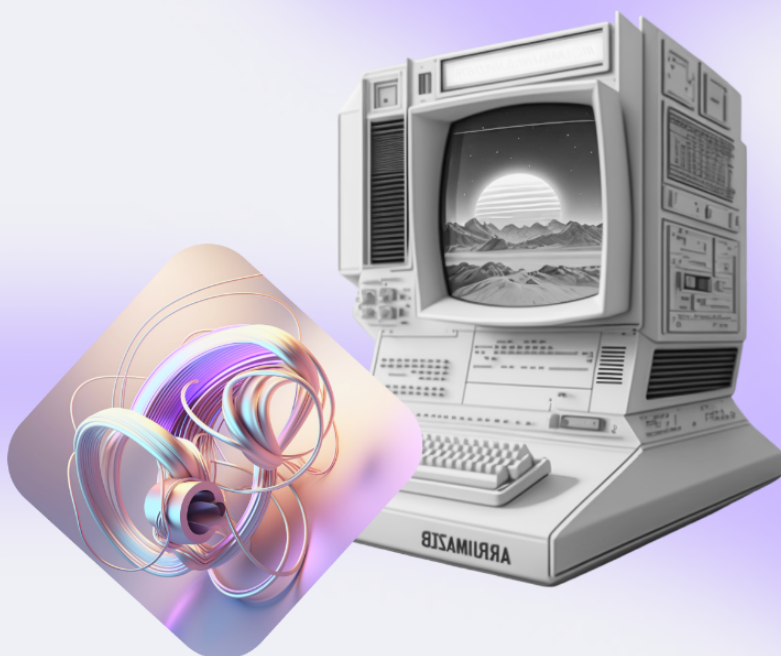


Каркас приложения

Основы PHP



Оглавление

Введение	2
Словарь терминов	3
Разделение на слои	3
MVC	4
Модель	5
Вид	6
Контроллер	6
Организация кода	7
Точка входа	7
Обработка маршрутов	9
Автозагрузка	11
Метод вызова	16
Представление	20
Генерация частей страницы	23
Воспроизводим первое приложение в ООП стиле	25
Заключение	29
Домашнее задание	29
Что можно почитать еще?	30

Введение

На прошлой лекции мы ознакомились с фундаментальными концепциями ООП. Классы совместно с пространствами имен помогают организовать структурированное понятное приложение, которое можно легко разрабатывать и развивать.

Поэтому в рамках этой лекции мы с вами должны:

- понять, почему приложение нужно делить на составные части
- освоить приемы такого разделения
- научиться применять автозагрузку в контексте ООП
- собрать каркас приложения

Словарь терминов

Разделение слоев MVC – это простой способ организации структуры приложения.

Слой модели (Model) – это данные и правила бизнес-логики.

Слой представления (View) – пользовательский интерфейс.

Контроллер (Controller) – взаимодействие между моделью и представлением.

Частичные шаблоны – блоки сайта с контентом, генерируемым для конкретных типов страниц.

Функция `spl_autoload_register()` – эта функция, которая содержит логику, которая по имени файла будет искать его на диске.

TWIG – это шаблонный движок, разработанный для языка программирования PHP. Он предоставляет простой и гибкий способ создания и управления шаблонами веб-страниц. Он основывается на концепции разделения логики приложения и представления данных, что помогает создавать более модульный и легко поддерживаемый код.

Разделение на слои

Пока мы с вами разрабатываем относительно несложные приложения в рамках обучения, может показаться, что структурность кода не так важна. Главное – побыстрее создать что-то, чем будут пользоваться. Однако в мире IT приложения развиваются настолько динамично, что за одну рабочую неделю в них может появляться множество новых функциональных частей. При этом любое приложение начинается с маленькой разработки, предлагающей минимальный функционал (так называемый MVP – minimum viable product).

Представьте, что вы – владелец склада с ценным товаром, который у вас активно покупают. Чем активнее товар покупается, тем больше его надо размещать на

складе. Но если товар будет размещен по складу хаотично, то вы не сможете в адекватные сроки понять, сколько же единиц каждого наименования товара у вас имеется.

Также и с кодом. Он требует порядка. Иначе изменения в нем с каждым обновлением будут производиться все дольше и дольше. А значит, приложение будет либо доставляться пользователю недопустимо долго, либо качество будет хромать.

Например, если наш PHP-код будет перемешан с элементами верстки, то при наличии большого приложения будет крайне сложно отыскать место, в котором генерируется та или иная часть сайта.

Более того, многие современные приложения уже следуют принципам, о которых мы с вами поговорим в рамках этой лекции. А ведь с большой вероятностью вы будете работать в команде с уже существующей кодовой базой. И это означает, что вам надо понимать, почему код организован именно так, а никак иначе.

В нашей первой лекции мы встраивали PHP-логику прямо в HTML-код. И там мы уже отмечали, что это не самая лучшая практика. Поэтому давайте погружаться в механики того, как же код должен быть организован.

С какими сущностями мы работали до этого?

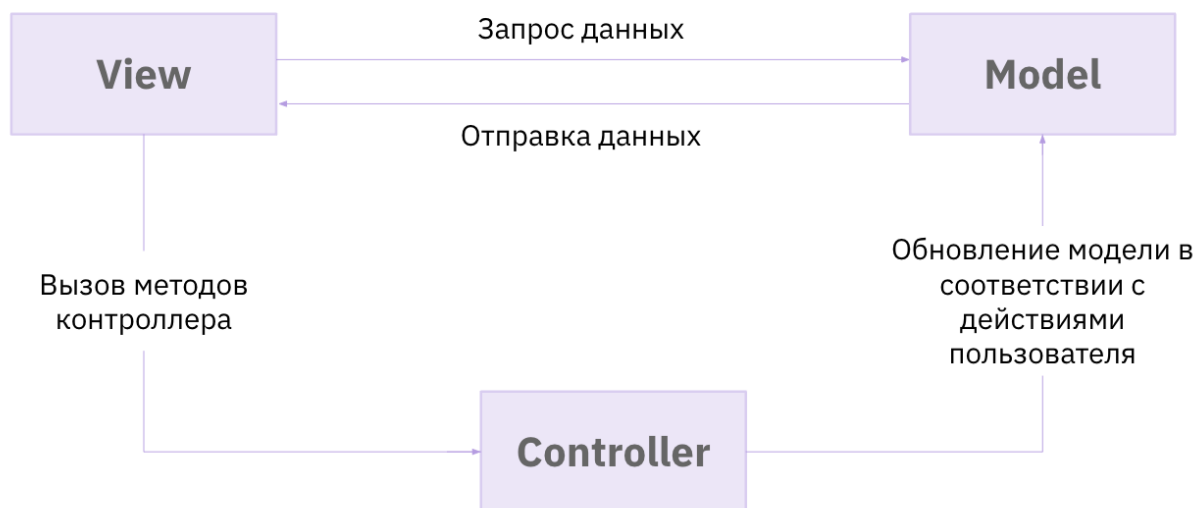
С точки зрения логики нам нужно было уметь обработать HTTP-запрос, собрать для него данные и вывести для пользователя какой-то ответ.

Эти блоки действий можно представить в виде прототипов наших слоев. Но, как и многое в программировании, эта задача уже имеет классическое решение.

MVC

💡 Разделение слоев MVC – это простой способ организации структуры приложения.

💡 Его цель – отделение бизнес-логики от техники работы приложения и пользовательского интерфейса. Такое разделение позволяет облегчить поддержку, расширение и тестирование.



В MVC слой модели (Model) – это данные и правила бизнес-логики, слой представления (View) – пользовательский интерфейс, а контроллер (Controller) – взаимодействие между моделью и представлением.

💡 Как здесь происходит работа:

Пользователь отправляет нам URI с заголовками, который приходит в наш PHP-скрипт. Он создает экземпляр приложения и запускает его на выполнение. По итогам работы формируется вид выбранной при помощи URI страницы сайта.

Модель

Модель содержит бизнес-логику приложения. Если проводить аналогию с нашим приложением хранилища пользователей, то модель будет отвечать за то, какие данные о пользователе нужно собирать приложению, как и куда они будут сохраняться.

Модель не имеет права напрямую взаимодействовать с HTTP-запросом. Все данные, которые надо передать, должны сформироваться в контроллере.

Также модель не должна заниматься генерацией HTML или любого другого вывода. Это задача слоя view.

Вид

Вид применяется для создания того, что увидит пользователь. Виды могут содержать HTML-разметку и элементарные вставки PHP-кода: например: вывод переменных или цикл для обхода массивов.

В виде не должно быть сложных логических конструкций и, тем более, прямого описания бизнес или технической логики.

Как и модели, видам не нужно ничего знать о том, какой именно запрос пришел от пользователя. Эту задачу должен выполнять контроллер.

При этом вид имеет право напрямую обращаться к свойствам и методам контроллеров или моделей, чтобы получить нужный пакет данных.

Как правило, виды имеют иерархическую структуру, в которой выделяются:

- общий шаблон, в котором содержится разметка, общая для всех страниц;
- частичные шаблоны – блоки сайта с контентом, генерируемым для конкретных типов страниц.

В нашем приложении хранилища роль слоя выполнял код в `template`-файле.

Контроллер

Контроллер – это тот краеугольный камень, клей, который соединяет модели, виды и другие компоненты в рабочее приложение. Контроллер отвечает за обработку запросов пользователя, но не должен содержать в себе работу с хранилищами, HTML-разметку. Основная задача контроллера – принять запрос, вытащить из него данные, согласно конфигурации приложения понять, какие действия нужно совершить и передать всю информацию модели для выполнения бизнес-логики.



В хорошем MVC контроллеры маленькие и содержат немного кода. Признаком плохого проектирования будет антипаттерн Fat Stupid Ugly Controllers (FSUC – толстый тупой уродливый контроллер).

Модели будут большими, так как содержат большую часть кода.

Организация кода

Теперь нам нужно организовать код так, чтобы все наши запросы приходили в одну точку приложения, попадали на слой контроллера, передавались на выполнение моделям и выводились в виде ответа при помощи представлений.

Точка входа

И снова мы уже знакомы с такой организацией кода, ведь мы делали её в нашем прошлом приложении.

Но в web-приложении нам нужно будет немного модифицировать уже знакомый нам паттерн Front Controller, чтобы web-сервер Nginx мог знать, что все запросы к нашему сайту будут обрабатываться нашим приложением. Исключение будут составлять только файлы статики, которые должны отдаваться сразу же средствами Nginx.

Если посмотреть в исходники из первой лекции, то можно увидеть там файл настроек нашего сайта по адресу:

```
nginx/hosts/mysite.local.conf
```

Вернемся к его содержимому:

```
server {
    # указываем 80 порт для соединения
    listen 80;
    # нужно указать, какому доменному имени принадлежит наш конфиг
    server_name mysite.local;

    # задаём корневую директорию
    root /data/mysite.local;

    # стартовый файл
    index index.php index.html;

    # при обращении к статическим файлам логи не нужны, равно как и обращение к
    fpm
    # http://mysite.local/static/some.png
    location ~* \.(jpg|jpeg|gif|css|png|js|ico|html)$ {
        access_log off;
        expires max;
    }
}
```

```

# помним про единую точку доступа
# все запросы заворачиваются в корневую директорию root на index.php
location / {
    try_files $uri $uri/ /index.php?$query_string;
}

# и наконец правило обращения к php-fpm
location ~* .php$ {
    try_files $uri = 404;
    fastcgi_split_path_info ^(.+\.php) (/.+)$;
    fastcgi_pass app:9000;
    #fastcgi_pass unix:/var/run/php-fpm.sock;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    include fastcgi_params;
}
}

```

Мы уже знаем, что директивы `location`, отвечающие за обработку адресов в Nginx, обрабатываются по порядку следования, а выбирается первая подходящая под адрес.

Таким образом, в начале указываются частные директивы, а затем – более общие.

Первая директива – это обработка статики. Как мы уже условились, статика не должна вызывать PHP:

```
location ~* \.(jpg|jpeg|gif|css|png|js|ico|html)$
```

Вторая директива – это ключ к реализации нашего паттерна Front Controller. Условие этой директивы:

```
location / {
```

говорит нам о том, что она будет принимать на себя абсолютно все запросы, которые не попали в обработку первой директивы (символ обратного слэша).

В теле директивы мы говорим, что адрес, переданный нам, должен быть принудительно направлен на файл `index.php`. Но как же тогда мы узнаем, какой адрес хотел вызвать пользователь? Тут надо обратить внимание на то, что после `index.php` будет идти дополнение в виде серверных параметров вызова:

```
try_files $uri $uri/ /index.php?$query_string;
```


Посмотрим, что у нас получается. Добавим в index.php всего одну строчку:

```
<pre><? var_dump($_SERVER); ?> </pre>
```

Теперь мы видим, что если вызвать в браузере адрес:

<http://mysite.local/controller/action>

то мы увидим, что:

- в ключе SCRIPT_FILENAME будет содержаться index.php, то есть мы вызвали ожидаемый файл;
- в ключе REQUEST_URI будет содержаться вызванный пользователем адрес – /controller/action.

Эти знания уже позволяют нам заключить соглашение в отношении адресов.



Каждый адрес будет строиться по следующему правилу:

/контроллер/действие/?параметры

То есть, первый блок адреса всегда будет указывать на контроллер, который надо вызвать. Второй – на метод в нем. А для передачи дополнительных данных будут использоваться GET-переменные в адресе.

Например, если мы заходим сохранить в хранилище нашего пользователя, то URI будет сформирован так:

```
/user/save/?name=Иван&birthday=05-05-1991
```

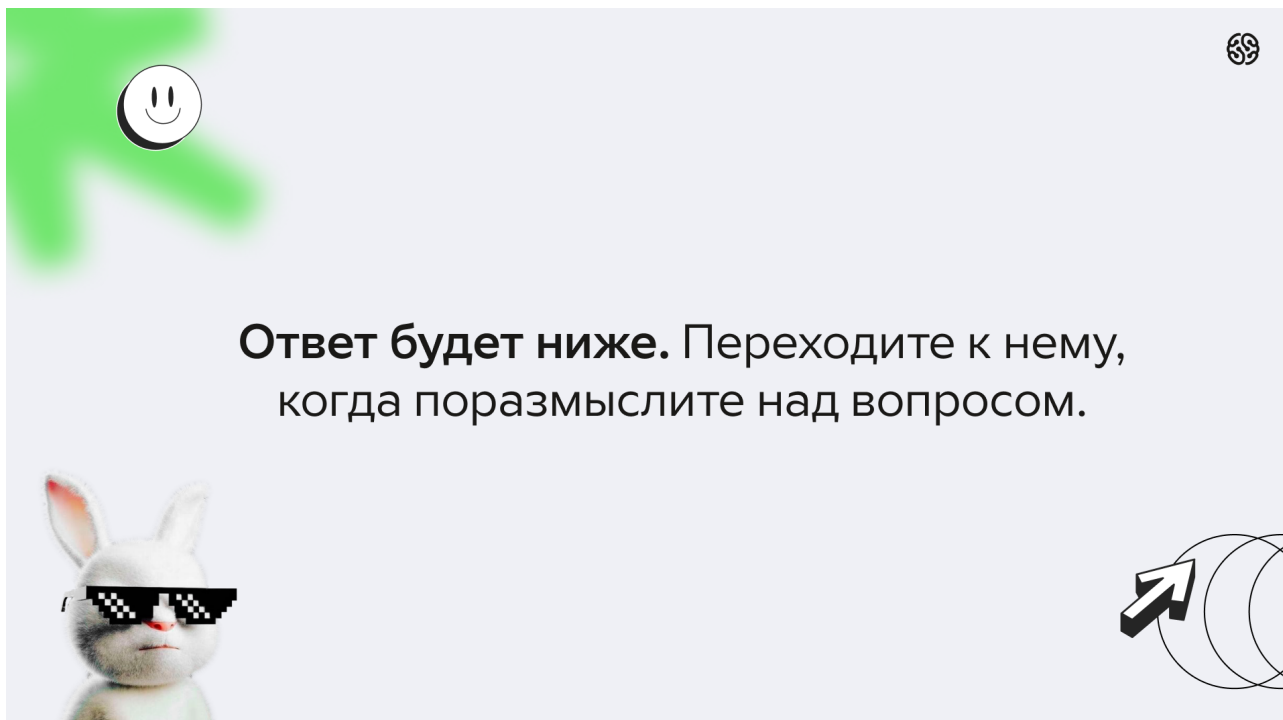
Конечно, можно передавать параметры и в POST, если применять HTML-формы, к чему мы еще вернемся.

Обработка маршрутов

Теперь мы можем обрабатывать маршруты. Как мы условились в контракте выше, нам надо разбивать наш адрес и выяснять, что же нужно будет вызвать. Для начала мы научимся генерировать главную страницу нашего сайта.

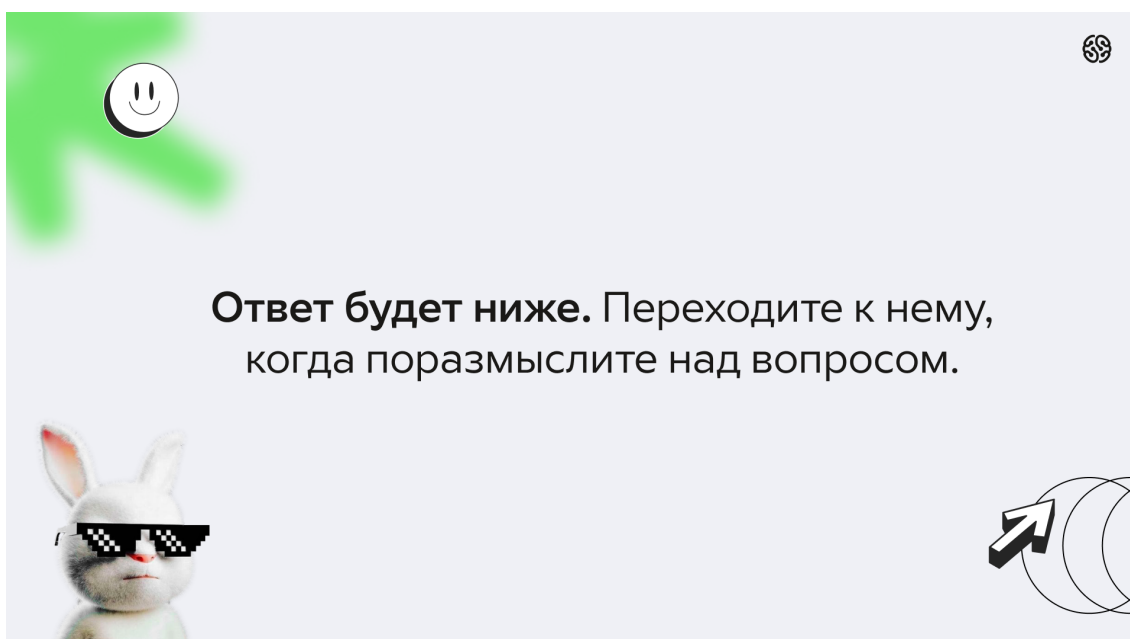
Для того, чтобы реализовать эту задачу нам надо ответить на несколько вопросов.

Вопрос 1. На какой адрес она будет отвечать?



Очевидно, что главная страница в нашей настройке будет отвечать на адрес <http://mysite.local>. Но в нем нет адреса, соответствующего нашему правилу. Значит, мы считаем, что если у нас нет имени контроллера и действия, то будем вызывать в будущем классе PageController метод `actionIndex`. То есть, это будет действие по-умолчанию.

Вопрос 2. Что делать, если указанному адресу не сопоставлены метод или контроллер?



Например, пользователь пытается вызвать адрес

<http://mysite.local/some/save/>

которого у нас нет (да и вовсе непонятно, за что он может отвечать). В таком случае согласно спецификации HTTP мы должны возвращать 404 ошибку (Страница не найдена). Более того, мы будем возвращать её даже если контроллер есть, но в нем нет нужного действия.

Таким образом, наше правило в области реализации будет выглядеть следующим образом:

Для адреса:

```
/user/save/?name=Иван&birthday=05-05-1991
```

должен вызываться контроллер UserController, в котором будет метод actionSave.

Мы можем начать описывать это правило поведения в классах. Но мы уже знаем, что классы надо как-то подключать. Значит, нам надо применить здесь уже знакомую нам автозагрузку. Ведь в нашем index.php мы не хотим подключать уйму файлов в явном виде.

Автозагрузка

В ООП, согласно конвенциям PSR, каждый класс размещается в отдельном файле. Это регламентируется правилом PSR-1.

Класс, как мы уже знаем, может вызывать экземпляры других классов. Чтобы код не выдавал ошибки, для каждого вызываемого класса нам пришлось бы в начале каждого файла подключать файлы, где описаны вызываемые классы.

Вообще говоря, PHP позволяет описать самостоятельную функцию-автозагрузчик при помощи spl_autoload_register. Эта функция будет содержать логику, которая по имени файла будет искать его на диске. Но реализация самостоятельного загрузчика – не самая лучшая идея, так как удобно, когда в разных проектах правила отыскания файла на диске общие. Тогда каждый разработчик вместо изобретения велосипеда будет пользоваться общей конвенцией.

Задачу загрузки файлов решает рекомендация PSR-4. Она требует именовать файл, взяв название класса без пространства имен. При этом пространство имен определяет директорию, в которой будет лежать наш класс.

Например, класс:

```
GeekBrains\Application\Router
```

будет лежать в файле по адресу:

```
GeekBrains/Application/Router.php
```

Для того, чтобы такая логика заработала, нам надо модифицировать уже знакомый нам файл `composer.json`, описав в нем корневой namespace.

Для этого мы будем использовать уже знакомый нам образ `php-gb-cli`, в котором у нас установлен `composer`. Если вы вдруг удалили его, то вот его `Dockerfile`:

```
FROM php:8.2
VOLUME /data

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin
--filename=composer

ENV COMPOSER_ALLOW_SUPERUSER 1

WORKDIR /data/mysite.local
```

Теперь, находясь в директории нашего проекта, выполним команду:

```
docker container run -it -v ${pwd}/code:/data/mysite.local/ php-gb-cli composer init
```

Она запустит создание нашего проекта. Назовем его `Application 1`:

```
Package name (<vendor>/<name>) [root/code]: geekbrains/application1
```

Ниже приведен весь диалог создания приложения:

```
Package name (<vendor>/<name>) [root/code]: geekbrains/application1
Description []:
Author [n to skip]:
Invalid author string. Must be in the formats: Jane Doe or John Smith
<john@example.com>
Author [n to skip]: n
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []:
License []:
Define your dependencies.
Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively
[yes]? no
Add PSR-4 autoload mapping? Maps namespace "Geekbrains\Application1" to the
entered relative path. [src/, n to skip]:
{
    "name": "geekbrains/application1",
    "autoload": {
        "psr-4": {
            "Geekbrains\\Application1\\": "src/"
        }
    },
    "require": {}
}
Do you confirm generation [yes]? yes
```


Обратите внимание на то, что при вызове контейнера мы указали для него volume в директории code. Так что, если вы все сделали правильно, то в code у вас должны будут появиться:

- директория src – в ней будут лежать исходники наших классов
- директория vendor – там будут лежать автогенерируемые Composer-ом файлы
- файл composer.json.

В нем появилась секция:

```
"autoload": {
    "psr-4": {
        "Geekbrains\\Application1\\": "src/"
    }
}
```

Она говорит о том, что у нас есть директория src, в которой будут лежать все классы, чей namespace начинается с Geekbrains\Application1.

 Регистр написания важен – если написать PSR-4 вместо psr-4, то Composer не сможет создать автозагрузчик.

В имени пространства имен два раза пишется backslash, так как это правила формата JSON (чтобы в строку вставить backslash, его надо написать 2 раза, backslash в JSON – это экранирующий символ).

Настало время протестировать подключение классов с методами. В файл index.php запишем подключение автозагрузчика:

```
<?php
require_once('./vendor/autoload.php');
```

Теперь создадим наш первый контроллер. В директории src разместим вложенную директорию Controllers, в которой будет лежать наш PageController в файле PageController.php. Согласно правилу PSR-4 для него мы зададим пространство имен:

```
<?php
// файл лежит в code/src/Controllers/PageController.php
namespace Geekbrains\Application1\Controllers;
```

Опишем в нем метод actionIndex:

```
class PageController {
    public function actionIndex() {
        echo "Hello";
    }
}
```

Теперь мы готовы к тому, чтобы вызвать наш метод. Для этого в `index.php` нам надо указать пространство имен, которое мы будем использовать:

```
<?php
require_once('..../vendor/autoload.php');

use Geekbrains\Application1\Controllers\PageController;

$app = new PageController();
$app->actionIndex();
```

После запуска наших контейнеров командой:

```
docker-compose up
```

в браузере мы можем открыть адрес <http://mysite.local> и увидеть там слово Hello.

Но это только первый шаг. Мы ведь хотим, чтобы у нас было полноценное приложение, которое реагирует на адреса, а не требует явного вызова контроллера.

Поэтому давайте создадим класс `Application` в директории `src`, который будет отвечать за обработку запросов.

```
<?php

namespace Geekbrains\Application1;

class Application {

}
```

Именно этот класс мы будем вызывать каждый раз, когда пользователь вызывает в строке браузера какой-либо адрес.

При вызове адреса нам нужно:

1. Считать адрес и разбить его на параметры вызова
2. Проверить, что контроллер и метод доступны к вызову
3. Создать экземпляр контроллера
4. Вызвать у него метод
5. Вернуть ответ пользователю

Метод вызова

В классе Application начнем с определения свойств и констант. Контроллеры согласно нашему правилу автозагрузки PSR-4 всегда будут храниться в одном и том же namespace. Значит, это константа:

```
private const APP_NAMESPACE = 'Geekbrains\Application1\Controllers\\';
```

Теперь создадим свойства класса, в которых будем хранить имя контроллера, который надо будет создать и имя метода, который надо будет вызвать.

```
private string $controllerName;  
  
private string $methodName;
```

Теперь можем приступить к описанию логики. Пока она вся будет уместиться в один метод run:

```
public function run() {  
    // разбиваем адрес по символу слеша  
  
    // определяем имя контроллера  
  
    // проверяем контроллер на существование  
  
    // создаем экземпляр контроллера, если класс существует  
  
    // проверяем метод на существование  
  
    // вызываем метод, если он существует  
}
```

Мы получили понимание того, как будет реализован метод. Давайте пошагово его создадим.

Разбить строку по символам можно встроенной в PHP функцией [explode](#).

```
// разбиваем адрес по символу слеша  
  
$routeArray = explode('/', $_SERVER['REQUEST_URI']);
```


В полученном массиве в нулевом элементе всегда будет пустота, так как строка в REQUEST_URI всегда начинается со слеша. Дальше будут элементы адреса. Так, если пользователь вызовет страницу <http://mysite.local/page/index/>, то в массиве будет лежать:

```
0 => "  
1 => page  
2 => index
```

Но пользователь может не задать и ничего. Тогда элемент с номером 1 будет пуст. В таком случае мы должны вызывать контроллер по умолчанию (PageController).

```
// определяем имя контроллера  
if(isset($routeArray[1]) && $routeArray[1] != "") {  
    $controllerName = $routeArray[1];  
}  
else{  
    $controllerName = "page";  
}  
  
$this->controllerName = Application::APP_NAMESPACE . ucfirst($controllerName) . "Controller";
```

Мы проверяем, задан ли в адресе контроллер.

Дальше мы получаем полное имя класса. Оно всегда содержит в себе namespace, который хранится у нас в константе. Далее имя контроллера всегда должно быть с большой буквы. Так строку трансформирует встроенная функция [ucfirst](#). Она делает заглавной первую букву переданной строки, а остальные оставляет строчными. В конце мы добавляем стандартное слово Controller.

Для указанного адреса <http://mysite.local/page/index/> будет сформировано значение:

```
Geekbrains\Application1\Controllers\PageController
```

Однако пользователь может ввести в строку что угодно. Значит, надо проверить класс на существование. Это можно сделать при помощи встроенной в PHP функции [class_exists](#).

```
if(class_exists($this->controllerName)){  
}  
  
else {  
    return "Класс не существует";  
}
```

В случае, если класс существует, мы должны сформировать похожим образом имя метода и проверить его на существование:

```
if(class_exists($this->controllerName)) {  
    // пытаемся вызвать метод  
    if(isset($routeArray[2]) && $routeArray[2] != '') {  
        $methodName = $routeArray[2];  
    }  
    else {  
        $methodName = "index";  
    }  
  
    $this->methodName = "action" . ucfirst($methodName);  
  
    if(method_exists($this->controllerName, $this->methodName)) {  
    }  
    else {  
        return "Метод не существует";  
    }  
}
```



Обратите внимание на то, что в `method_exists` нужно передать имя класса, в котором мы будем искать метод.

Теперь, если класс и метод в нем существуют, мы можем создать экземпляр класса и вызвать у него нужный метод.

Это можно сделать при помощи функции [call_user_func_array](#)

```
if(method_exists($this->controllerName, $this->methodName)){
    $controllerInstance = new $this->controllerName();
    return call_user_func_array(
        [$controllerInstance, $this->methodName],
        []
    );
}
```

В нее первым параметром передается массив, содержащий экземпляр класса и метод, который нужно вызвать. Во втором параметре в массиве можно передать свойства вызова. Но для нас это пока неактуально.

Итак, у нас получился метод вызова:

```
public function run() : string {
    $routeArray = explode('/', $_SERVER['REQUEST_URI']);

    if(isset($routeArray[1]) && $routeArray[1] != '') {
        $controllerName = $routeArray[1];
    }
    else{
        $controllerName = "page";
    }

    $this->controllerName = Application::APP_NAMESPACE .
ucfirst($controllerName) . "Controller";

    if(class_exists($this->controllerName)){
        // пытаемся вызвать метод
        if(isset($routeArray[2]) && $routeArray[2] != '') {
            $methodName = $routeArray[2];
        }
        else {
            $methodName = "index";
        }

        $this->methodName = "action" . ucfirst($methodName);

        if(method_exists($this->controllerName, $this->methodName)){
            $controllerInstance = new $this->controllerName();
            return call_user_func_array(
                [$controllerInstance, $this->methodName],
                []
            );
        }
        else {
            return "Метод не существует";
        }
    }
}
```

```
        else{
            return "Класс $this->controllerName не существует";
        }
    }
}
```

Теперь надо отредактировать обращение уже существующих файлов. Начнем с index.php:

```
<?php

require_once('./vendor/autoload.php');

use Geekbrains\Application1\Application;

$app = new Application();
echo $app->run();
```

Теперь мы обращаемся к нашему главному классу и вызываем в нем главный метод. Результат выводится на экран.

Чуть отредактируем наш PageController, заменив в нем echo на return.

```
class PageController {

    public function actionIndex() {
        return "Hello";
    }

}
```

Теперь наш каркас роутинга работает так, как ожидается. Посмотрите, что получится, если теперь вызвать адрес <http://mysite.local/page/index/>

Представление

Мы уже условились, что будем использовать слой представления для того, чтобы писать HTML-верстку и заниматься выводом на экран. Конечно, здесь также можно написать свою логику, но мы будем использовать готовые библиотеки для шаблонизации (то есть, для подготовки стандартных файлов вывода).

TWIG – это шаблонный движок, разработанный для языка программирования PHP. Он предоставляет простой и гибкий способ создания и управления шаблонами веб-страниц. Он основывается на концепции разделения логики приложения и представления данных, что помогает создавать более модульный и легко поддерживаемый код.

TWIG широко используется в популярных PHP-фреймворках, например, таких как Symfony, для рендеринга представлений веб-приложений.

TWIG можно подключить к приложению при помощи composer. TWIG потребует дополнительные компоненты для установки и выполнения, поэтому нам потребуются расширения Docker-контейнеров, которые надо добавить в Dockerfile для cli и fpm:

```
FROM php:8.2-cli

COPY ./php.ini /usr/local/etc/php/conf.d/php-custom.ini

RUN curl -sS https://getcomposer.org/installer | php --
--install-dir=/usr/local/bin --filename=composer

ENV COMPOSER_ALLOW_SUPERUSER 1

RUN apt-get update && apt-get install zip unzip

WORKDIR /data/mysite.local

VOLUME /data/mysite.local
```

После пересборки образа контейнера мы выполним команду:

```
docker container run -it -v ${pwd}/code:/data/mysite.local/ php-gb composer
require "twig/twig:^3.0"
```

У нас обновится файл composer.json и появится возможность использовать функционал шаблонизатора. Также нужно создать в директории code вложенную директорию cache, куда TWIG будет складывать сгенерированные странички.

Сразу же будем создавать наш сайт так, чтобы он мог иметь общий стиль. Поэтому нам надо будет создать

1. Основной шаблон – то, что будет общим у всех страниц (заголовки, шапка)
2. Подключаемые шаблоны – то, что может переиспользоваться

Шаблоны будем хранить в директории src/Views.

Создадим там наш первый шаблон – main.tpl:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Geekbrains Application 1</title>
  </head>
  <body>
  </body>
</html>
Для работы с TWIG нам лучше всего создать отдельный класс рендеринга – Render.
<?php

namespace Geekbrains\Application1;

use Twig\Loader\FilesystemLoader;
use Twig\Environment;

class Render {

    private string $viewFolder = '/src/Views/';
    private FilesystemLoader $loader;
    private Environment $environment;
}
```

В нем мы уже будем хранить адрес папки с шаблонами, а также ссылки на необходимые для TWIG переменные. Обязательно ознакомьтесь с [документацией](#) TWIG, чтобы иметь возможность обращаться к ней при необходимости.

Теперь создадим простой метод генерации нашего ответа:

```
public function __construct(){
    $this->loader = new FilesystemLoader($_SERVER['DOCUMENT_ROOT'] .
    $this->viewFolder);
    $this->environment = new Environment($this->loader, [
        'cache' => $_SERVER['DOCUMENT_ROOT'].'/cache/',
    ]);
}

public function renderPage() {
    $template = $this->environment->load('main.tpl');
    return $template->render(['title' => "Title for my site"]);
}
```

В конструкторе мы будем создавать окружение TWIG и настраивать его. FilesystemLoader по указанному пути будет давать возможность загружать файлы шаблонов. Environment будет отвечать за сам вызов TWIG.

Метод `renderPage` будет генерировать страницу на базе нашего базового шаблона, передавая в него пока только переменную `title`. К ней мы уже можем обратиться внутри самого шаблона:

```
<head>

    <title>{{ title }}</title>

</head>
```

Переменные в TWIG берутся в двойные фигурные скобки.

Теперь чуть изменим наш `PageController`, чтобы он мог вызывать TWIG:

```
public function actionIndex() {

    $render = new Render();

    return $render->renderPage();

}
```

Не забудьте в начале этого файла подключить нужный namespace

`use Geekbrains\Application1\Render;`

Если все сделано правильно, то наша страница <http://mysite.local> теперь будет генерироваться из шаблона.

Генерация частей страницы

Очевидно, что для каждого вызванного адреса потребуется генерация своей собственной части контента. Поэтому универсальным `main.tpl` не сделать. Но мы можем подключить к нашей странице другие шаблоны.

Пусть внутри нашей страницы будет идти вызов нужного шаблона, который будет определяться логикой вызова.

Чуть изменим наш метод `renderPage` в классе `Render`:

```
public function renderPage(string $contentTemplateName = 'page-index.tpl', array
$templateVariables = []) {
    $template = $this->environment->load('main.tpl');

    $templateVariables['content_template_name'] = $contentTemplateName;

    return $template->render($templateVariables);
}
```

Он все также подключает основной шаблон, но принимает в качестве параметров имя подключаемого шаблона и массив переменных.

В самом шаблоне `main.tpl` используем конструкцию подключения шаблона. Она похожа на встроенную функцию подключения файлов в PHP:

```
<body>
    {% include content_template_name %}
</body>
```

Теперь наш главный шаблон, задавая каркас верстки всему сайту, может передавать генерацию контента частичным шаблонам.

Доработаем наш метод `actionIndex` в `PageController`:

```
public function actionIndex() {
    $render = new Render();

    return $render->renderPage('page-index.tpl', ['title' => 'Главная
страница']);
}
```

Теперь мы можем указывать, какой именно шаблон контента нужно подключить. Осталось лишь создать шаблон `page-index.tpl`:

```
<p>Наше первое приложение!</p>
```

Если все сделано правильно, то главная страница вернет нам ожидаемый результат.

Обратите внимание, что TWIG кэширует шаблоны, так что изменения могут быть видны не сразу. Это сделано для ускорения работы шаблонизатора. Если вы не видите изменения, для начала очистите директорию `cache`.

Воспроизводим первое приложение в ООП стиле

Давайте воспроизведем часть нашего приложения с прошлого занятия. У нас было хранилище пользователей. Научимся выводить его содержимое на экран.

Для этого нам потребуются

1. Контроллер обработки запросов к хранилищу
2. Модель работы с хранилищем
3. Шаблон вывода данных из хранилища

Создадим в директории code вложенную директорию storage, куда разместим наш файл birthdays.txt:

Иван Иванов, 05-05-1991

Петр Петров, 07-07-1990

Настало время создать нашу первую модель. Это будет модель User.

Что мы знаем о пользователе? У него есть имя и день рождения. Это уже свойства нашего класса:

```
<?php
namespace Geekbrains\Application1\Models;
class User {
    private string $userName;
    private ?int $userBirthday;
?>
```

Обратите внимание на то, как мы будем хранить день рождения.

Во-первых, это будет integer-значение.



Timestamp — это числовая информация о дате и времени, указывающая на конкретный момент времени или временной интервал.

Обычно timestamp представляется в виде числа или строки, которые представляют количество прошедших секунд (или миллисекунд) с определенного начального момента времени, известного как "эпоха".

Типичное использование timestamp связано с фиксацией или индексацией событий, например, в журналах событий (логах) или в базах данных. Он может использоваться для определения порядка событий, измерения временных интервалов или вычисления продолжительности между двумя моментами времени.

Timestamps могут быть представлены в различных форматах, таких как UNIX timestamp, представляющий количество секунд, прошедших с начала эпохи UNIX (1 января 1970 года, 00:00:00 UTC).

Во-вторых, оба свойства могут принимать значение null. Поэтому перед их типов ставится знак вопроса.

Далее создадим стандартные методы для класса с приватными свойствами:

```
public function __construct(string $name = null, int $birthday = null){
    $this->userName = $name;
    $this->userBirthday = $birthday;
}

public function setName(string $userName) : void {
    $this->userName = $userName;
}

public function getUserName(): string {
    return $this->userName;
}

public function getUserBirthday(): int {
    return $this->userBirthday;
}
```

Но день рождения в нашем хранилище записан в виде строки формата «День-месяц-год». Поэтому её надо приводить к типу метки времени timestamp:

```
public function setBirthdayFromString(string $birthdayString) : void {
    $this->userBirthday = strtotime($birthdayString);
}
```

В этом нам поможет встроенная функция [strtotime](#).

Теперь надо реализовать подключение к хранилищу. Оно у нас уже готово. При этом оно не зависит от состояния User, так что метод будет статическим.

Также в статическом свойстве будем хранить адрес нашего хранилища:

```
private static string $storageAddress = '/storage/birthdays.txt';
public static function getAllUsersFromStorage(): array|false {
    $address = $_SERVER['DOCUMENT_ROOT'] . User::$storageAddress;

    if (file_exists($address) && is_readable($address)) {
        $file = fopen($address, "r");

        $users = [];

        while (!feof($file)) {
            $userString = fgets($file);
            $userArray = explode(",", $userString);

            $user = new User(
                $userArray[0]
            );
            $user->setBirthdayFromString($userArray[1]);

            $users[] = $user;
        }

        fclose($file);

        return $users;
    }
    else {
        return false;
    }
}
```

По сути, мы переиспользовали логику нашего приложения из 3 лекции, немного её модифицировав. Чтение у нас идет построчное. Каждую строку мы разбиваем на две части по запятой. Из каждой строки мы генерируем объект класса User. Полученные объекты складываем в массив и передаём в return. В случае ошибки метод вернет значение false.

Теперь создадим контроллер UserController. В нем будет только один метод.

Его задача будет состоять в том, чтобы обратиться к модели и получить у нее необходимые данные:

```
public function actionIndex() {
    $users = User::getAllUsersFromStorage();

    $render = new Render();

    if(!$users){
        return $render->renderPage(
            'user-empty.tpl',
            [
                'title' => 'Список пользователей в хранилище',
                'message' => "Список не найден"
            ]);
    }
    else{
        return $render->renderPage(
            'user-index.tpl',
            [
                'title' => 'Список пользователей в хранилище',
                'users' => $users
            ]);
    }
}
```

В случае, если на уровне модели произошла ошибка, мы вызовем шаблон user-empty, тогда как для вывода на экран списка пользователей мы будем использовать шаблон user-index.

Шаблон user-empty будет очень простым:

```
<p>{{ message }}</p>
Подробнее остановимся на user-index.
<p>Список пользователей в хранилище</p>

<ul id="navigation">
    {% for user in users %}
        <li>{{ user.getUserName() }}. День рождения: {{ user.getUserBirthday() |
date('d.m.Y') }}</li>
    {% endfor %}
</ul>
```

Здесь мы применяем новые механики.

Во-первых, TWIG позволяет использовать циклы. А они нам нужны, так как users – это массив. Циклы и другие конструкции, отличные от переменных, обрамляются фигурными скобками и символом процента.

Внутри реализуется простой цикл `for`, который очень похож на `foreach` в PHP. Внутри каждой итерации мы вызываем у каждого элемента массива методы получения данных. При этом для дня рождения мы применяем также фильтр, который превращает `timestamp` в читаемую человеком дату:

```
user.getUserBirthday() | date('d.m.Y')
```

Если мы все сделали правильно, то по адресу <http://mysite.local/user/index> мы сможем увидеть нужные нам данные

Заключение

В рамках лекции мы погрузились в практическое применение аспектов ООП, с которыми познакомились на прошлой лекции. Мы освоили автозагрузку классов, шаблонизацию, подход MVC к разделению приложения на слои. И получили в итоге рабочий каркас приложения, с которым будем работать в рамках следующих лекций.

Домашнее задание

1. Добавьте к шаблону подключение файлов стилей так, чтобы в дальнейшем можно было дорабатывать внешний вид системы
2. Сформируйте еще три подключаемых к скелету блока – шапку сайта (она всегда будет одинаковой по стилю и располагаться в самой верхней части), подвал сайта (также одинаковый, но в нижней части) и sidebar (боковая колонка, которую можно наполнять новыми элементами).
3. Средствами TWIG выводите на экран текущее время.
4. Создайте обработку страницы ошибки. Например, если контроллер не найден, то нужно вызывать специальный метод рендеринга, который сформирует специальную [страницу ошибок](#).
5. Для страницы ошибок формируйте HTTP-ответ 404. Это можно сделать при помощи функции [header](#).
6. Реализуйте функционал сохранения пользователя в хранилище. Сохранение будет происходить при помощи GET-запроса.

```
/user/save/?name=Иван&birthday=05-05-1991
```

Что можно почитать еще?

1. <https://phptherrightway.com/>
2. <https://php.net>
3. PHP 8 | Котеров Дмитрий Владимирович
4. Объектно-ориентированное мышление | Мэтт Вайсфельд
5. <https://stackoverflow.com/questions/34034730/how-to-enable-color-for-php-cli> - как покрасить вывод консоли
6. <https://phptoday.ru/post/gotovim-lokalnuyu-sredu-docker-dlya-razrabotki-na-php> - варианты образа Composer
7. <https://anton-pribora.ru/articles/php/mvc/>