 5.2.x и ранее:
class Vendor_Model_Foo
{
}
```

4. Константы, свойства и методы классов

Здесь под «классом» следует понимать также интерфейсы (interface) и примеси (trait).

4.1. Константы

Константы классов ДОЛЖНЫ быть объявлены в верхнем регистре с использованием символа подчёркивания в качестве разделителя слов, например:

```
<?php
namespace Vendor\Model;
class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

4.2. Свойства

В данном руководстве намеренно не приводится никаких рекомендаций относительно использования $StudlyCaps, $camelCase или $under_score вариантов именования свойств.

Какой бы вариант именования ни был выбран, СЛЕДУЕТ сохранять его неизменным в рамках некоторого разумного объёма кода (например, на уровне производителя, пакета, класса или метода).

4.3. Методы

Имена методов ДОЛЖНЫ быть объявлены с использованием т.н. «camelCase» (первое слово пишется в нижнем регистре, далее каждое слово начинается с большой буквы, а между словами нет разделителей).

Домашнее задание

1. Разобраться с принципом работы движка.
2. По образу и подобию модуля авторизации из движка V1.0 создать модуль работы с пользователем;
    a. Пользователь должен уметь входить в систему;
    b. Пользователь должен уметь выходить из системы;
    c. У пользователя должен быть личный кабинет (пока пустой).
3. *Научить движок запоминать 5 последних просмотренных страниц. Выводить их в личном кабинете блоком “Вы недавно смотрели”.

Дополнительные материалы

1. http://svyatoslav.biz/misc/psr_translation/ - стандарты PSR

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Мэтт Зандстра - "PHP. Объекты, шаблоны и методики программирования"