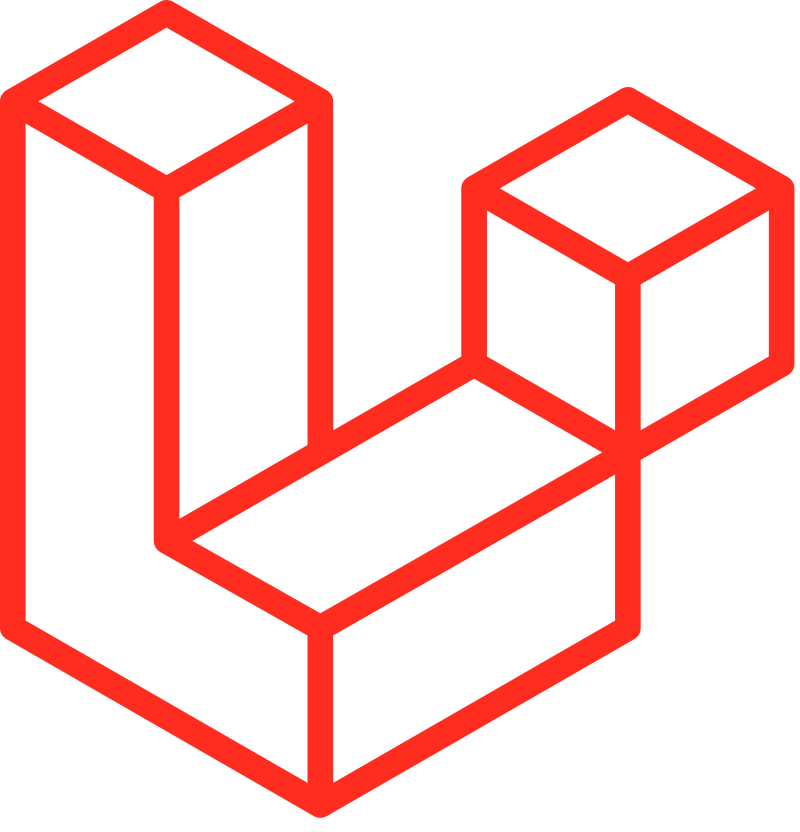
В новом обновление проект создавать нужно с указанной версией!  
composer create-project laravel/laravel:^10.0 example-app



Laravel — бесплатный веб-фреймворк с открытым кодом, предназначенный для разработки с использованием архитектурной модели MVC (модель-представление-контроллер). Laravel выпущен под лицензией MIT.

Исходный код проекта размещается на GitHub. В результате опроса sitepoint.com в декабре 2013 года о самых популярных PHP-фреймворках Laravel занял место самого многообещающего проекта на 2014 год.

В 2015 году в результате опроса sitepoint.com по использованию PHP-фреймворков среди программистов занял первое место в номинациях:

1. Фреймворк корпоративного уровня
2. Фреймворк для личных проектов
3. Кладезь антипаттернов

История

Laravel был создан Taylor Otwell как более функциональная альтернатива CodeIgniter, который не предусматривал различные дополнительные функции. Первый бета-релиз Laravel стал доступен 9 июня 2011 года, а Laravel 1 вышел в этом же месяце. Laravel 1 включает в себя встроенную поддержку для аутентификации, локализации, модели, представления, сессий, маршрутизации и других механизмов.

Laravel 2 был выпущен в сентябре 2011 года. Основные новые функции включают в себя поддержку контроллеров, которые сделали фреймворк полностью MVC-совместимым, встроенную поддержку для инверсии управления и шаблонизатор Blade.

Laravel 3 был выпущен в феврале 2012 года с набором новых функций, включая интерфейс командной строки (CLI) под именем «Artisan», встроенную поддержку нескольких систем управления базами данных, миграции баз данных в виде контроля версий, обработку событий. Выпуск Laravel 3 получил значительное увеличение числа пользователей, что повлияло на его популярность.

Laravel 4 был выпущен в мае 2013 года. Были произведены существенные изменения структуры Laravel, перенося ее макет в набор отдельных пакетов, распространяемых через Composer. Такая компоновка улучшила расширяемость Laravel 4, которая сочеталась с официальным регулярным расписанием релиза, охватывающим шесть месяцев между небольшими выпусками. Другие новые функции в выпуске Laravel 4 включают в себя выгрузка таблиц базы данных для первоначальной популяции, поддержку очередей сообщений, встроенную поддержку отправки различных типов электронной почты и поддержку «мягкого» удаления записей базы данных.

Laravel 5 был выпущен в феврале 2015 года в результате внутренних изменений, которые закончились перенумерацией релиза Laravel 4.3 в будущем. Новые возможности в выпуске Laravel 5 включают поддержку планирования периодически выполняемых задач через пакет Scheduler, слой абстракции Flysystem, который позволяет использовать удаленное хранилище так же, как и локальные файловые системы, улучшенную обработку активов пакета через Elixir и упрощенная аутентификация с внешней стороны через дополнительный пакет Socialite. Laravel 5 также представил новую внутреннюю структуру дерева каталогов для разработанных приложений.

Laravel 5.1, выпущенный в июне 2015 года. Является первым выпуском Laravel для долгосрочной поддержки (LTS) с запланированной возможностью исправления ошибок в течение трех лет и исправлений безопасности в течение трех лет.

Laravel 5.3, выпущенный 23 августа 2016 года. Новые функции в 5.3 сосредоточены на улучшении скорости разработки, добавив дополнительные усовершенствования для отдельных задач.

Laravel 5.4, выпущенный 24 января 2017 года. В этом выпуске есть много новых функций, таких как Laravel Dusk, Laravel Mix, Blade Components и Slots, Markdown Emails, автоматические фасады, улучшения маршрута.

Laravel 5.5, выпущенный 30 августа 2017 года. Возвращен пакет Whoops для отладки ошибок. Из нововведений, метод обработки запроса, новые директивы для шаблонизатора Blade, новая команда для миграции баз данных — migrate: fresh.

Laravel 7.2 вышел 19 марта 2020

Laravel 8.3 вышел 16 сентября 2020 года.

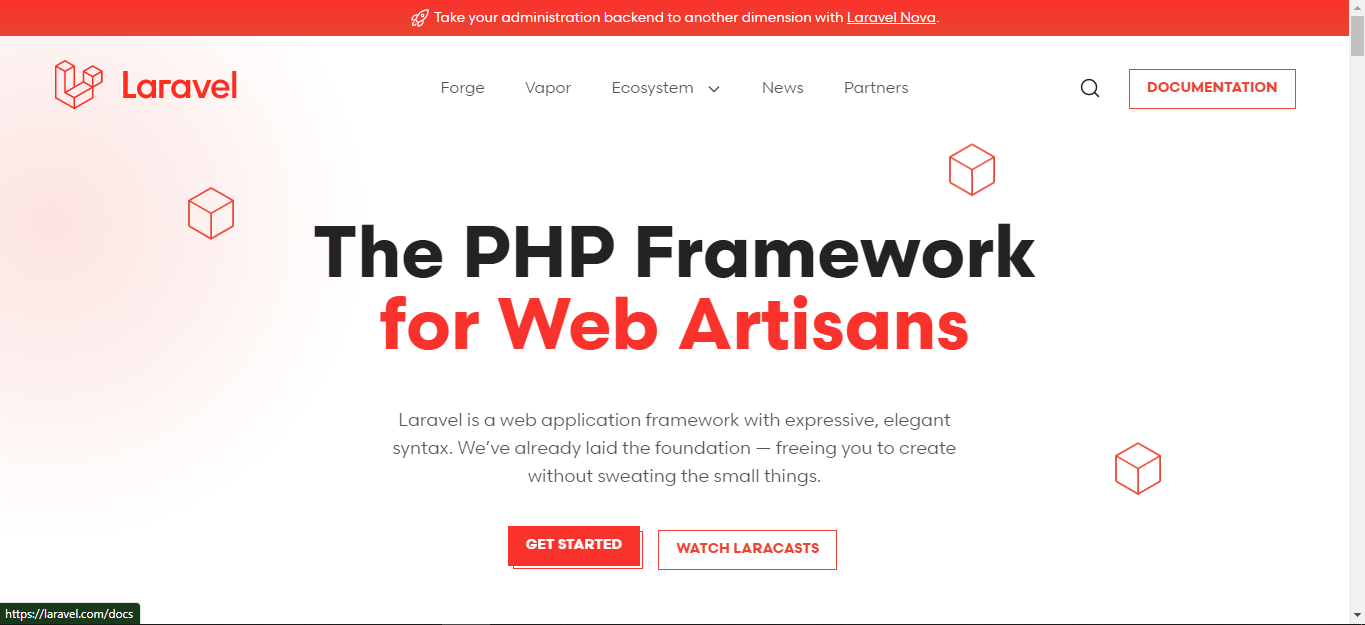
Laravel 9 вышел 8 февраля 2022 года.

Последние версии на текущий момент Laravel:

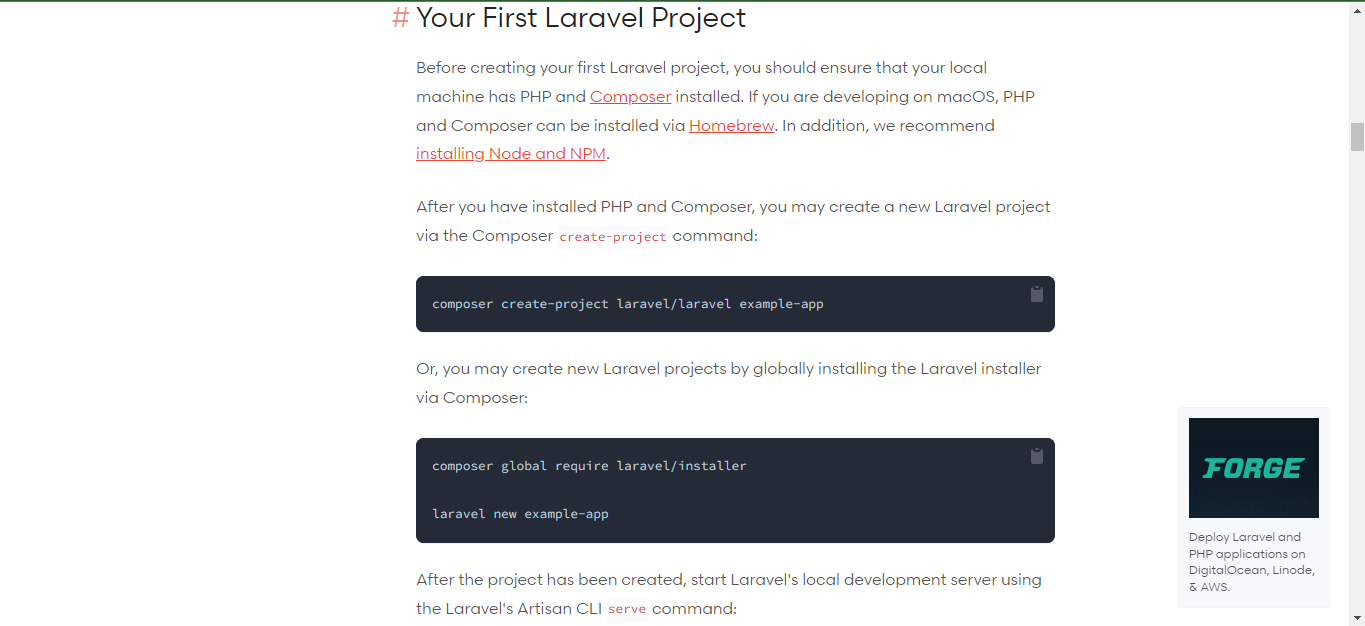
| **Version** | **PHP (\*)** | **Release** | **Bug Fixes Until** | **Security Fixes Until** |
| --- | --- | --- | --- | --- |
| 8 | 7.3 - 8.1 | September 8th, 2020 | July 26th, 2022 | January 24th, 2023 |
| 9 | 8.0 - 8.2 | February 8th, 2022 | August 8th, 2023 | February 6th, 2024 |
| 10 | 8.1 - 8.2 | February 14th, 2023 | August 6th, 2024 | February 4th, 2025 |
| 11 | 8.2 | Q1 2024 | August 5th, 2025 | February 3rd, 2026 |

Официальный сайт: <https://laravel.com/>

Get Started:



Your first project:



Для установки Laravel нам понадобится менеджер зависимостей composer:



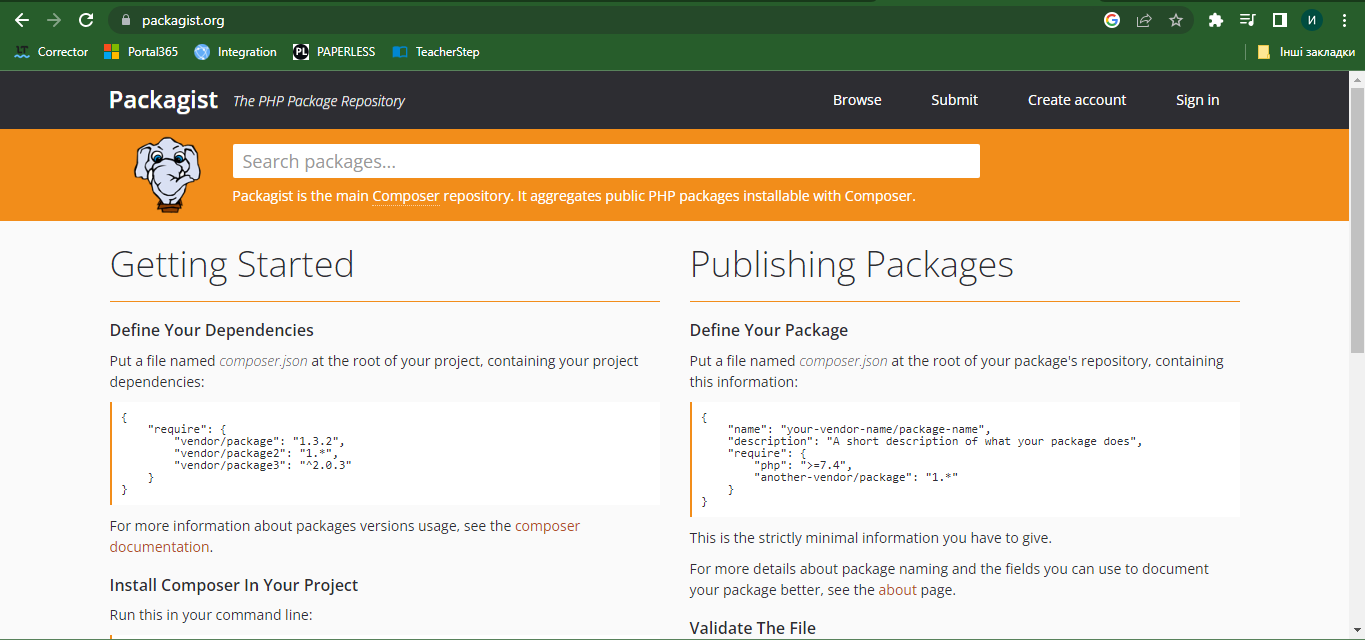
Composer

Composer — это пакетный менеджер уровня приложений для языка программирования PHP, который предоставляет средства по управлению зависимостями в PHP-приложении. Composer разработали и продолжают поддерживать два программиста Nils Adermann и Jordi Boggiano. Они начали разрабатывать Composer в апреле 2011, а первый релиз состоялся 1 марта 2012. Идея создания пакетных менеджеров уровня приложений не нова и его авторы вдохновлялись уже существовавшими на тот момент времени npm для Node.js и bundler для Ruby.

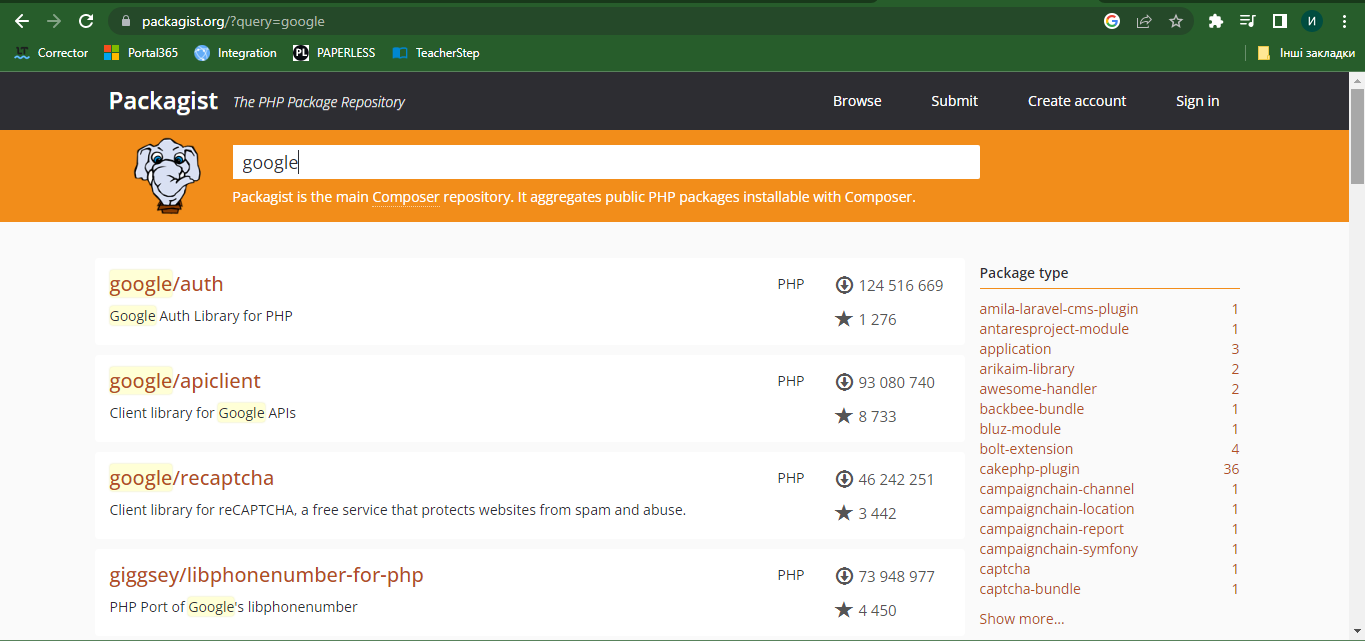
Composer работает через интерфейс командной строки и устанавливает зависимости (например библиотеки) для приложения. Он также позволяет пользователям устанавливать PHP-приложения, которые доступны на <https://packagist.org/>

, который является его основным репозиторием, где содержатся все доступные пакеты.

Здесь мы можем проводить анализ пакетов:



Например google auth:



Пакеты добавляются регулярно и Вы можете сами написать свой пакет и его сюда добавить:

Если кратко то вот порядок действий.

Зарегистрироваться на packagist

В проекте создать composer.json указав vendor/package (выбрать незанятый vendor).

На packagist нажать на кнопку submit или пройти по ссылке <https://packagist.org/packages/submit>

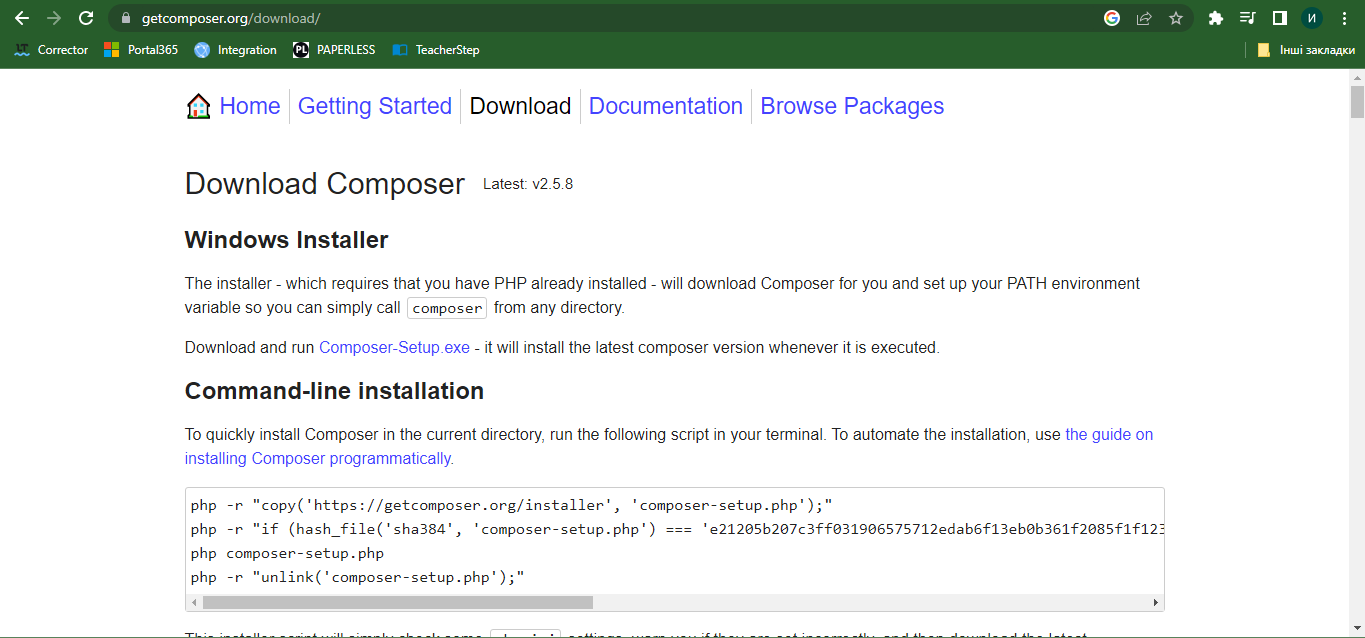
Указать адрес проекта на github или bitbucket

Он проверит composer.json если имя вендора свободно он предложит создать проект, также поищет похожие проекты

Есть еще и дополнительные шаги как сделать автообновления проекта и тд, но лучше почитать в доке на английском.

Для установки composer официальный сайт менеджера:

<https://getcomposer.org/download/>

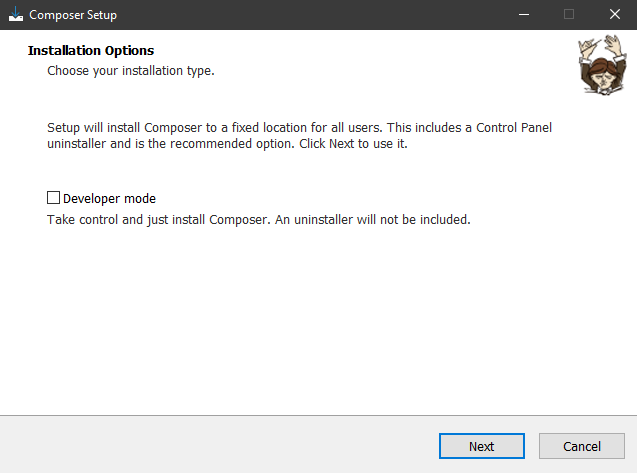


Так же здесь же Вы найдете отличную официальную документацию по всем командам и не только для composer:

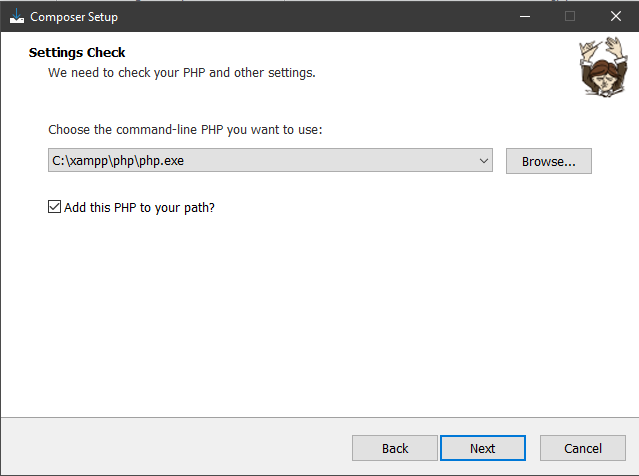
<https://getcomposer.org/doc/01-basic-usage.md>

Установка composer:

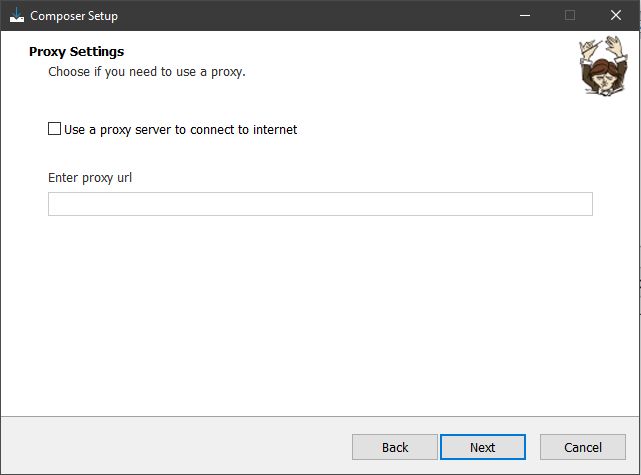
Next:



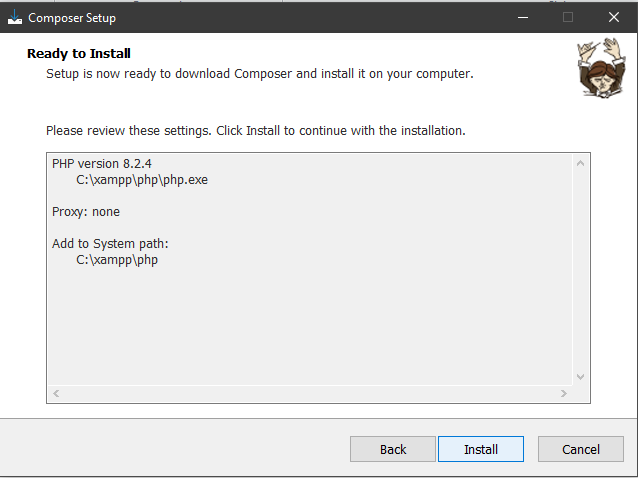
1. Прописываем правильный путь к папке php.exe
2. Поставить галочку Add Path (Переменные среды)
3. Next



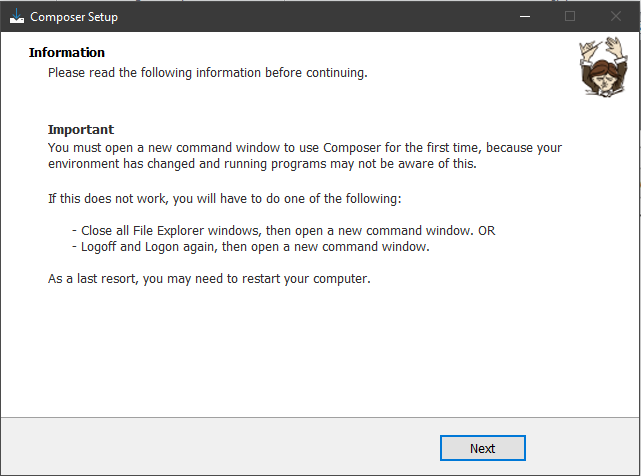
Next:



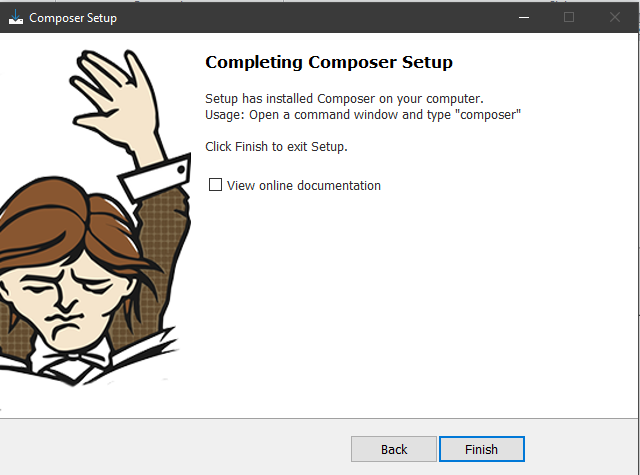
Install:



Next:



Finish:



Проверяем работу composer:

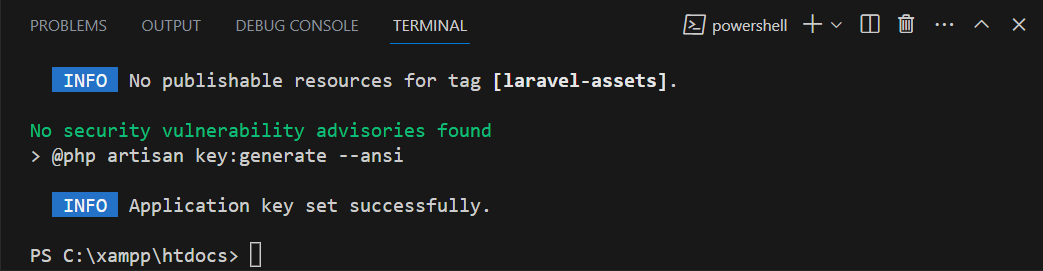
Win+R -> cmd



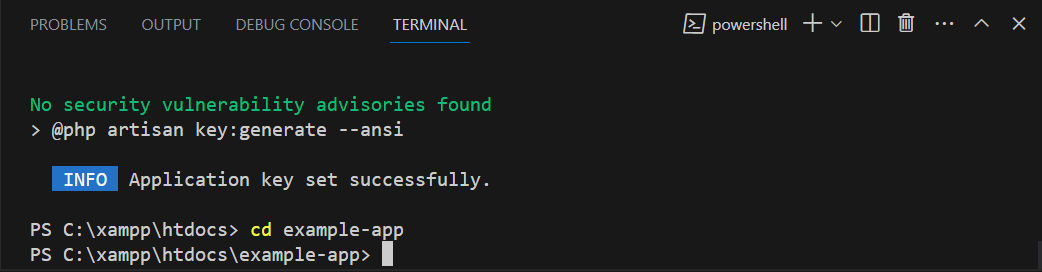
Продолжаем установку Laravel:

Открываем htdocs в vs code -> New terminal:

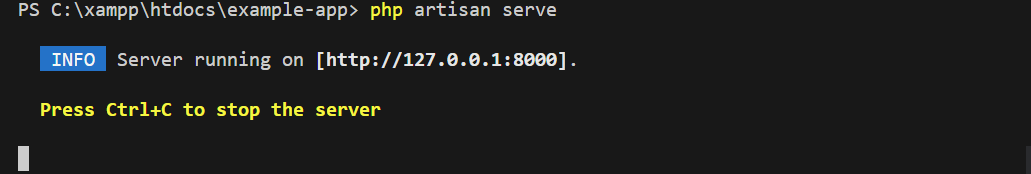
composer create-project laravel/laravel example-app



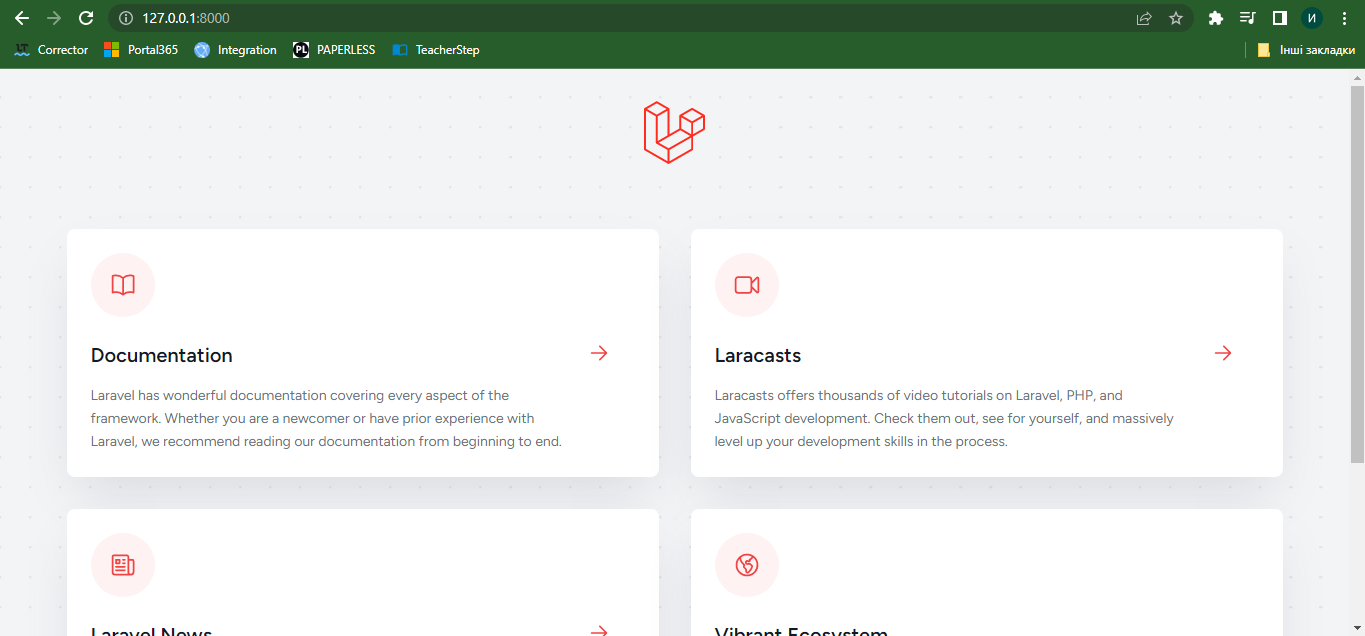
cd example-app



php artisan serve



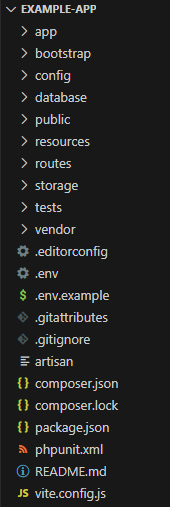
Заходим в браузер на <http://127.0.0.1:8000>



Laravel успешно установлен!

Структура проекта

Структура приложения Laravel по умолчанию предназначена для обеспечения отличной отправной точки как для больших, так и для малых приложений. Но вы можете организовать свое приложение так, как вам нравится. Laravel почти не накладывает ограничений на то, где находится тот или иной класс, пока Composer может автоматически загружать класс.



Корневой каталог

Каталог приложений

Каталог app содержит основной код вашего приложения. Вскоре мы рассмотрим этот каталог более подробно; однако почти все классы вашего приложения будут находиться в этом каталоге.

Каталог начальной загрузки

Каталог bootstrap содержит app.php файл, который загружает фреймворк. В этом каталоге также находится cacheкаталог, который содержит созданные фреймворком файлы для оптимизации производительности, такие как файлы кэша маршрутов и служб. Обычно вам не нужно изменять какие-либо файлы в этом каталоге.

Каталог конфигурации

Каталог config, как следует из названия, содержит все файлы конфигурации вашего приложения. Это отличная идея, чтобы прочитать все эти файлы и ознакомиться со всеми доступными вам вариантами.

Каталог базы данных

Каталог database содержит миграции вашей базы данных, фабрики моделей и исходные данные. При желании вы также можете использовать этот каталог для хранения базы данных SQLite.

Публичный каталог

Каталог public содержит index.php файл, который является точкой входа для всех запросов, поступающих в ваше приложение, и настраивает автозагрузку. В этом каталоге также хранятся ваши активы, такие как изображения, JavaScript и CSS.

Каталог ресурсов

Каталог resources содержит ваши представления , а также необработанные, не скомпилированные активы, такие как CSS или JavaScript.

Каталог маршрутов

Каталог routes содержит все определения маршрутов для вашего приложения. По умолчанию в Laravel включены несколько файлов маршрутов: web.php, api.php, console.php и channels.php.

Файл web.php содержит маршруты, которые RouteServiceProvider помещаются в web группу промежуточного программного обеспечения, которая обеспечивает состояние сеанса, защиту CSRF и шифрование файлов cookie. Если ваше приложение не предлагает RESTful API без сохранения состояния, то все ваши маршруты, скорее всего, будут определены в файле web.php.

Файл api.php содержит маршруты, которые RouteServiceProvider размещаются в api группе промежуточного программного обеспечения. Эти маршруты предназначены для сохранения состояния, поэтому запросы, поступающие в приложение по этим маршрутам, предназначены для аутентификации с помощью токенов и не будут иметь доступа к состоянию сеанса.

В этом console.php файле вы можете определить все ваши консольные команды на основе закрытия. Каждое замыкание привязано к экземпляру команды, что обеспечивает простой подход к взаимодействию с методами ввода-вывода каждой команды. Несмотря на то, что этот файл не определяет маршруты HTTP, он определяет точки входа (маршруты) на основе консоли в ваше приложение.

В этом channels.php файле вы можете зарегистрировать все каналы трансляции событий , поддерживаемые вашим приложением.

Каталог хранилища

Каталог storage содержит ваши журналы, скомпилированные шаблоны Blade, сеансы на основе файлов, файловые кеши и другие файлы, созданные платформой. Этот каталог разделен на каталоги app, framework и logs. Каталог app может использоваться для хранения любых файлов, созданных вашим приложением. Каталог framework используется для хранения сгенерированных фреймворком файлов и кешей. Наконец, logs каталог содержит файлы журналов вашего приложения.

Каталог storage/app/public может использоваться для хранения файлов, созданных пользователями, таких как аватары профилей, которые должны быть общедоступными. Вы должны создать символическую ссылку, по public/storage которой указывает на этот каталог. Вы можете создать ссылку с помощью php artisan storage:link команды Artisan.

Каталог тестов

Каталог tests содержит ваши автоматические тесты. Примеры модульных тестов PHPUnit и функциональных тестов предоставляются из коробки. Каждый тестовый класс должен иметь суффикс слова Test. Вы можете запускать свои тесты с помощью команд phpunitили php vendor/bin/phpunit. Или, если вы хотите более подробное и красивое представление результатов ваших тестов, вы можете запустить свои тесты с помощью php artisan test команды Artisan.

Каталог поставщиков

Каталог vendor содержит ваши зависимости Composer .

Каталог приложений

Большая часть вашего приложения размещена в app каталоге. По умолчанию этот каталог находится в пространстве имен App и автоматически загружается Composer с использованием стандарта автозагрузки PSR-4 . <https://www.php-fig.org/psr/psr-4/>

Каталог app содержит множество дополнительных каталогов, таких как Console, Http и Providers. Думайте о каталогах Console и Http как о предоставлении API в ядре вашего приложения. Протокол HTTP и интерфейс командной строки являются механизмами взаимодействия с вашим приложением, но фактически не содержат логики приложения. Другими словами, это два способа подачи команд вашему приложению. Каталог Console содержит все ваши команды Artisan, а Http каталог содержит ваши контроллеры, промежуточное ПО и запросы.

Множество других каталогов будет создано внутри app каталога, когда вы используете make Artisan-команды для создания классов. Так, например, app/Jobs каталог не будет существовать, пока вы не выполните make:job команду Artisan для создания класса задания.

Более подробно про структуру проекта можно прочитать здесь: <https://laravel.com/docs/10.x/structure#the-root-app-directory>

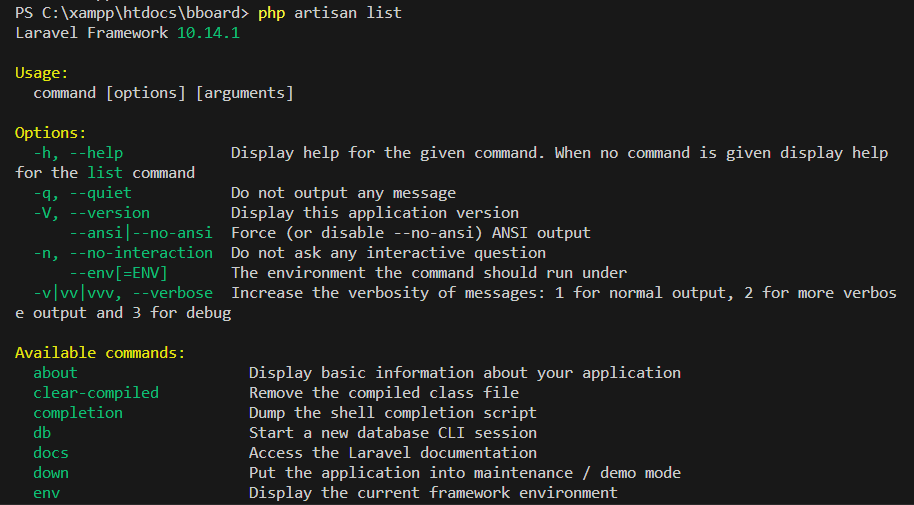
Artisan

Artisan - название интерфейса командной строки, входящей в состав Laravel. Он предоставляет полезные команды для использования во время разработки вашего приложения.

Здесь Вы найдете все нужные команды для использования консоли artisan: <https://laravel.com/docs/10.x/artisan>

Artisan — это утилита, написанная на PHP, хранящаяся непосредственно в папке проекта и служащая для выполнения различных действий над проектом. Команда serve этой утилиты запускает встроенный в PHP отладочный веб-сервер.

Например командой php artisan list можно посмотреть доступный список команд:



Vite



Vite — это современный инструмент для сборки фронтенд-приложений, созданный для ускорения процесса разработки и оптимизации сборки. Он использует нативную поддержку ES-модулей в браузере и компилирует модули "по требованию", что делает разработку более быстрой и эффективной.

2. Как работает Vite?

* Запуск сервера разработки: Vite запускает сервер разработки, который использует нативные ES-модули. Когда вы запускаете сервер (npm run dev), Vite предоставляет исходные файлы напрямую в браузер без предварительной сборки.
* Компиляция "по требованию": При запросе модуля браузером Vite компилирует и отправляет только этот модуль. Это позволяет избежать необходимости компиляции всего проекта и сокращает время ожидания изменений.
* Горячая замена модулей (HMR): Vite поддерживает HMR, что позволяет обновлять модули на лету без полной перезагрузки страницы. Это делает процесс разработки более продуктивным.
* Продакшн-сборка: Для продакшн-сборки Vite использует Rollup, который производит оптимизированные бандлы, применяет код-сплиттинг и другие оптимизации для минимизации размера и улучшения производительности.

Hot Module Replacement (HMR) в Vite позволяет обновлять модули во время выполнения без полной перезагрузки страницы. Это значительно ускоряет процесс разработки, так как изменения в коде мгновенно отражаются

в браузере, сохраняя текущее состояние приложения. Вот как это работает:

Основные принципы HMR в Vite:

* WebSocket сервер: Vite запускает WebSocket сервер, который следит за изменениями файлов в проекте. Когда файл изменяется, сервер отправляет уведомление в браузер через WebSocket соединение.
* HMR runtime: Браузер загружает специальный HMR runtime код, который управляет получением обновлений модулей от WebSocket сервера. Этот код встроен в каждый бандл, созданный Vite.
* Обновление модулей: Когда сервер обнаруживает изменения в файлах, он пересобирает только измененные модули и отправляет их в браузер. HMR runtime принимает эти обновления и пытается применить их к уже загруженным модулям.
* HMR API: Для некоторых библиотек и фреймворков (например, React, Vue) Vite предоставляет специальное HMR API, которое позволяет корректно обновлять состояние приложения без полной перезагрузки. Например, в React с помощью библиотеки react-refresh можно обновлять компоненты без потери состояния.

Сравнение Vite и Webpack

1. Производительность

* Vite: Быстрое время запуска и обновления благодаря использованию ES-модулей и компиляции "по требованию". Мгновенное обновление с помощью HMR.
* Webpack: Медленное время запуска и сборки, особенно для больших проектов, из-за необходимости предварительной компиляции всего приложения. HMR может быть медленным и требует дополнительной настройки.

2. Конфигурация

* Vite: Простая конфигурация с использованием vite.config.js. Поддержка плагинов основана на Rollup и собственных плагинах Vite.
* Webpack: Сложная конфигурация с использованием webpack.config.js. Требуется настройка загрузчиков и плагинов, что может усложнить процесс.

3. Горячая замена модулей (HMR)

* Vite: Эффективная HMR, основанная на ES-модулях, с быстрым обновлением модулей.
* Webpack: HMR может быть менее эффективным и медленным, требует настройки и может не всегда работать оптимально.

4. Плагины и расширяемость

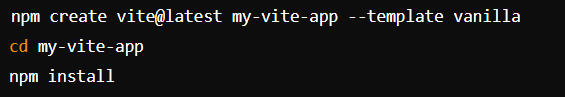
* Vite: Поддержка плагинов основана на Rollup и собственных плагинах Vite. Экосистема плагинов развивающаяся, но может быть менее разнообразной.
* Webpack: Обширная экосистема плагинов и загрузчиков, позволяющая реализовать множество функциональностей. Экосистема зрелая и хорошо поддерживается.

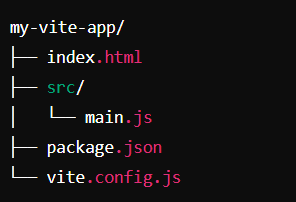
5. Поддержка фреймворков

* Vite: Встроенная поддержка популярных фреймворков (Vue, React, Svelte) и простота настройки.
* Webpack: Широкая поддержка фреймворков, но может требовать сложной настройки.

Пример приложения Vite + чистый JavaScript

Создайте новый проект с помощью Vite:





index.html

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Vite + Vanilla JS</title>

</head>

<body>

  <h1 id="greeting">Hello, Vite!</h1>

  <button id="changeText">Change Text</button>

  <script type="module" src="/src/main.js"></script>

</body>

</html>

src/main.js

document.getElementById('changeText').addEventListener('click', () => {

    const greeting = document.getElementById('greeting');

    greeting.textContent = 'Text changed with Vite!';

  });

Откройте браузер и перейдите по адресу http://localhost:3000. Вы увидите страницу с текстом "Hello, Vite!" и кнопкой. При нажатии на кнопку текст изменится на "Text changed with Vite!".



Сборка для продакшн

Выполните команду для сборки:



Результат будет находиться в папке dist, готовый для деплоя.

Official website: <https://vitejs.dev/>

CRUD

Перед началом обязательно удалить базу данных Laravel

composer create-project laravel/laravel crud\_app

cd crud\_app

php artisan make:model Task -m

php artisan make:controller TaskController

app/Http/Controllers/TaskController.php

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Models\Task;

class TaskController extends Controller

{

    public function index()

    {

        $tasks = Task::all();

        return view('tasks.index', ['tasks' => $tasks]);

    }

    public function create()

    {

        return view('tasks.create');

    }

    public function store(Request $request)

    {

        $this->validate($request, [

            'title' => 'required|max:255',

            'description' => 'required',

        ]);

        $task = new Task();

        $task->title = $request->input('title');

        $task->description = $request->input('description');

        $task->save();

        return redirect()->route('tasks.index')->with('success', 'Задача успешно создана.');

    }

    public function show($id)

    {

        $task = Task::findOrFail($id);

        return view('tasks.show', ['task' => $task]);

    }

    public function edit($id)

    {

        $task = Task::findOrFail($id);

        return view('tasks.edit', ['task' => $task]);

    }

    public function update(Request $request, $id)

    {

        $this->validate($request, [

            'title' => 'required|max:255',

            'description' => 'required',

        ]);

        $task = Task::findOrFail($id);

        $task->title = $request->input('title');

        $task->description = $request->input('description');

        $task->save();

        return redirect()->route('tasks.index')->with('success', 'Задача успешно обновлена.');

    }

    public function destroy($id)

    {

        $task = Task::findOrFail($id);

        $task->delete();

        return redirect()->route('tasks.index')->with('success', 'Задача успешно удалена.');

    }

}

app/Models/Task.php

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Task extends Model

{

    protected $fillable = ['title', 'description'];

    // Дополнительные методы и свойства модели могут быть добавлены здесь

}

database/Migrations/ 2023\_09\_07\_120815\_create\_tasks\_table.php

<?php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

return new class extends Migration

{

    // database/migrations/<timestamp>\_create\_tasks\_table.php

public function up()

{

    Schema::create('tasks', function (Blueprint $table) {

        $table->id();

        $table->string('title');

        $table->text('description');

        $table->timestamps();

    });

}

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

        Schema::dropIfExists('tasks');

    }

};

resources/index.blade.php

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC" crossorigin="anonymous">

    <title>Tasks</title>

</head>

<body>

    <table class="table">

        <tr scope="row">

            <th scope="col">id</th>

            <th scope="col">title</th>

            <th scope="col">description</th>

        </tr>

        @foreach ($tasks as $task)

            <tr scope="row">

                <td>{{ $task->id }}</td>

                <td>{{ $task->title }}</td>

                <td>{{ $task->description }}</td>

            </tr>

        @endforeach

    </table>

</body>

</html>

Если миграция была уже сделана, то можно обновить миграцию

php artisan migrate:refresh

php artisan migrate

npm install

удаляем ненужный плагин

npm uninstall laravel-vite-plugin

Убедитесь, что в файле resources/views/layouts/app.blade.php в секции "Scripts" отсутствует вызов @vite, и он выглядит примерно так:

Удалить строку из loyout

recourses/loyouts

<!-- Scripts -->

@vite(['resources/sass/app.scss', 'resources/js/app.js'])

app.blade.php

<!doctype html>

<html lang="{{ str\_replace('\_', '-', app()->getLocale()) }}">

<head>

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- CSRF Token -->

    <meta name="csrf-token" content="{{ csrf\_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Fonts -->

    <link rel="dns-prefetch" href="//fonts.bunny.net">

    <link href="https://fonts.bunny.net/css?family=Nunito" rel="stylesheet">

</head>

<body>

    <div id="app">

        <nav class="navbar navbar-expand-md navbar-light bg-white shadow-sm">

            <div class="container">

                <a class="navbar-brand" href="{{ url('/') }}">

                    {{ config('app.name', 'Laravel') }}

                </a>

                <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="{{ \_\_('Toggle navigation') }}">

                    <span class="navbar-toggler-icon"></span>

                </button>

                <div class="collapse navbar-collapse" id="navbarSupportedContent">

                    <!-- Left Side Of Navbar -->

                    <ul class="navbar-nav me-auto">

                    </ul>

                    <!-- Right Side Of Navbar -->

                    <ul class="navbar-nav ms-auto">

                        <!-- Authentication Links -->

                        @guest

                            @if (Route::has('login'))

                                <li class="nav-item">

                                    <a class="nav-link" href="{{ route('login') }}">{{ \_\_('Login') }}</a>

                                </li>

                            @endif

                            @if (Route::has('register'))

                                <li class="nav-item">

                                    <a class="nav-link" href="{{ route('register') }}">{{ \_\_('Register') }}</a>

                                </li>

                            @endif

                        @else

                            <li class="nav-item dropdown">

                                <a id="navbarDropdown" class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown" aria-haspopup="true" aria-expanded="false" v-pre>

                                    {{ Auth::user()->name }}

                                </a>

                                <div class="dropdown-menu dropdown-menu-end" aria-labelledby="navbarDropdown">

                                    <a class="dropdown-item" href="{{ route('logout') }}"

                                       onclick="event.preventDefault();

                                                     document.getElementById('logout-form').submit();">

                                        {{ \_\_('Logout') }}

                                    </a>

                                    <form id="logout-form" action="{{ route('logout') }}" method="POST" class="d-none">

                                        @csrf

                                    </form>

                                </div>

                            </li>

                        @endguest

                    </ul>

                </div>

            </div>

        </nav>

        <main class="py-4">

            @yield('content')

        </main>

    </div>

</body>

</html>

В Laravel, @csrf — это Blade-директива, используемая для генерации скрытого токена CSRF (Cross-Site Request Forgery), который помогает защитить ваше приложение от CSRF-атак. CSRF-атаки происходят, когда злоумышленник пытается выполнить несанкционированные команды от имени аутентифицированного пользователя.

routes/web.php

<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\TaskController;

/\*

|--------------------------------------------------------------------------

| Web Routes

|--------------------------------------------------------------------------

|

| Here is where you can register web routes for your application. These

| routes are loaded by the RouteServiceProvider and all of them will

| be assigned to the "web" middleware group. Make something great!

|

\*/

//Auth::routes();

// Routes CRUD

Route::resource('tasks', TaskController::class);

Route::get('tasks', [TaskController::class, 'index'])->name('tasks.index');

Route::get('tasks/create', [TaskController::class, 'create'])->name('tasks.create');

Route::post('tasks', [TaskController::class, 'store'])->name('tasks.store');

Route::get('tasks/{task}', [TaskController::class, 'show'])->name('tasks.show');

Route::get('tasks/{task}/edit', [TaskController::class, 'edit'])->name('tasks.edit');

Route::put('tasks/{task}', [TaskController::class, 'update'])->name('tasks.update');

Route::delete('tasks/{task}', [TaskController::class, 'destroy'])->name('tasks.destroy');

Создание сайта доска объявлений (Практика)

Контроллеры и действия

Контроллер — это программный модуль, реализующий функциональность одного из разделов сайта (например, раздела, выводящего объявления). Действие (action) — одна из операций, выполняемых контроллером (вывод страницы с перечнем объявлений, вывод отдельного объявления, вывод страницы для добавления объявления, сохранение добавленного объявления в базе и пр.).

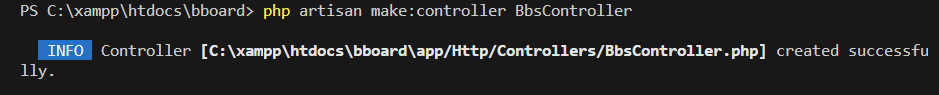
Сайт может содержать произвольное количество контроллеров (обычно по числу входящих в него разделов), а каждый контроллер — произвольное количество действий.

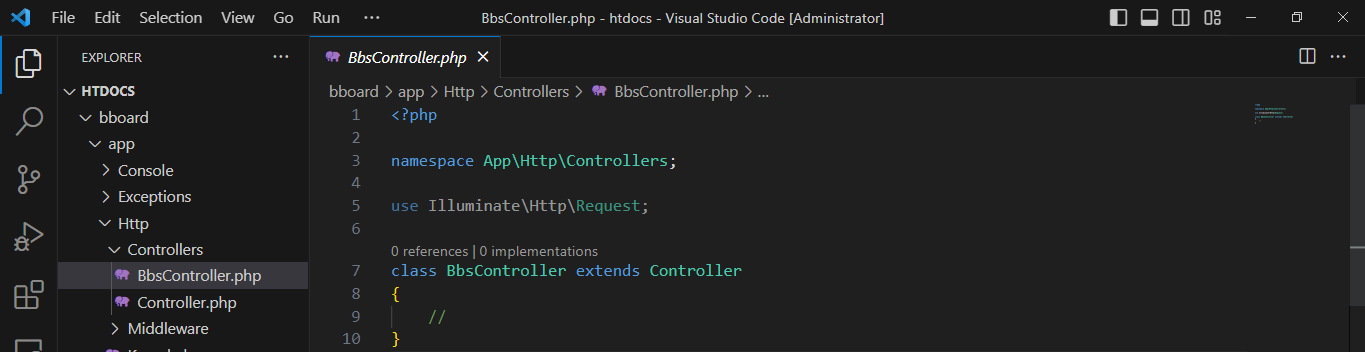
Laravel поддерживает три разновидности контроллеров:

1. контроллер-класс — реализуется в виде класса, а его действия — в виде методов этого класса. Позволяет свести всю функциональность раздела сайта в один программный модуль. По принятому в Laravel соглашению модули с контроллерами-классами сохраняются в папке app\Http\Controllers папки проекта, а имена их классов должны заканчиваться словом Controller;
2. контроллер-функция — реализуется в виде анонимной функции и содержит лишь одно действие. Применяется для выполнения самых простых операций, наподобие вывода служебной страницы. Контроллеры-функции записываются непосредственно в списках маршрутов (о них — чуть позже);
3. контроллеры одного действия — нечто промежуточное между классами и функциями — реализуются в виде классов, но содержат лишь одно действие.

Напишем контроллер BbsController, обрабатывающий список объявлений, с действием index(), которое в будущем станет выводить страницу с перечнем объявлений, а сейчас — временную «заглушку» в виде обычного текста.

php artisan make:controller BbsController





Контроллер объявлен в пространстве имен App\Http\Controllers и является производным от суперкласса Controller. Изначально он «пуст» — не содержит ни одного метода-действия.

Добавим метод index()

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class BbsController extends Controller

{

    public function index() {

        return response('Здесь будет перечень объявлений.')

        ->header('Content-Type', 'text/plain');

        }

}

Функция response() генерирует серверный ответ, представленный объектом класса Illuminate\Http\Response, на основе строки, переданной в параметре, и возвращает его в качестве результата. Метод header() класса Illuminate\Http\Response помещает в текущий серверный ответ заголовок с заданными в параметрах именем и значением. Мы используем этот метод, чтобы поместить в ответ заголовок Content-Type со значением text/plain, тем самым указав веб-обозревателю, что ответ содержит обычный текст. Готовый ответ следует вернуть из метода-действия в качестве результата — чтобы Laravel смог отправить его клиенту.

Маршруты и списки маршрутов

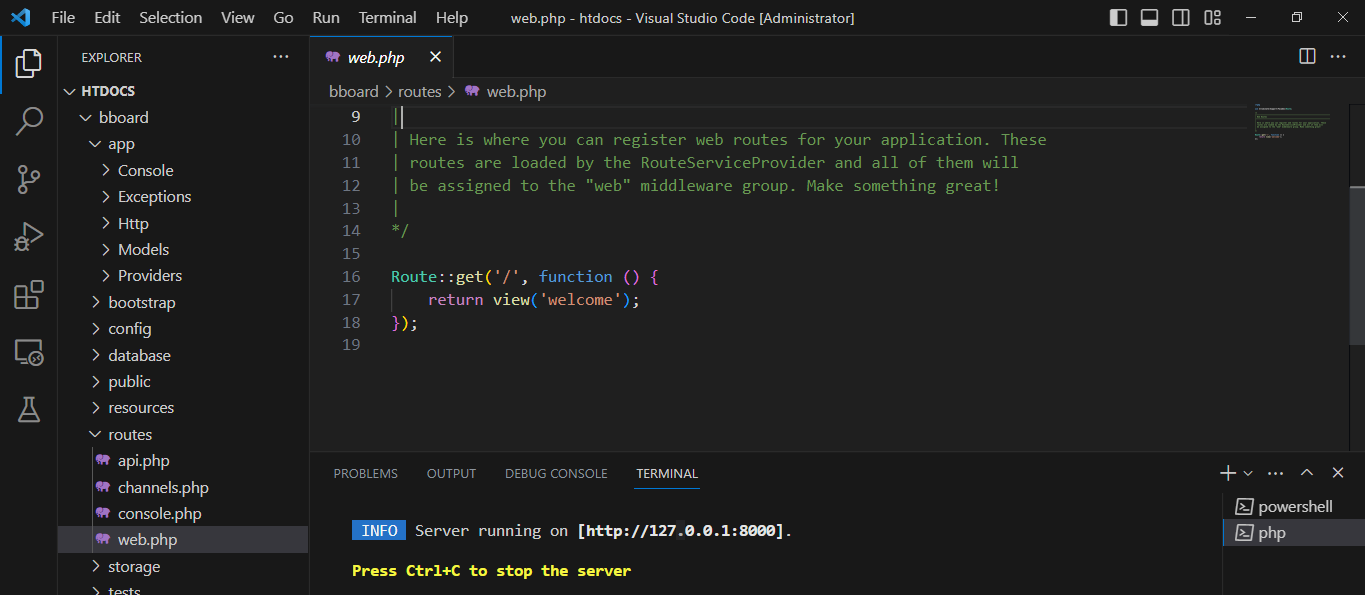
Фасады

Каждая операция, производимая сайтом (вывод страницы, сохранение введенных данных в базе и пр.), выполняется при получении им от веб-обозревателя клиентского запроса по определенному пути, выполненного с применением определенного HTTP-метода (GET, POST, PATCH и др.). Путь — это часть интернет-адреса, находящаяся между адресом хоста и набором GET-параметров и идентифицирующая запрашиваемую страницу (например, интернет-адрес http://localhost:8000/items/34?from=index содержит путь items/34).

Следовательно, чтобы какое-либо действие контроллера выполнилось при получении запроса по определенному пути, выполненного определенным HTTP-методом, его следует связать с этими путем и методом, создав маршрут. Маршрут Laravel — это объект особого класса, содержащий следующие сведения:

1. шаблонный путь — задает нужный формат путей;
2. допустимый HTTP-метод — которым должен быть выполнен клиентский запрос;
3. действие контроллера — выполняется при совпадении шаблонного пути и допустимого метода с путем и методом, извлеченными из запроса (т. е. если маршрут является совпавшим).

Создадим веб-маршрут, связывающий шаблонный путь / («корень» сайта) и допустимый HTTP-метод GET с действием index() контроллера BbsController.



Класс Route — это фасад маршрутизатора (фасадом называется класс, служащий своего рода «пультом управления» одной из подсистем фреймворка). Статический метод get(), вызванный у этого фасада, указывает маршрутизатору создать новый объект маршрута, связывающий допустимый HTTP-метод GET (одноименный методу фасада), шаблонный путь из первого параметра (у нас — /, «корень» сайта) и контроллер-функцию, заданную вторым параметром. Последний генерирует на основе шаблона welcome.blade.php

<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\BbsController;

/\*

|--------------------------------------------------------------------------

| Web Routes

|--------------------------------------------------------------------------

|

| Here is where you can register web routes for your application. These

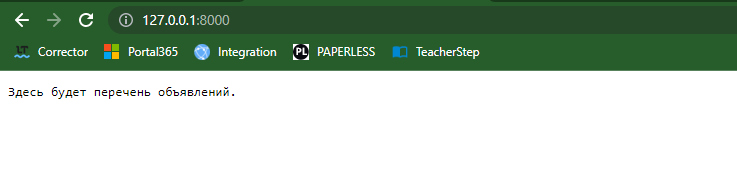
| routes are loaded by the RouteServiceProvider and all of them will

| be assigned to the "web" middleware group. Make something great!

|

\*/

Route::get('/', [BbsController::class, 'index']);



Настройки проекта. Подготовка проекта к работе с базой данных

в файле .env — локальные, задающие параметры текущей платформы (сведения для подключения к базам данных, серверу электронной почты, службе кэширования и пр.);

в папке config — рабочие, затрагивающие все аспекты функционирования сайта и непосредственно используемые Laravel для получения всех сведений о нем. Рабочие настройки хранятся в 14 разных PHP-модулях. Так, модуль app.php содержит настройки самого проекта (его имя, режим работы и др.), модуль database.php — настройки подключения к базе данных, а cache.php — настройки подсистемы кэширования. Каждый из этих модулей включает ассоциативный массив, элементы которых содержат значения соответствующих настроек — как обычные скалярные величины, так и массивы, в том числе ассоциативные. Значения некоторых рабочих настроек загружаются из файла .env. Таким образом, Laravel фактически объединяет локальные и рабочие настройки для удобства программирования.

.env

APP\_NAME=Laravel

APP\_ENV=local

APP\_KEY=base64:+rlvtltiZX6X8wggoLUFx9Gmno/833i3rEWEMIEewvw=

APP\_DEBUG=true

APP\_URL=http://localhost

LOG\_CHANNEL=stack

LOG\_DEPRECATIONS\_CHANNEL=null

LOG\_LEVEL=debug

DB\_CONNECTION=mysql

DB\_HOST=127.0.0.1

DB\_PORT=3306

DB\_DATABASE=bboard

DB\_USERNAME=root

DB\_PASSWORD=

BROADCAST\_DRIVER=log

CACHE\_DRIVER=file

FILESYSTEM\_DISK=local

QUEUE\_CONNECTION=sync

SESSION\_DRIVER=file

SESSION\_LIFETIME=120

MEMCACHED\_HOST=127.0.0.1

REDIS\_HOST=127.0.0.1

REDIS\_PASSWORD=null

REDIS\_PORT=6379

MAIL\_MAILER=smtp

MAIL\_HOST=mailpit

MAIL\_PORT=1025

MAIL\_USERNAME=null

MAIL\_PASSWORD=null

MAIL\_ENCRYPTION=null

MAIL\_FROM\_ADDRESS="hello@example.com"

MAIL\_FROM\_NAME="${APP\_NAME}"

AWS\_ACCESS\_KEY\_ID=

AWS\_SECRET\_ACCESS\_KEY=

AWS\_DEFAULT\_REGION=us-east-1

AWS\_BUCKET=

AWS\_USE\_PATH\_STYLE\_ENDPOINT=false

PUSHER\_APP\_ID=

PUSHER\_APP\_KEY=

PUSHER\_APP\_SECRET=

PUSHER\_HOST=

PUSHER\_PORT=443

PUSHER\_SCHEME=https

PUSHER\_APP\_CLUSTER=mt1

VITE\_PUSHER\_APP\_KEY="${PUSHER\_APP\_KEY}"

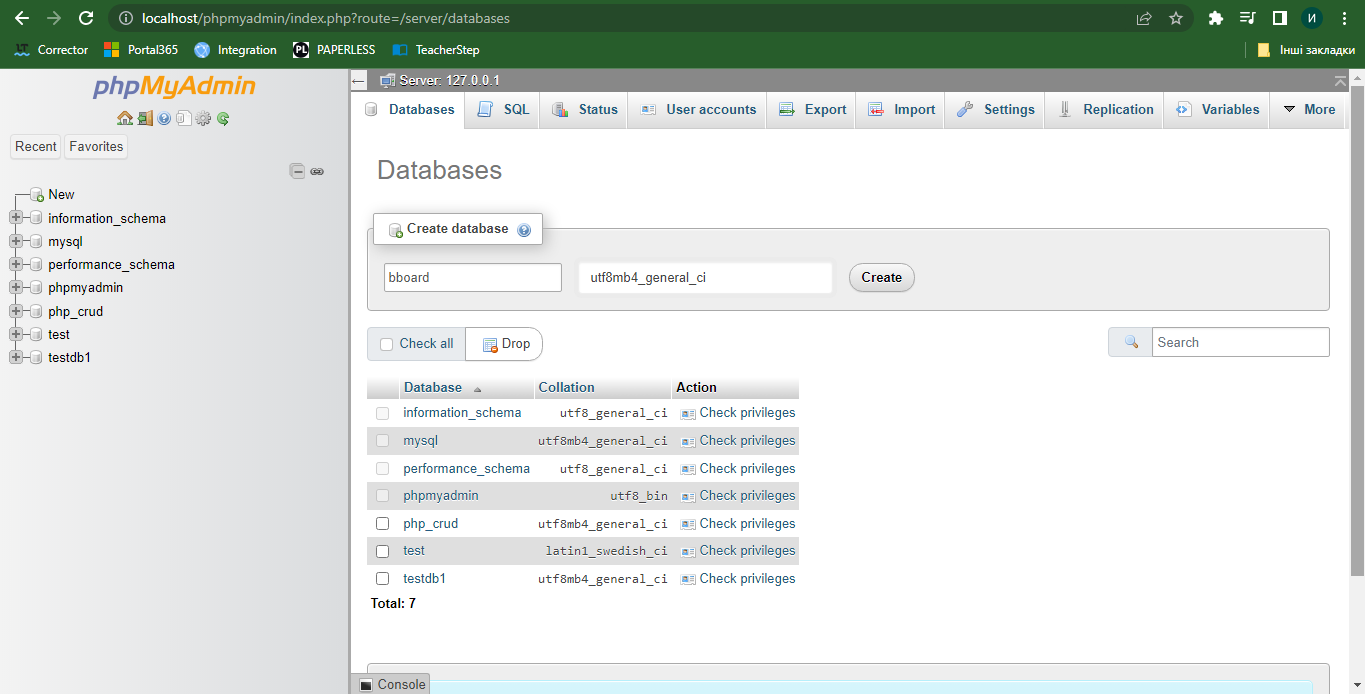
VITE\_PUSHER\_HOST="${PUSHER\_HOST}"

VITE\_PUSHER\_PORT="${PUSHER\_PORT}"

VITE\_PUSHER\_SCHEME="${PUSHER\_SCHEME}"

VITE\_PUSHER\_APP\_CLUSTER="${PUSHER\_APP\_CLUSTER}"

Создаем базу данных:



database.php:



Проверяем подключение к бд:

Routes -> web.php

Добавим новый маршрут:

use Illuminate\Support\Facades\DB;

Route::get('/test', function() {

    if (DB::connection()->getDatabaseName())  {

        $name = DB::connection()->getDatabaseName();

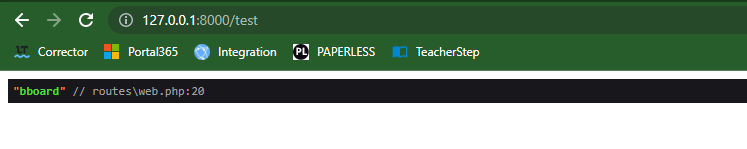
      dd($name);

    } else {

      return 'Error';

    }});

Проверяем:



Все ОК!

Миграции

Миграция — программный PHP-модуль, вносящий какие-либо изменения в структуру базы данных. Миграция может, например, создать таблицу вместе со всеми полями и индексами, исправить имя или тип поля в существующей таблице, создать или удалить индекс. Миграция реализуется в виде класса.

Миграцию можно применить или откатить. При применении миграция выполняет все описанные в ней действия. При откате же она возвращает базу данных в состояние, существовавшее перед применением этой миграции. Понятно, что откатить можно лишь миграцию, примененную ранее. Программные модули с миграциями хранятся в папке database\migrations. Список всех примененных миграций в хронологическом порядке сохраняется в особой таблице базы данных, создаваемой перед применением самой первой миграции.

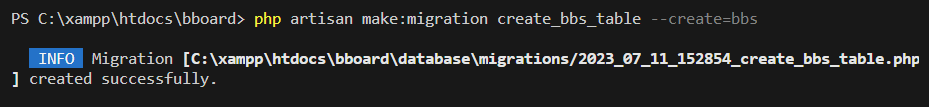
Напишем миграцию create\_bbs\_table, создающую в базе данных таблицу bbs со следующими полями:

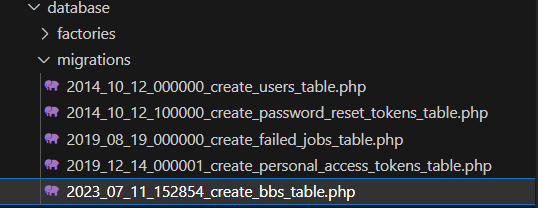
1. title — заголовок объявления с названием продаваемого товара (тип — строковый, длина — 50 символов);
2. content — сам текст объявления, описание товара (тип — memo);
3. price — цена (тип — вещественное число).

В командной строке создадим «пустую» миграцию create\_bbs\_table:

php artisan make:migration create\_bbs\_table --create=bbs

Команда make:migration утилиты artisan создает миграцию с заданным именем. Дополнительный параметр --create предписывает вставить в миграцию код, создающий таблицу, чье имя указано в параметре.





Код миграции:

<?php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

return new class extends Migration

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

        Schema::create('bbs', function (Blueprint $table) {

            $table->id();

            $table->timestamps();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

        Schema::dropIfExists('bbs');

    }

};

Класс миграции содержит два метода: up(), выполняющийся при применении миграции, и down(), запускаемый при ее откате. Параметр --create предписывает утилите artisan сразу же вставить в эти методы код, соответственно создающий и удаляющий заданную в параметре таблицу.

В методе up() миграции выполняется создание таблицы. Для этого у фасада Schema, за которым «прячется» подсистема, манипулирующая структурой базы данных, вызывается метод create(). В первом параметре методу передается имя создаваемой таблицы, а во втором — анонимная функция, в качестве параметра принимающая объект класса Blueprint, который представляет структуру создаваемой таблицы. Добавление полей в нее выполняется вызовом различных методов у этого объекта. Параметр --create также предписывает утилите artisan вставить в код этой функции вызовы методов id() и timestamps(). Первый метод добавляет в формируемую таблицу поле id, хранящее ключ (какое-либо уникальное значение, однозначно идентифицирующее запись, обычно — уникальный номер, генерируемый встроенным автоинкрементом. Поле, хранящее ключ, называется ключевым), а второй — поля для хранения временны´ х отметок создания записи (отметка создания) и ее последней правки (отметка правки).

В методе down() производится удаление таблицы. Для этого у фасада Schema вызывается метод dropIfExists(), которому передается имя удаляемой таблицы.

Добавим в метод up() миграции код, создающий поля title, content и price:

    public function up()

    {

        Schema::create('bbs', function (Blueprint $table) {

            $table->id();

            $table->string('title', 50);

            $table->text('content');

            $table->float('price');

            $table->timestamps();

        });

    }

Метод string() класса Blueprint создает строковое поле типа VARCHAR с именем и длиной, заданными в параметрах метода. Для создания поля memo типа TEXT и поля вещественного типа FLOAT используются методы text() и float().

Мы будем выводить объявления в хронологическом порядке, отсортированными по значению поля отметки создания записи. Это поле создается вызовом метода timestamps() и по умолчанию имеет имя created\_at. Для ускорения сортировки создадим по нему индекс.

Добавим код, создающий обычный индекс по полю created\_at:

    public function up()

    {

        Schema::create('bbs', function (Blueprint $table) {

            $table->id();

            $table->string('title', 50);

            $table->text('content');

            $table->float('price');

            $table->timestamps();

            $table->index('created\_at');

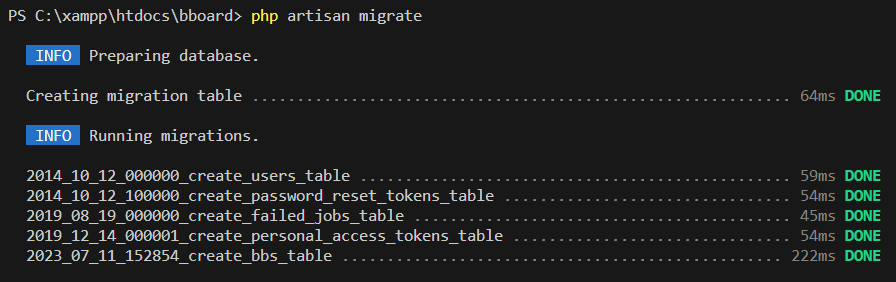
        });

    }

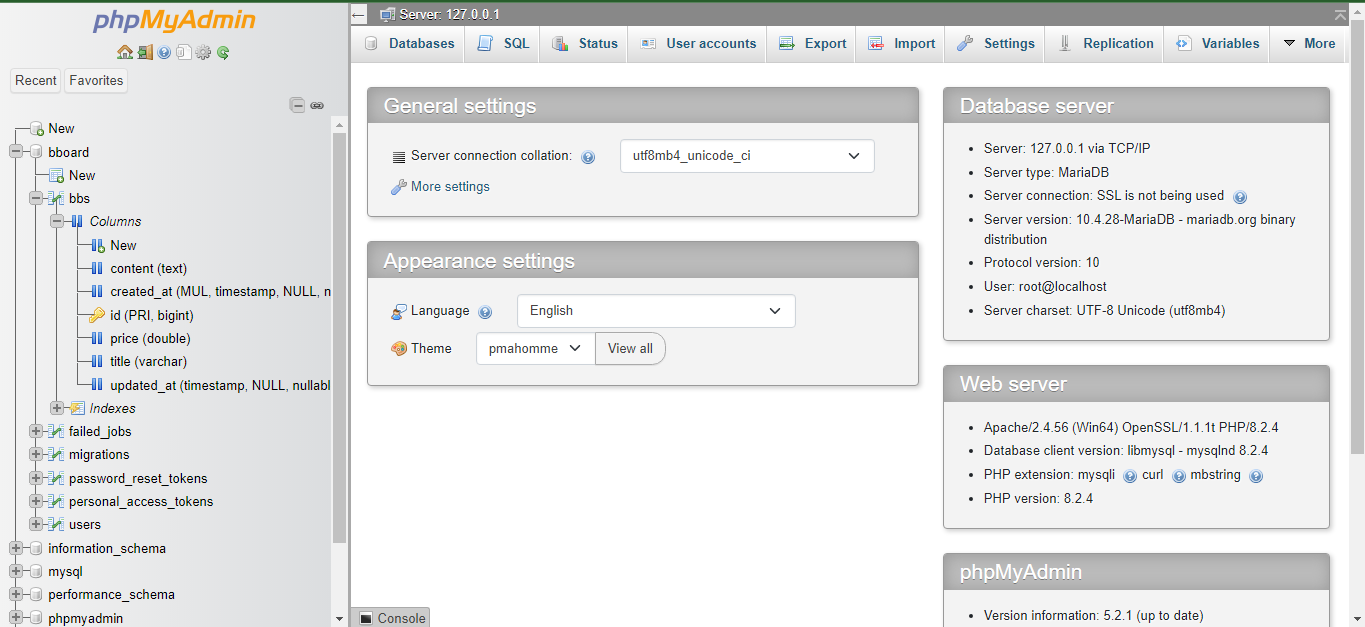
Применим только что созданную миграцию, набрав в командной строке команду:

php artisan migrate

В результате Laravel выполнит все еще не выполненные миграции, находящиеся в папке database\migrations.



Проверим результат в phpmyadmin:



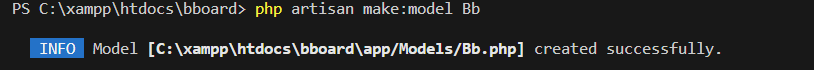
Модели

Модель — программный модуль, служащий для взаимодействия с обслуживаемой им таблицей базы данных: выборки записей, извлечения значений полей, добавления, правки и удаления записей. Модель реализуется в виде класса. Файлы с моделями по умолчанию хранятся в папке app\Models.

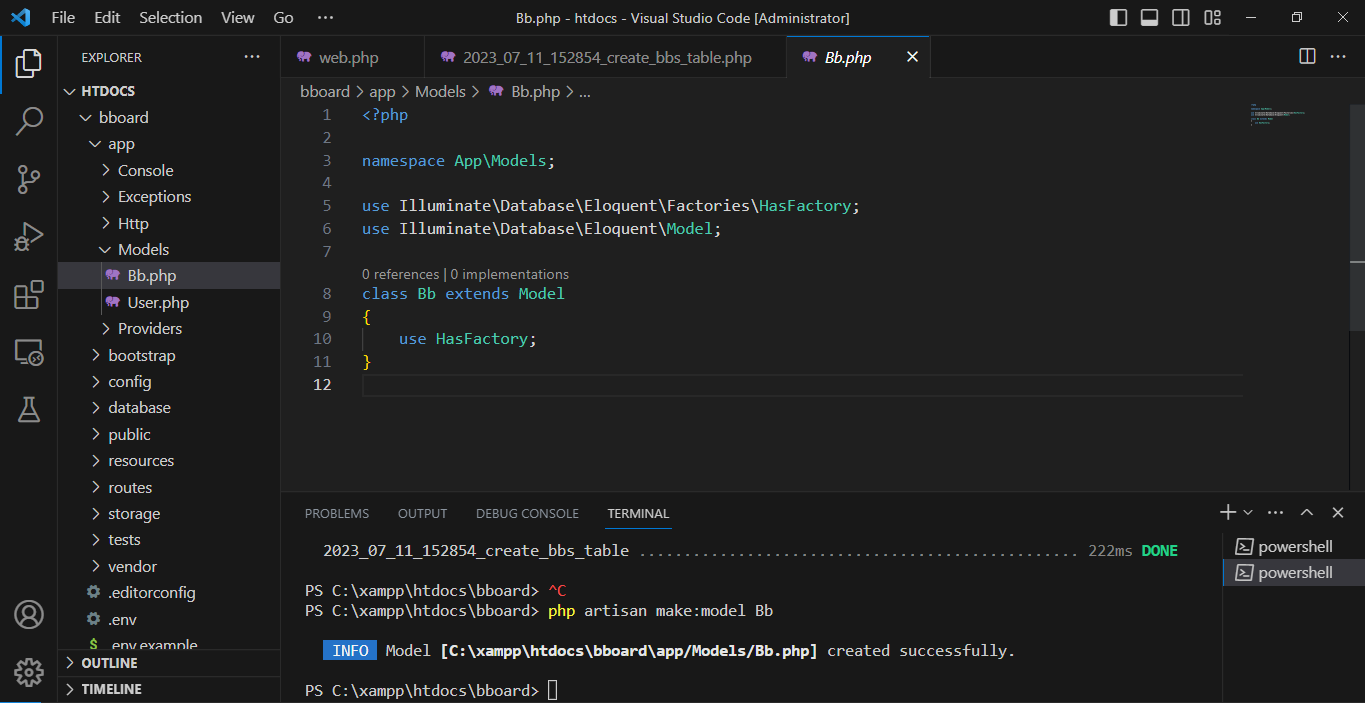
Напишем модель Bb, предназначенную для работы с таблицей bb базы данных:

В командной строке создадим модуль с классом модели:

php artisan make:model Bb



Откроем модуль app\Models\bb.php, хранящий созданную модель, в текстовом редакторе и посмотрим на его содержимое:



Всю функциональность модели, необходимую для работы с базой данных, реализует базовый класс Illuminate\Database\Eloquent\Model. Трейт Illuminate\ Database\Eloquent\Factories\HasFactory используется лишь при автоматизированном тестировании

Сделаем поля title, content и price доступными для массового присваивания (о нем — чуть позже), добавив в класс модели выделенный полужирным шрифтом код, и заодно удалим трейт HasFactory:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Bb extends Model

{

    protected $fillable = ['title', 'content', 'price'];

}

Массив с именами полей, доступных для массового присваивания, заносится в защищенное свойство fillable.

Консоль Laravel

Консоль Laravel позволяет работать с классами фреймворка в интерактивном режиме. В частности, с ее помощью удобно заносить в информационную базу какие-либо отладочные данные.

В PHP встроен REPL, который позволяет быстро поэкспериментировать с языком.

В командной строке запустим консоль Laravel:

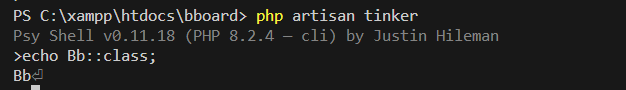
php artisan tinker

Приглашение к вводу команды обозначается префиксом >. Вывод результатов производится без всякого префикса.

Проверим консоль в работе, выведя полное имя класса модели Bb, для чего наберем следующие выражения, завершая каждое из них нажатием клавиши :

> use App\Models\Bb;

> echo Bb::class;

Следует помнить, что префиксом > обозначается приглашение к вводу команды. Результат выполнения команды выводится без префикса. Консоль Laravel позволяет набирать выражения в несколько строк, разрывая их нажатием клавиши .

Введем последнее из набранных выражений в две строки:

> use ... App\Models\Bb;

Префиксом ... (три точки) обозначается приглашение к вводу следующей строки многострочного выражения.

Завершим работу консоли Laravel, нажав комбинацию клавиш Ctrl + Break или Ctrl + C. Также можно набрать в консоли команду exit.

Работа с базой данных

Добавим несколько записей в таблицу объявлений bb, исправим какую-либо запись, удалим другую и попробуем выполнить выборку записей с фильтрацией и сортировкой. Затем переделаем действие index() контроллера HomeController таким образом, чтобы оно выводило перечень объявлений из базы данных.

1. Запустим консоль Laravel
2. Добавим первое объявление, набрав код:

> use App\Models\Bb;

> $bb = new Bb();

=> App\Models\Bb {#3019}

> $bb->title = 'Шкаф';

=> "Шкаф"

> $bb->content = 'Совсем новый, полированный, двухстворчатый';

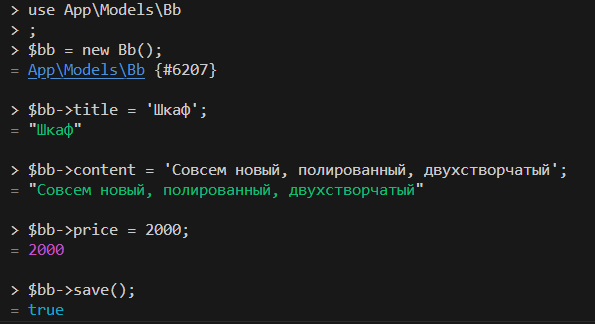
=> "Совсем новый, полированный, двухстворчатый"

> $bb->price = 2000;

=> 2000

> $bb->save();

=> true



Проверяем сохранилась ли запись в таблице:

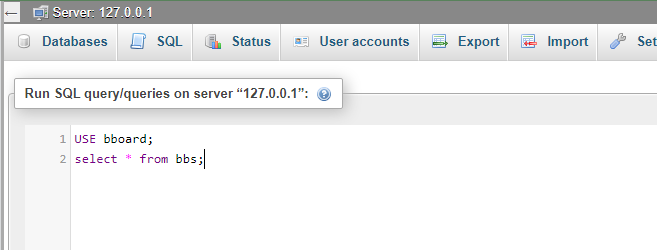
Обратимся id:

> echo $bb->id;

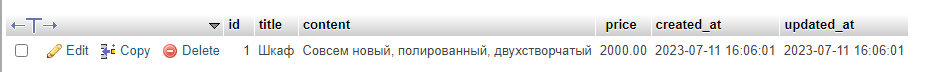


Все ок!

Проверим так же в самом phpmyadmin:



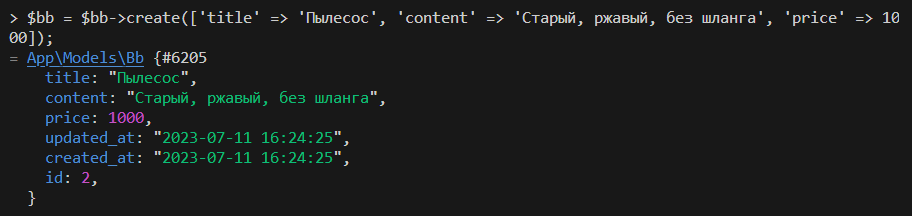
Результат ниже:



Все ОК!

Добавим другое объявление — другим способом:

$bb = $bb->create(['title' => 'Пылесос', 'content' => 'Старый, ржавый, без шланга', 'price' => 1000]);

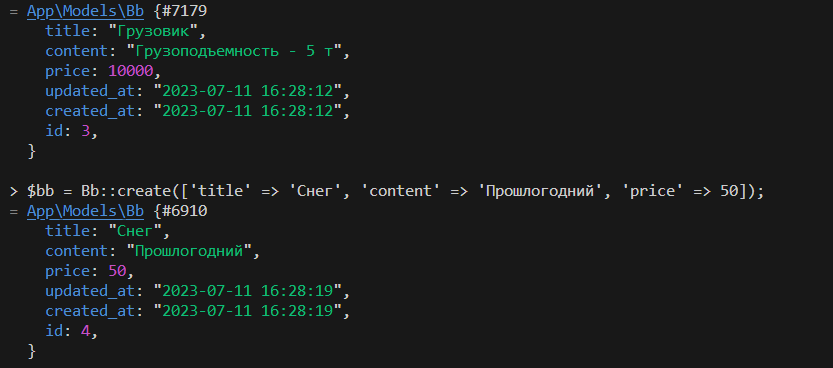


Метод create() создает новую запись, заносит в ее поля значения из указанного ассоциативного массива, сохраняет запись и возвращает ее в качестве результата. Он использует массовое присваивание, при котором значения заносятся сразу в несколько полей создаваемой записи. Отметим, что таким образом можно занести значения только в поля, перечисленные в списке доступных для массового присваивания

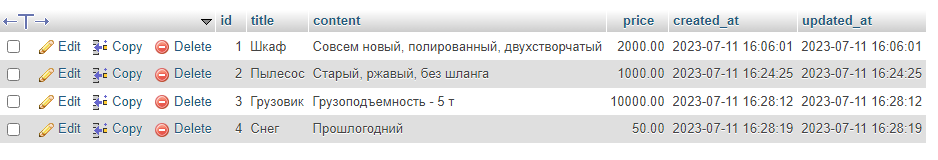
Добавим еще 2 записи:

$bb = Bb::create(['title' => 'Грузовик', 'content' => 'Грузоподъемность - 5 т', 'price' => 10000]);

$bb = Bb::create(['title' => 'Снег', 'content' => 'Прошлогодний', 'price' => 50]);



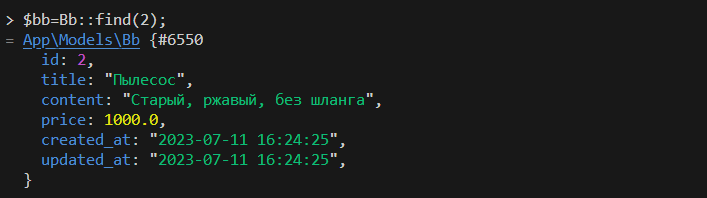
Проверим наличие записей в phpmyadmin:



Методы построителя запроса можно вызывать не только у объекта модели, но и у ее класса — как статические.

Извлечем запись № 2:

$bb=Bb::find(2);



Метод find(), опять же, выполняемый построителем запросов, ищет и возвращает объект модели, хранящий запись с заданным ключом

Тысяча гривен за старый пылесос без шланга — не многовато ли?.. Уценим его.

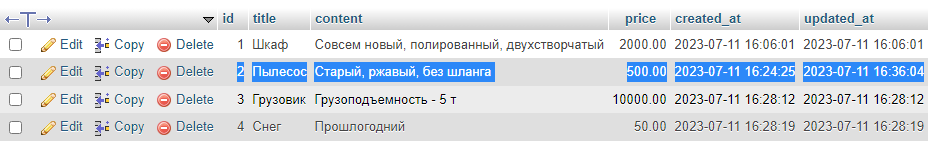
Изменим цену товара (значение поля price):

$bb->price = 500;

$bb->save();



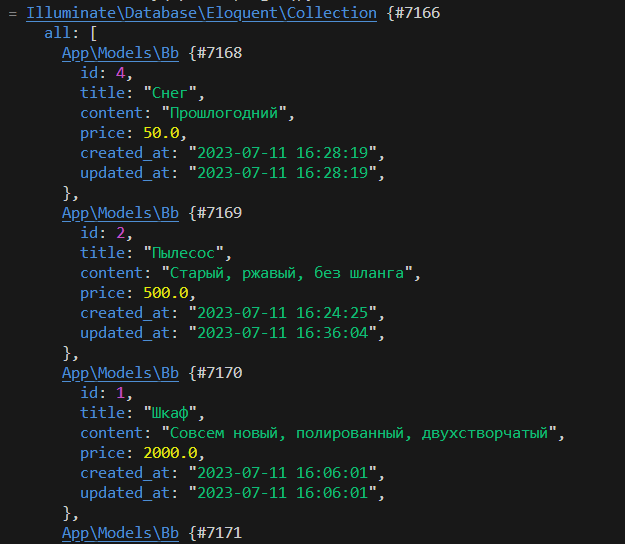
Проверим результат в бд:



Мы занесли новое значение в нужное свойство модели и вызвали у объекта записи метод save().

Извлечем все объявления, отсортированные по увеличению цены:

$bbs=Bb::orderBy('price')->get();



Метод orderBy() построителя запросов сортирует записи по указанному полю.

В качестве результата он возвращает тот же объект построителя запросов, что позволяет записывать «цепочки» методов.

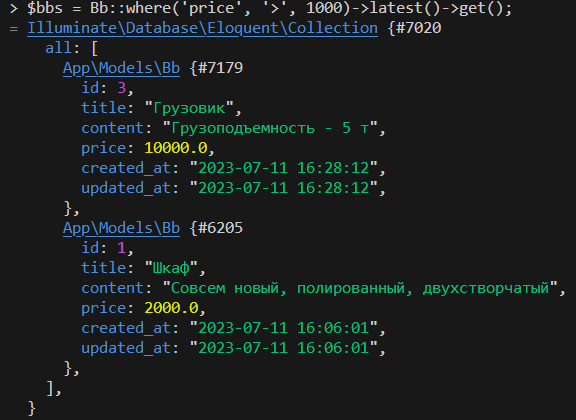
Мы и так сделали, «сцепив» с методом orderBy() метод get(), который отправит базе данных готовый SQL-запрос и вернет результат его выполнения в виде

коллекции объектов модели Bb, хранящих выбранные записи.

Коллекцию записей, возвращенную методом get(), как и любую другую, можно перебрать в цикле.

Извлечем объявления с ценами более 1000 рублей и выведем их в обратном хронологическом порядке:

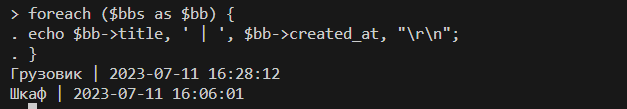
$bbs = Bb::where('price', '>', 1000)->latest()->get();



foreach ($bbs as $bb) {

echo $bb->title, ' | ', $bb->created\_at, "\r\n";

}

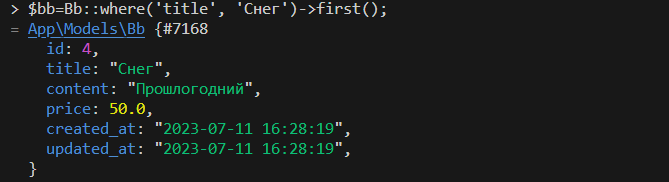


Метод where() построителя запросов фильтрует записи по значению поля, указанного в первом параметре. Второй параметр задает SQL-оператор сравнения, а третий — сравниваемое значение.

Метод latest() сортирует записи по убыванию значения поля отметки создания (т. е. в обратном хронологическом порядке). Знакомый нам метод get() выполнит запрос и вернет результат.

Удалим объявление о продаже прошлогоднего снега (вряд ли на него будет спрос):

$bb=Bb::where('title', 'Снег')->first();



$bb->delete();



Ищем объявление вызовом метода where() (поскольку оператор сравнения в нем не указан, будет использован оператор равенства =). Метод first() возвращает первую запись из полученного результата. А метод delete() модели удаляет текущую запись.

Откроем модуль app\Http\Controllers\BbsController.php, хранящий код контроллера

BbsController, в текстовом редакторе и переделаем действие index(), чтобы оно

выводило перечень объявлений, взятых из базы данных, в обратном хронологическом порядке:

<?php

namespace App\Http\Controllers;

use App\Models\Bb;

use Illuminate\Http\Request;

class BbsController extends Controller

{

    public function index()

    {

        $bbs = Bb::latest()->get();

        $s = "Объявления<hr>";

        foreach ($bbs as $bb) {

            $s .= $bb->title . "<br>";

            $s .= $bb->price . " грн.<br>";

            $s .= "<br>";

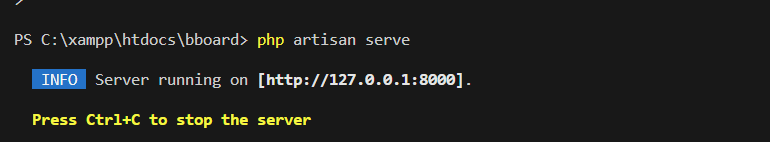
        }

        return response($s);

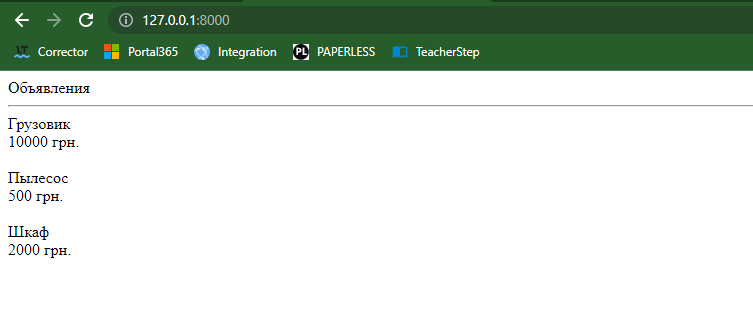
    }

}

Не забываем все сохранить и запустить сервер:



Проверяем результат:



Материалы к уроку:

1. <https://laravel.com/>
2. Book <https://drive.google.com/file/d/1LAvxZpOiQ9rsw2KJAvgEoWvGsJJYLsKL/view?usp=sharing>

Домашнее задание:

1. Настроить рабочую среду для Laravel
2. Пройти все практические задачи в этом документе (выслать photo в mystat)
3. Добавьте 10 различных товаров в проект и покажите соответствующие photo в mystat через tinker
4. В разделе материалы к уроку есть ссылка на оф. Сайт фреймворка Laravel, Ваша задача ознакомится с документацией. Показать несколько photo в mystat доказательства факта прочтения.
5. Создайте новый Laravel проект, добавить пользователей поля придумать самостоятельно, сделать миграции в бд. Добавить новых, удалить и обновить. Для добавления используйте Laravel console CRUD (Проект поместить в архив и отправить в mystat) Либо Tinker, blade страницы

