

Автореферат

Наименование Серверные технологии разработки

Автор А.В.Михалькевич

Специальность Web-дизайн и компьютерная графика,

Анотация

Цель учебной дисциплины - изучение технологий, необходимых для разработки и технической поддержке серверной части web-приложения. В процессе изучения дисциплины на практике рассматриваются использование сервера в типичных задач программирования, таких как запуск и остановка серверов Apache, Nginx, изучение серверного языка программирования PHP и взаимодействия с базой MySQL, использование Laravel в разработке бэкендов, разработка серверной маршрутизации web-приложения, токен-авторизация, ресурсы бэкенда, api-запросы, CRUD и т.д, решение которых складывается в полноценный универсальный бэкенд, клиентами которого могут выступать различные фронтенд-приложения.

Anotation in English

-

Ключевые слова Docker, сервер, Apache, HMVC, MVC, контроллер, модель, маршрутизация, Серверные технологии разработки, СТР,

Количество символов 99047

Содержание

[Введение](#)

[1 Обзор технологий](#)

[1.1 Серверная и клиентская части сайта](#)

[1.2 Фронтенд и бэкенд](#)

[1.3 Docker и Docker-compose](#)

[2 Разработка web-приложения с серверной маршрутизацией](#)

[2.1 Фрэймворк Laravel](#)

[2.2 Маршрутизация](#)

[2.3 Middleware](#)

[2.3 .1 Создание middleware AlwaysAcceptJson](#)

[2.3 .2 Подключение middleware](#)

[2.4 Контроллеры](#)

[2.5 Модели](#)

[2.6 Представления](#)

[3 Разработка backend API](#)

[3.1 Ресурсы в Laravel](#)

[3.2 Авторизация API](#)

[3.2 .1 Маршруты авторизации](#)

[3.2 .2 Контроллер авторизации](#)

[3.2 .3 Правила валидации запросов авторизации](#)

[3.3 Загрузка изображений и файлов](#)

[4 Контроллеры и ресурсные контроллеры](#)

[5 Командная разработка](#)

5.1 [GIT](#)

5.2 [Удаленный репозиторий github.com](#)

[Заключение](#)

[Список использованных источников](#)

[Приложения](#)

Введение

1 Обзор технологий

FrontEnd, BackEnd и смежные технологии

2 направления: frontend и backend. И 3 группы технологий (для backend, для frontend и общий).

Инструментарий общий

1. IDE: PHPStorm, или VSCode, NetBeans (+PHPDoc), Adobe Brackets, Sublime Text 2, Notepad++, WebMatrix...
2. Docker
3. OpenServer
4. Firefox, Firebug
5. Системы контроля версий: GIT, Mercurial, SVN, Subversion. Сайты github и bitbucket
6. Postman

Далее рассмотрим две группы компетенций, которыми необходимо обладать для понимания этой дисциплины:

FrontEnd:

1. HTML+CSS. Селекторы. Адаптивная верстка. Гибкая блочная верстка. Резиновая и фиксированная верстка, традиционная блочная и табличная.
2. HTML5
3. JSON
4. архитектурный шаблон MVVM
5. JavaScript
6. Node.js
7. Системы сборки FrontEnd-a. Gulp, Grunt, webpack
8. Bootstrap 3, API Bootstrap
9. Основы SEO.
10. Schema.org, генератор schema.org микроформат данных, микроданные

BackEnd

1) Языки программирования:

PHP +

Node.js +

Java +

Python -

Perl -

Ruby -

Asp.net -

2) Сервера (это ПО, получающая запросы от клиента, либо компьютеры, где это ПО находится): apache, nginx

3) СУБД (это ПО предназначенное для работы с базами данных):

MySQL (PHPMyAdmin)

NoSQL, SQL-lite

Oracle

MS-SQL

6) фреймворки Laravel и др.

7) CMS - WordPress и др.

8) Менеджер зависимостей: Composer.

9) Умение работать с запрос-ответом. Request - response. Типы запросов и варианты ответов.

Вопросы: Серверные языки программирования. Сравнение Объектно-ориентированное мышление

1.1 Серверная и клиентская части сайта

Веб браузеры взаимодействуют с [веб-серверами](#) при помощи гипертекстового транспортного протокола ([HTTP](#)). Когда вы нажимаете на ссылку на веб-странице, заполняете форму или запускаете поиск, **HTTP запрос** отправляется из вашего браузера на целевой сервер.

Запрос включает в себя URL, определяющий затронутый ресурс, метод, определяющий требуемое действие (например, получить, удалить или опубликовать ресурс) и может включать дополнительную информацию, закодированную в параметрах URL (пары поле-значение, опрарвленные как [строка запроса](#)), как POST запрос (данные, отправленные методом [HTTP POST](#)), или в [куки-файлах](#).

Веб серверы ожидают сообщений с клиентскими запросами, обрабатывают их по прибытию и отвечают веб-браузеру при помощи ответного HTTP сообщения. Ответ содержит строку состояния, показывающую, был ли запрос успешным, или нет (например, "HTTP/1.1 200 OK" в случае успеха.

Тело успешного ответа на запрос может содержать запрашиваемые данные (например, новую HTML страницу, или изображение, и т.п), который может отображаться через веб-браузер.

Теперь обратим внимание на код, задействованный в серверной части и клиентской части. В каждом случае код существенно различается:

Они имеют различные цели и назначение.

Как правило, они не используют одни и те же языки программирования (исключение составляет JavaScript, который можно использовать на стороне сервера и клиента).

Они выполняются в разных средах операционной системы.

Код, который выполняется в браузере, известный как **код клиентской части**, прежде всего связан с улучшением внешнего вида и поведения отображаемой веб-страницы. Это включает в себя выбор и стилизацию компонентов пользовательского интерфейса, создание макетов,

навигацию, проверку форм и т. д. Напротив, программирование веб-сайта на стороне сервера в основном включает выбор содержимого, которое возвращается браузеру в ответ на запросы. Код на стороне сервера обрабатывает такие задачи, как проверка отправленных данных и запросов, использование баз данных для хранения и извлечения данных и отправка правильных данных клиенту по мере необходимости.

Код клиентской части написан с использованием [HTML](#), [CSS](#) и [JavaScript](#) - он запускается в веб-браузере и практически не имеет доступа к базовой операционной системе (включая ограниченный доступ к файловой системе).

Веб-разработчики не могут контролировать, какой браузер может использовать каждый пользователь для просмотра веб-сайта - браузеры обеспечивают противоречивые уровни совместимости с функциями кода на стороне клиента, и одной из задач программирования на стороне клиента является изящная обработка различий в поддержке браузера.

Код серверной части может быть написан на любом количестве языков программирования - примеры популярных языков серверной части включают в себя PHP, Python, Ruby, C# и NodeJS(JavaScript). Код серверной части имеет полный доступ к операционной системе сервера, и разработчик может выбрать какой язык программирования (и какую версию) он хотел бы использовать.

Разработчики обычно пишут свой код, используя **веб-фреймворки**. Веб-фреймворки - это наборы функций, объектов, правил и других конструкций кода, предназначенных для решения общих проблем, ускорения разработки и упрощения различных типов задач, стоящих в конкретной области.

И снова, поскольку и клиентская и серверная части используют фреймворки, области очень разные и, следовательно, фреймворки тоже разные. Фреймворки клиентской части упрощают верстку и представление данных, тогда как фреймворки серверной части обеспечивают много «обычного» функционала веб-сервера, который вы возможно в противном случае должны были осуществлять самостоятельно (например, поддержка сессий, поддержка пользователей и аутентификация, простой доступ к базе данных, шаблонам библиотек, и т.д.).

Вопросы: Серверная часть сайта Клиентская часть сайта Сохранение данных на стороне клиента Сохранение данных на стороне сервера Типы HTTP-запросов Заголовки HTTP Строка запроса в протоколе HTTP Язык программирования PHP Язык программирования Node.js Глобальные объекты Node.js Суперглобальные переменные PHP Сервера для PHP Сервер Apache Сервер Nginx Особенности написания кода в Node.js Менеджер зависимостей Composer Менеджер зависимостей Npm

1.2 Фронтенд и бэкенд

С появлением динамики на стороне клиента (браузера) выделились такие понятия, как фронтенд и бэкенд. Бэкенд - программирование на стороне сервера. Фронтенд - программирование на стороне клиента. Постепенно обозначились основные задачи фронтенда:

- шаблонизация, или создание результирующего HTML;
- SEO;
- динамика и запросы на сервер.

Задачи бэкенда:

- серверная маршрутизация;
- хранение данных и формирование ответов на запросы фронтенда;
- авторизация;
- сложная логика, требующая вычислений;
- рассылка;
- авторизация пользователей;
- запуск скриптов по времени
- и многие другие задачи, вплоть до считывания данных датчиков и запусков сервлетов.

Как в бэкенд, так и в фронтенд разработке широкое распространение получил паттерн MVC (Model-View-Controller), разделяющий все компоненты на предназначенные для получения, хранения и отображения данных. Появилось множество фреймворков и библиотек, использующие в своих реализациях частично или полностью этот паттерн.

Вопросы: Инструментарий web-разработчика

1.3 Docker и Docker-compose

Snap - это пакет приложения для Linux систем, который легко устанавливать без дополнительных зависимостей.

Сперва убеждаемся в том, что snap установлен, либо устанавливаем его с помощью apt.

```
sudo apt update
sudo apt install snapd
```

После чего устанавливаем Docker

```
sudo snap install docker
```

Далее необходимо установить Docker-compose

[Docker Compose](#) - это инструмент, разработанный сообществом, который позволяет вам запускать мультиконтейнерные приложения, основанные на определениях из файла YAML. Определения сервисов позволяют собирать гибкие пользовательские среды с большим количеством контейнеров, которые могут совместно использовать сети и тома данных.

Установить Docker Compose можно из официального репозитория Ubuntu, однако тогда вы получите не самую свежую версию, потому лучше установить программу из [GitHub-репозитория Docker](#).

Найдите ссылку на свежий релиз на этой [странице](#).

После установки Docker и Docker-compose, можете перейти по ссылке и установить Nginx и PHP, MySQL, PHPMyAdmin

[Nginx и PHP](#)

[MySQL и PHPMyAdmin](#)

Вопросы: Использование bootstrap для разработки макетов

2 Разработка web-приложения с серверной маршрутизацией

Используем PHP + Laravel

2.1 Фреймворк Laravel

Фреймворк Laravel позволяет быстро, а, главное, грамотно создать web-приложение любой сложности (от сайта-визитки до порталов, чатов, магазинов...).

Изучать новый незнакомый фреймворк - нелегкое занятие, но учить новое всегда интересно. Чтобы это занятие оказалось не только интересным, но и эффективным, желательно придерживаться следующих рекомендаций по изучению нового фреймворка.

Изучаем вспомогательные технологии. Если это php-фреймворк, предварительно нужно изучить PHP и ООП. Фреймворк использует HMVC, поэтому необходимо понимать основные принципы использования данного паттерна. Устанавливается через Composer (уметь работать с этим менеджером зависимостей).

В процессе изучения разрабатываем проект, тогда обучение станет более эффективным, не только интересным, но, возможно, и прибыльным занятием.

Придерживаемся определённой архитектуре в разработке. Как правило, для бэкенд - это HMVC, для фронтенд - MV-VM. В HMVC - основную логику приложения держим в контроллерах, а все остальные классы можно воспринимать как вспомогательные, предназначение которых облегчить контроллер. В MV-VM нет контроллеров, приложение состоит из компонентов (тэгов), каждый из которых и содержит всю необходимую логику, которая не должна выходить за рамки компонента.

Начинаем разработку с главной страницы, потом обозначем маршруты, модели, макеты страниц.

После изучения этих документов вы будете иметь представление о том, как во фреймворке происходит обработка цикла запроса, а также создать простой сайт-визитку.

Затем нужно изучить про настройку соединения и построение запросов к базе данных, а также про встроенный Eloquent ORM, облегчающий работу с БД.

Затем, когда вы получите знания основ и приобретете некоторый опыт использования фреймворка, можно приступать к углубленному изучению: использование и разработка модулей, библиотек, хелперов, работа с ресурсами Laravel, разработка realtime-приложений и многое другое, возможности Laravel безграничны.

Важно, чтобы изучение теории сопровождалось с практикой.

Философия Laravel

Laravel - фреймворк для построения веб-приложений с выразительным и элегантным синтаксисом. Процесс разработки только тогда наиболее продуктивен, когда работа с фреймворком приносит радость и удовольствие. *Счастливые разработчики пишут лучший код.*

Laravel - попытка сгладить все острые и неприятные моменты в работе php-разработчика. Он берет на себя аутентификацию, роутинг, работу с сессиями, кеширование, внедрение

зависимостей и многое другое, что встречается в большинстве приложений, оставив вам только фокус на вашей задаче.

Laravel стремится сделать процесс разработки приятным для разработчика без ущерба для функциональности приложений. Для этого мы попытались объединить все самое лучшее из того, что мы видели в других фреймворках, - RubyOnRails, ASP.NET и Sinatra, Kohana, Yii. Превосходный IoCcontainer, встроенные миграции и интегрированная поддержка юнит-тестов дают вам мощные инструменты для того, чтобы сделать именно тот функционал, который вам нужен.

Требования к установке

Официальный сайт laravel - <http://laravel.com>

У Laravel всего несколько требований к вашему серверу:

PHP >= 8.1

Mcrypt PHP Extension

OpenSSL PHP Extension

MbstringPHPExtension

В некоторых операционных системах может понадобиться ручная установка PHP JSON extension.

```
composer create-project laravel/laravel
```

Composer создаст папку laravel, куда установится проект laravel

Вопросы: Фрэймворк Laravel Серверные MVC фрэймворки

2.2 Маршрутизация

Запрос из адресной строки попадает в так называемый обработчик маршрутов, или маршрутизатор, или роутер (routes). Маршрутизатор определяет, какой контроллер необходимо вызывать.

Маршруты определяются в файле app/routes.php

Простейший get-маршрут.

```
Route::get('/', function () {  
    return 'Hello World';  
});
```

Для перехвата POST-данных можно воспользоваться методом Route::post

Простейший post-маршрут.

```
Route::post('foo/bar', function () {  
    return 'Hello World';  
});
```

Метод Route::any перехватывает и POSTи GET данные.

Маршрут любого http-запроса.

```
Route::any('foo', function () {  
    return 'Hello World';  
});
```

Для перехвата маршрутов только по протоколу HTTPS, вторым входящим параметром можно

передать не функцию, а массив, первым элементом которого является тип протокола, а вторым – функция.

Маршрут любого https-запроса.

```
Route::get('foo', array('https', function() {  
    return 'Must be over HTTPS';  
}));
```

Для генерации URL к какому-нибудь маршруту можно воспользоваться методом URL::to(). Это может пригодиться для генерации путей к ссылкам (значение атрибута href), картинкам (src), стилям (href), скриптам(src), обработчику форм (action) либо при переадресации.

Генерация URL к маршруту.

```
$url = URL::to('foo');
```

Для этих же целей можно использовать хелпер url

Использование хелпера url.

```
$url = url('foo');
```

Параметры маршрутов

Добавление к маршруту обязательного параметра id.

Добавление к маршруту обязательного параметра id.

```
Route::get('user/{id}', function ($id) {  
    return 'User '.$id;  
});
```

Добавление к маршруту необязательного параметра name.

Добавление к маршруту необязательного параметра name.

```
Route::get('user/{name?}', function ($name = null) {  
    return $name;  
});
```

Вместо \$name = null можно использовать любое значение по умолчанию.

Добавление к маршруту необязательного параметра name.

```
Route::get('user/{name?}', function ($name = 'Jhon') {  
    return $name;  
});
```

Использование регулярных выражений в маршрутах.

Маршруты с соответствием пути регулярному выражению.

```
Route::get('user/{name}', function ($name) {  
})->where('name', '[A-Za-z]+');
```



```
Route::get('user/{id}', function ($id) {  
})->where('id', '[0-9]+');
```

Вместо последовательного вызова метода where, можно передать массив ограничений.

Использование массива в регулярных выражениях.

```
Route::get('user/{id}/{name}', function ($name) {  
})->where(['name'=>'[A-Za-z]+', 'id'=>'[0-9]+']);
```

Если какие-то регулярные выражения нужно связать со всеми параметрами, то можно использовать метод pattern:

Использование шаблона регулярного выражения.

```
public function boot(Router $router) {    $router->pattern('id', '[0-9]+' );
parent::boot($router); }
```

Таким образом, будет осуществляться проверка всех параметров с именем id.

Именованные маршруты

Задать имя маршруту можно следующим способом:

Назначение имени текущего исполняемого маршрута.

```
Route::get('user/profile', array('as' => 'profile', function () {
//
}));
```

Также можно задать контроллер и его экшн, который будет выполняется по данному маршруту.

Назначение контроллера и экшна.

```
Route::get('user/profile', array('as' => 'profile',
'uses' => 'UserController@showProfile'));
```

Теперь можно использовать имя маршрута при генерации URL либо при перенаправлении.

Генерация URL.

```
$url = URL::route('profile');
$redirect = Redirect::route('profile');
```

Получить имя текущего выполняемого маршрута можно методом `currentRouteName()`:

Получить имя текущего исполняемого маршрута.

```
$name = Route::currentRouteName();
```

Вопросы: Маршрутизация в MVC

2.3 Middleware

HTTP Middleware (посредники) - это промежуточный класс между маршрутизатором и контроллером приложения, который еще является фильтром обработки HTTP-запроса. Так, например, в Laravel включены middlewares для проверки аутентификации пользователя. Если пользователь не залогинен, middleware перенаправляет его на страницу логина. Если же залогинен - middleware не вмешивается в прохождение запроса, передавая его дальше по цепочке middleware-посредников к собственно приложению.

Проверка авторизации - не единственная задача, которую способны выполнять middlewares. Это также добавление особых заголовков (например, CORS http-ответ вашего приложения) или логирование всех http-запросов.

В Laravel есть несколько дефолтных middleware, которые находятся в папке `app/Http/Middleware`. Это middlewares для реализации режима обслуживания сайта ("сайт временно не работает, зайдите позже"), проверки авторизации, CSRF-защиты и т.п.

Вопросы: Архитектурный шаблон проектирования HMVC Простой шаблон проектирования
Функциональный шаблон проектирования Архитектурные шаблоны проектирования
Архитектурные шаблоны проектирования Миграции базы данных Промежуточное
программное обеспечение middleware

2.3.1 Создание middleware AlwaysAcceptJson

Итак, в нашем бэкенд-приложении все ответы, не зависимо от заголовков фронтенда, должны быть в формате JSON. Давайте создадим middleware, который ответ ошибок будет выводить в формате json, а если клиент забыл указать в заголовке Асепт 'application/json', то middleware изменит заголовок.

Создать middleware можно с помощью artisan

```
php artisan make:middleware AlwaysAcceptJson
```

В созданный файл добавим следующий код

```
$request->headers->set('Accept', 'application/json');
```

2.3.2 Подключение middleware

Существует несколько способов вызова middleware, в том числе и в конструкторе контроллера (что не желательно делать, поэтому такой вариант не рассматриваем).

1. Создать группу для этого middleware в маршрутизаторах web.php или api.php

```
Route::middleware(['api', 'acceptJson'])->group(function () {  
    Route::get('/', function () {  
        // Uses first & second middleware...  
    });  
  
    Route::get('/user/profile', function () {  
        // Uses first & second middleware...  
    });  
});
```

и в файле kernel.php определить значение

```
'acceptJson' => \App\Http\Middleware\AlwaysAcceptJson::class,
```

2. В RouteServiceProvider:

```
Route::middleware(['api', AlwaysAcceptJson::class])
```

3. Непосредственно в файле kernel.php в группе api

2.4 Контроллеры

Контроллеры хранятся в папке app/controllers/. Этот путь, в свою очередь определен в файле composer.json в настройке classmap.

Все контроллеры должны наследовать класс BaseController. Этот класс также может храниться в папке app/controllers, и в него можно поместить общую логику для других контроллеров. BaseController расширяет базовый класс Controller.

Создаем контроллер

Создание контроллера с помощью artisan-команды.

```
php artisan make:controller StaticController
```

Есть несколько способов определения маршрута для контроллера.

В файле app/routes.php.

Определение маршрута для контроллера с помощью метода get.

```
Route::get('static', 'StaticController@index');
```

Вот так будет выглядеть сам контроллер:

Простейший контроллер StaticController.

```
namespace App\Http\Controllers;
class StaticController extends BaseController {
    public function getIndex()
    {
        echo 'Ok';
    }
}
```

Обратите внимание на namespace (пространство имен) в начале файла.

Теперь, если в адресной строке браузера набираем cabinet, получаем ответ сервера Ok

Создание RESTfull-контроллера через artisan

Сперва в консоли перейдем в папку с файлом artisan. Находясь в этой папке, выполним следующую консольную команду.

Можно также создавать RESTfull-контроллеры:

Создание RESTfull-контроллеров.

```
php artisan make:controller NewsController --resource
```

Получим такой контроллер:

RESTfull-контроллер.

```

class NewsController extends \BaseController {
    public function index()
    {
        //
    }
    public function create()
    {
        //
    }
    public function store()
    {
        //
    }
    public function show($id)
    {
        //
    }
    public function edit($id)
    {
        //
    }
    public function update($id)
    {
        //
    }
    public function destroy($id)
    {
        //
    }
}

```

Также необходимо зарегистрировать роут к этому контроллеру.

Регистрация роута в файле app/routes.php.

```
Route::resource('news', 'NewsController');
```

Такая регистрация дает множество маршрутов для обработки RESTfull экшнов контроллера News. Сам сгенерированный контроллер уже имеет методы-заглушки для каждого из этих маршрутов с комментариями, которые напоминают вам о том, какие типы запросов они обрабатывают.

Тип	Путь	Действие	Имя маршрута
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{id}	show	resource.show
GET	/resource/{id}/edit	edit	resource.edit
PUT/PATCH	/resource/{id}	update	resource.update
DELETE	/resource/{id}	destroy	resource.destroy

Где resource - имя контроллера.

Чтобы создать только часть из возможных экшнов, можно воспользоваться настройками `-only` или `--except` :

Ключевые слова `only` и `except` можно указать при регистрации маршрута:

Создание маршрутов для группы экшнов.

```
Route::resource('news', 'PhotoController',
    array('only' => array('index', 'show')));
// либо:
Route::resource('news', 'PhotoController',
    array('except' => array('create', 'store', 'update', 'delete')));
```

Вложенная папка для контроллеров

Нужно ли создавать вложенную папку для контроллеров - каждый программист решает для себя сам. Здесь мы рассмотрим как это сделать. Сперва просто создаем папки, где будут находиться контроллеры.

```
app/controllers
app/controllers/auth
app/controllers/adminka
...
```

В самих контроллерах ничего не меняем. Из корневой папки проекта открываем консоль и выполняем следующую команду composer:

Перестройка автозагрузки контроллеров.

```
composer dump-autoload
```

Вопросы: Контроллеры в MVC

2.5 Модели

Модель - это класс, ассоциированный с именем таблицы из базы данных.

Причем, по-умолчанию, имя класса модели равняется имени таблицы из базы из которой убирается окончание "s". Таким образом, модель Product ассоциирована с таблицей products.

У artisan имеется специальная команда для создания моделей.

```
php artisan make:model Flight
```

Если модель связывана с таблицей, а миграции создают таблицы, то логично было бы предположить, что имеется специальная команда связывающая модель с миграцией. Действительно, у artisan имеется специальная команда, создающая сразу два файла: файл модели и файл миграции.

```
php artisan make:model Flight --migration
//или
php artisan make:model Flight -m
```

Не важно, какой командой мы воспользовались, в папке models проекта появится файл Flight.php со следующим содержимым:

```
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{
    //
}
```

В отличие от контроллера, который без методов бесполезен, такой пустой класс модели, мы уже можем использовать. Через данный класс модели, мы можем использовать методы конструктора запросов.

В Laravel имеется встроенная модель User, которая нас связывает с таблице users (используемая в авторизации).

На примере этой модели, рассмотрим основные CRUD-операции (Create, Read, Update и Delete).

Извлечение данных

Как только модель определена, у вас всё готово для того, чтобы можно было выбирать и создавать записи. Обратите внимание, что вам нужно создать в этой таблице поля updated_at и created_at. Если вы не хотите, чтобы они были автоматически используемы, установите свойство \$timestamps класса модели в false.

Получение всех записей модели

```
$users = User::all();
```

Получение записи по первичному ключу:

```
$user = User::find(1);  
var_dump($user->name);
```

Метод find может принимать входящим параметром массив:

```
$user = User::find([1,2,4]);
```

Для извлечения данных по другим полям можно воспользоваться методом where.

```
$user = User::where('email',$email)->get();
```

При этом конечный метод get возвращает массив объектов, для вывода которых в последствии необходимо использовать foreach.

Метод where() может использоваться с тремя входящими параметрами, тогда вторым входящим параметром передается ключевой символ =, !=, >, <, <> или ключевое слово LIKE.

```
$user = User::where('name','!=',$name)->get()
```

Метод where можно использовать в запросе несколько раз подряд, тогда последующие where работают как AND WHERE.

Множественный вызов метода where:

```
$user = Товар::where('cat_id',1)->where('showhide','show')->get()
```

Существует еще один полезный метод, связанный с подзапросом where - это whereIn. WhereIn вторым (или третьим) входящим параметром принимает массив значений, по которым необходимо формировать запрос.

```
$models = Model::whereIn('id', [1, 2, 3])->get();
```

Все методы, доступные в конструкторе запросов, также доступны в запросах с использованием моделей.

Иногда вам нужно возбудить исключение, если определённое значение не найдено, что позволит вам его отловить в обработчике `App::error()` и вывести страницу 404 («Не найдено»).

```
$model = User::findOrFail(1);  
$model = User::where('votes', '>', 100)->findOrFail();
```

Вставка

Данные через модель могут быть вставлены двумя способами:

Множественная вставка

Одиночная вставка

Пример одиночной вставки:

```
$user = new User; // создаем объект  
$user->name = 'Джон'; // данные  
$user->save(); // сохранение
```

Пример множественной вставки:

```
User::create(['name'=>'Джон']);
```

Считается, что способ множественной вставки менее безопасен одиночной. Поэтому для реализации множественной вставки, нужно специальное разрешение модели. В модели нужно объявить свойство `$fillable`, содержащее массив элементов полей, разрешенных для вставки.

```
$fillable = ['name'];
```

Обновление

Обновление данных по id:

```
$user = User::find(1);  
$user->email = 'alex@ya.ru';  
$user->save();
```

Удаление

Удаление данных по id

```
$user = User::find(1);  
$user->delete();
```

Заготовки запросов

Заготовки позволяют повторно использовать логику запросов в моделях. Для создания заготовки просто начните имя метода со `scope`:

Создание заготовок запроса

```
class User extends Model {

    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    public function scopeWomen($query)
    {
        return $query->whereGender('W');
    }

}
```

Использование заготовок запросов:

```
$users = User::popular()->women()->orderBy('created_at')->get();
```

Можно также использовать заготовки запросов с входящими параметрами:

```
class User extends Model {

    public function scopeOfType($query, $type)
    {
        return $query->whereType($type);
    }

}
```

Вызов заготовок запросов с входящими параметрами:

```
$users = User::ofType('member')->get();
```

Вопросы: Модели в MVC

2 .6 Представления

По умолчанию, laravel работает с шаблонизатором blade. Шаблоны создаются в папке app/views и имеют расширение blade.php. Шаблоны подключаются в экшне через хелпер view(), входящим параметром в который передается имя шаблона без расширения blade.php.

Сперва создадим в папке view папку layouts для хранения базовых шаблонов. В папке layouts создадим файл defaults.blade.php

Базовый шаблон defaults.blade.php.

```

@include('layouts.header')
<div class="container">

<div class="masthead">
<h3 class="text-muted">Project name</h3>
<ul class="nav nav-justified">
<li class="active"><a href="#">Home</a></li>
--блок ссылок --
</ul>
</div>
@yield('content')
<!-- Site footer -->
<div class="footer">
<p>&copy; Company 2014</p>
</div>
</div><!-- /container -->
@include('layouts.footer')

```

В файлах layouts/header.blade.php и layouts/footer.blade.php находится обычный html-код для шапки и футера сайта. Эти 3 файла - это неизменная часть шаблона.

@yield('content') - вывод переменной content. Саму переменную определим в меняющейся части шаблона.

Меняющуюся часть шаблона вынесем в отдельный файл index.blade.php.

Меняющаяся часть шаблона.

```

@extends('layouts.default')
@section('content')
<h1>Добро пожаловать на сайт</h1>
<div>Текст на страницу</div>
@stop

```

Если необходимо создать часть кода в шаблоне, которая в последствии будет либо заменена, либо добавлена, то можно воспользоваться директивой @show

Объявление переменной с директивой @show в базовом шаблоне.

```

@section('styles')
<link href="{ { asset('/css/bootstrap.css') } }">
@show

```

Далее к данному стилю можно добавить другие стили в файлах подшаблона. В подшаблоне обращаемся к директиве @parent переменной styles.

Объявление переменной с директивой @show в базовом шаблоне.

```

@extends('public')
@section('styles')
    @parent
<link href="{ { asset('/js/fancybox/jquery.fancybox.css') } }">
@stop

```

Обратите внимание на @extends в начале кода. В шаблонизаторе blade из контроллера мы обращаемся к подшаблону, а подшаблон с помощью директивы @extends сам себя вставляет в базовый шаблон public.

Подключение подшаблона index.blade.php осуществляется в экшне контроллера.

Подключение подшаблона index.blade.php.

```

return view('index');

```

Передача массива в шаблон:

Передача массива в шаблон.

```
$posts = array(1=>'One', 2=>'Two');  
return view('index', $posts);
```

Имеется также ещё один способ передачи переменных в шаблон, через метод with():

Передача переменной в шаблон с помощью метода with.

```
$posts = array(1=>'One', 2=>'Two');  
  
return view('index')->with('posts', $posts)
```

Если вы не уверены в существовании передаваемой переменной, то нужно использовать with(), тогда не будет выводиться ошибка.

И ещё один способ передачи переменных:

Магический метод передачи данных в шаблон.

```
$view = view('greeting')->withName('Victoria');
```

Для вывода переменных в шаблоне можно воспользоваться директивой {{}}

Вывод переменных шаблона.

```
{{ $name }} // простой вывод переменной  
  
{{ isset($name)?$name:'Default' }} // вывод либо переменной либо  
значения по умолчанию.  
  
{{ $name or 'Default' }} // еще один способ вывода значения по  
умолчанию
```

Для предотвращения XSS-атак директива {{}} экранирует html-тэги. Если всё же необходимо вывести html, то можно воспользоваться другой директивой:

Директива {!! !!}.

```
Hello, {!! $name !!}.
```

Вывод всех элементов массива на экран

Вывод элементов массива на экран.

```
@if($posts->count())  
@foreach($posts as $post)  
    <div>{{ $post }}</div>  
@endforeach  
@endif
```

Проверка шаблона на существование

Проверить существует ли шаблон по заданному пути, можно с помощью хелпера view() и метода exists(), входящим параметром в который передается путь к шаблону.

Проверка шаблона на существование.

```
if (view()->exists('templates.base')) {  
    // шаблон существует  
}else{  
    // шаблон не существует  
}
```

Передача данных во все шаблоны

Для передачи переменных во все шаблоны, можно воспользоваться либо хелпером view():

Передача данных во все шаблоны с помощью хелперов.

```
view()->share('data', [1, 2, 3]);
```

либо фасадом:

Передача данных во все шаблоны с помощью фасадов.

```
View::share('data', [1, 2, 3]);
```

Этот код можно положить в метод boot() сервис-провайдера общего сервис-провайдера приложения AppServiceProvider или своего собственного.

View composer

Композеры (view composers) - функции-замыкания или методы класса, которые вызываются, когда шаблон рендерится в строку. Если у вас есть данные, которые вы хотите привязать к шаблону при каждом его рендеринге, то композеры помогут вам выделить такую логику в отдельное место.

Регистрировать композеры можно внутри сервис-провайдера. Мы будем использовать фасад View для того, чтобы получить доступ к имплементации контракта Illuminate\Contracts\View\Factory:

Определение композера.

```

<?php namespace App\Providers;
use View;
use Illuminate\Support\ServiceProvider;
use App\Http\ViewComposers\SiteComposer;
class ComposerServiceProvider extends ServiceProvider {
    public function boot()
    {
        View::composer('*', 'App\Http\ViewComposers\SiteComposer');
    }
    public function register()
    {
        //
    }
}

```

Файл композера необходимо зарегистрировать в config/app.php

Регистрация файла композера.

```
'App\Providers\ComposerServiceProvider',
```

Определение класса SiteComposer:

Класс SiteComposer.

```

<?php
namespace App\Http\ViewComposers;
use Illuminate\Contracts\View\View;
class SiteComposer
{
    public function compose(View $view)
    {
        $view->with('menu_items', 'TEST');
    }
}

```

Метод `compose` должен получать в качестве аргумента инстанс `Illuminate\Contracts\View\View`. Для передачи переменных в шаблон используйте метод `with()`.

Назначение композера для нескольких шаблонов

Вместо имени шаблона можно использовать массив имен.

Назначение композера для нескольких шаблонов.

```
View::composer(['profile', 'dashboard'],  
'App\Http\ViewComposers\MyViewComposer');
```

Композер для всех шаблонов

А вот так можно назначить композер для всех шаблонов:

Назначение композера для всех шаблонов.

```
View::composer('*', function()  
{  
    //  
});
```

Регистрация нескольких шаблонов

Можно использовать метод `composers`, чтобы зарегистрировать несколько композеров одновременно:

Регистрация нескольких шаблонов одновременно.

```
View::composers([  
    'App\Http\ViewComposers\AdminComposer' =>  
    ['admin.index', 'admin.profile'],  
    'App\Http\ViewComposers\UserComposer' => 'user',  
    'App\Http\ViewComposers\ProductComposer' => 'product'  
]);
```

ViewCreator

Создатели шаблонов работают почти так же, как композеры, но вызываются сразу после создания объекта шаблона, а не во время его рендеринга в строку. Для регистрации используйте метод `creator`:

Регистрация view creator.

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

Вопросы: Элементы представлений в MVC Шаблонизатор blade

3 Разработка backend API

Встроенный функционал Laravel позволяет разрабатывать API бэкенд, к которому могут подключаться любые фронтенды. При таком разделении на фронтенд и бэкенд, на стороне

бэкенда формируются ресурсы, на стороне фронтенда - компоненты. Таким образом, формирование ресурсов, которые должны быть доступны по определенным маршрутам - это основная задача возлагаемая на Laravel.

3 .1 Ресурсы в Laravel

При создании API вам может потребоваться уровень преобразования, который находится между вашими моделями Eloquent и ответами JSON, которые фактически возвращаются пользователям вашего приложения. Например, вы можете захотеть отображать определенные атрибуты для подмножества пользователей, а не других, или вы можете всегда включать определенные отношения в JSON-представление ваших моделей. Классы ресурсов Eloquent позволяют легко и выразительно преобразовывать ваши модели и коллекции моделей в JSON. Конечно, вы всегда можете преобразовать модели или коллекции Eloquent в JSON, используя их методы toJson; однако ресурсы Eloquent обеспечивают более детальный и надежный контроль над сериализацией JSON ваших моделей и их взаимосвязями, и делают возможным множественное использование одного ресурса.

Чтобы сгенерировать класс ресурсов, вы можете использовать Artisan-команду `make:resource`. По умолчанию ресурсы будут помещены в каталог `app/Http/Resources` приложения. Ресурсы расширяют класс `Illuminate\Http\Resources\Json\JsonResource`:

```
php artisan make:resource UserResource
```

Помимо создания ресурсов, преобразующих отдельные модели, вы можете создавать ресурсы, отвечающие за преобразование коллекций моделей. Это позволяет вашим ответам JSON включать ссылки и другую метаданную, имеющую отношение ко всей коллекции данного ресурса. Чтобы создать коллекцию ресурсов, вы должны использовать флаг `--collection` при создании ресурса. Или включение слова `Collection` в имя ресурса укажет Laravel, что он должен создать ресурс коллекции. Ресурсы коллекции расширяют класс `Illuminate\Http\Resources\Json\ResourceCollection`:

```
php artisan make:resource User --collection
```

```
php artisan make:resource UserCollection
```

Каждый класс ресурсов определяет метод `toArray`, который возвращает массив атрибутов, которые должны быть преобразованы в JSON, когда ресурс возвращается в качестве ответа от метода маршрута или контроллера.

```
namespace App\Http\Resources;
```

```
use Illuminate\Http\Resources\Json\JsonResource;
```

```
class UserResource extends JsonResource
{
    /**
```

```

    * Transform the resource into an array.
    *
    * @param \Illuminate\Http\Request $request
    * @return array
    */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}

```

Обратите внимание, что мы можем получить доступ к свойствам модели непосредственно из переменной `$this`. Это связано с тем, что класс ресурсов автоматически передает свойства и методы доступа к базовой модели для удобного доступа. Как только ресурс определен, он может быть возвращен из маршрута или контроллера. Причём его вызов может быть либо множественным, либо одиночным.

Одиночный вызов ресурса через конструктор:

```

Route::get('/user/{id}', function ($id) {
    return new UserResource(User::findOrFail($id));
});

```

Множественный вызов ресурса с помощью метода collection

```

Route::get('/users', function () {
    return UserResource::collection(User::all());
});

```

Связи в ресурсах

Если вы хотите включить связанные ресурсы в свой ответ, вы можете добавить их в массив, возвращаемый методом `toArray` вашего ресурса. В этом примере мы будем использовать метод коллекции ресурса `PostResource`, чтобы добавить сообщения блога пользователя в ответ ресурса:

```

use App\Http\Resources\PostResource;

/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)

```

```
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($this->posts),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

Вопросы: Шаблонизация

3.2 Авторизация API

[Laravel Sanctum](https://laravel.com/docs/9.x/sanctum) (<https://laravel.com/docs/9.x/sanctum>) предоставляет простую систему аутентификации для SPA (одностраничных приложений), мобильных приложений и API на основе токенов. Sanctum позволяет каждому пользователю вашего приложения генерировать несколько токенов API для своей учетной записи. Токены определяют доступы пользователей к определенным методам.

Модуль Jetstream включает в себя авторизацию с помощью Sanctum. Если вы уже устанавливали Jetstream, то Sanctum устанавливать не надо.

Для включения функции api токен, необходимо раскомментировать соответствующую запись в опции конфигурации features в файле конфигурации config/jetstream.php приложения:

```
'features' => [
    Features::profilePhotos(),
    Features::api(),
    Features::teams(),
],
```

Каждый запрос, сделанный к приложению Jetstream, даже к аутентифицированным маршрутам в файле routes/web.php, будет связан с объектом токена Sanctum. И мы можем определить, имеет ли связанный токен данное разрешение, используя метод tokenCan, предоставленный трейтом Laravel\Sanctum\HasApiTokens.

Это свойство HasApiTokens автоматически применяется к модели приложения App\Models\User во время установки Jetstream. Обычно метод tokenCan вызывается в контроллерах приложения, компонентах Livewire или политиках авторизации.

Например, таким образом, мы можем создать токен для пользователя:

```
$user = User::first();
$user->create_token('developer_access')->plainTextToken;
```

Свойство plainTextToken содержит созданный токен.

Использование токена в middleware:

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request)
{
    return $request->user();
});
```

Т.е. если мы сейчас переходя по ссылке /user не передадим в запросе значение токена (свойство plainTextToken), то будет ошибка доступа.

Удаление токенов пользователя:

```
$user->tokens()->delete();
```

Еще одна полезная возможность - это создание токена на определенные действия:

```
$user->createToken('token-name', ['category_list']);
```

Используем токен так:

```
if ($user->tokenCan('category_list')) {
    //
}
```

Вопросы: Синтаксис JavaScript Типы данных Программирование с помощью callback-функций
Область видимость переменных Взаимодействие классов в ООП

3 .2 .1 Маршруты авторизации

```
Route::middleware('auth:sanctum')->group(function (){
    Route::post('logout', [Controllers\AuthController::class, 'logout']);
    Route::get('profile', [Controllers\AuthController::class, 'profile']);
});
Route::post('register', [Controllers\AuthController::class, 'register']);
Route::post('login', [Controllers\AuthController::class, 'login']);
```

3 .2 .2 Контроллер авторизации

Сперва необходимо создать контроллер авторизации. Сделать это можно с помощью artisan-команды.

```
php artisan make:controller AuthController
```

Контроллер авторизации должен содержать следующие методы авторизации: register(), login(), logout(), profile().

Для реализации задач авторизации в Laravel имеется встроенный механизм - фасад Auth.

Данные пользователей можем хранить в таблице users.

Рассмотрим методы подробнее:

```
public function register(UserRequest $r)
{
```

```

    $r['password'] = Hash::make($r->password);
    $user = User::create($r->all());
    $token = $user->createToken('myapptoken')->plainTextToken;
    $answer = [
        'user' => $user,
        'token' => $token
    ];
    return response()->json($answer);
}

```

Метод login()

```

public function login(Request $request){
    abort_if(!$request->email, '401', 'email is empty');
    abort_if(!$request->password, '401', 'password is empty');
    $user = User::where('email', $request->email)->first();
    if(!$user || !Hash::check($request->password, $user->password)){
        return response()->json([
            'message' => 'bad credits'
        ]);
    }
    $token = $user->createToken('myapptoken')->plainTextToken;
    $answer = [
        'user' => $user,
        'token' => $token
    ];
    return response()->json($answer);
}

```

Методы logout() и profile() соответственно:

```

public function logout(){
    Auth::user()->tokens()->delete();
    return response()->json([
        'message' => 'user logout'
    ]);
}
public function profile(){
    return response()->json(Auth::user());
}

```

3.2.3 Правила валидации запросов авторизации

Request-классы - это файлы для хранения правил валидации запросов. Создаются такие файлы с помощью artisan:

```
php artisan make:request UserRequest
```

Содержимое файле UserRequest

```

public function authorize()
{

```

```

        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'name'=>'required|string',
            'email'=>'required|string|unique:users|email',
            'password'=>'required|confirmed'
        ];
    }
}

```

3.3 Загрузка изображений и файлов

Модуль intervention/image

Сперва произведем декомпозицию задачи по разработке модуля загрузки изображения. Разобьём эту задачу на 7 простых шагов. Вот они:

1. Обновить composer

```
composer self-update
```

2. Установить зависимость intervention image

```
composer require intervention/image
```

3. В файле config/app.php добавить провайдер зависимости (после подключения зависимостей ядра, но до провайдеров приложения). Лучше всего это делать сразу после комментария Package Service Providers...:

```
Intervention\Image\ImageServiceProvider::class
```

В этом же файле добавить ссылку на фасад Image:

```
'Image' => Intervention\Image\Facades\Image::class
```

С прочими особенностями установки можно ознакомиться по адресу официальной документации пакета - http://image.intervention.io/getting_started/installation

4. В директории /app создать еще одну папку libs, в которой будут храниться вспомогательные классы приложения.

5. В папке libs создать файл Imap.php

Рассмотрим содержимое файла:

```

namespace App\Libs;

use Image;
use Auth;

class Imag{
    public function url($path = null, $dirrectory = null, $name = null){
        if($path != null){
            if($dirrectory != null){
                $dir = public_path() . $dirrectory;
            }else{
                if(!Auth::guest()){
                    $dir = public_path() . '/uploads/'. Auth::user()->id . '/';
                }else{
                    $dir = public_path() . '/uploads/0/';
                }
                if(!file_exists($dir)){
                    mkdir($dir, 0777, true);
                }
            }
            if($name != null){
                $filename = $name;
            }else{
                $filename = date('y_m_d_h_i_s').'.jpg';
            }
            $img = Image::make($path);
            $img->resize(600, null, function ($constraint) {
                $constraint->aspectRatio();
            });
            $img->save($dir . $filename);
            $img->resize(300, null, function ($constraint) {
                $constraint->aspectRatio();
            });
            $pic_small = 's_'. $filename;
            $img->save($dir . $pic_small);
            $img->resize(100, null, function ($constraint) {
                $constraint->aspectRatio();
            });
            $pic_small2 = 'ss_'. $filename;
            $img->save($dir . $pic_small2);
            return $filename;
        }else{
            return false;
        }
    }
}

```

6. Далее необходимо подготовить форму с элементом `input type="file" name="picture1"`.

Для самого тега `form` необходимо добавить атрибут `enctype="multipart/form-data"`

7.Использование класса Imag в контроллере, обрабатывающем форму:

```
$pic = \App::make('\App\Libs\Imag')->url($_FILES['picture1']['tmp_name']);
```

Медиабиблиотека Spatie

Для связывания файлов (изображений, видео, аудио, текстовых и других форматов) с моделями Laravel можно воспользоваться модулем Spatie [Laravel-medialibrary](#)

Вопросы: Объекты jQuery

4 Контроллеры и ресурсные контроллеры

Один из простейших способов создания контроллера - это с помощью artisan:

```
php artisan make:controller BaseController
```

Результат выполненной команды:

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class BaseController extends Controller
{
    //
}
```

Реализация экшна в контроллере:

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class BaseController extends Controller
{
    public function index(){
        echo 'Ok';
    }
}
```

Созданные экшны необходимо привязывать в маршрутизаторе routes/web.php.

Привязка экшна контроллера к get-запросу:

```
Route::get('/', 'BaseController@index');
```

Ресурсные контроллеры

Привязка ресурсного контроллера:

```
Route::resource('tasks', 'TasksController');
```

Создание ресурсного контроллера:

```
php artisan make:controller TasksController --resource
```

Привязка массива ресурсных контроллеров:

```
Route::resources(['tasks' => 'TasksController']);
```

Ресурсные контроллеры содержат следующие методы запроса и обрабатывающие их экшны:

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

RESTFull API контроллеры

Создание контроллера для API

```
php artisan make:controller MySampleResourceController --api
```

Привязка контроллера-api:

```
Route::apiResource('tasks', 'TasksController');
```

Имеется возможность создать контроллер с подключенной моделью.

```
php artisan make:controller TestController --api --model=User
```

Или:

```
php artisan make:controller TestController --model=User
```

5 Командная разработка

Система контроля версий GIT.

Используем репозиторий github.com

5 .1 GIT

Основы git

<https://proglib.io/p/git-for-half-an-hour/>

Вопросы: Система контроля версий GIT

5 .2 Удаленный репозиторий github.com

Команды при создании нового репозитория в командной строке:

```
cd domains // перейти
```

создадим папку project.

В папке project – тестовый файл test.html

cd project // \domains\project

git init // команда создаёт в текущем каталоге новый подкаталог с именем .git
содержащий все необходимые файлы репозитория — основу Git-репозитория, на этом этапе
проект ещё не находится под версионным контролем

git status // статус, означает, что Git видит файл,
отсутствующий в предыдущем снимке состояния (коммите)

git add *

git commit -m "first message" // сохраняет снимок сцены в виде коммита

git remote add origin https://git // добавить новый удалённый Git-репозиторий

git push -u origin master // отправить свои наработки в главный репозиторий

Вопросы: Удалённый репозиторий github.com

Заключение

Список использованных источников

1. [url] **Laravel** <http://laravel.com>

Приложения

1. [picture] [5c3dbd22aa5ba_вопросы.txt](#)
2. [Вопросы] **Вопросы** [5c3dbd6a28ca7_Questions.docx](#)
3. [picture] [5e2059770a90c_bilets_2020.docx](#)