Урок 5

Парадигма MVC. Обновления движка

Знакомство с парадигмой-паттерном «Model-View-Controller». Обновление архитектуры системы. Стандартизация кода.

Парадигма MVC

Архитектура системы

Структура БД

Немного о стандартах

PSR-0 – стандарт автозагрузки

PSR-1 – базовый стандарт оформления кода

1. Общие положения

2. Файлы

3. Имена пространств имен и имена классов

4. Константы, свойства и методы классов

Практическое задание

Дополнительные материалы

Используемая литература

Многие начинают писать проект для работы с единственной задачей, не подразумевая, что он может вырасти в многопользовательскую систему управления – например, контентом или производством. Это кажется приемлемым и работает, пока не становится очевидно, что код полностью состоит из костылей и хардкода.

Код, перемешанный с версткой, запросами и костылями, не поддается иногда даже прочтению. При добавлении новых фич приходится долго вспоминать, «а что же там такое написано-то было?», и проклинать себя.

Если посмотреть на движок, написанный на курсе «PHP. Уровень 1» с нынешних позиций, приходит понимание, что он хорош для решения относительно простых задач, но непригоден для использования в сложных бизнес-процессах. В нем неудобно настраивать поведение страниц, он не обладает гибкостью при создании новых шаблонных модулей.

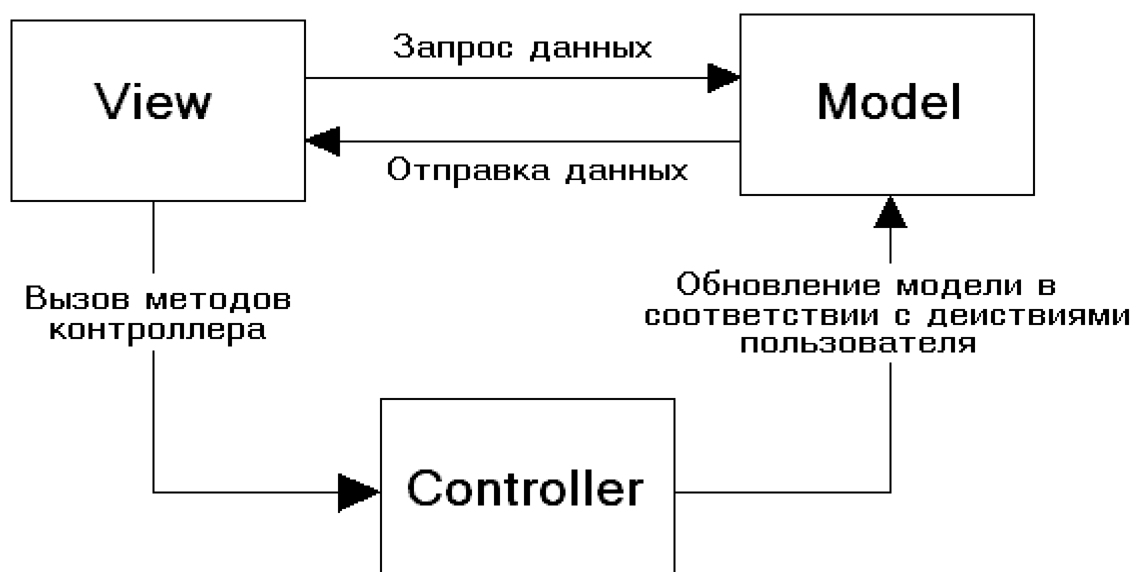
Может быть, вы слышали о шаблонах проектирования. И даже, не испугавшись огромных руководств и документаций, пытались изучить какой-либо из современных фреймворков. Из-за множества архитектурных концепций, хитро увязанных между собой, многие откладывают изучение и применение современных инструментов в «долгий ящик».

На этом уроке мы поговорим, как правильно структурировать код, и познакомимся с апгрейдом нашего движка, версией 2.0, на базе которой будем развивать проект.

Парадигма MVC

Шаблон **MVC** описывает простой способ построения структуры приложения, цель которого – отделить бизнес-логику от пользовательского интерфейса. В результате приложение легче масштабируется, тестируется, сопровождается и 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. Когда пользователь заходит на веб-ресурс, скрипт инициализации создает экземпляр приложения и запускает его на выполнение. При этом отображается вид, например, главной страницы сайта.
2. Приложение получает запрос от пользователя и определяет запрошенные контроллер и действие. В случае с главной страницей выполняется действие по умолчанию (**index**).
3. Приложение создает экземпляр контроллера и запускает метод действия. В нем могут содержаться, например, 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. Поэтому нормально, если она объемная.

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, 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, чтобы задавать внешнее отображение данных, полученных из контроллера и модели.

Виды содержат HTML-разметку и небольшие вставки PHP-кода для обхода, форматирования и отображения данных. Они не должны напрямую обращаться к базе данных – этим занимаются модели. Не должны работать с данными, полученными из запроса пользователя – эту задачу выполняет контроллер. Вид 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 – это связующее звено, соединяющее модели, виды и другие компоненты в рабочее приложение. Контроллер отвечает за обработку запросов пользователя. Он не должен содержать SQL-запросов – их лучше держать в моделях. В нем также не должно быть HTML и другой разметки – ее 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. Структура данных и бизнес-логика, содержащаяся в них, обычно специфична для конкретного приложения.

Одним из важных аспектов 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 не придется подключать **global.php** каждый раз, когда нужно создать новую страницу. Эта «единая точка входа» будет называться **index.php** и на данный момент будет такой:

```
<?php
// Здесь выполняем действия
?>
```

Похожее поведение реализовано в первой версии движка, но там оно имеет слабую структуру кода, так как не используются преимущества ООП. В данном случае мы стремимся сделать прозрачную, но в то же время четкую и удобную структуру кода.

Этот скрипт пока еще ничего не делает. Чтобы направить все запросы на главную страницу, 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**. Если нет – 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.

Если нет возможности использовать **.htaccess** или **mod_rewrite**, то придется вручную адресовать все запросы к **index.php**. Все ссылки должны будут иметь вид «**index.php?path=[здесь-идет-запрос]**». Например, «**index.php?path=chat/index**».

Архитектура системы

Важная часть реализации – 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 формируется из трех частей по шаблону:

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

Таким образом, адрес **/catalog/index/** вызовет контроллер **CatalogController.class.php**, после чего в нем произойдет выполнение метода **index**.

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, чтобы каждый раз не задумываться, какие файлы надо включить в **require**. Используем функцию **spl_autoload_register**, которая позволяет создавать много раздельных автозагрузчиков. Но 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, что пользователь хочет получить. Затем, согласно парадигме MVC, мы создаем экземпляр контроллера, который готовит систему к генерации представления, используя данные из моделей.

Структура БД

Чтобы быстро ознакомиться с современными методиками разработки, ограничим возможности магазина до минимума. Пусть посетитель сайта видит каталог товаров, имеет возможность собрать несколько товаров в корзину и оформить заказ. Стандартные функции по регистрации пользователя портируем из движка V1.

Для реализации такого функционала 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 или неактивен; 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, удален, подтвержден).

Хранить удаленные и неактивные записи может быть необходимо для аналитики. Есть и чисто технический аспект: операция удаления из БД гораздо дороже, чем редактирование.

Немного о стандартах

PHP достаточно поздно обзавелся стандартами написания кода, однако уже успело появиться 8 описаний различных его аспектов. Рассмотрим два из них: **PSR-0** и **PSR-1**.

PSR-0 – стандарт автозагрузки

Требования, обязательные к исполнению, чтобы обеспечить совместимость механизмов автозагрузки:

- Полностью определенное пространство имен и имя класса должны иметь следующую структуру: **\<Vendor Name>\(<Namespace>\)*<Class Name>**.
- Каждое пространство имен должно начинаться с пространства имен высшего уровня, указывающего на разработчика кода («имя производителя»).
- Каждое пространство имен может включать в себя неограниченное количество вложенных подпространств имен.
- Каждый разделитель пространства имен при обращении к файловой системе преобразуется в РАЗДЕЛИТЕЛЬ_ИМЕН_КАТАЛОГОВ.
- Каждый символ «_» («знак подчеркивания») в ИМЕНИ_КЛАССА преобразуется в РАЗДЕЛИТЕЛЬ_ИМЕН_КАТАЛОГОВ. При этом символ «_» не обладает особенным значением в имени пространства имен (и не претерпевает преобразований).
- При обращении к файловой системе полностью определенное пространство имен и имя класса дополняются суффиксом **.php**.
- В имени производителя, имени пространства имен и имени класса допускается использование буквенных символов в любых комбинациях нижнего и верхнего регистров.

PSR-1 – базовый стандарт оформления кода

Этот раздел описывает стандартные элементы, существенные 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 имен и имена классов **должны** следовать стандарту **PSR-0**.
- Имена классов **должны** быть объявлены с использованием «**StudlyCaps**» (каждое слово начинается с большой буквы, между словами нет разделителей).

- Константы классов **должны** быть объявлены исключительно в верхнем регистре с использованием символа подчеркивания для разделения слов.
- Имена методов **должны** быть объявлены с использованием «**camelCase**» (первое слово пишется в нижнем регистре, далее каждое слово начинается с большой буквы, между словами нет разделителей).

2. Файлы

2.1. PHP-теги

PHP-код **обязательно** следует заключать в полную версию тегов (**<?php ?>**) или укороченную версию тегов (**<?= ?>**) (сокращенную запись **echo**) и **недопустимо** заключать его 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.

3. Имена пространств имен и имена классов

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

Имена классов **должны** быть объявлены с использованием «**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 имен, 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 имен 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 подчеркивания 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 вариантов именования свойств **$StudlyCaps**, **$camelCase** или **$under_score**.

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

4.3. Методы

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

Практическое задание

1. Разобраться с принципом работы движка.
2. По образу 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 пять последних просмотренных страниц. Выводить их в личном кабинете блоком «Вы недавно смотрели».

Дополнительные материалы

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

   К сожалению ссылка устарела.

   Для разработчиков на современном этапе наиболее интересен стандарт psr-4. Собственно он интересен тем, что позволяет не писать свой автозагрузчик классов, а взять уже готовый, например встроенный в composer/

   https://github.com/codedokode/pasta/blob/master/php/autoload.md

2. htaccess - небольшая шпаргалка

   https://ruseller.com/lessons.php?rub=29&id=2860
   https://semantica.in/blog/chto-takoe-rewriterule-i-dlya-chego-on-nuzhen.html

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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