**PAŃSTWOWA WYŻSZA SZKOŁA ZAWODOWA   
W NOWYM SĄCZU**

**INSTYTUT TECHNICZNY**

**PRACA DYPLOMOWA**

**PORTAL WSPOMAGAJĄCY KOMUNIKACJĘ   
ZE STUDENTAMI**

**Autor: Jakub Bogdański**

**Kierunek: Informatyka stosowana**

**Nr albumu: 26050**

**Promotor: dr hab. Witold Przygoda**

**NOWY SĄCZ 2019**

Spis treści.

[1. Wstęp. 4](#_Toc31723883)

[1.1. Uzasadnienie wyboru tematu. 4](#_Toc31723884)

[1.2. Cel i zakres pracy. 5](#_Toc31723885)

[1.3. Przedstawienie użytych narzędzi. 6](#_Toc31723886)

[2. Wzorzec projektowy MVC. 7](#_Toc31723887)

[2.1. Model. 7](#_Toc31723888)

[2.2. Widok. 7](#_Toc31723889)

[2.3. Kontroler. 7](#_Toc31723890)

[3. Baza danych. 8](#_Toc31723891)

[3.1. Schemat bazy danych. 8](#_Toc31723892)

[3.2. Połączenie z bazą danych. 9](#_Toc31723893)

[3.3. Określenie relacji w bazie danych. 9](#_Toc31723894)

[3.4. Relacje polimorficzne. 10](#_Toc31723895)

[3.5. Tworzenie relacji polimorficznych. 10](#_Toc31723896)

[3.6. Migracje. 11](#_Toc31723897)

[3.7. Schemat migracji. 12](#_Toc31723898)

[4. Uprawnienia użytkowników. 13](#_Toc31723899)

[4.1. Hierarchia użytkowników. 13](#_Toc31723900)

[4.2. Rozwiązanie technologiczne. 14](#_Toc31723901)

[4.3. Uprawnienia grup. 16](#_Toc31723902)

[5. Funkcjonalność portalu. 17](#_Toc31723903)

[5.1. Strona główna. 17](#_Toc31723904)

[5.1.1. Widok. 18](#_Toc31723905)

[5.1.2. Kontroler. 18](#_Toc31723906)

[5.2. Logowanie oraz rejestracja. 19](#_Toc31723907)

[5.2.1. Widok. 20](#_Toc31723908)

[5.2.2. Kontroler. 21](#_Toc31723909)

[5.3. Użytkownik zalogowany. 22](#_Toc31723910)

[5.3.1. Widok. 23](#_Toc31723911)

[5.3.2. Kontroler oraz skrypty. 25](#_Toc31723912)

[5.3.3. Tworzenie nowego wpisu. 28](#_Toc31723913)

[5.4. Wiadomości. 29](#_Toc31723914)

[5.4.1. Widok. 30](#_Toc31723915)

[5.4.2. Kontroler i skrypty. 30](#_Toc31723916)

[5.5. Grupy. 32](#_Toc31723917)

[5.5.1. Widok. 35](#_Toc31723918)

[5.5.2. Kontroler oraz skrypty. 38](#_Toc31723919)

[5.6. Pliki. 40](#_Toc31723920)

[5.6.1. Widok. 40](#_Toc31723921)

[5.6.2. Kontroler. 41](#_Toc31723922)

[5.7. Zarządzanie użytkownikami oraz ich aktywacja. 42](#_Toc31723923)

[5.7.1. Widok. 43](#_Toc31723924)

[5.7.2. Kontroler oraz skrypty. 44](#_Toc31723925)

1. Wstęp.

W czasach postępującej informatyzacji oraz powszechnego dostępu do sieci internetowej, większość wiadomości, które ze sobą wymieniamy są w formie cyfrowej. Dobrym przykładem mogą być chociażby wiadomości e-mail lub portale społecznościowe cieszące się ogromną popularnością. Informatyzacja dosięgła również instytucje takie jak urzędy, szkoły i uczelnie wyższe.

W skład pracy wchodzą rozdziały:

Rozdział 2. Wzorzec projektowy MVC.

Rozdział 3. Baza danych.

Rozdział 4. Uprawnienia użytkowników.

Rozdział 5. Funkcjonalność portalu.

Rozdział 6.

* 1. Uzasadnienie wyboru tematu.

Głównym powodem, dla którego zdecydowałem się realizować portal wspomagający komunikację ze studentami była potrzeba dostarczenia przejrzystego systemu wiadomości między wykładowcą i studentami. Istniejące rozwiązanie występują w niewielkiej skali oraz niekiedy brakuje im niektórych funkcjonalności.

Często na stronach uczelni wyższych istnieją strony prowadzone przez samych pracowników, w których sami muszą zmieniać kod pliku HTML. Rozwiązanie to sprawia, że strony te wyglądają niezbyt schludnie, szczególnie jeśli mówimy o instytutach nietechnicznych. Nierzadko wykładowcy mają minimalistyczne podejście do tychże stron, które tworzą jako listę linków, gdzie każdy ma do nich dostęp nawet osoby spoza uczelni.

Portal wspomagający komunikację ze studentami, który tworzę w ramach tej pracy dyplomowej sprawi, że wykładowcy nie będą musieli tracić czasu na edytowanie plików odpowiedzialnych za treść na stronie, a studentom da czytelny dostęp do materiałów z zajęć oraz dobrze działający system komunikacji.

* 1. Cel i zakres pracy.

Celem niniejszej pracy jest zaprojektowanie oraz implementacja portalu komunikacyjnego między prowadzącymi zajęcia i studentami.

Podstawowe funkcjonalności portalu:

- system kont użytkowników – w skład, którego wchodzą logowanie, rejestracja nowych użytkowników, edycja istniejących kont oraz blokowanie/usuwanie kont.

- tworzenie oraz zarządzanie grupami – prowadzący lub administrator będą mogli tworzyć grupy. Grupy te będą podzielone na: laboratoryjne, ćwiczeniowe, wykładowe. Użytkownicy z odpowiednimi uprawnieniami będą mogli definiować własne grupy. System grup będzie działał z systemem kontroli dostępu.

- system kontroli dostępu – użytkownicy będą podzieleni na odpowiadające im role tj. administrator, wykładowca, student. Każda z tych grup będzie posiadać inne uprawnienia na stronie. Administrator będzie mógł korzystać ze wszystkich opcji dostępnych na portalu. Wykładowca będzie mógł: tworzyć grupy, dodawać lub usuwać użytkowników z grup, wysyłać wiadomości do studentów, udostępniać ogłoszenia oraz pliki grupom. Student będzie mógł: wymieniać korespondencje z prowadzącym zajęcia, pobierać materiały przeznaczone dla niego lub grupy do której należy, czytać ogłoszenia przeznaczone dla niego

- prywatne tablice ogłoszeń – system wiadomości między wykładowcą, a studentem. System pozwoli na powiadomienia przez e-mail oraz wysyłanie dowolnych załączników w wiadomości.

- grupowe tablice ogłoszeń – komunikacja prowadzącego zajęcia z grupą studentów. Prowadzący będzie miał możliwość udostępniania materiałów oraz ogłoszeń dla całej grupy.

- publiczne ogłoszenia – wykładowca będzie mógł udostępnić informację lub pliki dla wszystkich użytkowników systemu.

Dodatkowe funkcjonalności portalu:

- wyszukiwarka użytkowników – działająca asynchronicznie wyszukiwarka pozwalająca wykładowcy wyszukać studentów według numeru albumu lub danych osobowych.

- profil użytkownika – pozwala zobaczyć podstawowe dane na temat wybranego użytkownika

* 1. Przedstawienie użytych narzędzi.

Do stworzenia niniejszej pracy wykorzystano podstawowe technologie wykorzystywane przy tworzeniu stron internetowych takie jak: PHP, HTML, MySQL, AJAX, CSS, JavaScript. Dodatkowo zostały użyte rozszerzenia: Laravel oraz Bootstrap.

PHP – popularny, skryptowy język programowania witryn internetowych. Duża część składni PHP wywodzi się z C, Javy oraz języka Perl. Technologia ta pozwala na dynamiczne generowanie stron internetowych. PHP działa po stronie serwera.

HTML – język znaczników służący do tworzenia stron internetowych. Jedno z najbardziej podstawowych narzędzi wykorzystywanych przy tworzeniu witryn. HTML odpowiada za strukturę strony internetowej. Standardy języka ustalane są przez specyfikację W3C. Aktualną wersją jest HTML5.

CSS – jest językiem wykorzystywanym do opisywania wyglądu stron internetowych. Podobnie do HTML jest opisany przez standardy W3C.

JavaScript – skryptowy język programowania służący głównie do tworzenia stron internetowych. Kod JavaScriptu wykonywany jest po stronie klienta i jest on interpretowany przez przeglądarkę internetową

AJAX – technologia pozwalająca na tworzenie asynchronicznych stron internetowych. AJAX pozwala na tworzenie stron, które odświeżają się automatycznie bez konieczności ingerencji użytkownika.

MySQL – strukturalny język zapytań rozwijany przez firmę Oracle. Służy do zarządzania relacyjnymi bazami danych. MySQL jest udostępniany na licencji GNU GPL.

Bootstrap – jedno z najpopularniejszych rozszerzeń służących do tworzenia wyglądu stron internetowych. Bootstrap ułatwia tworzenia responsywnych portali. Dodatkowym jego atutem jest otwarto źródłowość.

Laravel – system stworzony w języku PHP ułatwiający tworzenie portali internetowych. Jego główną cechą jest wykorzystanie wzorca Model View Controller.

1. Wzorzec projektowy MVC.

Wzorzec projektowy MVC (z ang. Model-View-Controller) pozwala w sposób przejrzysty zorganizować zarys aplikacji. Wzorzec ten jest stosowany w przypadku tworzenia aplikacji z graficznym interfejsem użytkownika. Założenia MVC często wykorzystywane są podczas tworzenia portali internetowych, wówczas mają one nieco zmienioną formę. Poprzez zastosowanie technologii HTTP przy tworzeniu stron nie ma możliwości użycia aktywnych modeli.

* 1. Model.

We wzorcu MVC model jest odpowiedzialny za przechowywanie danych, które są wykorzystywane przez kontroler. Modele najczęściej przedstawione są za pomocą klas danego języka programowania.

* 1. Widok.

Zadaniem widoku jest obsługa graficznego interfejsu użytkownika. W skład widoku mogą wchodzić tzw. widoki częściowe, odpowiadają one za odpowiednie fragmenty strony. Widok komunikuje się z kontrolerem poprzez przekazywanie danych z formularzy oraz pobieranie żądania odświeżenia. Widok w żaden sposób nie powinien modyfikować danych.

* 1. Kontroler.

Ostatnim elementem wzorca projektowego MVC jest kontroler. Kontroler odbiera, analizuje oraz przetwarza dane wysłane przez użytkownika za pomocą widoku. Kontroler wchodzi w interakcję z modelem poprzez wysyłanie danych do niego. Kontroler ma bezpośrednie powiązanie z modelem i widokami, z którymi kooperuje. W przypadku aplikacji internetowych zgodnie ze wzorcem MVC kontroler może wysyłać zapytania do bazy danych.

1. Baza danych.

Portal umożliwiający komunikację ze studentami korzysta z bazy danych zarządzanej za pomocą systemu zarządzania bazami danych MySQL. Warto odnotować, że relacje występujące w bazie danych zdefiniowane są na poziomie aplikacji w tak zwanych modelach.

* 1. Schemat bazy danych.

Tabele bazy danych oraz relacje pomiędzy nimi zostały przedstawione na schemacie poniżej(Rysunek 1).



Rysunek 1. Diagram ERD. (DO ZMIANY/TYMCZASOWY)

* 1. Połączenie z bazą danych.

Konfiguracja połączenia z bazą danych odbywa się poprzez rozszerzenie PHP – Laravel. W tym rozszerzeniu wszelakie ustawienia bazy danych znajdują się w pliku .env. Za połączenie z bazą danych odpowiada następujący fragment kodu:

1. DB\_CONNECTION=mysql
2. DB\_HOST=127.0.0.1
3. DB\_PORT=3306
4. DB\_DATABASE=portal
5. DB\_USERNAME=root
6. DB\_PASSWORD=

Na powyższym przykładzie można zauważyć, że konfiguracja odbywa się za pomocą zmiennych środowiskowych.

* 1. Określenie relacji w bazie danych.

Relacje między poszczególnymi tabelami zostały określone w migracjach oraz w modelach za pomocą odpowiednich metod. Do zdefiniowania relacji należy utworzyć metodę w modelu, która zwraca wynik metody odpowiedzialnej za typ relacji, kolejno dla relacji:

- jeden do jednego – korzysta z metody hasOne, odwrotność relacji belongsTo.

- jeden do wielu – używa metody hasMany, odwrotność relacji belongsTo.

- wiele do wielu – korzysta z metody belongsToMany zarówno dla określenia relacji jak i jej odwrotności.

Każda z powyższych metod przyjmuje jako parametr podstawowy odwołanie do modeli, z którymi zachodzi relacja. Dodatkowo możemy podać parametry jeśli chcemy skorzystać ze zdefiniowanych przez siebie kluczy. Metoda belongsToMany może przyjmować dodatkowo nazwę dla tabeli łączącej.

Jeżeli nie korzystamy z dodatkowych parametrów należy pamiętać o odpowiednim nazewnictwie tabel oraz jej kolumn.

Na poniższym przykładzie można zauważyć sposób określania relacji. Za przykład posłuży fragment modelu odpowiedzialnego za użytkownika – User.php, w którym została zdefiniowana relacja typu „wiele do wielu” z grupami.

1. **public** function groups()
2. {
3. **return** $**this**->belongsToMany('App\Group');
4. }

Podobnie prezentuje się fragment odpowiedzialny za odwrotność tej relacji. W modelu grup - Group.php korzystamy z tej samej metody belongsToMany, która przyjmuje model User jako parametr.

1. **public** function users()
2. {
3. **return** $**this**->belongsToMany('App\User');
4. }
   1. Relacje polimorficzne.

Szczególnym typem relacji występującym w bazie danych są tak zwane relacje polimorficzne. W niniejszym projekcie relacje polimorficzne wykorzystywane są do obsługiwania plików. Użytkownik korzystający z aplikacji może załączyć plik do wiadomości lub postu. Zastosowanie relacji polimorficznej sprawia, że zamiast dwóch tabel dla plików z wyróżnieniem relacji(np. file\_post, file\_message), można stworzyć jedną tabelę, która będzie posiadała informację, z którym modelem będzie plik będzie powiązany.

Poniższa tabela(Rysunek 2) przedstawia strukturę tabeli odpowiedzialnej za pliki.



Rysunek 2. Struktura tabeli o nazwie file.

Tabela w relacji polimorficznej posiada dodatkowe pola wskazujące id modelu jego typ. W bazie danych jako typ przechowywana jest nazwa modelu z przedrostkiem ‘App\’, na przykład ‘App\Post’.

* 1. Tworzenie relacji polimorficznych.

Tworząc relacje polimorficzne należy stworzyć odpowiednie kolumny w bazie danych. Odpowiednio przygotowaną tabelę można zobaczyć na Rysunku 2.

Tworzenie relacji tego typu zostało opisane na podstawie relacji tabel odpowiedzialnych za pliki, posty oraz wiadomości niniejszego projektu

W modelu File należy utworzyć metodą zwracającą wynik metody morphTo.

1. **public** function filetest()
2. {
3. **return** $**this**->morphTo();
4. }

W modelach Message oraz Post metodą odpowiadająca za polimorficzną relację jest metoda morphMany, która przyjmuje jako parametry model oraz nazwę metody z tego modelu odpowiedzialną za definicje relacji.

1. **public** function files()
2. {
3. **return** $**this**->morphMany('App\File', 'filetest');
4. }

Odwołanie do relacji odbywa się podobnie jak w przypadku normalnej relacji.

* 1. Migracje.

Migracje w dodatku Laravel działają na podobnej zasadzie jak systemu kontroli wersji. Pozwalają w łatwy sposób zarządzać schematem bazy danych.

Tworzenie migracji odbywa się za pomocą polecenia:

php artisan make:migration create\_files\_table

Za każdym razem przy wywołaniu tego polecenia do nazwy pliku z migracją dopisywana jest informacja o czasie utworzenia tejże migracji. Migracje wywoływane są w kolejności od najstarszej do najnowszej.

Wykonanie migracji odbywa się poleceniem:

php artisan migrate

Dodatkowo to polecenie może przyjmować flagę --force, wymusza ona wykonanie migracji nawet w przypadku gdy może to powodować utratę niektórych danych.

W przypadku błędu każdą migrację można cofnąć za pomocą polecenia:

php artisan migrate:rollback

Wszystkie migracje można wycofać następującym poleceniem:

php artisan migrate:reset

Polecenie php artisan migrate:refresh działa na takiej samej zasadzie jak reset, z tą różnicą, że po resecie wykonywana jest opcja migrate.

Usuwanie wszystkich tabeli a następnie wykonanie migracji odbywa się za pomocą polecenia: php artisan migrate:fresh

* 1. Schemat migracji.

Migracje mają format klasy, która rozszerza klasę Mirgation. Stworzony plik migracji domyślnie składa się z dwóch metod: up, down. Pierwsza z nich wywoływana jest przy okazji uruchomienia migracji, natomiast druga w przypadku cofanie tej migracji.

Migracje powiązanie są z fasadą Schema, która dostarcza narzędzia do zarządzania tabelami bazy danych.

1. Schema::create('messages', function (Blueprint $table) {
2. $table->bigIncrements('id');
3. $table->**char**('title', 64);
4. $table->string('content');
5. $table->integer('from\_id')->unsigned();
6. $table->integer('to\_id')->unsigned();
7. $table->dateTime('date');
8. $table->boolean('status');
10. $table->foreign('from\_id')->references('id')->on('users');
11. $table->foreign('to\_id')->references('id')->on('users');
12. });

Powyższy kod przedstawia zawartość metody up dla pliku migracji odpowiedzialnego za tworzenie tabeli messages.

Na powyższym przykładzie można zauważyć sposób tworzenia tabeli oraz jej kolumn. Tworzenie tabeli odbywa się za pomocą metody create, która jaka parametry przyjmuję nazwę tworzonej tabeli oraz funkcję, która przyjmuje jako parametr przyjmuje obiekt typu Blueprint, który jest używany do określenia nowej tabeli.

Do utworzenia kolumn należy wywołać metodę elementu typu Blueprint. Najczęściej używane metody to:

* $table->bigIncrements(‘nazwa’) – tworzenie klucza głównego typu UNSIGNED BIG INT
* $table->char(‘nazwa’, ilość znaków) – pole znaków o określonej długości
* $table->string(‘nazwa’) – typ tekstowy
* $table->integer(‘nazwa’) - określa liczbę całkowitą, może dodatkowo wywoływać metodę unsigned, która oznacza liczby tylko nieujemne
* $table->dateTime(‘nazwa’) – pole daty i czasu.
* $table->boolean(‘nazwa’) -> pole przyjmujące wartości true lub false

W przypadku tworzenia relacji polimorficznej należy skorzystać z metody $table->morphs(‘nazwa’). Tworzy ona automatycznie pola „nazwa\_type” oraz „nazwa\_id” określające model oraz klucz obcy do tego modelu.

Podczas korzystania z migracji tworzącej relację polimorficzną należy spełnić określone wymagania serwera. W przypadku korzystania z MySQL wersja musi być nowsza niż 5.7.7.

W przeciwnym razie otrzymamy następujący błąd:

SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 767 bytes

Jego rozwiązaniem może być aktualizacja MySQL do najnowszej wersji lub ręczne ustawienie domyślnej długości zmiennej typu tekstowego.

W celu określenia relacji na poziomie bazy danych należy skorzystać z metody

$table->foreign():

$table->foreign('from\_id')->references('id')->on('users');

Powyższy kod pokazuje sposób określania klucza obcego dla kolumny o nazwie „from\_id”. Należy wskazać do jakiej kolumny odnosi się wybrana kolumna w obcej tabeli.

Dodatkowo można określić zachowanie klucza obcego podczas usuwania lub zmiany wartości w tabeli. W tym celu należy zastosować metodę onDelete lub onUpdate, która za parametr przyjmuje określone zachowanie.

$table->foreign('user\_id')->references('id')->on('users')->onDelete('set null');

Powyższy przykład opisuje ustawienie wartości null przy usunięciu użytkownika. Należy pamiętać, żeby klucz obcy był ustawiony jak nullable.

Metody onDelete oraz onUpdate mogą przyjmować wartości znane z systemu zarządzania bazą danych MySQL, to jest:

* Cascade – usuwa element powiązany
* Set null – ustawia wartość null dla klucza obcego
* Restrict – oznacza odrzucenie wykonywanej akcji

1. Uprawnienia użytkowników.

Użytkownicy korzystający z systemu przypisani są do grup różniącymi się poziomami dostępu do funkcjonalności dostępu.

* 1. Hierarchia użytkowników.

Najniższy poziom dostępu do informacji na stronie internetowej będą mieli użytkownicy niezalogowani tak zwani „goście”. Goście będą mieli dostęp jedynie do plików i materiałów oznaczonych przez udostępniającego jako publiczne.

Kolejni w hierarchii są studenci. Podczas rejestracji będą mogli oni dołączyć do grupy wykładowej odpowiadającej ich specjalizacji. Oprócz tego mogą wysyłać wiadomości wraz z załącznikami do innych użytkowników systemu.

Użytkownicy zalogowani mogą wybrać czy chcą być powiadamiani drogą mailową o nowej wiadomości w systemie. Oprócz komunikacji z innymi użytkownikami student może zobaczyć do jakich grup należy oraz posty przeznaczone dla tych grup. Oprócz tego mogą zobaczyć pliki, które zostały do nich wysłane w postach bez konieczności ich przeszukiwania.

Grupą z wyższymi uprawnieniami jest grupa prowadzących zajęcia. Oprócz przywilejów przypisanych do studentów mogą oni dodatkowo definiować własne grupy oraz dodawać do nich użytkowników. Mogą oni również wyrzucać wybrane osoby z grupy oraz usuwać całe grupy.

Najwyższy poziom uprawnień mają administratorzy. Oprócz uprawnień niższych grup otrzymują oni dodatkowo dostęp do zarządzania uprawnieniami użytkowników. Domyślnie każdy zarejestrowany użytkownik ma rolę studenta. Administrator systemu może ją zmienić. Dodatkowo mają oni podgląd wszystkich grup.

* 1. Rozwiązanie technologiczne.

System uprawnień oparty jest na podstawie zmiennej w tabeli użytkowników. Zmienna ta przyjmuje wartości całkowite od 1 do 3. Każda z nich odpowiada innemu uprawnieniu, kolejno:

* 1 – student
* 2 – prowadzący
* 3 – administrator

Sprawdzanie roli użytkownika odbywa się za pomocą tak zwanych Middleware. Middleware to mechanizm pozwalający filtrować wywołania użytkownika.

Żeby utworzyć klasę middleware należy wywołać następujące polecenie:

php artisan make:middleware RoleTeacher

Polecenie to utworzy plik z klasą RoleTeacher z metodą handle przyjmującą jako parametry: żądanie użytkownika oraz następną stronę.

Poniższy kod przedstawia metodę handle dla middleware RoleTeacher.

1. **public** function handle($request, Closure $next)
2. {
3. **if**(Auth::user()->role>=2)
4. **return** $next($request);
5. **else**
6. **return** redirect()->back();
7. }

Metoda ta odpowiada za sprawdzanie czy zalogowany użytkownik ma co najmniej uprawnienia prowadzącego zajęcia. W przypadku gdy zalogowana osoba jest prowadzącym lub administratorem system pozwala przejść użytkownikowi do żądanej przez niego podstrony. W przeciwnym razie następuje przekierowanie do poprzedniej strony.

W przypadku gdy należy sprawdzić czy zalogowany użytkownik jest administratorem należy utworzyć odpowiednią klasę. Wówczas metoda handle będzie wyglądać podobnie do tej z powyższego przykładu z tą różnicą, że warunkiem będzie równość z liczbą 3. Poniżej znajduje się fragment kodu klasy IsAdmin

1. **if**(Auth::user()->role>=3)
2. **return** $next($request);
3. **else**
4. **return** redirect()->back();

Niektóre z podstron nie wymagają oceny uprawnień. Wtedy konieczne jest zabezpieczenie adresów przed dostępem użytkowników niezalogowanych. Należy wówczas utworzyć middleware sprawdzający czy użytkownik jest zalogowany.

php artisan make:middleware LoggedIn

Metoda handle klasy LoggedIn prezentuje się następująco:

public function handle($request, Closure $next)

{

if(Auth::Check()==true)

return $next($request);

else

return redirect()->back();

}

Na powyższym przykładzie warunkiem do przekierowania dalej jest sprawdzenie czy użytkownik jest zalogowany, odpowiada za to metoda Auth::check(), która zwraca wartości true, gdy użytkownik jest zalogowany lub false, w przeciwnym przypadku.

Middleware dodatkowo mogą być wykorzystywane do zabezpieczenia adresów wywoływanych asynchronicznie. W takim przypadku należy utworzyć klasę, która sprawdzi czy aktualne żądanie zostało wywołane przez technologię AJAX.

1. **public** function handle($request, Closure $next)
2. {
3. **if**($request->ajax())
4. **return** $next($request);
5. **else**
6. **return** redirect()->back();
7. }

Powyższy kod przedstawia metodę handle klasy AjaxMiddleware. W warunku wywoływana jest metoda, która sprawdza czy żądanie wywołane jest asynchronicznie, jeśli tak odbywa się przekierowanie do żądanej podstrony, w przeciwnym razie użytkownik zostanie cofnięty do strony poprzedniej.

Rejestracja middleware odbywa się w pliku Kernel.php, który znajduje się pod ścieżką App\Http. Należy do tablicy o nazwie routeMiddleware dodać wpis z nową klasą. Przykładowy wpis prezentuje się następująco:

'ajax' => \App\Http\Middleware\AjaxMiddleware::class,

Przypisanie middleware do adresu url odbywa się w pliku web.php znajdującym się w folderze routes. Middleware można przypisać na dwa sposoby:

* Dla grupy adresów – wówczas korzystamy z klasy Route wywołując metodę middleware a następnie metodę group. Przykład przedstawia grupę tras wywoływanych przez technologię AJAX:

1. Route::middleware('ajax')->group(function () {
2. Route::get('/message/x/{id}', 'AjaxController@messages')->middleware('loggedin');
3. Route::get('/message/read/{type}/{id}', 'MessageController@read')->middleware('loggedin');
4. Route::get('/post/filter/{id}', 'AjaxController@posts');
5. Route::get('/users/x/{id}', 'AjaxController@users')->middleware('admin');
6. Route::get('/ajax/search{id}', 'AjaxController@search')->name('search');
7. Route::get('/ajax/user/search', 'AjaxController@searchuser')->name('searchuser');
8. Route::get('/userinfo/{id}', 'AjaxController@userinfo')->name('userinfo')->middleware('teacher');
9. Route::get('/group/add/{id}/{uid}', 'GroupController@addto')->name('groups.addto')->middleware('teacher');
10. Route::get('/message/add/{id}', 'MessageController@add')->name('message.add');
11. Route::get('/group/info/{id}', 'AjaxController@groups')->name('grouptest');
12. });

* Dla pojedynczego adresu – wtedy należy wywołać metodę middleware dla zarejestrowanej trasy

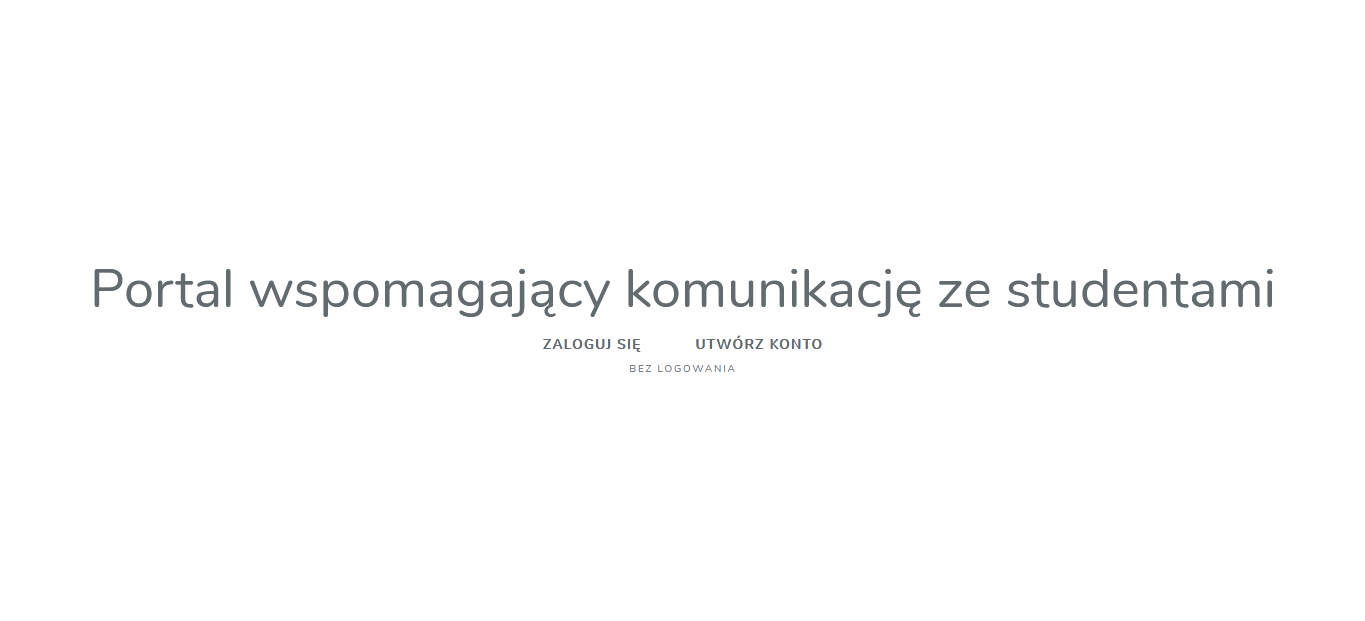
1. Route::post('/message/postreply/{type}/{id}', 'MessageController@postreply')->name('message.postreply')->middleware('loggedin');
   1. Uprawnienia grup.

W przypadku uprawnień, grupa definiuje dostęp do wyświetlanych danych. Dostęp do plików i postów uzależniony jest od przynależności do grupy zdefiniowanej przez prowadzącego. Jeśli użytkownik nie należy do grupy nie będzie miał możliwości pobrania plików, ani odczytania postu.

1. Funkcjonalność portalu.
   1. Strona główna.

Po przejściu pod adres portalu użytkownikowi wyświetlana jest strona startowa. Na tej stronie osoba odwiedzająca witrynę ma możliwość wyboru czy chce się zalogować, założyć konto lub odwiedzać stronę jako gość.

Strona ta ma przejrzysty, minimalistyczny wygląd.

Rysunek 3. Strona startowa portalu.

W przypadku nieudanego logowania lub rejestracji użytkownik jest o tym informowany odpowiednimi komunikatami.



Rysunek 4. Informacja o nieudanej walidacji rejestracji.

* + 1. Widok.

Za wygląd strony startowej odpowiada widok o nazwie welcome. Widok to plik odpowiadający za prezentacje danych.

Ciało strony prezentuje się następująco:

1. **<div** class="flex-center position-ref full-height"**>**
2. **<div** class="content"**>**
3. @if ($errors-**>**any())
4. **<div** class="alert alert-danger"**>**
5. **<ul>**
6. @foreach ($errors-**>**all() as $error)
7. **<li>**{{ $error }}**</li>**
8. @endforeach
9. **</ul>**
10. **</div>**
11. @endif
12. **<div** class="title m-b-md"**>**
13. Portal wspomagający komunikację ze studentami
14. **</div>**
16. **<div** class="links"**>**
17. **<a** href="#" data-toggle="modal" data-target="#loginModal"**>**Zaloguj się**</a>**
18. **<a** href="#" data-toggle="modal" data-target="#registerModal"**>**Utwórz konto**</a><br>**
19. **<a** href="{{route('guest')}}"**><small** class="text-muted"**>**Bez logowania**</small></a>**
20. **</div>**
21. **</div>**
    * 1. Kontroler.

W kontrolerze UserController za stronę główną odpowiada metoda home, która przentuje się w następujący sposób:

1. **public** **function** home()
2. {
3. $groups=Group::where('type', 1)->get();
4. **if**(!Auth::check())
5. **return** view('welcome')->with('groups', $groups);
6. **else**
7. **return** redirect()->route('main');
8. }

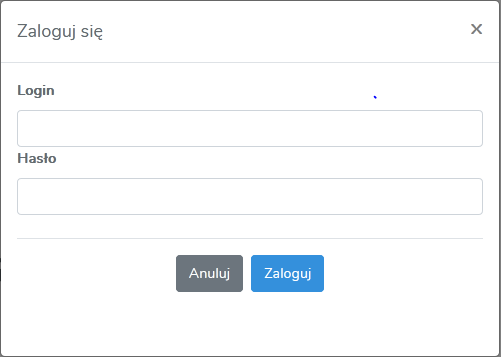
W metodzie home pobierane są grupy, które są ustawione jako wykładowe. Następnie sprawdzane jest czy użytkownik jest zalogowany.

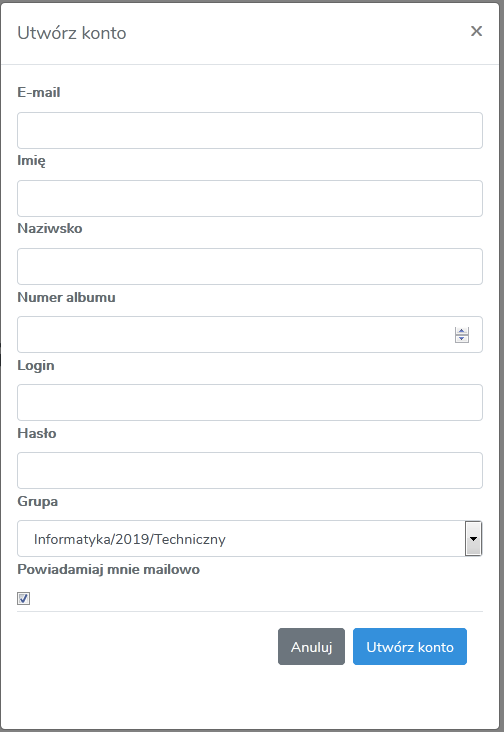
W przypadku gdy użytkownik nie jest zalogowany zwracany jest widok welcome do którego przekazana jest zmienna z grupami wykładowymi. Zawartość tej zmiennej wyświetlana jest w formularzu rejestracji.

Jeśli użytkownik jest zalogowany, zostaje przekierowany do trasy o nazwie main.

5.2. Logowanie oraz rejestracja.

Po wybraniu opcji zaloguj się lub utwórz konto zostanie wyświetlone okienko z formularzem umożliwiającym logowanie(Rysunek 4) lub rejestrację nowego użytkownika (Rysunek 5).

 Rysunek 4. Logowanie.

 Rysunek 5. Formularz rejestracji.

5.2.1. Widok.

Kliknięcie w link do logowania lub rejestracji na stronie głównej wywołuje okienko typu modal. Kod modal odpowiedzialny za logowanie prezentuje się następująco.

1. **<div** class="modal smooth-transition" id="loginModal" tabindex="-1" role="dialog" aria-labelledby="loginModalLabel" aria-hidden="true"**>**
2. **<div** class="modal-dialog" role="document"**>**
3. **<div** class="modal-content"**>**
4. **<div** class="modal-header"**>**
5. **<h5** class="modal-title" id="loginModalLabel"**>**Zaloguj się**</h5>**
6. **<button** type="button" class="close" data-dismiss="modal" aria-label="Close"**>**
7. **<span** aria-hidden="true"**>**×**</span>**
8. **</button>**
9. **</div>**
10. **<div** class="modal-body form-group"**>**
11. **<form** method="post" action="{{route('login')}}"**>**
12. @csrf
13. **<div** class="form-group text-left font-weight-bold"**>**
14. **<label** for="login"**>**Login**</label>**
15. **<input** class="form-control" name="login" type="text"**>**
16. **<label** for="password"**>**Hasło**</label>**
17. **<input** class="form-control" name="password" type="password"**>**
18. **<br>**
19. **<div** class="modal-footer justify-content-center"**>**
20. **<button** type="button" class="btn btn-secondary" data-dismiss="modal"**>**Anuluj**</button>**
21. **<button** type="submit" class="btn btn-primary"**>**Zaloguj**</button>**
22. **</div>**
23. **</div>**
24. **</form>**
25. **</div>**
26. **</div>**
27. **</div>**
28. **</div>**

W skład obiektu modal wchodzą:

* Nagłówek – w nagłówku znajduje jego tytuł oraz przycisk ‘x’ do zamknięcia okienka
* Ciało – w skład ciała wchodzi formularz odpowiedzialny za logowanie
* Stopka – w stopce znajdują się dwa przycisku: pierwszy z nich zamyka okno logowania, a drugi z nich wysyła dane do serwera pod adres znajdujący się pod trasą o nazwie login

Podobnie prezentuje się modal odpowiedzialny za rejestrację użytkownika. Jedyną różnicą jest tytuł oraz formularz, który ma więcej pól i odwołuje się do trasy o nazwie register.

5.2.2. Kontroler.

Dane wpisywane w formularzach logowania i rejestracji są sprawdzane pod kątem poprawności wpisanych danych. Walidacja danych odbywa się w kontrolerze UserController. Za tworzenie nowego konta użytkownika odpowiada metoda register, która jako parametr przyjmuje obiekt typu Request. Obiekt ten odpowiada za żądania użytkownika. Metoda register prezentuje się następująco:

1. **public** **function** register(Request $request)
2. {
4. $request->validate([
5. 'login' => 'bail|unique:users|required',
6. 'password' => 'min:8|required',
7. 'email' => 'unique:users|required|email',
8. 'name' => 'required',
9. 'surname' => 'required',
10. 'album' => 'nullable|unique:users',
11. ]);

14. $user = User::create([
15. 'name' => $request->name,
16. 'surname' => $request->surname,
17. 'login' => $request->login,
18. 'password' => Hash::make($request->password),
19. 'email' => $request->email,
20. 'album' => $request->album,
21. 'role' => 1,
22. 'notify' => $request->check=='on'?true:false,
23. ]);
25. **if**(isset($request->group))
26. {
27. $group = Group::find($request->group);
28. $group->users()->attach($user->id);
29. }

32. **return** $this->login($request);
33. }

Na początku metody odbywa się walidacja wpisanych przez użytkownika danych. W przypadku pól login, email oraz album sprawdzane jest czy wpisane dane znajdują się już w bazie danych. Odpowiada za to wpis unique:users. Oprócz tego sprawdzane jest czy hasło jest odpowiedniej długości oraz czy email ma odpowiedni format. Dodatkowo album ma wartość nullable, co oznacza, że może przyjmować wartość null.

Po sprawdzeniu poprawności danych, można dodać użytkownika do bazy danych. Odpowiada za to metoda create. Zanim będzie możliwe dodanie użytkownika z podanymi danymi do bazy danych, najpierw należy ustawić atrybuty w modelu jako fillable lub protected. W metodzie create następuje przypisanie danych do odpowiadających im kolumnom w bazie danych.

Po utworzeniu sprawdzane jest czy użytkownik wyraził chęć dołączenia do grup użytkowników. Jeśli tak, następuje przypisanie użytkownika do wybranej grupy.

Na końcu wywoływana jest metoda login, która prezentuje się następująco:

1. **public** **function** login(Request $request)
2. {
3. $credentials = $request->only('login', 'password');
4. $request->validate([
5. 'login' => 'required',
6. ‘password' => 'required',
7. ]);
9. **if** (Auth::attempt($credentials))
10. {
12. **return** redirect()->intended('/main');
13. }
14. **else**
15. {
16. **return** redirect()->back();
17. }
18. }

Metoda login podobnie do metody register przyjmuje jako parametr żądanie użytkownika.

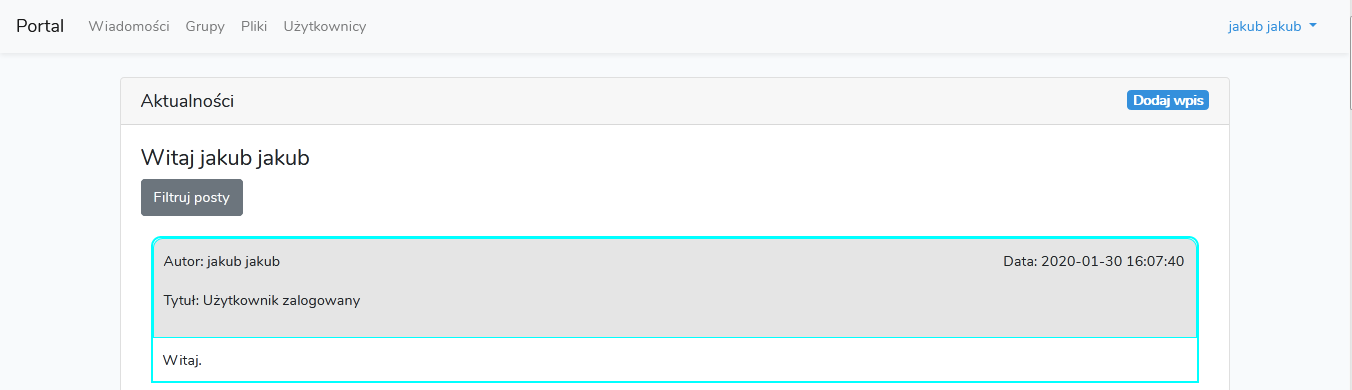
Następnie w zmiennej creditentals wybierane są tylko dane dotyczące logowanie są to pola login oraz password. Kolejnym krokiem jest walidacja danych, w tym przypadku sprawdzane jest tylko czy dane zostały wprowadzone.

Następnie wykonywana jest metoda attempt, klasy Auth, która przyjmuje jako parametr dane, które należy porównać z tym w bazie danych. Jeśli próba logowania przebiegnie pomyślnie, użytkownik przekierowywany jest pod adres ‘/main’, w przeciwnym razie użytkownik zostaje cofnięty do poprzedniej strony.

5.3. Użytkownik zalogowany.

Zarejestrowany użytkownik musi czekać na aktywacje konta przez administratora.

Udane logowanie powoduje przekierowanie użytkownika pod adres ‘/main’. Wyświetlana jest użytkownikowi tablica z postami udostępnionymi dla niego. Dodatkowo po zalogowaniu użytkownik otrzymuje dostęp do menu. Menu różni się w zależności od uprawnień użytkownika, jednak elementami stałymi dla użytkownika są odnośniki prowadzące do wiadomości, grup oraz plików. Po prawej menu stronie użytkownik ma do dyspozycji rozwijane menu w skład którego wchodzą informacje o użytkowniku tzw. profil oraz możliwość wylogowania się z systemu.



Rysunek 6. Interfejs użytkownika (po zalogowaniu na konto administratora).

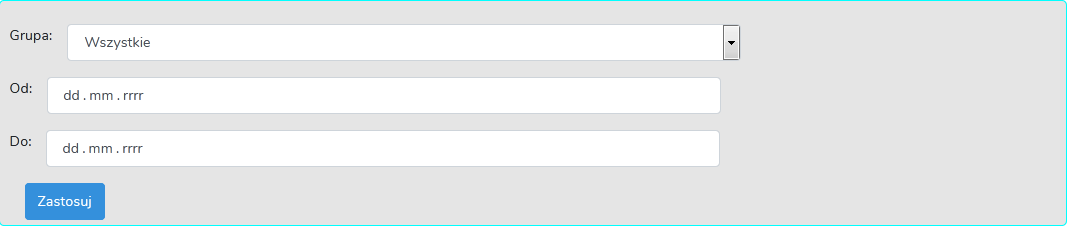
Powyższy rysunek przedstawia interfejs użytkownika dla administratora. W przeciwieństwie do studenta, administrator oraz prowadzący zajęcia mają możliwość dodawania wpisów.

5.3.1. Widok.

Za stronę użytkownika zalogowanego odpowiada widok o nazwie mainpage. W skład zawartości karty wchodzą następujące elementy:

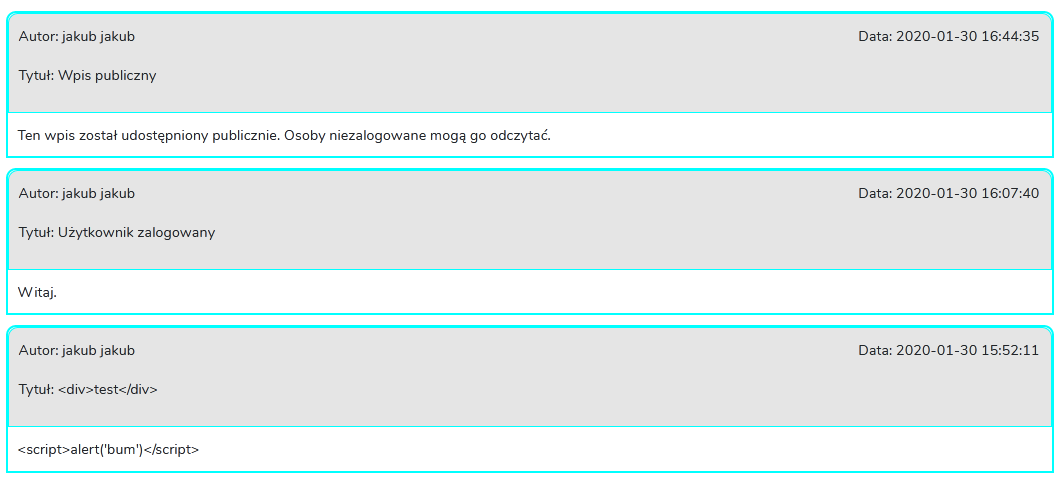
* Przycisk do dodawania wpisu (pod warunkiem, że użytkownik ma większe uprawnienia niż student)
* Wiadomość powitalna
* Posty (jeśli są udostępnione zalogowanemu użytkownikowi)
* Przycisk do filtrowania postów

Filtrowanie postów jest dostępne jeśli są dostępne posty, w przeciwnym przypadku przycisk jest niewidoczny dla użytkownika. Po kliknięciu w przycisk wysuwa się formularz, które prezentuje się następująco:



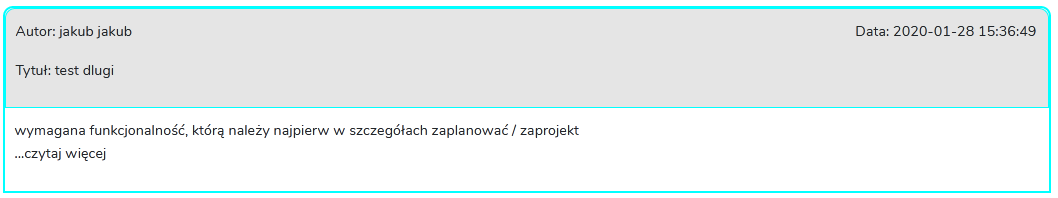
Rysunek 7. Formularz filtrowania postów.

Jeśli użytkownik należy do grupy, dla której wpis został udostępnionym, wówczas wyświetli się on w postaci listy(rysunek 8).



Rysunek 8. Lista postów.

Jak można zauważyć na zrzucie ekranu (Rysunek 8). Dla istniejącego użytkownika zostało udostępnione kilka postów. Wyświetlone zostały posty zarówno posty dla jego grup oraz posty publiczne. Na powyższym rysunku można zaobserwować przykład gdzie został udostępniony kod programu. Zostanie on wyświetlony, ale nie będzie on uruchomiony. Dodatkowo jeśli post jest za długi, zostanie on skrócony i zostanie dodana opcja rozwinięcia całej zawartości.



Rysunek 9. Długi post.

Za wyświetlanie postów odpowiada następujący fragment kodu:

1. **<div** id="posts"**>**
2. @foreach($posts as $post)
3. **<article>**
4. **<div** class="posthead"**>**
5. **<p** class="row justify-content-between"**>**
6. @if(is\_null($post-**>**user\_id))
7. **<span** class="col text-danger"**>**Użytkownik usunięty**</span><span** class="mr-3"**>**Data: {{$post-**>**date}}**</span>**
8. @else
9. **<span** class="col"**>**Autor: {{App\User::Where('id',$post-**>**user\_id)-**>**pluck('name')[0] .' ' . App\User::Where('id',$post-**>**user\_id)
10. -**>**pluck('surname')[0]}}**</span><span** class="mr-3"**>**Data: {{$post-**>**date}}**</span>**
11. @endif
12. **</p>**
13. **<p>**Tytuł: {{htmlspecialchars\_decode($post-**>**title)}}**</p>**
14. **</div>**
15. **<div** class="singlepost"**>**
16. {{htmlspecialchars\_decode($post-**>**content)}}
17. **</div>**
18. @if(isset($post-**>**files[0]))
19. **<hr>**
20. **<div** class="files"**>**
21. Załączone pliki:
22. **<small** class="text-muted"**>**
23. @foreach($post-**>**files as $file)
24. **<a** href="files/{{$file->name}}"**>**{{$file-**>**name}}**</a>**
25. @endforeach
26. **</small>**
27. **</div>**
28. @endif
29. **</article>**
30. @endforeach
31. @else
32. **<br>**
33. **<h6>**Nie znaleziono postów**</h6>**
34. @endif
35. @endif
36. **</div>**

5.3.2. Kontroler oraz skrypty.

Za wyświetlanie odpowiedniego widoku oraz przesyłanie danych do niego odpowiada kontroler o nazwie PostController. Metoda posts odpowiada za zwracanie odpowiedniego widoku w zależności od tego czy użytkownik jest zalogowany. W przypadku udanej autentykacji użytkownika grupy, do których użytkownik należy zapisywane są w tablicy, a następnie wykonywane jest zapytanie do bazy danych, które sprawdza czy są dostępne posty za interesującymi użytkownika parametrami. Wyniki tego zapytania zapisywane są do zmiennej, która kolejno jest zwracana z widokiem o nazwie mainpage.

W przypadku gdy użytkownik nie jest zalogowany zwracany jest widok odpowiedzialny za stronę gościa wraz z postami udostępnionymi publicznie.

Zawartość metody posts wygląda następująco:

1. **if**(Auth::check())
2. {
3. $groups = Auth::user()->groups;
4. $groupsid = **array**();
6. **foreach** ($groups **as** $group)
7. {
8. array\_push($groupsid, $group->id);
9. }
11. $posts = Post::with('groups')->whereHas('groups.users', **function**($q) **use**($groupsid){
12. $q->whereIn('group\_user.group\_id', $groupsid);
13. })->orWhere('public', true)->get()->sortByDesc('date');
14. }
15. **else**
16. $posts = Post::where('public', true)->get()->sortByDesc('date');
18. $view = Auth::check()?'mainpage':'guest';
20. **return** view($view)->with('posts', $posts);

Po wyświetleniu postów wykonywany jest skrypt po stronie klienta, który odpowiada za sprawdzanie długości postów. W przypadku gdy wpis jest za długi zostaje on przycięty oraz zostaje dodana możliwość rozwinięcia oraz zwinięcia tego postu. Odpowiada za to skrypt napisany z wykorzystaniem biblioteki jQuery, który prezentuje się następująco:

1. **function** readpost()
2. {
3. $('.singlepost').each(**function**(index){
4. **var** text = $(**this**).html();
5. **var** lines = text.split("\n");
6. **if**(lines.length>4)
7. {
8. **var** res = text.substr(0, 100);
10. $(**this**).html('<div id="res'+ index + '">' + res + '</div>');
11. $(**this**).append("<a id='more" + index + "'>...czytaj więcej</a>");
12. $(document).on('click', '#more'+index, **function**(){
13. $('#res' + index).html(text);
14. $(**this**).remove();
15. $('#res' + index).append("<p id='less" + index + "'>...czytaj mniej</p>");
16. });
17. $(document).on('click','#less'+index, **function**(){
18. $('#res' + index).html(res);
19. $(**this**).remove();
20. $('#res' + index).append("<p id='more" + index + "'>...czytaj więcej</p>");
21. });
22. }
23. **else**
24. $(**this**).html(text);
25. });
26. }

Za filtrowanie postów odpowiada metoda posts, kontrolera AjaxController. Metoda przyjmuje dwa parametry: id grupy oraz żądanie użytkownika.

Wywołanie metody sprawdza ustawione parametry filtrowania postów, a następnie wywołuje zapytanie do bazy danych. Zarówno dla wybranej grupy jak i dla wszystkich grup sprawdzane są następujące warunki:

* Data od, której wyświetlane są posty jest pusta. Data do jest ustawiona.
* Data od, której wyświetlane są posty jest ustawiona. Data do jest pusta.
* Data od i do, której wyświetlane są posty jest ustawiona.
* Data od i do, której wyświetlane są posty jest pusta.

Fragment odpowiadający za filtrowanie postów w przypadku gdy obie daty są ustawione oraz grupa została wybrana, wygląda następująco:

1. **if**(!is\_null($datefrom)&&!is\_null($dateto))
2. {
3. $posts = Post::with('groups')->whereHas('groups.users', **function**($q) **use**($id, $dateto, $datefrom){
4. $q->where('group\_user.group\_id', $id)->whereBetween('date', [$datefrom . ' 00:00:00', $dateto.' 23:59:59']);
5. })->whereBetween('date', [$datefrom . ' 00:00:00', $dateto.' 23:59:59'])->get()->sortByDesc('date');
6. }

Dodatkowo za obsługę filtrowania postów odpowiada skrypt po stronie klienta. Skrypt po kliknięciu przycisku „Zastosuj” pobiera dane z trzech elementów: data od, data do oraz id grupy, dla której wyświetlane mają być posty. Następnie metoda z AjaxControllera zostaje wywołana asynchronicznie oraz przesłane są do niej wcześniej pobrane lokalnie dane.

Odpowiada za to następujący skrypt:

1. $('#filterposts').on("click", **function**(){
2. **var** id=$('#filtercond').val();
3. **var** datefrom = $('#filterdate\_from').val();
4. **var** dateto = $('#filterdate\_to').val();
5. $.ajax({
6. type : 'get',
7. url : 'post/filter/' + id ,
8. data: {'datefrom': datefrom, 'dateto':dateto},
9. success:**function**(data){
10. $('#posts').html(data);
11. readpost();
12. }
13. });
14. });

Skrypt zwraca częściowy widok o nazwie posts. W jego skład wchodzi pętla wyświetlająca wszystkie elementy zmiennej posts metody posts z AjaxControllera.

5.3.3. Tworzenie nowego wpisu.

Za tworzenie nowego wpisu odpowiada formularz, który obsługuje metoda add znajdująca się w kontrolerze PostController. Metoda ta przyjmuje za parametr żądanie użytkownika. Tworzenie postu odbywa się od sprawdzenia poprawności wpisanych danych:

1. $request->validate([
2. 'title' => 'required',
3. 'content' => 'required',
4. 'recievers' => 'required\_without:public',
5. 'public' => 'required\_without:recievers',
6. 'files.\*' => 'mimes:rar,zip,7z,doc,docx,ppt,pptx,odt,txt,jpeg,bmp,png,gif,svg,pdf',
7. ]);

Tytuł oraz zawartość postu są wymagane. Odbiorcy są wymagani jeśli nie zostało zaznaczone, że post jest publiczny i odwrotnie. Wysłane pliki są sprawdzane pod kątem rozszerzenia pliku.

Posty tworzone są za pomocą metody create.

1. $post = Post::create([
2. 'title' => htmlspecialchars($request->title),
3. 'content' => htmlspecialchars($request->content),
4. 'user\_id' => Auth::id(),
5. 'date' => date(now()),
6. 'public'=>0,
7. ]);

Tytuł oraz zawartość postu są kodowane, żeby zapobiec złośliwym atakom.

Po stworzeniu wpisu dołączane są do niego pliki, jeśli zostały dodane.

1. **if**($request->file('files'))
2. {
3. **foreach**($request->file('files') **as** $file)
4. {
5. $name=$file->getClientOriginalName();
6. $file->move(public\_path().'\\files\\', $name);
8. $filex = **new** File;
9. $filex->name = $name;
10. $filex->path =  public\_path().'\\files\\' . $name;
11. $filex->filetest\_id =  $post->id;
12. $filex->filetest\_type = get\_class($postx);
14. $filex->save();
15. }
16. }

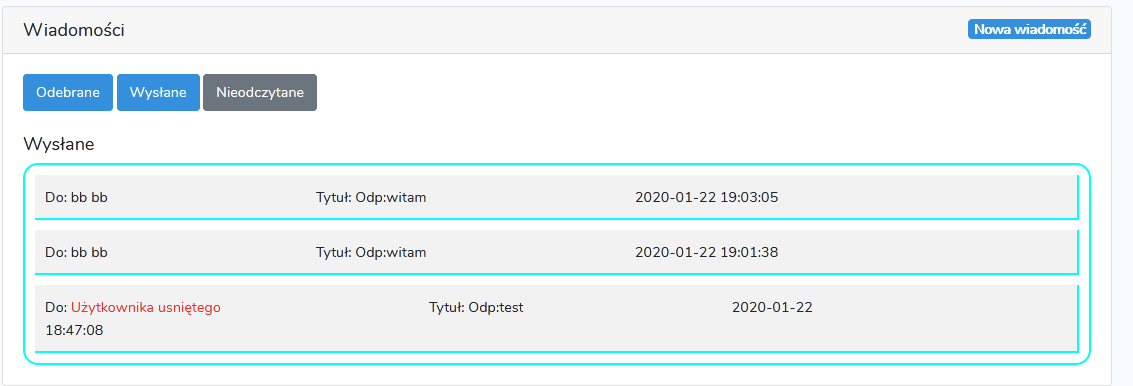
5.4. Wiadomości.

Po wybraniu z menu opcji „Wiadomości” użytkownik zostaje przekierowany do strony, na której może odczytać oraz napisać wiadomość do dowolnego użytkownika systemu. Użytkownik otrzymuje do dyspozycji trzy przyciski: Odebrane, Wysłane, Nieodebrane. Po ich wciśnięciu wczytywane są wiadomości z odpowiadającymi parametrami. Podstrona odpowiedzialna za wiadomości prezentuje się następująco:



Rysunek 10. Wiadomości.

Po kliknięciu w przycisk wiadomości zostają załadowane, pod warunkiem że istnieją.



Rysunek 11. Wiadomości wysłane.

Na powyższym rysunku (Rysunek 11) można zauważyć wiadomość od użytkownika usuniętego. Dzieje się tak w przypadku gdy użytkownik wyśle wiadomość, a następnie usunie swoje konto.

W przypadku wybrania wiadomości odebranych tło wiadomości różni się w zależności czy wiadomość ta ma status odebranej lub nieodebranej. Dodatkowo jeśli nie ma żadnej wiadomości skierowanej do użytkownika, zostaje poinformowany o tym komunikatem „Brak wiadomości”

W prawym górnym rogu menu wiadomości użytkownik ma dostęp do tworzenia nowej wiadomości. Po kliknięciu nowa wiadomość użytkownik zostaje przekierowany do podstrony z formularzem. Odbiorca wiadomości zostanie poinformowany o nowej wiadomości drogą mailową tylko w przypadku gdy wyrazi taką intencje.

5.4.1. Widok.

Za stronę wiadomości odpowiada widok o nazwie all, który znajduje się w folderze messages. Ciało karty tworzą tylko trzy przyciski oraz element blokowy o nadanym id messagebox. Dodatkowo wiadomości korzystają z częściowego widoku messages, który znajduje się w folderze partial.

W skład częściowego widoku wchodzą:

* Instrukcja warunkowa sprawdzająca czy istnieją wiadomości z podanymi parametrami
* Pętla foreach przechodząca po wybranych przez użytkownika wiadomościach
* Pętla sprawdzająca czy do wiadomości zostały przypisane odpowiedzi
* Warunek sprawdzający czy zostały dołączone pliki do wiadomości
* Pętla zwracająca pliki przypisane do wiadomości

5.4.2. Kontroler i skrypty.

Za obsługę wiadomości odpowiada kontroler o nazwie MessageController. W jego skład wchodzą następujące metody:

* Messages – zwraca widok z wiadomościami
* Create – zwraca widok odpowiedzialny za tworzenie wiadomości
* Add – asynchronicznie dodaje użytkownika jako odbiorcę
* Postadd – tworzy wiadomość oraz przypisuje do niej pliki
* Read – zmienia status wiadomości lub odpowiedzi na odczytany
* Reply – zwraca widok odpowiedzialny za odpisywanie na daną wiadomość
* Postreply – tworzy odpowiedź oraz dołącza pliki

W metodach postreply oraz postadd dochodzi do sprawdzenia poprawności wpisanych danych oraz zostają sprawdzone rozszerzenia udostępnionych plików. Odpowiada za to następujący fragment kodu:

1. $request->validate([
2. 'title' => 'required',
3. 'content' => 'required',
4. 'files.\*' => 'mimes:rar,zip,7z,doc,docx,ppt,pptx,odt,txt,jpeg,bmp,png,gif,svg,pdf',
5. ]);

Natomiast za tworzenie wiadomości odpowiada następujący kod:

1. $message = Message::create([
2. 'title' => htmlspecialchars($request->title),
3. 'content' => htmlspecialchars($request->content),
4. 'to\_id' => $request->cookie('user'),
5. 'from\_id' => Auth::id(),
6. 'date' => date(now()),
7. 'status' => false,
8. ]);

Podobnie prezentuje się kod odpowiedzialny za tworzenie odpowiedzi na wiadomość.

W obu przypadkach tytuł oraz zawartość wiadomości zostają przefiltrowane pod kątem znaków specjalnych, które mogłyby zostać zinterpretowany jako kod, który należy wykonać.

Warto odnotować, że dane o użytkowniku, do którego ma zostać wysłana wiadomość przechowywane są w pliku cookie. Po wysłaniu wiadomości ciasteczko to jest kasowane. Dodatkowo użytkownik może ręcznie usunąć ciasteczko, jeśli pomyli się podczas wyboru użytkownika. W tym przypadku wystarczy nacisnąć odpowiedni przycisk na stronie, który pojawi się przy nazwisku adresata.

Za tworzenie pliku cookie odpowiada metoda add.

1. **public** **function** add($id)
2. {
3. Cookie::queue('user', $id, 10);
4. }

Natomiast za usunięcie odpowiada następujący fragment metody tworzącej nową wiadomość:

1. Cookie::queue(Cookie::forget('user'));

Dodawanie plików do wiadomości i odpowiedzi odbywa się na takiej samej zasadzie jak w przypadku postu (rozdział 5.3.3). Tylko w tym przypadku dochodzi do odwołania się do obiektu klasy Message lub Reply.

Po stronie klienta w przypadku wiadomości wykonywane jest pięć skryptów odpowiedzialnych kolejno za:

* Wyszukanie użytkownika
* Asynchroniczne dodanie odbiorcy
* Zmianę odbiorcy
* Odczytanie wiadomości (zmiana statusu)
* Wyświetlenie listy wiadomości z odpowiednim statusem

Skrypt odpowiedzialny za wyszukiwanie użytkownika wywołuje metodę searchuser z kontrolera AjaxController. Skrypt ten wygląda następująco:

1. $("#searchuser").keyup(**function**(){
2. **var** value=$(this).val();
3. $.ajax({
4. type : 'get',
5. url : '../ajax/user/search',
6. data:{'search':value},
7. success:**function**(data){
8. $('#tezt').html(data);
9. }
10. });
11. });

Metoda searchuser znajduje się pod adresem podanym w parametrze url.

Zmiana statusu wiadomości oraz jej odczytanie wywoływane jest następującym skryptem:

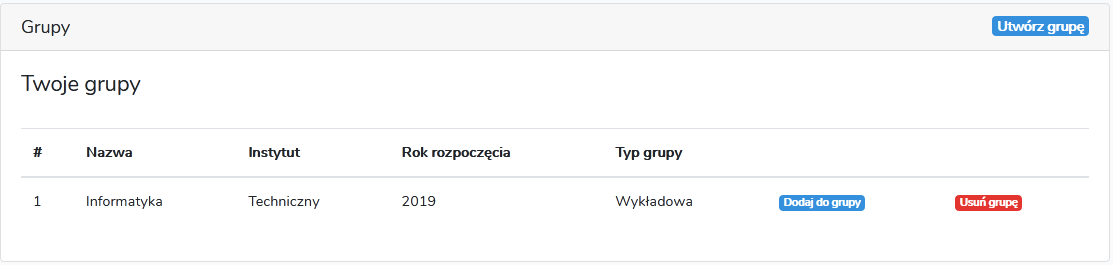
1. $(document).on("click",".show\_message" ,**function**(){
2. **var** val = $(this).attr('val');
3. **var** type = $(this).attr('type');
4. $.ajax({
5. type:'GET',
6. url: './message/read/'+ type +'/'+val,
7. success:**function**(data) {
8. }
9. });
10. $('#message\_content'+type+'\_'+val).removeClass('d-none');
11. $('#message\_content'+type+'\_'+val).toggle();
12. });

Skrypt ten pobiera dane id oraz typ wiadomości (wiadomość lub odpowiedź). Następnie asynchronicznie wywoływana jest metoda read MessageControllera. Kolejno zostaje wyświetlona wiadomość użytkownikowi. Metoda read wygląda następująco:

1. **public** **function** read($type, $id)
2. {
3. **if**($type == 1)
4. {
5. $message=Message::find($id);
6. **if**($message->status==false && $message->to\_id == Auth::id())
7. {
8. $message->status = true;
9. $message->save();
10. }
11. }
12. **else**
13. {
14. $reply=Reply::where('id',$id)->first();
15. **if**($reply->status==false && $reply->to\_id == Auth::id())
16. {
17. $reply->status = true;
18. $reply->save();
19. }
20. }
21. }

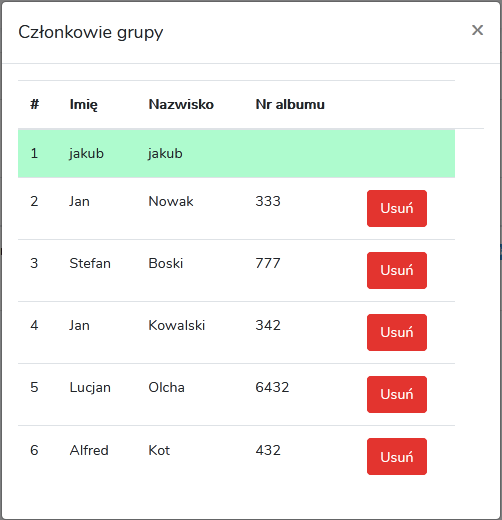
5.5. Grupy.

Kolejną opcją w menu jest opcja o nazwie „Grupy”. Po wybraniu tej opcji użytkownik jest zostaje przekierowany na stronę, która wyświetla grupy do których użytkownik jest przypisany lub wszystkie grupy w przypadku administratora. Dodatkowo administrator lub prowadzący zajęcia mają dostęp do przycisku odpowiedzialnego za tworzenie nowych grup. Grupy wyświetlane są w postaci tabeli.



Rysunek 12. Grupy.

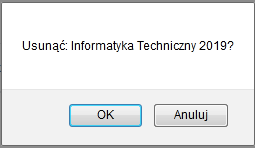
Po kliknięciu we wpis w tabeli użytkownik będzie miał możliwość zobaczenia kto oprócz niego przynależy do tej grupy. Właściciel grupy na liście użytkowników w grupie zostanie wyróżniony zielonym tłem. Lista użytkowników prezentuje się następująco:



Rysunek 13. Okno z użytkownikami grupy (wywołane przez administratora).

W przeciwieństwie do użytkownika zalogowanego jako student, administrator lub właściciel grupy ma dostęp do informacji o numerze albumu oraz ma możliwość usunięcia użytkownika z grupy.

Administrator lub nauczyciel mogą usunąć lub stworzyć grupę. Po wybraniu opcji usuń z tabeli grup, zostaje wyświetlone okienko wymagające potwierdzenia.



Rysunek 14. Potwierdzenie usunięcia grupy „Informatyka” na instytucie technicznym.

Po wciśnięciu „Utwórz grupę” użytkownik zostaje przekierowany do formularza tworzenia nowej grupy. Wymagane dane grupy są następujące:

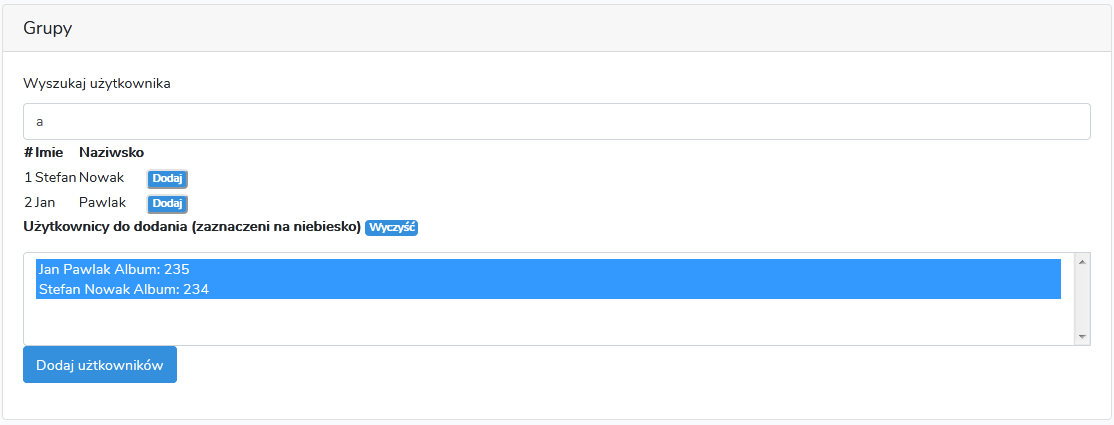
* Nazwa grupy
* Instytut
* Rok rozpoczęcia zajęć
* Typ grupy

Typy grupy, które użytkownik może wybrać są narzucone przez system. Są to następujące typy:

* Wykładowa
* Ćwiczeniowa
* Projektowa
* Laboratoryjna
* Seminaryjna
* Inna

Grupa wykładowa obejmuje cały rocznik, użytkownik może ją wybrać podczas rejestracji. Grupy określone jako ćwiczeniowa, projektowa, laboratoryjna oraz seminaryjna to grupy w skład których wchodzi część rocznika, powiązane są z zajęciami. Grupa, która ma typ inna dotyczy grup definiowanych przez użytkownika, której nie da się określić wcześniej wymienionymi typami grup.

Utworzona grupa jest pusta, dlatego należy dodać do niej użytkowników. Wciskając przycisk „Dodaj do grupy” użytkownik zostanie przekierowany na podstronę z odpowiednim formularzem. Wpisując w polu wyszukiwania imię, nazwisko lub numer albumu, użytkownik zostanie wyszukany. Można go dodać do listy klikając na przycisk obok jego nazwiska. Po dodaniu co najmniej jednego użytkownika, ukaże się lista użytkowników. W przypadku pomyłki należy kliknąć na dane użytkownika. Tło zmieni kolor na biały, co oznacza że użytkownik nie będzie brany pod uwagę po wciśnięciu przycisku „Dodaj użytkowników”.



Rysunek 15. Dodawanie użytkowników do grupy.

5.5.1. Widok.

Za wyświetlanie danych grup odpowiadają widoki zawarte w folderze group to jest add, all, create oraz widoki częściowe o nazwach search oraz groups. Widoki częściowe przyjmują dane zwracane przez asynchroniczne metody wykonywane w kontrolerach AjaxController oraz GroupController.

W skład widoku odpowiedzialnego za wyświetlanie danych o grupach wchodzą następujące elementy:

* Tabela grup – wyświetlana w przypadku gdy grupy istnieją, w przeciwnym razie zwracana jest informacja o braku grup (Rysunek 12)
* Okno „modal” – wyświetla osoby należące do klikniętej przez użytkownika grupy (Rysunek 13)

Fragment widoku odpowiedzialny za wyświetlanie okna typu modal prezentuje się następująco:

1. **<div** class="modal smooth-transition" id="groupModal" tabindex="-1" role="dialog" aria-labelledby="groupModalLabel" aria-hidden="true"**>**
2. **<div** class="modal-dialog" role="document"**>**
3. **<div** class="modal-content"**>**
4. **<div** class="modal-header"**>**
5. **<h5** class="modal-title" id="loginModalLabel"**>**Członkowie grupy**</h5>**
6. **<button** type="button" class="close" data-dismiss="modal" aria-label="Close"**>**
7. **<span** aria-hidden="true"**>**×**</span>**
8. **</button>**
9. **</div>**
10. **<div** class="modal-body"**>**
11. **<div** class="table-responsive"**>**
12. **<table** class="table"**>**
13. **<thead>**
14. **<th>**#**</th>**
15. **<th>**Imię**</th>**
16. **<th>**Nazwisko**</th>**
17. @if(Auth::user()-**>**role**>**=2)**<th>**Nr albumu**</th>**@endif
18. **<th></th>**
19. **</thead>**
20. **<tbody** id="groupmodal"**>**
21. **</tbody>**
22. **</table>**
23. **</div>**
24. **</div>**
25. **</div>**
26. **</div>**
27. **</div>**

W skład elementu o identyfikatorze „groupmodal” wchodzi pętla foreach wyświetlająca podstawowe dane użytkowników znajdujących się w grupie. Dane te są wyświetlane jako elementy tabeli.

Warto zwrócić uwagę na pętle wyświetlającą wszystkie grupy, do których należy użytkownik. To w niej znajdują się elementy wywołujące okno z użytkownikami.

1. @foreach($groups as $i=**>**$gr)
2. **<tr>**
3. **<td** data-toggle="modal" data-target="#groupModal" val="{{$gr->id}}" class="groupinfo"**>**{{$i+1}}**</td>**
4. **<td** data-toggle="modal" data-target="#groupModal" val="{{$gr->id}}" class="groupinfo"**>**{{$gr-**>**name}}**</td>**
5. **<td** data-toggle="modal" data-target="#groupModal" val="{{$gr->id}}" class="groupinfo"**>**{{$gr-**>**institute}}**</td>**
6. **<td** data-toggle="modal" data-target="#groupModal" val="{{$gr->id}}" class="groupinfo"**>**{{$gr-**>**year}}**</td>**
7. **<td** data-toggle="modal" data-target="#groupModal" val="{{$gr->id}}" class="groupinfo"**>**
8. @switch($gr-**>**type)
9. @case(1) Wykładowa @break
10. @case(2) Ćwiczeniowa @break
11. @case(3) Projektowa @break
12. @case(4) Laboratoryna @break
13. @case(5) Seminaryjna @break
14. @case(6) Inna @break
15. @endswitch
16. **</td>**
17. @if(Auth::user()-**>**role**>**2 || App\Group::find($gr-**>**id)-**>**owner==Auth::id())
18. **<td><a** class="badge badge-primary" href="{{route('groups.add', $gr->id)}}"**>**Dodaj do grupy**</a></td>**
19. **<td><a** class="badge badge-danger" onclick="return confirm('Usunąć: {{$gr->name}} {{$gr->institute}} {{$gr->year}}?');" href="{{route('groups.delete', $gr->id)}}"**>**Usuń grupę**</a></td>**
20. @endif
21. **</tr>**
22. @endforeach

Oprócz tego sprawdzane są uprawnienia użytkownika. Jeśli bieżący użytkownik ma uprawnienia administratora lub jest właścicielem grupy, wtedy zostają mu wyświetlone opcje dodawania użytkowników do grupy lub całkowitego usunięcia grupy.

Widok służący do tworzenia grup składa się jedynie z formularza. Dane, które przyjmuje formularz można zobaczyć w Rozdziale 5.5 na stronie 32.

Dodawanie do grupy obsługiwane jest przez widok o nazwie add. W skład widoku wchodzą:

* pole tekstowe służące do wyszukiwania użytkowników
* element blokowy, który przyjmuje dane wyszukiwania
* pole wielokrotnego wyboru wyświetlane w przypadku dodania użytkownika
* przycisk wysyłający dane do serwera

Widok częściowy odpowiedzialny za wyświetlanie rezultatów wyszukiwania składa się z tabeli do której dodawani są użytkownicy, których dane zgadzają się z parametrami wyszukiwania.

1. **<table>**
2. **<thead>**
3. **<th>**#**</th>**
4. **<th>**Imie**</th>**
5. **<th>**Naziwsko**</th>**
6. **<th></th>**
7. **</thead>**
8. **<tbody>**
9. @foreach($users as $i=**>**$user)
10. **<tr>**
11. **<td>**{{$i+1}}**</td>**
12. **<td>**{{$user-**>**name}}**</td>**
13. **<td>**{{$user-**>**surname}}**</td>**
14. @if($user-**>**id != Auth::id())**<td><button** class="usertogroup badge badge-primary" name="userid" val="{{$user->id}}"**>**Dodaj**</button></td>**@endif
15. **</tr>**
16. @endforeach
17. **</tbody>**
18. **</table>**

Kolejnym częściowym widokiem jest widok groups, który odpowiada za użytkowników aktualnie znajdujących się w grupie.

1. @foreach($users as $i=**>**$user)
2. **<tr>**
4. @if($user-**>**id == $owner)
5. **<td** class="highlight"**>**{{$i+1}}**</td>**
6. **<td** class="highlight"**>**{{$user-**>**name}}**</td>**
7. **<td** class="highlight"**>**{{$user-**>**surname}}**</td>**
8. **<td** class="highlight"**></td>**
9. **<td** class="highlight"**></td>**
10. @else
11. **<td>**{{$i+1}}**</td>**
12. **<td>**{{$user-**>**name}}**</td>**
13. **<td>**{{$user-**>**surname}}**</td>**
14. @if(Auth::user()-**>**role**>**=2)**<td>**{{$user-**>**album}}**</td>**@endif
15. @endif
16. @if(Auth::user()-**>**role**>**=2 && $user-**>**id!=$owner)**<td><form** method="post" action="{{route('groups.deletefrom',[$gid,$user->id])}}"**>**@csrf **<input** type="submit" class="btn btn-danger" value="Usuń"**></form></td>**@else**<td></td>**@endif
17. **</tr>**
18. @endforeach

W obu przypadkach widok częściowy ma postać pętli foreach przechodzących po elementach zmiennej users. Zmienne te jednak składają się z zupełnie innych danych. W pierwszym przypadku użytkownicy zwracani są na podstawie schematu wyszukiwania, natomiast w drugim sprawdzana jest przynależność do grup. Dane zawarte w zmiennej users w obu przypadkach wyświetlane są w postaci tabeli.

5.5.2. Kontroler oraz skrypty.

Za obsługę grup odpowiadają wcześniej wspomniane kontrolery GroupController oraz AjaxController. W tym drugim jedyną związaną z grupami metodą jest metoda o nazwie search. Odpowiada ona za wyszukiwanie użytkowników, których należy dodać do grupy.

Metoda ta przyjmuje dwa parametry: identyfikator grupy oraz żądanie użytkownika. Metoda wywołuje zapytanie do bazy danych, które sprawdza czy istnieją użytkownicy z podanymi parametrami oraz czy należy do grupy o podanym jako parametr identyfikatorze, jeśli oba warunki są spełnione zostaje pobrana lista użytkowników. Metoda zwraca widok częściowy wraz z listą pobranych użytkowników.

1. **public** **function** search(Request $request, $id)
2. {
4. $user = User::where(**function** ($query) **use**($request) {
5. $query->where('name', 'LIKE', '%' . $request->search . '%')->orWhere('album', 'LIKE', '%' . $request->search . '%')->orWhere('surname', 'LIKE', '%'. $request->search . '%');
6. })->whereDoesntHave('groups', **function**($q) **use**($request, $id){
7. $q->where('group\_user.group\_id',$id);
8. })->get();
10. **return** view('partial.search')->with('users', $user);
11. }

W kontrolerze GroupController znajdują się następujące metody:

* groups – metoda zwracająca widok z listą grup
* new – tworzy nową grupę
* addto – metoda dodająca użytkowników do listy. Metoda jej działania polega na plikach cookies
* add – metoda zwracająca widok z formularzem dodawania użytkowników do grupy
* postadd – przypisuje użytkowników do grupy na poziomie bazy danych
* clearusers – czyści listę użytkowników
* deletefrom – usuwa użytkownika z grupy
* deletegroup – metoda usuwająca całą grupę

Metoda addto dodająca użytkowników do listy przyjmuje trzy parametry: żądanie użytkownika, identyfikator grup oraz identyfikator użytkownika. Na początku tej metody sprawdzane jest czy plik ciasteczka został utworzony, jeśli istnieje wówczas jego dane rzutowane są z tekstu do tablicy. Następnie sprawdzane jest czy identyfikator użytkownika znajduje się w tablicy pobranej z pliku cookie, jeśli się w niej nie znajduje zostaje dodany do tablicy. Kolejno tablica rzutowana jest z powrotem do zmiennej tekstowej. Metoda ta wywoływana jest asynchronicznie.

1. **public** **function** addto(Request $request, $id, $uid)
2. {
4. $cookiename = 'group' . $id;
6. $cookie = $request->cookie($cookiename);
8. **if**($cookie == null)
9. {
10. Cookie::queue('group' . $id, $uid, 15);
11. }
12. **else**
13. {
14. $cookie = explode(';', $cookie);
16. **for** ($i=0; $i<count($cookie); $i++)
17. {
18. **if**(!in\_array($uid, $cookie))
19. {
20. array\_push($cookie, $uid);
21. $cookie = implode(';', $cookie);
22. Cookie::queue('group' . $id, $request->cookie($cookiename).';'. $uid, 15);
23. $cookie = explode(';', $cookie);
24. }
25. }
26. }
27. }

Po stronie klienta wywoływane są skrypty odpowiedzialne za:

* Wyszukiwanie użytkowników
* Dodawanie użytkowników do listy
* Czyszczenie listy
* Wyświetlanie użytkowników znajdujących się w wybranej grupie

Za wyszukiwanie użytkowników odpowiada następujący skrypt:

1. $("#search").keyup(**function**(){
2. **var** id=$(**this**).attr('val');
3. **var** value=$(**this**).val();
4. $.ajax({
5. type : 'get',
6. url : '../../ajax/search' + id,
7. data:{'search':value},
8. success:**function**(data){
9. $('#tezt').html(data);
10. }
11. });
12. });

Skrypt współpracuje z metodą search kontrolera AjaxController.

Przycisk obsługujący dodawanie użytkownika do pliku ciasteczka działa poprzez wykonanie następującego skryptu:

1. $(document).on( "click", ".usertogroup" ,**function**() {
2. **var** uid = $( **this** ).attr("val");
4. $.ajax({
5. type:'GET',
6. url: window.location.href + '/' +uid,
7. success:**function**(data) {
8. $("#userstoadd").load( location.href+" #userstoadd");
9. }
10. });
11. });

Skrypt ten odwołuje się do metody addto kontrolera GroupController.

Za czyszczenie listy odpowiada skrypt współpracujący z metodą clearusers. Skypt wysyła identyfikator do metody oraz odświeża element blokowy z ustawionym id jako „userstoadd”.

1. $(document).on("click", "#clearusers", **function**(){
2. **var** id = $(**this**).attr('val');
3. $.ajax({
4. type:'GET',
5. url: '../../clear/' +id,
6. success:**function**(data) {
7. $("#userstoadd").load( location.href+" #userstoadd");
8. }
9. });
10. });

5.6. Pliki.

Kolejną opcją, którą można wybrać są pliki. Pod tym linkiem użytkownik znajdzie pliki z wpisów przeznaczonych dla niego. Pliki przedstawione są w postaci listy odnośników.

5.6.1. Widok.

Za pliki odpowiedzialny jest widok w folderze files. Jedyną zawartością karty jest pętla foreach przechodząca przez element znajdujące się pod zmienną files.

1. **<ul>**
2. @foreach($files as $file)
3. **<li><a** href="files/{{$file->name}}"**>**{{$file-**>**name}}**</a></li>**
4. @endforeach
5. **</ul>**

5.6.2. Kontroler.

Za pliki odpowiada kontroler FileController. Jego jedyną metodą jest metoda o nazwie filelist, która nie przyjmuje żadnych parametrów.

Metoda ta pobiera na początku pobiera listę grup, do których użytkownik należy. Następnie identyfikatory tych grup przypisywane są do tablicy z grupami. Kolejno wykonywana jest kwerenda pobierająca posty, które są przypisane do aktualnego użytkownika. Po tych wpisach przechodzi podwójna pętla foreach, która dodaje pliki do tablicy. Przedostatnim krokiem jest dodanie plików z postów, które zostały udostępnione. Na końcu zostaje zwrócony widok z tablicą plików.

Kod źródłowy:

1. **public** **function** filelist()
2. {
3. $groups=Auth::user()->groups->all();
4. $g = **array**();
5. $files = **array**();
7. **foreach**($groups **as** $group)
8. {
9. array\_push($g, $group->id);
10. }
12. $posts = Post::with('files')->whereHas('groups.users', **function**($q) **use**($g){
13. $q->whereIn('group\_user.user\_id', $g);
14. })->get();

17. **foreach** ($posts **as** $p)
18. {
19. **foreach** ($p->files **as** $f)
20. {
21. **if**(!in\_array($f, $files))
22. array\_push($files, $f);
23. }
24. }
26. $publicpost = Post::where('public', true)->get();
27. **foreach**($publicpost **as** $ppost)
28. {
29. **foreach** ($ppost->files **as** $file)
30. {
31. **if**(!in\_array($file, $files))
32. array\_push($files, $file);
33. }
34. }
36. **return** view('files.all')->with('files', $files);
37. }

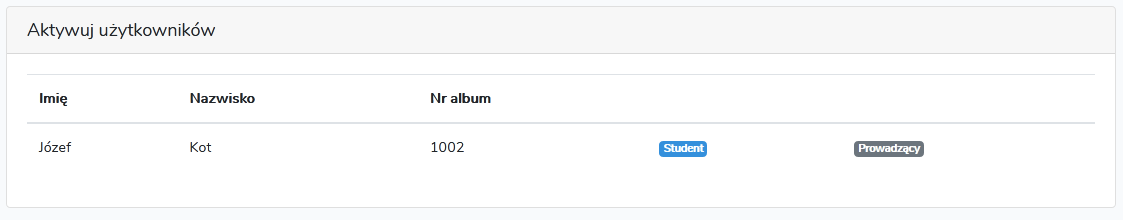
5.7. Zarządzanie użytkownikami oraz ich aktywacja.

Ostatnimi opcjami dostępnymi w menu jest zarządzanie użytkownikami i ich aktywacja. Pozycje te dostępne są po zalogowaniu na konto administratora. Dodatkowo aktywacja użytkowników nie będzie widoczna, jeśli nie ma kont użytkowników wymagających aktywacji. Administrator informowany jest o ilości użytkowników wymagających aktywacji.



Rysunek 16. Pozycja „Aktywuj użytkowników” w menu.

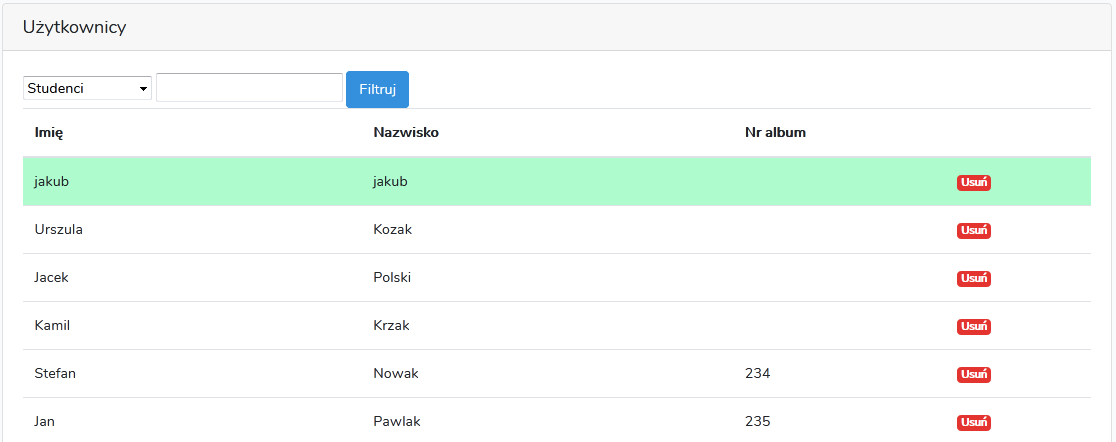
Lista użytkowników w obu przypadkach wyświetlana jest w postaci tabeli. Tabela aktywacji użytkowników przedstawiona została na poniższym rysunku (Rysunek 17).



Rysunek 17. Aktywacja użytkowników.

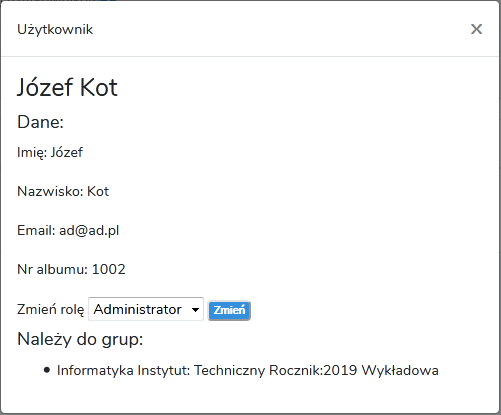
Aktywacja użytkowników odbywa się za pomocą dwóch przycisków. Przycisk opisany jako student przypisuje do konta rangę studenta, natomiast przycisk „Prowadzący” nadaje użytkownikowi uprawnienia nauczyciela.

Tabela użytkownicy składa się z informacji takich jak: imię, nazwisko, numer albumu oraz przycisku służącego do usuwania użytkownika z systemu. Ponadto użytkownik może filtrować użytkowników według ich uprawnień lub wyszukać ich na podstawie imienia, nazwiska lub numeru albumu.



Rysunek 18. Lista użytkowników.

Na stronach służących do zarządzania użytkownikami administrator ma możliwość sprawdzenia szczegółowych danych wybranego przez siebie użytkownika systemu. Odbywa się to poprzez kliknięcie we wpis w tabeli. Wówczas wyświetli się okienko ze szczegółowymi danymi użytkownika. Wewnątrz okienka administrator może zmienić uprawnienia użytkownika.



Rysunek 19. Szczegółowe dane użytkownika.

Jeśli zalogowany użytkownik chce sprawdzić swoje dane, wówczas musi kliknąć swoje imię i nazwisko w prawym górnym rogu, a następnie z rozwijanego menu wybrać „Profil”.

5.7.1. Widok.

Za wyświetlanie tabeli użytkowników odpowiada widok users znajdujący się w folderze users oraz widok częściowy o nazwie users. Za szczegółowe informacje użytkownika odpowiada widok częściowy o nazwie userinfo.

Fragment kodu odpowiadającego za tabele użytkowników dla widoków częściowego oraz pełnego ma taka samą postać. W obu przypadkach na początku sprawdzane jest czy zmienna tablicowa users ma przynajmniej jeden element. Jeżeli ma wówczas tworzona jest tabela wewnątrz której wpisywane są dane ze zmiennej users, w przeciwnym przypadku zwracany jest komunikat o braku użytkowników.

1. @if(isset($users[0]))
2. **<table** class="table"**>**
3. **<thead>**
4. **<th** scope="col"**>**Imię**</th>**
5. **<th** scope="col"**>**Nazwisko**</th>**
6. **<th** scope="col"**>**Nr album**</th>**
7. **<th** scope="col"**></th>**
8. **</thead>**
9. **<tbody** id="usersbox"**>**
10. @foreach($users as $user)
11. @if($user-**>**role==3)
12. **<tr** class="highlight"**>**
13. @else
14. **<tr>**
15. @endif
16. **<td** class="uinfo" val="{{$user->id}}" data-toggle="modal" data-target="#uinfoModal"**>**{{$user-**>**name}}**</td>**
17. **<td** class="uinfo" val="{{$user->id}}" data-toggle="modal" data-target="#uinfoModal"**>**{{$user-**>**surname}}**</td>**
18. **<td** class="uinfo" val="{{$user->id}}" data-toggle="modal" data-target="#uinfoModal"**>**{{$user-**>**album}}**</td>**
19. **<td><a** href="{{route('user.delete', $user->id)}}" class="badge badge-danger" onclick="return confirm('Usunąć: {{$user->name}}  {{$user->surname}} {{$user->album}}?');"**>**Usuń**</a></td>**
20. **</tr>**
21. @endforeach
22. **</tbody>**
23. **</table>**
24. @else
25. Brak wyników wyszukiwania
26. @endif

5.7.2. Kontroler oraz skrypty.

Za działania związane z kontami użytkowników odpowiada kontroler o nazwie UserController. Metody wykonywanie asynchronicznie przechowywane są w kontrolerze AjaxController.

Lista metod dotyczących zarządzania lub wyświetlania danych użytkownika kontroler UserController prezentuje się następująco:

* users – zwraca widok z listą użytkowników
* profile – wyświetla informację o aktualnie zalogowanym użytkowniku (Profil)
* editprofile – zwraca widok edycji profilu
* profilesave – waliduje oraz zapisuje dane z formularza edycji danych
* changerole – metoda zmieniająca uprawnienia użytkownika
* delete – usuwa użytkownika z systemu
* activateusers – zwraca listę użytkowników wymagających aktywacji
* activate – aktywuje użytkownika

Metoda aktywacji użytkownika działa na podstawie systemu ról. Po utworzeniu konta użytkownik przyjmuję role o identyfikatorze 0. Aktywacja użytkownika sprawia, że jego rola zmienia się na 1 – w przypadku studenta lub 2 – w przypadku prowadzącego.

1. **public** **function** activate($r, $id)
2. {
3. $user = User::find($id);
4. $user->role = $r;
5. $user->save();
7. **return** redirect()->back();
8. }

Metoda ta przyjmuje dwa argumenty: pierwszy z nich to jaką rolę należy przypisać do użytkownika, drugi argument to identyfikator użytkownika.

Metoda profilesave służąca do zapisywania edytowanych danych waliduje je przed wysłaniem do bazy danych. Warto zwrócić uwagę, że nie zawsze użytkownik chce zmienić wszystkie dane. W takim przypadku sprawdzanie danych, które są unikatowe ma nieco inną postać. Dane te muszą być sprawdzane z pominięciem tych dotyczących bieżącego użytkownika. Fragment kodu odpowiadający za walidacje wygląda następująco:

1. $request->validate([
2. 'login' => 'bail|required|unique:users,login,' . Auth::id(),
3. 'password' => 'min:8|required',
4. 'email' => 'required|email|unique:users,email,' . Auth::id(),
5. 'name' => 'required',
6. 'surname' => 'required',
7. 'album' => 'nullable|unique:users,album,' . Auth::id(),
8. ]);

W kontrolerze AjaxController za zarządzanie użytkownikiem odpowiadają dwie metody:

userinfo oraz users. Pierwsza z nich odpowiada za wyświetlanie szczegółowych danych

o użytkowniku, natomiast metoda users odpowiada za filtrowanie danych użytkowników.

Kod źródłowy metody usersinfo:

1. **public** **function** userinfo($id)
2. {
3. $user = User::find($id);
4. $groups = $user->groups;
5. **return** view('partial.userinfo', ['user'=> $user, 'groups'=>$groups]);
6. }

Metoda ta jedynie zwraca częściowy widok z dwoma zmiennymi user i groups. Zmienna user przechowuje dane o użytkowniku, natomiast zmienna groups zawiera w sobie informacje o grupach, do których ten użytkownik należy.

Kod źródłowy metody users:

1. **public** **function** users($id, Request $request)
2. {
4. **if**($id==1|$id==2|$id==3)
5. {
6. $users = User::where(**function**($q) **use**($request){
7. $q->where('name', 'LIKE' ,'%' . $request->search . '%')->orWhere('surname', 'LIKE' ,'%' . $request->search . '%')->orWhere('album', 'LIKE' ,'%' . $request->search . '%');
8. })->where('role', $id)->get();
10. **if**($request->search==null)
11. $users = User::where('role', $id)->get();
13. **return** view('partial.users')->with('users', $users);
14. }
15. **elseif**($id==4)
16. {
17. $users = User::where('name', 'LIKE' ,'%' . $request->search . '%')->orWhere('surname', 'LIKE' ,'%' . $request->search . '%')->orWhere('album', 'LIKE' ,'%' . $request->search . '%')->get();
19. **if**($request->search==null)
20. $users = User::all();
22. **return** view('partial.users')->with('users', $users);
23. }
24. **else**
25. **return** redirect()->back();
26. }

Metoda ta przyjmuje dwa parametry: identyfikator roli użytkownika w systemie oraz żądanie użytkownika. Na początku sprawdzana jest wartość roli użytkownika, jeśli spełnia określone warunki zostaje wywołana kwerenda, która zwraca użytkowników spełniających parametry wyszukiwania do zmiennej users. Jeśli pole wyszukiwania użytkowników jest puste, do zmiennej users przypisani zostają wszyscy użytkownicy z określonymi uprawnieniami. Na końcu zwracany zostaje widok ze zmienną users.

W powyższym przykładzie id o wartości 4 odpowiada za wszystkie grupy. Wówczas zwracani są wszyscy użytkownicy lub ci, którzy spełniają parametry wyszukiwania.

Po stronie użytkownika wykonywane są skrypty odpowiedzialne za wyświetlanie informacji o użytkowniku oraz filtrowanie użytkowników. Oba skrypty wywołują metody znajdujące się w kontrolerze o nazwie AjaxController

Skrypt odpowiedzialny za wyświetlanie szczegółowych informacji o użytkowniku prezentuje się następująco:

1. $(document).on("click",'.uinfo',**function**(){
2. $('#userinfo').html('<div class="d-flex justify-content-center"><div class="spinner-grow spinner-grow-md" role="status"><span class="sr-only">Loading...</span></div></div>');
3. **var** id = $(**this**).attr('val')
4. $.ajax({
5. type:'GET',
6. url: './userinfo/'+id,
7. success:**function**(data) {
8. $("#userinfo").html(data);
9. }
10. });
11. });

Skrypt ten pobiera lokalnie id użytkownika. Następnie przesyła je do wspomnianej wcześniej metody userinfo.

Skrypt odpowiedzialny za filtrowanie użytkowników pobiera lokalnie wartość pola wyszukiwania oraz rolę użytkowników, których ma wyświetlać. Skrypt ten współpracuje wysyła dane do metody users kontrolera AjaxController.