Laravel new app – rozpoczęcie nowego projektu

Jeśli tworzysz nowy projekt i z góry zakładasz że autentykacja będzie potrzebna to stwórz nowy projekt poleceniem **laravel new nazwaProjektu –auth** dzięki temu na starcie zainstaluje się moduł, a szablony widoku nie rozsypią się.

Php artisan serve – tworzy serwer www do developingu, zanm wykonamy komendę trzeba przejść do katalogu aplikacji np.: cd c:/laravel/app

Artisan dodaje do systemu kilka komend: php artisan – lista komend

Composer.json – plik konfiguracyjny  
composer.lock – plik zawiera informacje o zainstalowanych aktualnie pakietach i ich wersjach

**Dodawanie pakietu:**composer require [nazwa pakietu] np. composer require moneyphp/money ‘’^3.1’’ <- w cudzysłowiu numer wersji którą chcemy zainstalować, daszek oznacza najnowszą wersję z gałęzi 3.x

Dodać pakiet można też ręcznie, dopisując go do pliku composer.json a później wykonując polecenie composer update

**Usuwanie pakietu:  
c**omposer remove [nazwa pakietu]

**Composer update** – composer przechodzi po pakietach i sprawdza czy nie pojawiła się nowsza wersja każdego z pakietów, można wykonać to polecenie globalnie, ale można też aktualizować tylko jeden z pakietów np.: composer update moneyphp/money. Dane brane są z composer.json, na koniec aktualizuje composer.lock

**Composer install -**  dane brane są z composer.lock i instalowane są konkretne wersje

Katalog vendor – katalog w którym są ściągnięte wszystkie biblioteki, można go wywalić a później wklepać composer install

**Katalogi w projekcie lavela:**

**App** – główny katalog aplikacji:

* Console – tutaj tworzone będą nasze polecenia konsolowe wykonywane przez artisana np. jeśli tworzymy nową komendę, to domyślnie w tym katalogu ją umiesczamy.  
  Kernel.php – określa przykładowo gdzie system ma szukać komend
* Exceptions – dobre miejsce do umieszczania wyjątków. Plik handler – możemy zdefiniować domyślną obsługę wyjątków
* http – pliki i klasy powiązane z obsługą requestów http
* providers – pliki odpowiedzialne za wczytywanie i konfigurację modułów frameworka

Uwaga, domyślnie laravel nie wprowadza strict types (ścisła kontrola typów), ale warto to dodać samemu!

**Katalog vendor** – nie powinniśmy wprowadzać żadnych modyfikacji w tym katalogu!

**Katalog config**: sekcja konfiguracyjna przeznaczona dla nas

* app.php – nazwa aplikacji, domyślne środowisko, tryb, strefa czasowa itp. Konfiguracja jest przetrzymywana jako tablica, łączy wiele plików (pozostałe pliki z folderu)
* jeśli potrzebujemy jakąś nową konfigurację to tworzymy nowy plik, w nim tablicę ze strukturą jak pozostałe pliki, wtedy ten plik zostanie wczytany automatycznie

**katalog database** – kod odpowiedzialny za zarządzanie bazą danych

* migration – sekcja która odpowiada za modyfikacje struktury bazy danych, dodaje ważne elementy do danych w bazie. Można je wykonać albo zrollbackować czyli cofnąć smiany
* seeds – klasy które są odpowiedzialne za uzupełnienie bazy danymi
* factories – wygenerowanie zbioru danych

**katalog public** – document root na serwerze. Tutaj są pliki widoczne dla użytkownika. Tutaj jest index.php który jest bramą do naszego systemu

**katalog resources-**  część niewidoczna dla użytkownika, odpowiedzialna głównie za część widokową

**katalog routes –** route- mówi frameworkowi jaki kod ma się wykonać, gdy przyjdzie jakiś request http

**katalog storage -**  worek na różne typy danych, np. cache frameworka, możemy wrzucać tam jakieś graiki które chcemy wykorzystać na stronie, ale nie chcemy by użytkownik miał do nich dostęp, a dostęp do nich regulować np. w kontrolerze

**Oprócz katalogów są też pliki:**

**Package.json, package.lock -**alternatwa dla composera, ale dotyczy pakietów javascriptowych

**Webpack.mix.js –** kod odpowiedzialny za przetworzenie tego, co znajduje się w resources – część frontendowa, i umieszczenie tego w katalogu public

**.env –** konfiguracja naszego środowiska, te wszystkie zmienne nazywamy zmiennymi środowiskowymi

**.env.example** – szablon konfiguracji środowiska

Możemy sobie utworzyć kopię pliku .env, zmienić jej nazwę na .env.testing i wtedy:   
jeśli wchodzimy z url’a to odpalany jest domyślnie plik .env, plik .end.testing jest uruchamiany kiedy odpalamy testy unitowe. Te zmienne zastępują zmienne z pliku .env w przypadku testów

Zasada jest taka: plik .env nie jest podzielony pomiędzy użytkownikami, nie powinniśmy wrzucać go do repozytorium, możemy wrzucić .env.example. Jeżeli instalujemy projekt, np. przychodzi nowa osoba do projektu, zaciąga sobie projekt np. z githuba i przykładowo zmienia nazwę pliku .env.example na .env albo kopiuje sobie jego zawartość.

**composer require barryvdh/laravel-debugbar --dev** - fajny debug bar, flaga - - dev (minus minus dev) oznacza, że pakiet trafi do sekcji require dev w comsposer.json

**laravel dostarcza dwie użyteczne funkcje (tylko do developingu):**

**dump();** - printuje tekst i kontynuuje działanie strony

**dd();** - działa w taki sam sposób jak dump plus zatrzymuje działanie strony

php artisan route:list – wyświetla routing

możemy też klikąć F1 w visual studio code i wpisać artisan route:list – wtedy jest lepiej widoczny routing

middleware – to część kodu, która dodaje nam pewną funkcjonalność, zanim wykonamy kod odpowiedzialny za obsługę żądania

**Routes -> web.php :**

Route::get('/hello/{name}', 'HelloController@hello');

Route::get('/goodbye/{name}', function(string $name) {

return 'Goodbye: '. $name;

});

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

return 'jestem GET';

});

Route::get('example1', fn() => 'jestem arrow GET');

Route::match(['get', 'post'], '/match', function() {

});

Jeżeli ma się wykonać ta sama logika dla dwóch metod to możemy połączyć w match

Route::any('/all', fn() => 'wszystkie metody');

Wszystkie metody powyżej

Route::view('/view/route', 'route.view');

Szybki route, przekierowuje od razu do wskazanej strony, bez zbędnej logiki. Przydatne, gdy chcemy wyświetlić jakąś statyczną podstronę. Jest opcja przekazania parametru do widoku:

Route::view('/view/route/var1', 'route.viewParam', ['param1' => 'var1- to nasza dana']);

Ustawienie opcjonalnego parametru:

Route::get('users/{nick?}', function(string $nick = null){

    dd($nick);

});

Filtrowanie parametrów: (nick musi zawierać małe litery i mieć przynajmniej jeden znak, oraz lub cyfry)

Route::get('users/{nick}', function(string $nick){

    dd($nick);

})->where(['nick' => '[a-z0-9]+']);

**Nazywanie routy:**

Route::get('items', function(){

    return 'items';

})->name('shop.items');

Route::get('example', function(){

    $url = route('shop.items');

    dump($url);

})->name('shop.items');

**To samo tylko z przekazaniem parametrów:**

Route::get('elements/{id}', function(int $id){

    return 'Element: '. $id;

})->name('shop.item.single');

Route::get('example', function(){

    $url = route('shop.item.single', ['id' => 44]);

    dump($url);

})->name('shop.items');

 public function \_\_invoke(int $id)

\_\_invoke – to magiczna funkcja, która aktywuje się, gdy używamy naszego obiektu jak funkcji np.:

$obj = new ShowAdress();

$obj();

**php artisan make:controller TestController –** komenda artisana do tworzenia kontrolera

**php artisan make:controller test\testController –invokable** – tworzy kontroler Single Action z funkcją \_\_invoke

**Korzystanie z obiektu request w kontrolerze:**

use Illuminate\Http\Request;

    public function testShow(Request $request, int $id)

    {

        dd($id);

    }

Pierwszym argumentem musi być okiekt request!

**Funkcjonalności obiektu Request:**

$uri=$request->path();

$url=$request->url();

$fullUrl=$request->fullUrl();

dump($uri, $url, $fullUrl);

Output:

"**users/test/5**"

"**http://127.0.0.1:8000/users/test/5**"

"**http://127.0.0.1:8000/users/test/5**"

        $httpMethod=$request->method();

        dump($httpMethod);

Output:

"**GET**"

Kolejna metoda:

      if($request->isMethod('post'))

        {

            dump('to jest post');

        }

  //http://127.0.0.1:8000/users/test/5?name=Tom&nick=boss

        $all = $request->all();

        dd($all);

output:

array:2 [▼

"name" => "**Tom**"

"nick" => "**boss**"

]

$name=$request->input('name');

dd($name);

W przypadku powyżej nie obchodzi nas jaką metodą spłynęły do nas dane, czy to gettem, postem, putem itp. Po prostu wyciągamy dane.

[127.0.0.1:8000/users/test/5?name=Tom&nick=boss&games[]=quake&games[]=turok](http://127.0.0.1:8000/users/test/5?name=Tom&nick=boss&games%5b%5d=quake&games%5b%5d=turok)

 $game=$request->input('games.1');

W nawiasie wpisujemy nazwę tablicy a po kropce numer indeksu

Zagnieżdżanie tablic w url’u:

[http://127.0.0.1:8000/users/test/5?name=Tom&nick=boss&games[]=quake&games[][name]=turok](http://127.0.0.1:8000/users/test/5?name=Tom&nick=boss&games%5b%5d=quake&games%5b%5d%5bname%5d=turok)

$game=$request->input('games.1.name');

Używając metody input – nie ma znaczenia jaką metodą użytkownik wysłał do nas dane get, post, put…

$name=$request->query('name', ‘Nowak’); //druki param to wart. domyślna

dd($name);

Wyciągnięcie wartości true|false z url’a :

 $expired=$request->boolean('expired');

 dd($expired);

Sprawdzenie czy jest dany parametr: umożliwia wykonanie jakiejś logiki jeśli…

$hasName=$request->has('name');

        dd($hasName);

  $hasParam=$request->has(['name', 'nick']);

        dd($hasName);

wartość true zostanie zwrócowna tylko jeśli obydwa parametry zostaną przesłane

 $hasAnyParam=$request->hasAny(['name', 'nick']);

        dd($hasAnyParam);

to samo, tylko wystarczy jeden warunek spełniony

**Pobieranie ciasteczek z requestu:**

$cookies=$request->cookie(); //możemy wpisać nazwę ciasteczka w nawiasach

dd($cookies);

**RESPONSE:**

W UserController.php w funkcji np. testShow możemy zwracać użytkownikowi:

* zwykły text:  
  return „To jest zwykły tekst konwertowany przez framework na http response User: $id”;
* Obiekt response:
* return response (
* "<h3>To jest obiekt response: User $id</h3>",   //content
* 200,                        //status
* ['Content-Type' => 'text/html']
* );

* Inny sposób tzw chain:
* return response ("<h3>To jest obiekt response: User $id</h3>")
* ->setStatusCode(200)
* ->header('Content-Type', 'text/html')
* ->header('Own-Header', 'Laravel'); //własny nagłówek
* Mix obydwu podejść:

return response("<h3>To jest obiekt response: User $id</h3>", 200)

        ->header('Content-Type', 'text/html')

        ->cookie('my\_best\_cookie', 'brownie', 10); //czas w minutach

**REDIRECT:**

Zwykłe przekierowanie na stronę: ten string to adres po domienie do którego przekierowujemy

 return redirect('users');

redirect po nazwie routy:

        return redirect()->route('get.users');

Jeśli nasza trasa wymaga podania parametrów:

return redirect()->route('get.users.adress', ['id' => $id]);

Przekierowanie do kontrolera:

        return redirect()->action('UserController@list');

gdy kontroler wymaga argumentów:

 return redirect()->action('user\ShowAdress', ['id' => $id]);

przekierowanie do obcej strony:

    return redirect()->away('https://mentorzy.it');

**Zwracanie widoków:**

  return response()

        ->view('user.profil');

//albo inaczej:

Return view('user.profil');

Jeżeli będziemy potrzebowali argumentów: to wrzucamy je w tablicę

  return response()

        ->view('user.profil', ['id' => $id], 200);

Jeżeli chcemy dorzucić nagłówki:

        return response()

        ->view('user.profil', ['id' => $id], 200)

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

Zwracanie danych w formie dżejsonowej:

 return response() ->json(['id' => $id]);

**Widoki:**

**Dzielenie zmiennych pomiędzy widokami:**

Wchpdzimy do app->service->providers->AppServiceProvider.php

**Wpółdzielone zmienne widokowe wrzucamy do metody boot() za pomocą konstrukcji:**

view()->share('key', $value);

albo w ten sposób:

use Illuminate\Support\Facades\View;

View::share('applicationName', 'Laravel Games2');

**Optymalizacja**:

Domyślnie szablony są kompilowane „On Demand” tzn podczas wczytywania strony. Artisan ma komendę, która kompiluje wszystkie szablony i zapisuje je do cacha.

Jeśli chodzi o developowanie to zostawiamy tak jak jest. Jednak w środowisku produkcyjnym to wartko odpalić komendę która skompiluje i scachuje wszystkie widoki

**Php artisan view:cache**

Żeby usunąc wszystkie scachowane widoki wpisujemy: **php artisan view:clear**

**BLADE:**

Definiowanie sekcji:

<div class="sidebar">

            @section('sidebar')

            <ul>

                <li><a href="#">...</a></li>

            </ul>

            @show

        </div>

<div class="container">

            @yield('content')

        </div>

Jeżeli gdzieś w szablonie została zdefiniowana sekcja ‘content’ to zostanie ona wyświetlona w polu yield

Rozszerzanie głównego szablonu:

@extends('layout.main')

@section('content')

<h1>Application: {{ $applicationName }}</h1>

<div>

    id: {{$userId}}

</div>

<hr>

@endsection

@show – to tak jakby jednocześnie zakończyć sekcję i ją wyświetlić (endsection i yelt)

 @yield('title', $applicationName)

Jeśli nie ma zdefiniowanej sekcji title, to użyj tej wartości domyślnej

Wyświetlanie danych w blade:

public function show(int $userId)

    {

        $faker = Factory::create();

        $user= [

            'id' => $userId,

            'name' => $faker->name,

            'firstName' => $faker->firstName,

            'lastName' => $faker->lastName,

            'city' => $faker->city,

            'age' => $faker->numberBetween(12, 40),

            'html' => '<b>Bold HTML</b>'

        ];

        return view(

            'user.show',

            [

                'user' => $user

            ]

        );

    }

<h3>Informacje o użytkowniku: </h3>

    <ul>

        <li>Id: {{ $user['id'] }}</li>

        <li>imię: {{ $user['firstName'] }}</li>

        <li>nazwisko: {{ $user['lastName'] }}</li>

        <li>miasto: : {{ $user['city'] }}</li>

    </ul>

Wyświetlanie zawartości HTML w blade, bez escapowania:

{!! $user['html'] !!}

Instukcje warunkowe w blade:

@if ($user['age'] >= 18)

<div> Osoba dorosła</div>

@elseif($user['age'] >= 16)

<div>prawie dorosła</div>

@else

<div>nastolatek</div>

@endif

 @isset($nick)

        Nick: xxx

@else

        ISSET -false

@endisset

@auth

            Informuje nas czy użytkownik jest zalogowany

@endauth

@guest

            Użytkownik nie jest zalogowany

@endguest

Blade pętle:

FOREACH:

 @foreach ($users as $user)

                  <tr>

                    <td>{{ $user['id'] }}</td>

                    <td>{{ $user['name'] }}</td>

                    <td>

                        <a href="

                            {{

                                route('get.user.show',

                                ['userId' => $user['id']

                                ])

                            }}"

                        >

                        Szczegóły</a>

                    </td>

                </tr>

              @endforeach

FOR:

 @for ($i = 0; $i < count($users); $i++)

                  <tr>

                      <td>{{ $users[$i]['id'] }}</td>

                      <td>{{ $users[$i]['name'] }}</td>

                      <td>

                        <a href="

                        {{

                            route('get.user.show',

                            ['userId' => $users[$i]['id']

                            ])

                        }}"

                    >

                    Szczegóły</a>

                      </td>

                  </tr>

              @endfor

FORELSE:

 @forelse ($users as $user)

              <tr>

                    <td>{{ $user['id'] }}</td>

                    <td>{{ $user['name'] }}</td>

                    <td>

                        <a href="

                            {{

                                route('get.user.show',

                                ['userId' => $user['id']

                                ])

                            }}"

                        >

                        Szczegóły</a>

                    </td>

                </tr>

              @empty

              <tr>

                  <td colspan="3">Lista jest pusta</td>

              </tr>

              @endforelse

Jeśli dana lista nie zawiera elementów to przechodzimy do @empty

WHILE:

<tr><td colspan="3">WHILE</td></tr>

              @php

                  $j=0;

              @endphp

              @while ($j < count($users))

                <tr>

                    <td>{{ $users[$j]['id'] }}</td>

                    <td>{{ $users[$j]['name'] }}</td>

                    <td>

                    <a href="

                    {{

                        route('get.user.show',

                        ['userId' => $users[$j]['id']

                        ])

                    }}"

                >

                Szczegóły</a>

                    </td>

                </tr>

              @php

                  $j++;

              @endphp

              @endwhile

BREAK i CONTINUE:

  @if($i == 1)

      @break

   @endif

Skrócony zapis:

@break($i==1)

@if($i == 1)

    @continue

@endif

Skrócony zapis:

  @continue($i==1)

 @foreach ($users as $user)

            <tr>

                <td>{{ $loop->index }} </td>

                <td>{{ $loop ->iteration }}</td>

@endforeach

Index – index z tabeli po której przechodzimy  
iteration – iteracja po tabeli

@if($loop->first)

            <tr>

                <td>FIRST</td>

            </tr>

@endif

Jest dużo więcej opcji obiektu loop np. last itp. Poczytaj o tym

 @each('view.name', $collection, 'variable', 'view.empty')

I argument – nazwa szablonu który będzie renderowany dla każdego elementu  
II argument – to po czym iterujemy  
III argument – zmienna do której przypisujemy dane danego wiersza, danej iteracji  
IV argument – nazwa szablonu kiedy kolekcja jest pusta

Npm- menedżer pakietów javasrciptowych  
package.json – informacje o zainstalowanych bibliotekach js

**composer require laravel/ui –** instalacja UI Laravela (pozwala integrować laravela z bibliotekami frontentowymi)

**Instalacja Laravelowej obsługi bibliotek UI** (do wyboru: Vue, React, Bootstrap)

**php artisan ui bootstrap**

**Katalog node\_modules** – odpowiednik katalogu vendor w php. W vendor są rzymane biblioteki phpowe, a w node\_modules biblioteki javastciptowe

**Npm run dev –** run odpala skrypt z package.json , odpala laravel mix

**LARAVEL MIX - Webpack** – tzw module handler, potrafi on spakować wiele różnch typów zasobów do jednego zasobu. Np. wiele bibiliotek js do jednego pliku.

Laravel MIX jest nakładką na webpacka, która upraszcza kilka aspektów względem oryginalnego webpacka

Konfiguracja laravel mixa : webpack.mix.js

Dodajemy pliki które chcemy dorzucić: np.:

mix.js('resources/js/app.js', 'public/js')

    .sass('resources/sass/app.scss', 'public/css')

    .sass('resources/sass/admin.scss', 'public/css/admin')

    .sourceMaps();

**Npm run production –** również generuje javastript i cssa oraz pliki mimifikuje: pliki są bardziej zwięzłe, ale mniej czytelne

**Npm run watch –** komenda wychwytuje zmiany w plikach które edytujemy i automatycznie przegenerowuje pliku publiczne.

Dołączenie plików js i css do projektu:

<link rel="stylesheet" href="{{ mix(('css/app.css')) }}"/>

<script src="{{ mix('js/app.js') }}"></script>

Webpack i JS:

Pliki, moduły JS znajdujące się w katalogu resources nie są dosępne z poziomu naszej strony. Webpack kompiluje te pliki w jeden, mimifikuje i wrzuca potem do public/js.

Webpack obsługuje tzw pliki vanilla js, czyli pliki dostępne od razu w katalogu resources, je także możemy przepuścić przez WebPacka:

mix.js('resources/js/app.js', 'public/js')

    .scripts([

        'public/js/admin.js',

        'public/js/admin2.js',

    ],'public/js/all.js') //ten drugi parametr to plik wynikowy

Mix umożliwia wersjonowanie plików: pomaga to przy wymuszeniu przeglądarki do ponownego pobrania plików składowych, a nie korzystania z plików z cache przeglądarki:

mix.js('resources/js/app.js', 'public/js')

    .sass('resources/sass/app.scss', 'public/css')

    .sass('resources/sass/admin.scss', 'public/css/admin')

    .sourceMaps()

    .version();

Tylko pamiętaj że trzeba przebudować pliki strony np. npm run dev

Dodawanie wersji pliku na środowisku developerskim nie ma sensu!  
można zrobić coś takiego:

    if (mix.inProduction())

    {

        mix.version();

    } //to doda nam tokeny do plików tylko na produkcji, więc spoko

**BROWSER SYNC:**

mix.browserSync('127.0.0.1:8000') //tutaj wpisujemy domenę i port

potem robimy **npm run watch**, za pierwszym razem doinstalują się nam biblioteki, i dostaniemy informację pod jakim adresem i portem mamy browsersynca

za każdym razem gdy ściągniesz kod strony z udemy, odpal composer install i npm install

**KONFIGURACJA APLIKACJI:**

Konfiguracja: katalog config

Zmienne środowiskowe umieszczamy w pliku .env **Pliku .env nie wrzucamy do repozytorium!  
wrzucamy go do .gitignore**

**Zasady pracy ze zmiennymi środowiskowymi:**

* **Wszystkie dane w pliku .env są stringami**
* **Nazwy zmiennych środowiskowych mają konwencję stałych w php**
* **Zmienne środowiskowe odczytywane są za pomocą funkcji env()-** pierwszy argument – nazwa zmiennej środowiskowej, drugi – wartość domyślna (opcjonalne)
* **Używajmy funkcji env() tylko w plikach konfiguracyjnych** (tylko w katalogu config)

Dodawanie własnej zmiennej środowiskowej:

Plik .env wpisujemy swoją zmieną np.:

MY\_TEST\_ENV=true

Klucz z naszą konfiguracją możewy stworzyć w dowolnym pliku konfiguracyjnym, albo można stworzyć nowy plik w katalogu config np.: games.php i robimy:

<?php

return [

    'myTest' => env('MY\_TEST\_ENV', true)

];

Odczyt w dowolnym miejscu za pomocą funkcji config() :

 $myTestEnv = config('games.myTest');

        dd($myTestEnv);

odczytywanie danych konfiguracyjnych:

$config =config();

Dump($config); - zwróci nam tablicę z wszystkimi danymi konfiguracyjnymi

$config =config(‘app.name’); - przypisanie pojedynczego klucza do zmiennej config

**Configi cache’ujemy na środowiskach produkcyjnych, na developerskich niekoniecznie**

**Inne komendy artisan:**

Php artisan down – włączenie maintenance mode, czyli wyłączenie stronki, służy do wdrażania nowej wersji kodu na środowiska produkcyjne

Php artisan up – uruchomienie strony

**SESJE:**

Konfiguracja sesji: config/session.php

Uzyskanie dostępu do obiektu sesji:

* Za pomocą obiektu Request
* Za pomocą helpera dostarczonego przez laravela

Za pomocą request:

$session = $request->session();

Ale musimy mieć wstrzyknięty obiekt Request do funkcji:

use Illuminate\Http\Request;

 public function list(Request $request)

Zapisywanie do sesji:

 $session->put(‘key’, ‘value’);

Np.:

 $session->put('prevAction', \_\_METHOD\_\_ . ':'.time());

Pobranie danych z sesji:

  $prevAction= $request->session()->get('prevAction');

Inne metody:

dump($request->session()->has('test\_tt'));//sprawdza czy klucz istnieje i sprawdza jego wartość (null da false)

dump($request->session()->exists('test\_tt')); //sprawdza czy klucz istnieje

Zapomnienie klucza w sesji:

  $request->session()->forget('test\_tt');

Wyświetlenie szystkich danych sesji:

dump($request->session()->all());

usunięcie wszystkich danych sesji:

    $request->session()->flush();

 $session->flash('flashTestParam', 'Tu byłem');

//zapisze dane, które znikną po jednorazowym ich odczytaniu, albo po następnym requescie

dump($request->session()->get('flashTestParam'));

//zwróci „Tu byłem”, a później wartość zniknie

**Dostęp za pomocą helpera:**

 $prevAction= session('prevAction');

        dump($prevAction);

Zapis:

    session(['test\_11\_1' => 'foo']);

**BAZY DANYCH:**

Ściągnij sobie rozszeszenie: SQLite alexcvzz.vscode-sqlite

W pliku config->database.php ustawiamy sobie jakie połączenie nas interesuje: np. sqlite albo mysql, później w pliku .env wpisujemy to połączenie np.:

DB\_CONNECTION=gameworld

  $db = \DB::connection(); //przyjmuje jako parametr nazwę połączenia, można zastosować wiele połączeń wybierając je np. tak:

$db = \DB::connection('gameworld');

Jeżeli chcemy skorzystać z natywnego PDO to :

   $db = \DB::connection()->getPdo();

**Migracje:**

Database.php w katalogu config:

 'migrations' => 'migrations',

Określa miejsce w którym trzymane są migracje

Uruchomienie migracji: **php artisan migrate**

**Php artisan migrate:status** – status migracji

**Tworzenie nowej migracji:**

**php artisan make:migration create\_games\_table**

php artisan migrate:rollback – cofnięcie migracji

edytowanie własniej migracji: plik z nazwą w katalogu database/migrations

Schema::create('games', function (Blueprint $table) { //tworzenie tabeli

            $table->id(); //tworzenie kolumn tabeli

            $table->timestamps();

        });

Jeżeli chcemy zrobić migrację z innym połączeniem niż domyślnie to robimy tak:

 Schema::connection('mysql')->create('games', function (Blueprint $table) {

            $table->id();

            $table->timestamps();

        });

Usuwanie tabeli:

 Schema::dropIfExists('games'); //usuń jeśli istnieje

Lub:

Schema::drop('games'); //po prostu usuń

Zmiana nazwy tabeli:

Schema::rename('from', 'to');

if (Schema::hasTable('games')) //jeżli dana tabela istnieje to…

        {

            // ....

        }

Dorzucanie kolumn do tabeli:

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

            $table->id();

            $table->timestamps();

        });

 $table->string('title' , 100)->nullable;

Powyżej dodanie nowej kolumny typu varchar o tytule ‘title’, o długości 100, nullable to modyfikator, zezwala na wartość null

Modyfikatory można łączyć, tzn dodawać je po kolei

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

            $table->id();

            $table->string('title' , 100)->nullable;

            $table->text('description')->nullable;

            $table->string('publisher', 100)->comment('game publisher');

            $table->float('score')->nullable;

            $table->timestamps();

        });

Edycja tabeli:

Jeżeli chcemy modyfikować kolumny musimy doinstalować dodatkową paczkę:  
**composer require doctrine/dbal**

**Uwaga ściągnij dbala w wersji** doctrine/dbal": "^2.12 **bo nowsza krzaczy**

class AlterGameTable extends Migration

{

    /\*\*

     \* Run the migrations.

     \*

     \* @return void

     \*/

    public function up()

    {

        Schema::table('games', function (Blueprint $table) {

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

            $table->dropColumn('score');

            //$table->dropColumn(['score', 'title']);

            $table->index('title');

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*

     \* @return void

     \*

    public function down() //to jest cofnięcie zmian które wprowadziliśmy:

    {

        Schema::table('games', function (Blueprint $table) {

            $table->string('title', 100)->change();

            $table->float('score')->nullable();

            $table->dropIndex('games\_title\_index');

        })

    }

}

**Komenda do przywracania zmian:**

php artisan migrate:rollback

**Przełączanie sesji na silnik bazodanowy:**

php artisan session:table

php artisan migrate

**a później przełączamy .env na SESSION\_DRIVER=database**

**Przetrzymywanie w bazie jest szybsze niż przetrzywanie w pliku**

DB::table('users')->truncate(); //czyszczenie bazy

DB::table('genres')->delete(); //to też, tylko nie usuwa klucza głównego czyli np. id zostaje

Dodanie więcej niż jednego rekordu:

DB::table('genres')->insert([

            [

                'name' => 'Advewnenture',

                'created\_at' => Carbon::now(),

                'updated\_at' => Carbon::now()

            ],

            [

                'name' => 'FPS',

                'created\_at' => Carbon::now(),

                'updated\_at' => Carbon::now()

            ],

        ]);

DB::table('genres')->insertOrIgnore(

            [

                [

                    'name' => 'Advewnenture',

                    'created\_at' => Carbon::now(),

                    'updated\_at' => Carbon::now()

                ],

                [

                    'name' => 'FPS',

                    'created\_at' => Carbon::now(),

                    'updated\_at' => Carbon::now()

                ],

            ]

        );

insertOrIgnore różni się tym, że jeśli dany wpis już istnieje to nie dodaje danego rekordu

Automatyczne pobranie id:

$id = DB::table('genres')->insertGetID(

            [

                'name' => 'Sim',

                'created\_at' => Carbon::now(),

                'updated\_at' => Carbon::now(),

            ]

        );

**Aktualizacja:**

DB::table('genres')

        ->where('id', 6)

        ->update(['name' =>'strategy']);

**Usuwanie jednego rekordu:**

 DB::table('genres')

        ->where('id', 6)

        ->delete();

**Seed -**  automatyczne uzupełniane bazy danych wartościami testowymi

php artisan make:seeder GenresSeeder - **Tworzenie nowego seedera**

seeder tworzy się w katalogu: database -> seeds

Później musisz skonfigurować plik DarabaseSeeder.php i dodać swój plik z seedem do funkcji run():

$this->call(GenreSeeder::class);

**Uruchomienie seeda:** php artisan db:seed

**Wywoływanie kilku seedeów:** po prostu wrzucamy klasy do tablicy

 $this->call([

            GenresSeeder::class,

            GamesSeeder::class

        ]);

**(kolejność jest istotna)**

**Odpalenie jednego seeda z palca:**php artisan db:seed --class=GamesSeeder

**QUERY BUILDER: -SELECT**

Pobranie całej zawartości tabeli:

 $games = DB::table('games')->get();

Pobranie części tabeli: za pomocą select:

 $games = DB::table('games')

        ->select('id', 'title', 'score', 'genre\_id')

        ->get();

 public function show(int $gameId)

    {

        $game = DB::table('games')

        ->where('id', $gameId)

        ->get();

        dd($game);

        return view('game.show');

    }

II Sposób

 public function show(int $gameId)

    {

        $game = DB::table('games')

        ->where('id', $gameId)

        ->first();

        dd($game);

        return view('game.show');

    }

III Sposób

 $game = DB::table('games')->find($gameId);

**Funkcje agregujące:**

$maxScore = DB::table('games')->max('score');

$stat = [

                'count' => DB::table('games')->count(),

                'countScoreGtFive' =>DB::table('games')->where('score' , '>', 5)->count(),

                'max' => DB::table('games')->max('score'),

                'min' => DB::table('games')->min('score'),

                'avg' => DB::table('games')->avg('score'),

            ];

**JOIN’y**

->join(nazwa tabeli z którą łączymy', 'warunki połączenia'  )

Np.

 ->join('genres', 'games.genres\_id', '=', 'genres.id')

**Inner join**: zwróć wszystkie rekordy które spełniają warunek

**Left join –** zwraca wszystkie rekordy z tabeli po lewej stronie oraz rekordy z tabeli po prawej stronie, które udało się dopasować na podstawie warunku, a jeżeli nie udało się dopasować to zwraca nulle

**Right join**- to samo tylko na odwrót

Przykład:

 $games = DB::table('games')

            ->join('genres', 'games.genre\_id', '=', 'genres.id')

            ->select(

                'games.id', 'games.title', 'games.score', 'games.genre\_id',

                'genres.id as table\_genre\_id', 'genres.name'

                )

            ->get();

            dd($games->toarray());

ten alias ‘as’ jest potrzebny bo inaczej genres.id nadpisze games.id (bo kolumny mają taką samą nazwę)

**Where:**

->where('score', '>', 90)

**Krótsza forma zapisu (ale tylko jeśli chodzi o równość (=))**

->where('score', 90) //operatorem domyślnie jest =

**Podgląd zapytania do bazy danych:**

 $bestGames = DB::table('games')

            ->join('genres', 'games.genre\_id', '=', 'genres.id')

            ->select(

                'games.id',

                'games.title',

                'games.score',

                'games.genre\_id',

                'genres.id as table\_genre\_id',

                'genres.name as genre\_name'

            )

            ->where('score', '>', 90);

            dd($bestGames->toSql());

**Dodawanie kilku warunków where**: (wrzucamy tablicę tablic)

 $query = DB::table('games')

            ->select('id', 'title', 'score', 'genre\_id')

            ->where(

                [

                    ['score', '>', 90],

                    ['id', '=', 55]

                ]

            );

            dd($query->get());

**Pomiędzy warunkami jest AND** (oba warunki muszą być spełnione) !

**Łączenie warunków za pomocą OR:**

$query = DB::table('games')

            ->select('id', 'title', 'score', 'genre\_id')

            ->where( 'score', '>', 95)

            ->orWhere('id', 55); //metoda orWhere!

            dd($query->get());

**whereIn:**

        $query = DB::table('games')

            ->select('id', 'title', 'score', 'genre\_id')

            ->whereIn( 'id', [22,42,53]);

            dd($query->get());

//weź te wartości z tablicy isprawdź czy znajdują się w tabeli

Są jeszcze inne: whereBetween itd

**Sortowanie:**

->orderBy('score', 'desc')

**Grupowanie:**

->groupBy('score')

 ->select(DB::raw('count(\*)'), 'score') //wrzucenie polecenia SQL do selecta

 ->groupBy('score')

Plus alias:

 ->select(DB::raw('count(\*) as count'), 'score')

**Having: operuje na wartościach zwróconych z zapytania:**

$scoreStats=DB::table('games')

        ->select(DB::raw('count(\*) as count'), 'score')

        ->groupBy('score')

        ->having('count', '>', 10 ) //tego nie zrobimy za pomocą WHERE

        ->get();

**Limitowanie danych:**

 ->limit(5)

 ->offset(2)

**Paginacja:**

Zamiast metody -> get () wpisujemy ->paginate([limit wyników])

->paginate(10);

Następnie w widoku dodajemy:

 {{ $games ->links() }}

Jest jeszcze inna metoda:

 ->simplePaginate(10); //tylko przyciski na stronie

Paginacja dostarczana przez framework nie działa z grupowaniem! (ale obsługuje sortowanie)

**Własne stylowanie paginatora w sekcji widoku:**

Aby wczytać dodatkowe szablony paginatora, musimy je najpierw opublikować komendą:

*php artisan vendor:publish --tag=laravel-pagination*

i te szablony pojawiają się w : [\resources\views\vendor\pagination]

możemy wybrać jeden z nich albo je zmieniać itp.

**ELOQUENT**

Eloquent – ORM – dzięki niemu tworzymy klasy i obiekty klas które reprezentują dane w bazie danych

**Grupowanie Routes:**

Po staremu:

Route::get('b/games/dashboard', 'GameController@dashboard')

        ->name('games.dashboard');

    Route::get('b/games', 'GameController@index')

        ->name('games.list');

    Route::get('b/games/{game}', 'GameController@show')

        ->name('games.show');

Po grupowaniu:

Route::group(['prefix' => 'b/games'], function () {

    Route::get('dashboard', 'GameController@dashboard')

        ->name('games.dashboard');

    Route::get('', 'GameController@index')

        ->name('games.list');

    Route::get('{game}', 'GameController@show')

        ->name('games.show');

});

Plus prefix do nazwy kontrolera:

Route::group(['prefix' => 'b/games', 'namespace' => 'Game'], function () {

    Route::get('dashboard', 'BuilderController@dashboard')

        ->name('games.dashboard');

    Route::get('', 'BuilderController@index')

        ->name('games.list');

    Route::get('{game}', 'BuilderController@show')

        ->name('games.show');

});

Route::group([

    'prefix' => 'b/games',

    'namespace' => 'Game',

    'as' => 'games.b.' //prefix do nazwy routy

], function () {

    Route::get('dashboard', 'BuilderController@dashboard')

        ->name('dashboard'); //to jest nazwa routy

    Route::get('', 'BuilderController@index')

        ->name('list');

    Route::get('{game}', 'BuilderController@show')

        ->name('show');

});

**Tworzenie nowego modelu:**

**php artisan make:model game**

**php artisan make:model Model\Game** //utworzenie z właściwym namespacem i we właściwym katalogu

**Eloquent zmienia sobie nazwę klasy na liczbę mnogą (czyli z game robi games) i szuka takiej tabeli w bazie danych!**

**Jeżeli nasza naszej tabeli jest inna niż nazwa modelu to możemy ręcznie określić nazwę tabeli za pomocą właściwości:**

protected $table = "application\_game";

protected $primaryKey = '';

**tym ustawiamy nazwę klucza głównego w tabeli, domyślnie jest to id**

**Eloquent domyślnie zakłada, że w naszej tabeli mamy kolumny: „created\_at” oraz „updated\_at”.** Jeśli nie mamy i nie chcemy mieć tych kolumn to musimy ustawić właściwość:

protected $timestamps= false;

**Ustawienie domyślnych wartości dla kolumn:**

  protected $attributes = [

        'score' => 5,

    ];

**Implementacja Eloquent:**

use App\Model\Game;

 $games = Game::all();

        dd($games);

Możemy definiować warunki tak jak w Builderze:

    $games = Game::where('score', '>', 9)

        ->get();

        dd($games);

$stats = [

            'count' => Game::count(),

            'countScoreGtSeven' => Game::where('score', '>', 7)->count(),

            'max' => Game::max('score'),

            'min' => Game::min('score'),

            'avg' => Game::avg('score'),

        ];

**Pobranie pojedynczego rekordu: 4 sposoby:**

     $game = Game::find($gameId);

$game = Game::findOrFail($gameId);

     $game = Game::where('id' , $gameId) ->first();

     $game = Game::fistWhere('id' , $gameId);

**Inny przykład:**

        $games = Game::orderBy('created\_at')

        ->paginate(10);

**Relacje**

Najpierw nazywamy relację (nazywamy funkcję)

Definiujemy relację np. jeden do wielu

class Genre extends Model

{

   public function games()

   {

       return $this->hasMany('App\Model\Game');

       //return $this->hasMany(Game::class);   //to jest to samo

   }

}

Eloquent bierze nazwę klasy w której definiujemy relację, zmienia ją na snake\_case i dodaje sobie do tej nazwu surfix \_id. Czyli w powyższym przykładzie z klasy Genre zrobi sobie genre\_id

W kolumnie genre\_id znajduje się klucz obcy, który jest powiązany z kluczem głównym w kolumnie id (domyślnie) w tabeli genres

Jeżeli chcemy sami wskazać kolumnę z kluczem obcym i z głównym to:

return $this->hasMany(

           Game::class,

           'foreign\_key',

           'id'

        );

Następnie robimy relację w drugim modelu:

    public function genre()

    {

        return $this->belongsTo(Genre::class);

    }

}

Możemy tak jak poprzednio nadpisać skąd brać klucz główny i obcy:

return $this->belongsTo(Genre::class, 'foreign\_key', 'primary\_key');

dostanie się do danych:

 <td>{{ $game ->genre -> name }}</td>

Genre to nazwa relacji (public function genre()), name to nazwa kolumny)

**Iger loading: zachłanne pobieranie**

        $games = Game::with('genre')

        ->orderBy('created\_at')

        ->paginate(10

Dzięki temu zyskujemy na wydajności, program nie generuje kilkunastu zapytań sql, tylko pobiera więcej danych wcześniej

Możemy pobrać więcej niż jedną relację:

        $games = Game::with(['genre', 'movies'])

  …

**SCOPE** – METODA W MODELU W KTÓREJ ZAMYKAMY WARUNKI UŻYWANE DO WYCIĄGANIA DANYCH Z BAZY DANYCH

    public function scopeBest(Builder $query):Builder

    {

        return  $query->where('score', '>=', 9);

    }

 $bestGames = Game::with('genre')

        ->best()

        ->get();

Jest jeszcze global scope ale nie chce mi się go tu opisywać…

**Przekazywanie danych przez Eloquent:**

       $newGame = new Game();

        $newGame->title = 'Tomb Raider';

        $newGame->description = 'Przygoda, skarby itp';

        $newGame->score = 9;

        $newGame->RandomParameter = 'Edios';

        $newGame->publisher\_id = 5;

        $newGame->genre\_id = 4;

        $newGame->save();

Drugi sposób:

        Game::create([

            'title' => 'Tomb Raider 2',

            'description' => 'Gra...',

            'RandomParameter' => 'Edios',

            'movieId' =>6,

            'score' => 8,

            'publisher\_id' => 5,

            'genre\_id' =>4,

        ]);

**Ale uwaga!**  W modelu musimy wpisać:

protected $fillable =

[

    'title', 'description', 'score', 'publisher\_id', 'genre\_id', 'RandomParameter'

];

Czyli zezwolenie na nadpisywanie kolumn

**Trzeci sposób:**

        $newGame=new Game(

            [

                'title' => 'Tomb Raider 3',

                'description' => 'Gra...',

                'RandomParameter' => 'Edios',

                'movieId' =>6,

                'score' => 8,

                'publisher\_id' => 5,

                'genre\_id' =>4,

            ]);

        $newGame->save();

**Żeby tak zrobić również musimy mieć uzupełnioną właściwość fillable, tak jak w drugim sposobie**

**Aktualizacja danych:**

Najpierw musimy pobrać sobie wpis który chcemy aktualizować

$game = Game::find(115); //tu jest id

        $game->description= 'Opis po aktualizacji';

        $game->save();

**Aktualizowanie wielu wpisów:**

$gameIds= [ 101, 102, 114, 108];

        Game::whereIn('id' , $gameIds)

        ->update([

            'description' => 'Nowy Opis gierki'

        ]);

**Usuwanie danych:**

        $game = Game::find(115); //id wpisu

        $game->delete();

Drugi sposób:

     Game::destroy(114);

Różnica jest taka, że jeśli podamy nieprawidłowy id to pierwszy sposób wywali błąd, a drugi sposób nic nie zrobi 😊

Można też usuwać wiele, podając id po przecinku:

        Game::destroy(114, 113);

Uwaga, te sposoby będą generowały osobne żądanie dla każdego usuwanego wpisu! Rozwiązanie:

Game::whereIn('id', [106, 107]) ->delete();

//wywali te wpisy za pomocą jednego zapytania, najbardziej wydajne

**Middleware**

php artisan make:middleware log

Lokalizacja: app\Http\Middleware

lass RequestLog

{

    /\*\*

     \* Handle an incoming request.

     \*

     \* @param  \Illuminate\Http\Request  $request

     \* @param  \Closure  $next

     \* @return mixed

     \*/

    public function handle($request, Closure $next)

    {

        //kod przed

        $currentDate= Carbon::now();

        $timeStart= microtime(true);

        Log::info("==================================");

        Log::info($currentDate.': BEFORE: '.$timeStart);

        $response = $next($request);

         //kod po

        $timeEnd = microTime(true);

        Log::info($currentDate.': AFTER: '.$timeEnd);

        Log::info($currentDate.': RESULT: '.($timeEnd-$timeStart));

        return $response;

    }

}

Później musimy edytować Kernel.php:

protected $middleware = [

        // \App\Http\Middleware\TrustHosts::class,

        \App\Http\Middleware\TrustProxies::class,

        \Fruitcake\Cors\HandleCors::class,

        \App\Http\Middleware\CheckForMaintenanceMode::class,

        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,

        \App\Http\Middleware\TrimStrings::class,

        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,

**RequestLog::class // <-**

    ];

**AUTENTYKACJA**

php artisan ui bootstrap –-auth

Jeżeli tworzysz nowy projekt i planujesz mieć autentykacje, to od razu możesz stworzyć projekt komendą: laravel new [nazwaProjektu] –auth

Konfiguracja modułu: Config -> auth.php

Guard- w jaki sposób aplikacja ma autentykować

Provider – co ma się dziać po poprawnej autentykacji

**Po każdej zmianie konfiguracji auth.php wklep: composer dump-autoload**

**Zabezpieczenie stron przed niezalogowanym użytkownikiem:**

Wejdź w routing i wszystkie swoje routy zamknij w grupę, następnie do tej grupy dodaj middleware ‘auth’ o tak:

Route::group(['middleware' => ['auth']], function () {

    // GAMES

    Route::group([

        'prefix' => 'b/games',

        'namespace' => 'Game',

        'as' => 'games.b.'

    ], function () { //bla bla bla, dalej jest dalsza część kodu

**Drugi sposób w routingu:**

   Route::middleware(['auth'])->group(function(){

// ten sam kod w środku

    });

**Gdzie przekierować po udanym zalogowaniu?**

Plik Auth->loginController.php właściwość $redirectTo

**Edycja szablonu w zależności od tego czy użytkownik jest zalogowany:**

@auth

@else

@endauth

**Albo na odwrót**

@guest

@else

@endguest

**Pobranie obiektu użytkownika:**

$user = Auth::user();

Drugi sposób, za pomocą obiektu request:

public function \_\_invoke(Request $request)

    {

        $user = $request->user();

        dd($user);

Pobranie samego id:

    $id = Auth::id();

        DD($id);

Sprawdzenie czy użytkownik jest zalogowany:

        if(Auth::check()) {

            dd('jestem zalogowany');

        }

Ale lepszym sposobem jest zastosowanie middlewaru tak jak wcześniej

Wylogowanie użytkownika:

Auth::logout();

Dodawanie nowego pola do rejestracji:

Dodaj kolumnę do tabeli

Edytuj szablon, dodaj w nim pole

W pliku registerController.php popatrz na metody validator i create

Validator to reguły walidacji, druga to tworzenie, dodaj tam swoje pola

Później w Model\User.php dodaj swoje pola do właściwości $fillable

**Kontener**

Przechowuje informacje o wszystkich serwisach i zależnościach w aplikacji

Jeżeli jeden serwis wymaga drugiego, to korzystamy z kontenera. Kiedyś tworzyło się obiekt w i przypisywało go w konstruktorze klasy, teraz już się tak nie robi!

Są trzy metody wstrzykiwania zależności:

- poprzez konstruktor (w większości przypadków, najbardziej popularna):

class Log extends Controller

{

    private AdapterInterface $adapter;

    public function \_\_construct(AdapterInterface $adapter)

    {

        $this->adapter= $adapter;

    }

-bezpośrednio przez metodę np. wstrzykujemy zależność i nigdzie jej nie przypisujemy do właściwości:

    public function show(int $gameId, Request $request): View

-wstrzykiwanie poprzez właściwość

**Providery:**

Provider to dostawca usług, czyli jakieś moduły które będziemy później wykorzystwać w aplikacji np. moduł do logowania

Service Provider to miejsce gdzie te usługi są ładowane, czyli usługa jest ładowana a potem bootowana. Usługa jest wpakowana do kontenera, dzięki czemu jest dostępna w całej aplikacji, łącznie ze swoimi zależnościami.

Lista providrów jest w config/app.php

Providery mają metody register i boot

Register służy do bindowania klas, obiektów z kontenerem, nie należy rejestrować w tej metodzie listenerów ani routów, ani wstrzykiwać innych serwisów

Metoda bind wiąże dany interfejs z jego implementacją

Tworzenie:

php artisan make:provider GameServiceProvider

Kontener jest wstrzyknięty na starcie do właściwości $this->app

Pamiętaj żeby dodać provider w pliku app.php:

App\Providers\GameServiceProvider::class

Bindowanie:

    public function register()

    {

        $this->app->bind(

            GameRepository::class, EloquentGameRepository::class

        ); //pierwszy argument to nazwa interfejsu, drugi to klasa która ma być przypisana do tego interfejsu

    }

Bindowanie za pomocą callback’a:

public function register()

    {

        $this->app->bind(

            FakeService::class,

            function($app) {

                return new FakeService('parametr');

            }

        );

    }

Uwaga: jeśli coś mamy wykonać tylko raz, np. połączenie z bazą danych, to lepiej to zrobić poza bindem, i przerzucić do metody boot tak jak tu:

public function boot()

    {

        dump('Game -Boot');

        $fakeService = $this->app->make(FakeService::class);

        $this->app->bind(

            RepositoryGameRepository::class,

            function ($app) use ($fakeService) {

                dump('Game -Register - bind');

                $gameModel = $app->make(Game::class);

                return new GameRepository($gameModel, $fakeService);

            }

        );

    }

**Singleton:**

Metoda singleton – jeśli nie chcemy żeby za każdym razem zwracany był nowy obiekt. Przykład:

public function register()

    {

        dump('Game -Register');

        $this->app->singleton( // <- o tutaj zamiast bind

            RepositoryGameRepository::class,

            function($app) {

                dump('game -register- bind');

                $gameModel = $app->make(Game::class);

                return new GameRepository($gameModel);

            }

        );

Albo bez callbacka:

    $this->app->singleton(

            RepositoryGameRepository::class, GameRepository::class);

I wtedy zwracana jest ta sama instancja – referencja obiektu

Musimy przeanalizować kiedy warto tworzyć nowy obiekt, a kiedy wystarczy jedna instancja

**Bindowanie kontekstowe:**  wstrzykujemy konkretną implementację w zależności od kontekstu wywołania

 $this->app->when(BuilderController::class)

        ->needs(GameRepository::class)

        ->give(BuilderGameRepository::class);

**Fasady:**

Fasada to statyczny sposób wywołania klas dostępnych w kontenerze zależności

Log::alert('to jest fasada');

Fasady są bardzo zwięzłe, nie trzeba ich wstrzykiwać

**Tworzenie:**

nie ma polecenia artisanowego do tworzenia fasady, musimy sami ręcznie ją utworzyć.

Fasada musi zawierać statyczną funkcję getFacadeAccessor !

<?php

declare(strict\_types=1);

namespace App\Facade;

use App\Repository\GameRepository;

use Illuminate\Support\Facades\Facade;

class Game extends Facade

{

    protected static function getFacadeAccessor()

    {

        return GameRepository::class;

    }

}

A później korzystanie z fasady:

public function index(): View

    {

        Log::alert('to jest fasada');

        return view('game.list', [

            //'games' => $this->gameRepository->allPaginated(10)

            'games' => Game::allPaginated(10) //o tutaj

        ]);

    }

Można też zrobić tak:

class Game extends Facade

{

    protected static function getFacadeAccessor()

    {

        //return GameRepository::class;

        return 'game';

    }

}

A później w serviceProviderze zbindować string ‘game’ do klasy GameRepository:

 $this->app->singleton('game', RepositoryGameRepository::class);

Unikaj fasad na rzecz dependency injection

**Commands**

**Lista wszystkich komend**: php artisan

**Tworzenie komendy**: php artisan make:command LoadSteamGames

Komenda utworzy się w aktalogu app/console/commands

php artisan make:command Steam\LoadGames – komenda utworzy się w katalogu app/console/commands/Steam

$signature: to nazwa komendy np.

protected $signature = 'steam:load-games';

kod który ma się wykonać po wywołaniu komendy znajduje się w metodzie handle()

Dodawanie argumentu do komendy:

protected $signature = 'steam:update-game {game}';

argument opcjonalny:

protected $signature = 'steam:update-game {game?}'; //znak zapytania na końcu

Pobranie argumentu w funkcji handle() :

public function handle()

    {

        $game = $this->argument('game');

        dump($game);

        return 0;

    }

Jeśli chcemy przekazać argument wielowyrazowy to otaczamy go cudzysłowami

**Pytanie do użytkownika:**

 public function handle()

    {

        $game = $this->argument('game');

        dump($game);

        $answer = $this->ask("Czy to Twoja ulubiona gra?");

        dump($answer);

        return 0;

    }

**Confirm w handle():**

        $result = $this->confirm('Czy chcesz zaktualizować?');

        dump($result); //true albo false

**Wypisywanie tekstu w konsoli, metody wbudowane Laravel:**

$this->error('Error ...');

$this->question('Question ...');

        $this->comment('Comment ...');

        $this->info('info ...');

        $this->line('line ...');

Różnią się stylem formatowania w konsoli

http CLIENT

Jest to biblioteka do komunikacji z zewnętrznym API np. Facebooka albo innym, możemy wysyłać pliki na serwer, pobierać pliki z serwera, autoryzacja za pomocą tokenów

Http::get('url', []);

Przykład

 public function handle()

    {

        $response = Http::get('https://postman-echo.com/get', [

            'foo' => 'bar',

            'alpha' => 'omega'

        ]);

        dump($response);

        return 0;

    }

Dostajemy obiekt **Illuminate\Http\Client\Response,** który udostępnia m.in. metody:

/\*

        $resonse->body() : string;

        $resonse->json() : array|mixed;

        $resonse->status() : int;

        $resonse->ok() : bool;

        $resonse->successful() : bool;

        $resonse->failed() : bool;

        $resonse->serverError() : bool;

        $resonse->clientError() : bool;

        $resonse->header($header) : string;

        $resonse->headers() : array;

To było używanie za pomocą fasady, możemy też wstrzyknąć zależność:

private Factory $httpClient;

    public function \_\_construct(Factory $httpClient)

    {

$this->httpClient = $httpClient;

        parent::\_\_construct();

    }

public function handle()

    {

        // $response = Http::get('https://postman-echo.com/get', [

        //     'foo' => 'bar',

        //     'alpha' => 'omega'

        // ]); //to jest fasada

        $response = $this->httpClient->get('https://postman-echo.com/get', [

                'foo' => 'bar',

                'alpha' => 'omega'

            ]); //to jest wstrzyknięcie zależności

        dump($response->json());

System transakcji bazodanowych: jeśli któreś działanie się nie powiedzie to system wycofa wszystkie zmiany wprowadzone w bazie danych

 $result = DB::transaction(function () use ($data) {

}

**Generowanie URL:**

$url = url('path/to/something');

$url = url()->current(); //zwraca bieżący adres na którym jesteśmy

$url = url()->full(); //zwróci pełną ścieżkę z get np. localhost:3000/?foo=bar

$url = url()->previous(); //zwraca poprzedni adres z którego przyszliśmy

 $routeUrl = route('games.show', ['game' => $gameId]); //wygeneruje adres do jakiejś routy

$actionUrl = action([GameController::class, 'dashboard']); //pierwszy parametr to nazwa kontrolera, drugi nazwa akcji //nie polecane bo nazwy kontrolerów często się zmienia

walidacja

 $request->validate(

            [

                'email' => 'required|unique:users|email',

                'name' => 'required|max:2'

            ]

            );

        dd($request->all());

    }

$request->validate kluczem jest nazwa pola które walidujemy, wartość to reguła walidacji

Reguły walidacji są wpisane w laravela, szczegóły w dokumentacji

Alternatywny zapis który daje więcej możliwości:

  [

  'email' => ['required','unique:users','email'],

'name' => ['required','max:2']

    ]

return redirect()

        ->route('me.profile')

        ->with('status','Profil zaktualizowany');

Jeśli walidacja zwraca jakieś błędy to powróci do szablonu formularza i możemy zrobić coś takiego:

@if ($errors->any())

                <div class="alert alert-danger">

                    <ul>

                        @foreach ($errors->all() as $error)

                            <li>{{ $error }}</li>

                        @endforeach

                    </ul>

                </div>

            @endif

Drugi sposób: Pod polem wpisujemy np.:

@error('name')

<div class="invalid-feedback d-block">{{ $message }}

</div>

@enderror

Możemy też dodać klasę do pola:

 class="form-control @error('name') is-invalid @enderror"

**Tworzenie własnej klasy do walidacji:**

**php artisan make:request UpdateUserProfile**

dostęp do walidowanych danych:

$data = $request -> validated();

        dd($data);

**Tworzenie własnych komunikatów błędów:**

public function messages()

    {

        return [

            'email.unique' => 'Podany adres email jest zajęty',

            'name.max' => 'Maksymalna liczba znaków to :max' // <- laravel pobierze sobie max z reguły

        ]; //pierwszy parametr to nazwa reguły walidacji, drugi to konkretna reguła

    }

**Bardziej rozbudowane zasady:**

public function rules()

    {

        $userId = Auth::id();

        return [

            'email' => [

                'required',

                //'unique:users',

                Rule::unique('users')->ignore($userId),

                'email'

            ],

            'name' => 'required|max:2'

        ];

    }

Powyżej zasada odnośnie maila zignoruje zalogowanego użytkownika

**Tworzenie własnej zasady walidacji:**

**php artisan make:rule AlphaSpaces**

później dodajemy swoją zasadę:

'name' => [

                'required',

                'max:50',

                new AlphaSpaces()

            ]

I wtedy w tej zasadzie np.:

public function passes($attribute, $value)

    {

        return (bool) preg\_match('/(^[A-Za-z0-0 ] + $)+/', $value);

    }

Aby nie stracić naszych danych z formularza po nieudanej walidacji, w szablonie robimy coś takiego:

value="{{ old('name', $user->name) }}"

**Zabezpieczenie naszej strony przed podszywającymi formularzami z zewnątrz:**

W dowolnym miejscu w szablonie formularza wpisujemy :

  @csrf

Później włączamy middleware:

\App\Http\Middleware\VerifyCsrfToken::class,

W app/http/kernel.php

I to wszystko

Obsługa Plików

Laravel korzysta z FlySystem do operacji na plikach, konfiguracja w config/flysystems.php

Upublicznienie katalogu np. public: **php artisan storage:link**

dodajemy w formularzu input np.:

<div class="form-group">

                        <label for="avatar">Wybierz avatar...</label>

                        <input type="file" class="form-control-file" id="avatar" name="avatar">

                        @error('avatar')

                        <div class="invalid-feedback d-block">{{ $message }}</div>

                        @enderror

                    </div>

Dodajemy za pomocą migracji kolumnę do bazy danych

Dodajemy walidację np.:

//'avatar' =>'dimensions:min\_width=100, mind\_height=200, max\_width=100,max\_height=200',

            //'avatar' =>'nullable|file|image|dimensions:ratio=3/2',

            'avatar' =>'nullable|file|image',

Przyjmujemy avatar w kontrolerze np.:

 public function update(UpdateUserProfile $request)

    {

        $data= $request->validated();

        $data['avatar']->store('avatars', 'public');

        dd($data);

nadawanie nazwy dla pliku np.:

$path=$data['avatar']->storeAs('avatars',Auth::id(), 'public');

**Tworzenie nowego requestu:**

Php artisan make:request [Nazwa]

**Mechanizmy autoryzacji użytkowników**

**Gate – bramka**

Bramki tworzy się w app\Providers\AuthServiceProvider.php

public function boot()

    {

        $this->registerPolicies();

        Gate::define('admin-level', function(User $user) {

            return true;

        });

    }

Blade ma specjalną dyrektywę to wyświetlania:

@can('admin-level') //to jest nazwa bramki

<div class="sb-sidenav-menu-heading">Admin panel</div>

<a class="nav-link" href="{{ route('get.users') }}">

    <div class="sb-nav-link-icon"><i class="fas fa-user"></i></div>

    Użytkownicy

</a>

@endcan

Jest jeszcze jedna szybsza metoda na sprawdzenie bramki:

Gate::authorize('admin-level');

**Polityki** zapewniają system autoryzacji powiązany z danym modelem lub jakimś zasobem. Są reprezentowane przez klasę.

php artisan make:policy UserPolicy

polityki tworzą się w katalogu App\Policies

**Politykę możemy utworzyć od razu wiążąc ją z modelem:**

php artisan make:policy UserPolicy --model=User

Politykę trzeba zarejestrować, robimy to w AuthServiceProvider.php

protected $policies = [

        // 'App\Model' => 'App\Policies\ModelPolicy',

        User::class => UserPolicy::class,

    ];

Bramki i polityki mogą istnieć jednocześnie

Testowanie

Phpunit.xml

 <server name="APP\_ENV" value="testing"/>

        <server name="BCRYPT\_ROUNDS" value="4"/>

Itd. To są zmienne środowiskowe które nadpisują domyślne zmienne środowiskowe

Testy unitowe- testy jednostek (metod lub funkcjonalności) odizolowanych od otoczenia

Testy funkcjonalne – testy konkretnej funkcjonalności biznesowej

**Uruchomienie testów:** php artisan test

Podczas testowania aplikacja przełącza się na zmienne środowiskowe testów

php artisan make:test UserTest - **tworzenie testu funkcjonalnego (feature)**

php artisan make:test UserTest –unit -**tworzenie testu jednostkowego**

**inna metoda uruchamiania testów: wpisz w terminalu:**

.\vendor\bin\phpunit

**Tworzenie fabryk do testów:**

php artisan make:factory GameFactory

fabryki tworzą się w database\factories

**Dodawanie fabryki z modelem:**

php artisan make:factory PublisherFactory --model=App\Model\Publisher

Przykładowa fabryka:

$factory->define(Game::class, function (Faker $faker) {

    return [

        'id'=> $faker->numberBetween(1, 1000000),

        'steam\_appid'=> $faker->randomNumber,

        'name'=> $faker->words(rand(1, 3),true),

        'type'=> $faker->randomElement(['game','dlc','demo']),

        'description'=> $faker->text,

        'short\_description'=> $faker->text,

        'about'=> $faker->text,

        'image'=> $faker->imageUrl(),

        'website'=> $faker->url,

        'price\_amount'=> $faker->numberBetween(1, 5000),

        'price\_currency'=> $faker->randomElement(['PLN']),

        'metacritic\_score'=> $faker->numberBetween(1, 100),

        'metacritic\_url'=> $faker->url,

        'release\_date'=> $faker->date(),

        'language'=> 'Polish, English',

    ];

});

Przykładowe użycie fabryki:

namespace Tests\Feature;

use Tests\TestCase;

use App\Model\Game;

class ExampleTest extends TestCase

{

    public function testExample()

    {

        $game=factory(Game::class)->make();

        print\_r($game);

        exit;

utworzenie kilku elementów:

$games=factory(Game::class, 3)->make(); //ta trójka to liczba elementów do wygenerowania