

LABORATORIUM 12 CZĘŚĆ 1. IMPLEMENTACJA WZORCA PROJEKTOWEGO MVC W PRZYKŁADOWEJ APLIKACJI TYPU CRUD (LARAVEL V11)

Cel laboratorium:

Celem zajęć jest przygotowanie aplikacji w oparciu o wytyczne wzorca projektowego MVC i poznanie podstawowych elementów implementacji tego wzorca w języku PHP na przykładzie szkieletu programistycznego Laravel.

Zakres tematyczny zajęć:

Poznanie podstawowych elementów wzorca MVC na przykładzie szkieletu programistycznego Laravel w PHP. Utworzenie prostej aplikacji z autoryzacją użytkowników do zarządzania komentarzami. Obsługa akcji dodawania do bazy i pobierania danych z bazy.

Pytania kontrolne:

1. Co to są wzorce projektowe?
2. Wyjaśnij skrót MVC i scharakteryzuj jego elementy.
3. Wyjaśnij skrót ORM. Kiedy warto skorzystać z narzędzi ORM?

Zadanie 12.1. Przygotowanie aplikacji z autoryzacją użytkownika do obsługi akcji dodawania i wyświetlania komentarzy

Do wykonania przykładowej aplikacji skorzystamy z pakietu **XAMPP**, menedżera pakietów **Composer**, środowiska **Node.js** oraz **Laravel** w wersji 11.

W celu utworzenia projektu Laravel (<https://laravel.com/docs/11.x/installation>) należy:

- Zainstalować menadżer pakietów Composer, używanego przez nowoczesne aplikacje PHP do zarządzania zależnościami w aplikacjach i do instalowania komponentów w projektach PHP. Można go pobrać ze strony: <https://getcomposer.org/>.
- Zainstalować Node i NPM - środowisko uruchomieniowe JavaScript do pobrania ze strony: <https://nodejs.org/en>
- Za pomocą composera zainstalować instalator Laravla, wydając w konsoli polecenie:
`composer global require laravel/installer`

UWAGA! W salach laboratoryjnych w systemie Windows jest zainstalowany Composer, ale skonfigurowany do pracy z *php* zainstalowanym w katalogu *C:/tools/php81*. Aby wszystko działało poprawnie trzeba zmodyfikować ustawienia w pliku *php.ini* z katalogu *C:/tools/php81* (**NIE Z XAMPP!!!**). W tym celu należy:

- odkomentować ustawienie: `extension = fileinfo`
- odkomentować ustawienie: `extension = pdo_mysql`
- odkomentować i ustawić bezwzględną ścieżkę do katalogu, w którym znajdują się dodatkowe moduły: `extension_dir = "c:/tools/php81/ext"`



Instalując Composer w swoim środowisku należy wskazać własne miejsce instalacji PHP. W przypadku XAMPP w systemie Windows jest to katalog *C:/xampp/php*. Wtedy nie będą konieczne opisane wcześniej zmiany w pliku *php.ini* (ale proszę sprawdzić te ustawienia).

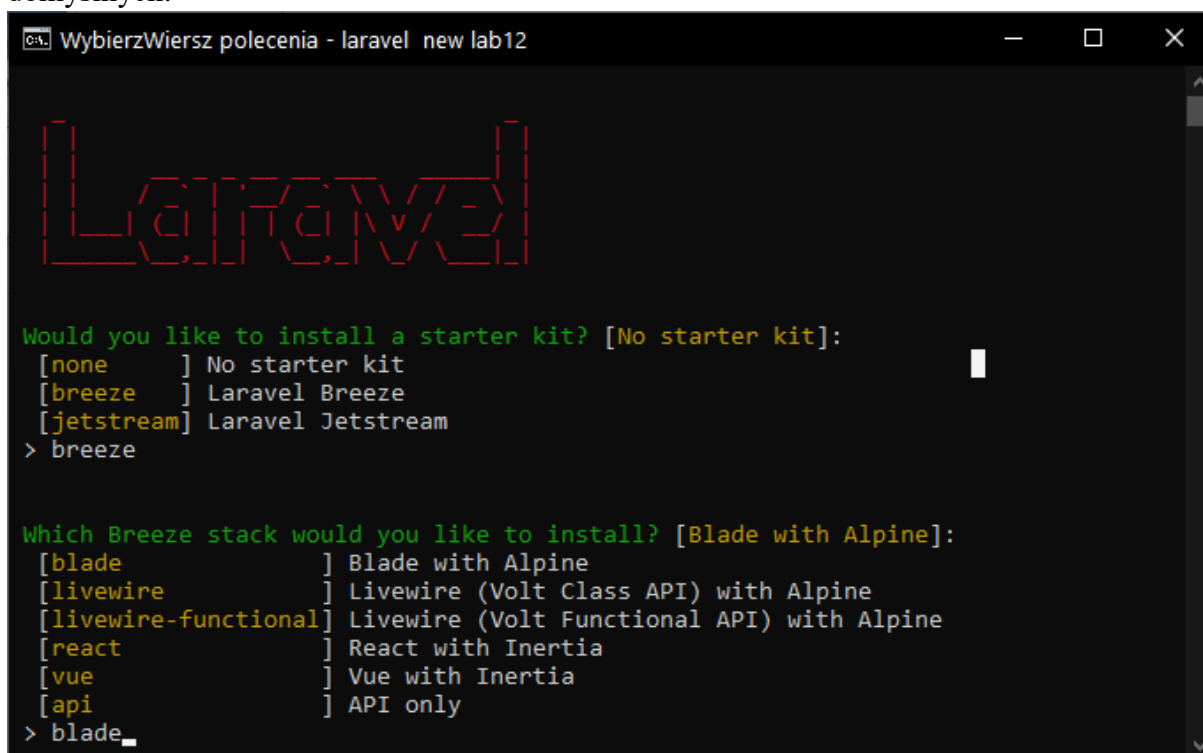
1. Utworzenie pierwszej aplikacji w Laravel

Lokalizacja projektu nie ma znaczenia, gdyż uruchamiać go będziemy za pomocą serwera deweloperskiego, który działa na porcie 8000.

Z wiersza poleceń (np. z poziomu folderu *C:\xampp\htdocs*) utwórz nowy projekt Laravel o nazwie np. *lab12* za pomocą polecenia:

```
laravel new lab12
```

Tworząc projekt wybierz *Laravel Breeze* (moduł do obsługi uwierzytelniania użytkownika), *blade* jako silnik widoków, jasny lub ciemny motyw. Resztę opcji pozostaw na ustawieniach domyślnych:



```
WybierzWiersz polecenia - laravel new lab12

Laravel

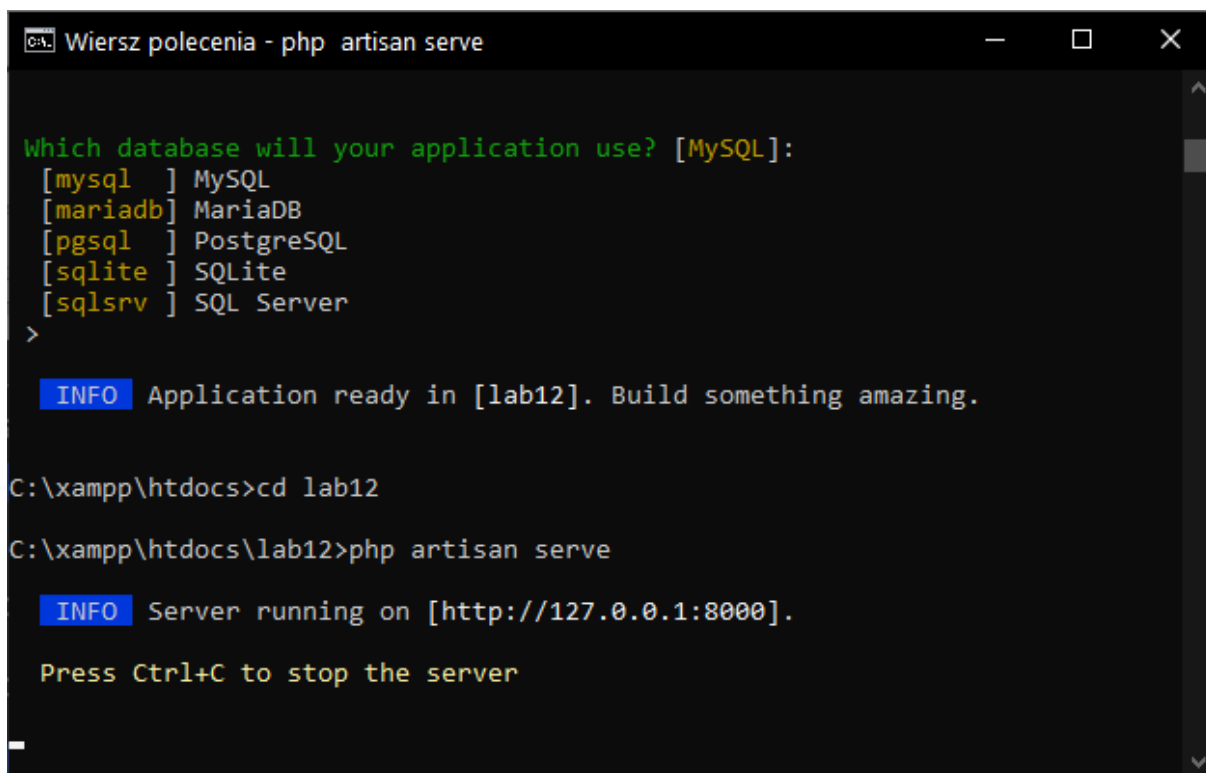
Would you like to install a starter kit? [No starter kit]:
[none] No starter kit
[breeze] Laravel Breeze
[jetstream] Laravel Jetstream
> breeze

Which Breeze stack would you like to install? [Blade with Alpine]:
[blade] Blade with Alpine
[livewire] Livewire (Volt Class API) with Alpine
[livewire-functional] Livewire (Volt Functional API) with Alpine
[react] React with Inertia
[vue] Vue with Inertia
[api] API only
> blade_
```

Przejdź do folderu *lab12* utworzonego projektu (będzie to nasz główny katalog, z którego będziemy wydawać kolejne polecenia). Za pomocą narzędzia *artisan* (interfejs wiersza poleceń dołączony do programu Laravel) uruchom serwer deweloperski (*localhost:8000*), na którym będziemy testować tworzoną aplikację (Rys. 12.1):

```
php artisan serve
```





```
Wiersz polecenia - php artisan serve

Which database will your application use? [MySQL]:
[mysql ] MySQL
[mariadb] MariaDB
[pgsql ] PostgreSQL
[sqlite] SQLite
[sqlsrv] SQL Server
>

INFO Application ready in [lab12]. Build something amazing.

C:\xampp\htdocs>cd lab12
C:\xampp\htdocs\lab12>php artisan serve

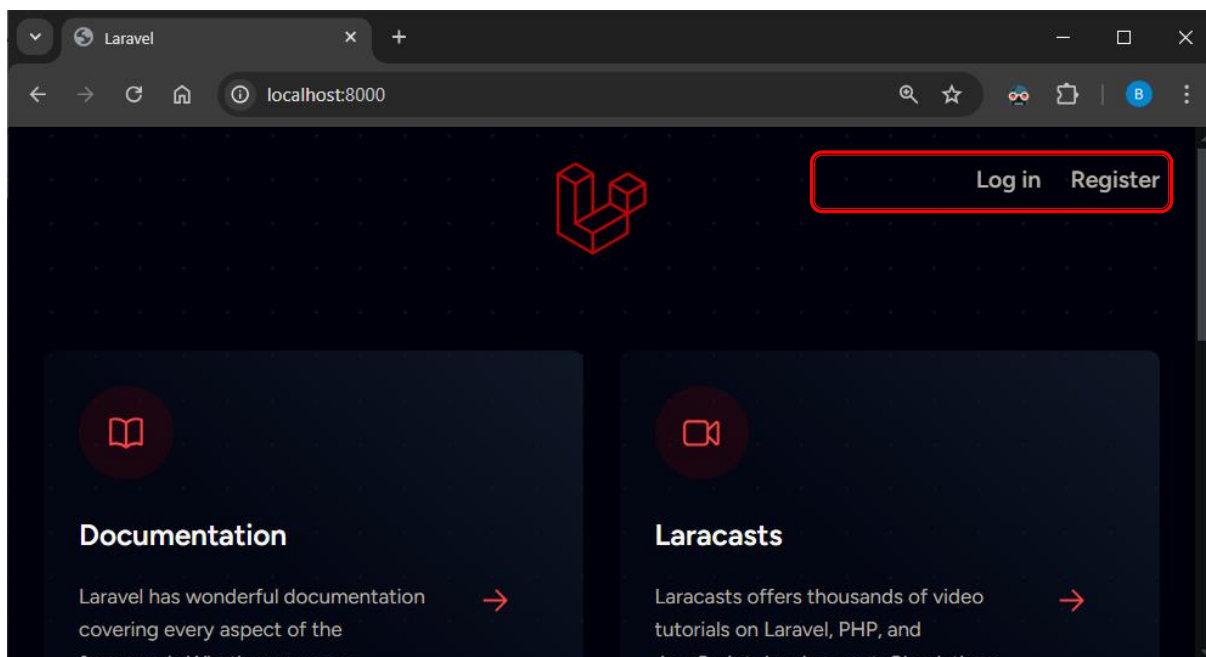
INFO Server running on [http://127.0.0.1:8000].

Press Ctrl+C to stop the server
```

Rys. 12.1. Polecenie uruchomienia serwera deweloperskiego

Uruchom przeglądarkę i przejdź pod adres **localhost:8000**. Widok jak na Rys. 12.2 oznacza, że instalacja przebiegła pomyślnie i utworzony został pierwszy projekt Laravel, którego zasoby znajdują się w katalogu o nazwie **lab12**. **Zwróć uwagę, że na górze po prawej stronie zostały dodane przyciski logowania i rejestracji.**

Korzystając z serwera deweloperskiego za każdym razem w celu wydania nowej komendy trzeba stopować serwer (Ctrl+C), wydać komendę i ponownie uruchamiać serwer do testowania aplikacji.

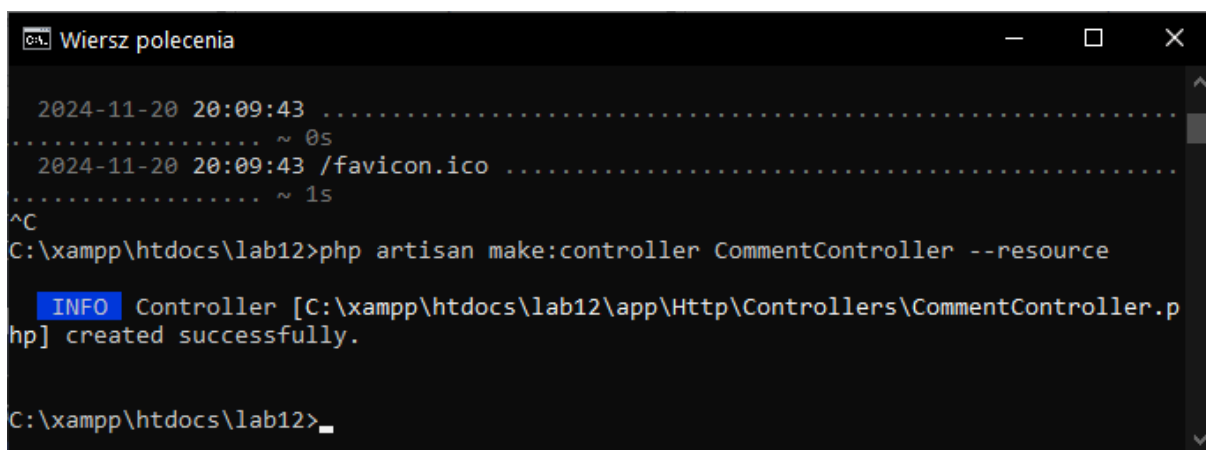


Rys. 12.2. Widok po uruchomieniu projektu w przeglądarce

2. Pierwszy kontroler, routing i widok

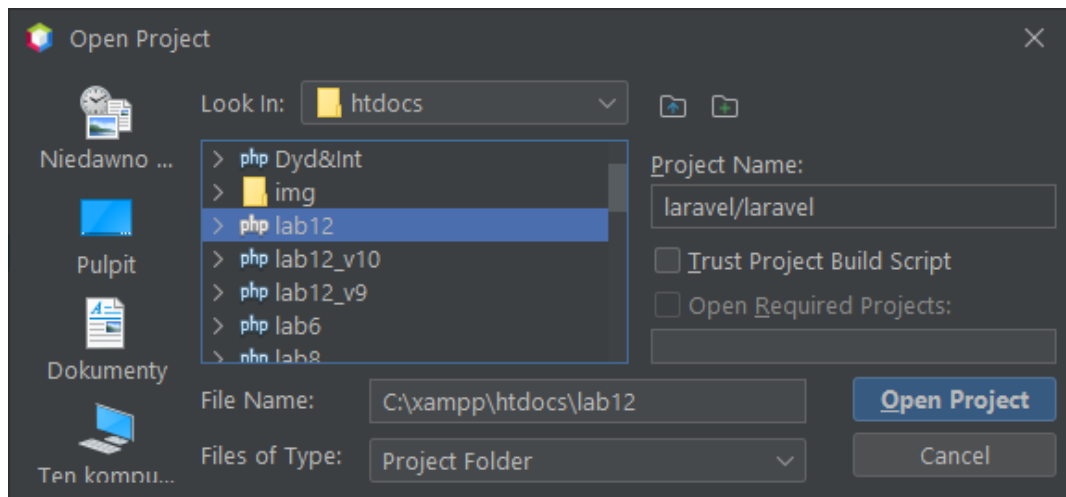
W programowaniu webowym, **routing** (inaczej trasowanie) jest silnikiem napędowym każdej witryny czy aplikacji. Podstawowy routing służy do kierowania żądania HTTP do odpowiedniego kontrolera i jego metody. W celu zobrazowania jak działa routing, utwórz pierwszy **kontroler**. W tym celu przejdź do wiersza poleceń (zastąp uruchomiony na porcie 8000 serwer deweloperski - **Ctrl+C**) i ponownie korzystając z narzędzia **artisan** w katalogu z projektem wydaj polecenie (Rys. 12.3):

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



Rys. 12.3. Polecenie tworzenia kontrolera

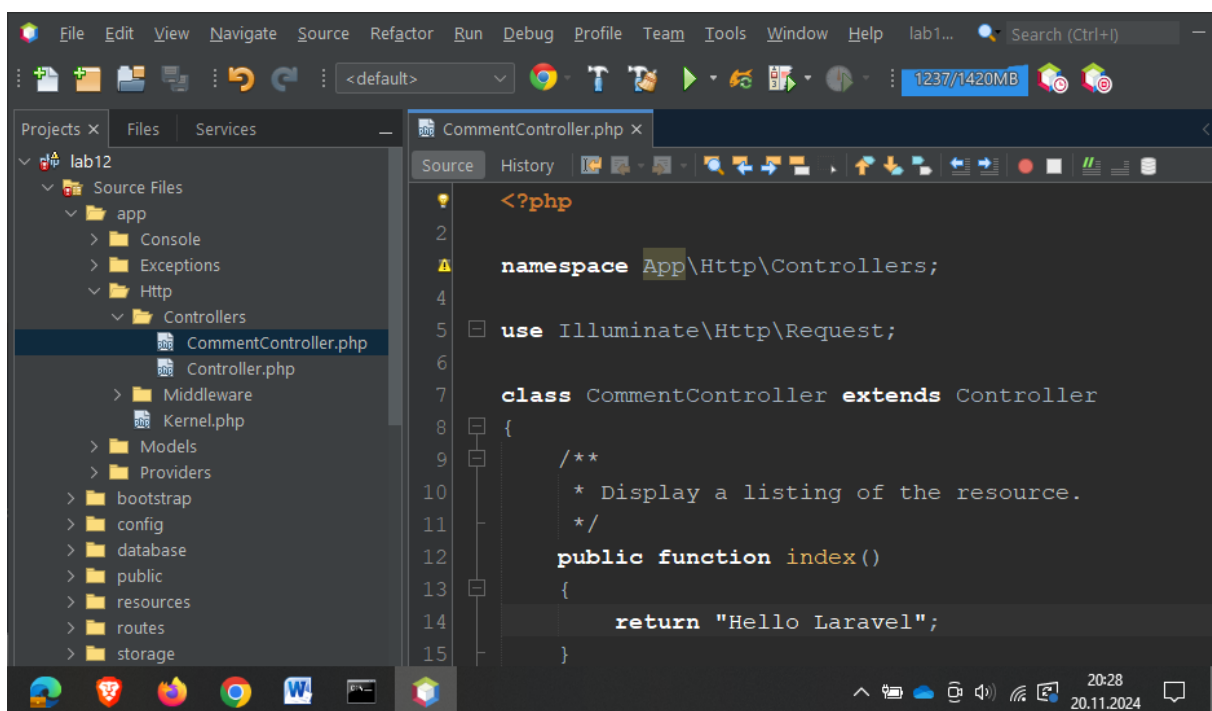
Polecenie to wydane z parametrem **--resource** utworzy klasę kontrolera z metodami do obsługi podstawowych akcji CRUD. Do dalszej pracy z projektem dobrze jest skorzystać ze środowiska programistycznego. Na przykład, korzystając z NetBeans, otwórz utworzony projekt (**File->Open project**) jak pokazano na Rys. 12.4.



Rys. 12.4. Wybór projektu do otworzenia w NetBeans

Przejrzyj zawartość folderu **app\Http\Controllers**, gdzie został utworzony plik **CommentController.php**, awierający automatycznie wygenerowany kod klasy kontrolera i kilka użytecznych (na razie pustych), odpowiednio nazwanych metod. Metody te będziemy uzupełniać w kolejnych punktach. Do funkcji **index()** dodaj instrukcję (Rys.. 12.5):

```
return "Hello Laravel";
```



Rys. 12.5. Uzupełniona metoda **index()** w utworzonym kontrolerze **CommentController**

Do pliku **routes/web.php** dodamy teraz regułę, która określi, że żądanie o URL: **localhost:8000/comments** powinno być obsługane przez metodę **index()** kontrolera **CommentController** (Rys. 12.6):

```
Route::get('/comments',[CommentController::class,'index']);
```



Fundusze Europejskie
Wiedza Edukacja Rozwój

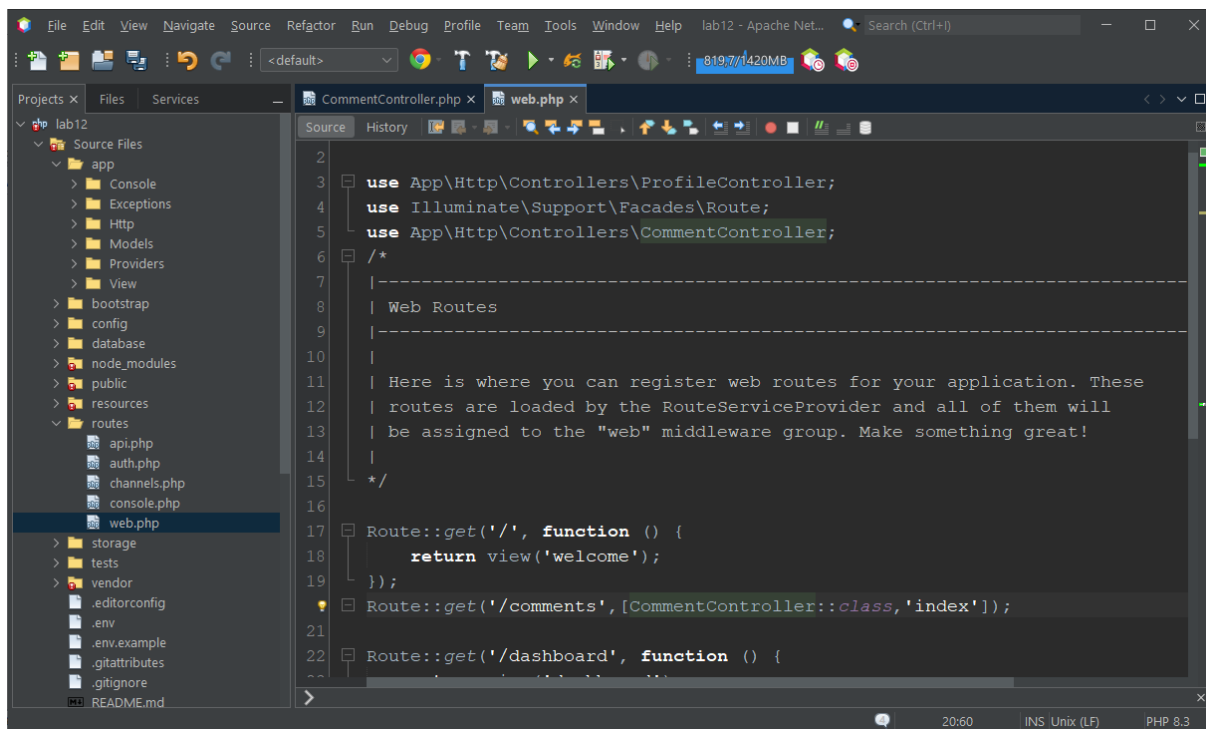


Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



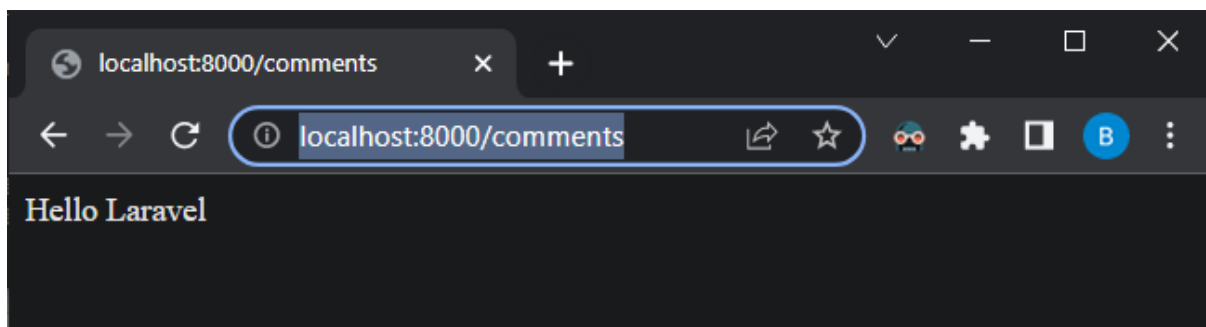
Zwróć uwagę na konieczność importu klasy *CommentController* oraz na istniejące już tam inne reguły routingu dla strony startowej i inne konieczne do obsługi rejestracji i logowania użytkownika (Rys. 12.6).



Rys. 12.6. Reguły routingu w pliku routes/web.php

Przetestuj działanie nowej reguły routingu w przeglądarce - uruchom stronę <http://localhost:8000/comments> (Rys. 12.7). **Pamiętaj o ponownym uruchomieniu serwera developerskiego za pomocą:**

php artisan serve



Rys. 12.7. Efekt działania metody index kontrolera CommentsController

3. Autoryzacja użytkowników i pierwsza migracja

Laravel pozwala na szybką implementację autoryzacji użytkowników. Laravel Breeze implementuje logowanie, rejestrację, zmianę hasła i weryfikację przez emaila. Dodatkowo Breeze oferuje gotowe strony ("profile" page), za pomocą których użytkownik może dokonać rejestracji, logowania oraz edycji swoich danych.

Laravel Breeze domyślnie generuje strony za pomocą widoków Blade, ale można skonfigurować go do wykorzystania widoków wykorzystujących Vue, React lub Inertia. Najprostsza konfiguracja korzysta z domyślnego silnika widoków Blade. Ten silnik oraz Breeze wskazaliśmy przy tworzeniu nowego projektu.

Aby poprawnie działało uwierzytelnienie użytkownika, należy wykorzystać (utworzone już w tym celu przez Laravel) elementy związane z zapisem danych użytkownika w bazie danych (domyślnie MySQL). W celu ustawienia połączenia do bazy danych, w pliku `.env` (w głównym folderze projektu) należy podać dane autoryzujące dostęp do bazy danych. Dla MySQL domyślne ustawienia w pliku `.env` (DB_CONNECTION, DB_HOST, itd.) są już gotowe (Rys. 12.8).

```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:WfyTxAehv1Pw/2SSGaX89H5CyO8Zz5gtsTAyoHvUCdY=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=lab12
15 DB_USERNAME=root
16 DB_PASSWORD=
```

Rys. 12.8. Domyślne ustawienia połączenia do bazy o nazwie *lab12* na serwerze MySQL

Korzystając z narzędzia *phpMyAdmin* utwórz nową bazę danych o nazwie **lab12** (taka nazwa bazy podana jest jako domyślna w ustawieniu **DB_DATABASE=lab12** (Rys. 12.8).

Do pracy z bazą danych wykorzystamy tzw. migracje. **Migracje** to pliki wykonujące określone operacje na bazie danych, takie jak np. tworzenie tabel.

Pliki migracji znajdują się w katalogu *database/migrations*. Laravel zadbał już o to, by utworzyć odpowiedni plik migracji dla tabeli użytkowników **users**. Gotowe pliki migracji można podejrzeć w projekcie (Rys. 12.9). Migracja z Rys. 12.9 tworzy tabelę **users**. Nazwy i typy pól tabeli (kolumn) zdefiniowano bezpośrednio z poziomu kodu w metodzie **up()** klasy o nazwie **CreateUserTable**, która dziedziczy po klasie **Migration**.


```
<?php

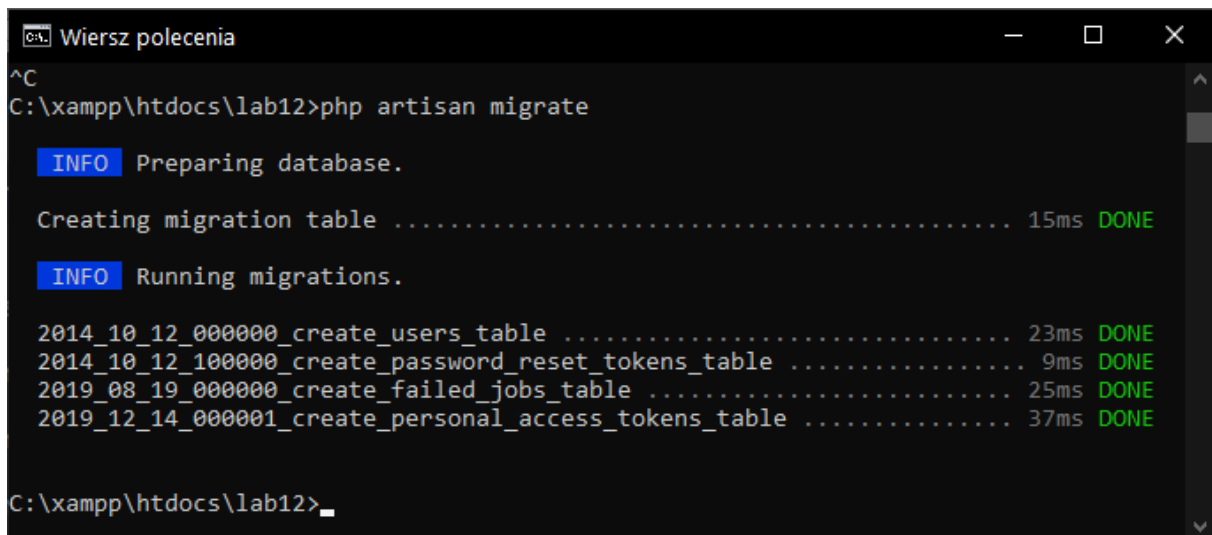
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
}
```

Rys. 12.9. Gotowa klasa migracji dla tabeli *users* (plik 2014_10_12_000000_create_users_table.php)

W celu fizycznego utworzenia tabeli *users* i innych zdefiniowanych tam tabel w naszej bazie danych *lab12*, wystarczy już tylko wywołać komendę migracji z poziomu wiersza poleceń (Rys. 12.10):

php artisan migrate



Rys. 12.10. Wynik działania polecenia migracji

Sprawdź w **phpMyAdmin** jakie tabele zostały utworzone po uruchomieniu migracji. Uruchom w przeglądarce stronę startową i zarejestruj (korzystając z przycisku **Register** na stronie głównej) nowego użytkownika, sprawdź efekt logowania i przetestuj możliwość edycji profilu. Przejrzyj jakie rekordy zostały dodane do tabeli **users** i **migrations**. Na koniec wyloguj użytkownika.

4. Widoki dla komentarzy

Kolejny etap to utworzenie widoków. W klasycznym wzorcu MVC, nazwy **widoków** są zwracane jako wynik działania metody (akcji) kontrolera. W przykładzie utworzymy stronę widoku z formularzem do wprowadzenia komentarza do książki gości.

Zadanie podzielimy na dwa etapy:

- a) utworzenie widoku dla książki gości,
- b) utworzenie logiki dodawania komentarzy przez użytkowników.

Otwórz plik kontrolera **CommentsController**, i zmodyfikuj jego metodę **index()** która zwracała prosty napis „**Hello Laravel**”. Teraz zmienimy to tak, aby zwracany był specjalny **widok** za pomocą funkcji **view()**. W tym celu zmień instrukcję **return** w metodzie **index()** kontrolera **CommentController** jak pokazano na Rys. 12.11.

```
use Illuminate\Http\Request;

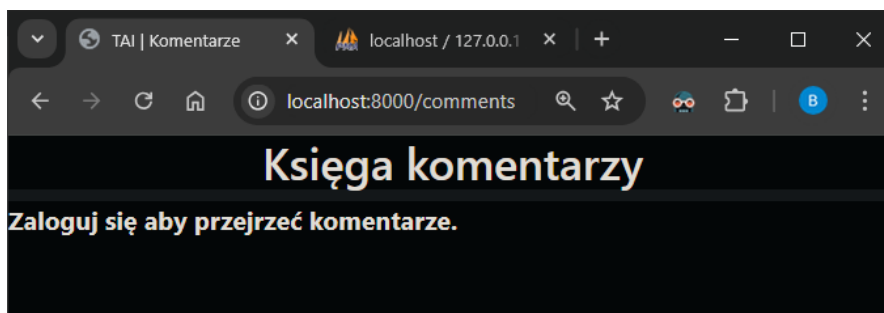
class CommentController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        return view("comments");
    }
}
```

Rys. 12.11. Modyfikacja metody **index()** kontrolera **CommentsController**

Następnie, w katalogu **resources/views** utwórz plik widoku **comments.blade.php** z treścią jak w załączonym do paczki z ćwiczeniem pliku. W Laravel widoki (dokumenty o strukturze HTML) są domyślnie obsługiwane przez silnik Blade. Finalna postać pliku będzie korzystała z silnika Blade do generowania dynamicznie zawartości na podstawie danych przesłanych z metod kontrolera. Na razie pokażemy statyczną zawartość komentarza testowego.

Ponownie uruchom widok dla komentarzy **'comments'** w przeglądarce (Rys. 12.12).





Rys. 12.12. Widok w przeglądarce strony *comments.blade.php*

Laravel pozwala łatwo określić sekcje na stronie, które mają być dostępne tylko dla zalogowanych użytkowników. Do pliku *comments.blade.php* dodano odpowiednie adnotacje *@auth* tuż przed widokiem dla zalogowanych użytkowników oraz *@endauth* tuż po sekcji widocznej dla zalogowanych. W adnotacjach *@guest* i *@endguest* można z kolei ustawić treść widoczną dla wszystkich użytkowników, np. z informacją o konieczności zalogowania (Rys. 12.13).

```
<body>
  <div class="table-container">
    <div class="title">
      <h3>Księga komentarzy</h3>
    </div>
    @auth
      <table data-toggle="table">
        <thead>
          <...6 lines />
        </thead>
        <...7 lines />
      </table>
      <br>
      <div class="footer-button">
        <a href="#" class="btn btn-secondary">Dodaj</a>
      </div>
    @endauth
  </div>

  @guest
    <div class="table-container">
      <b>Zaloguj się aby przejrzeć komentarze.</b>
    </div>
  @endguest
</body>
```

Rys. 12.13. Autoryzacja dostępu do elementów strony w pliku widoku

W celu założenia konta użytkownika korzystaliśmy z gotowego formularza rejestracji, który jest dostępny pod adresem <http://localhost:8000/register> (można nadać mu własny styl w odpowiednim pliku Blade, co będzie zrealizowane w późniejszych zadaniach). Mając zarejestrowanego użytkownika, można się już zalogować i przetestować ponownie działanie strony *comments*. Tym razem efekt powinien być taki jak na rys. 12.14.



#	Użytkownik	Data dodania	Komentarz
1	Demaio	10-01-2021	Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

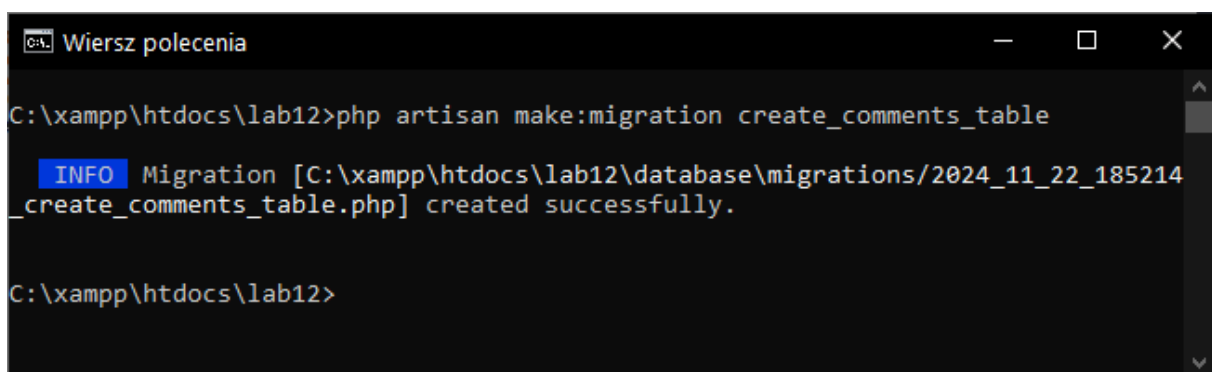
Rys. 12.14. Widok w przeglądarce strony *comments.blade.php* po zalogowaniu

Silnik widoków Blade oferuje wiele użytecznych metod szybkiego generowania widoków, co pozwala na łatwe tworzenie treści bezpośrednio w dokumencie HTML za pomocą różnych instrukcji sterujących, np. `@if`, `@endif`, `@foreach`, `@endforeach`. Te możliwości wykorzystamy w dalszych etapach ćwiczenia do pobrania i wyświetlenia komentarzy zapisanych w bazie danych.

5. Migracja tabeli dla komentarzy. Model Comments

Do przechowywania komentarzy potrzebujemy tabeli w bazie danych. W tym celu ponownie skorzystamy z odpowiedniej migracji, tyle, że tym razem sami musimy przygotować plik migracji. W wierszu poleceń, w katalogu projektu wpisz komendę, która utworzy taki plik dla tabeli komentarzy (Rys. 12.15):

```
php artisan make:migration create_comments_table
```



```

C:\xampp\htdocs\lab12>php artisan make:migration create_comments_table

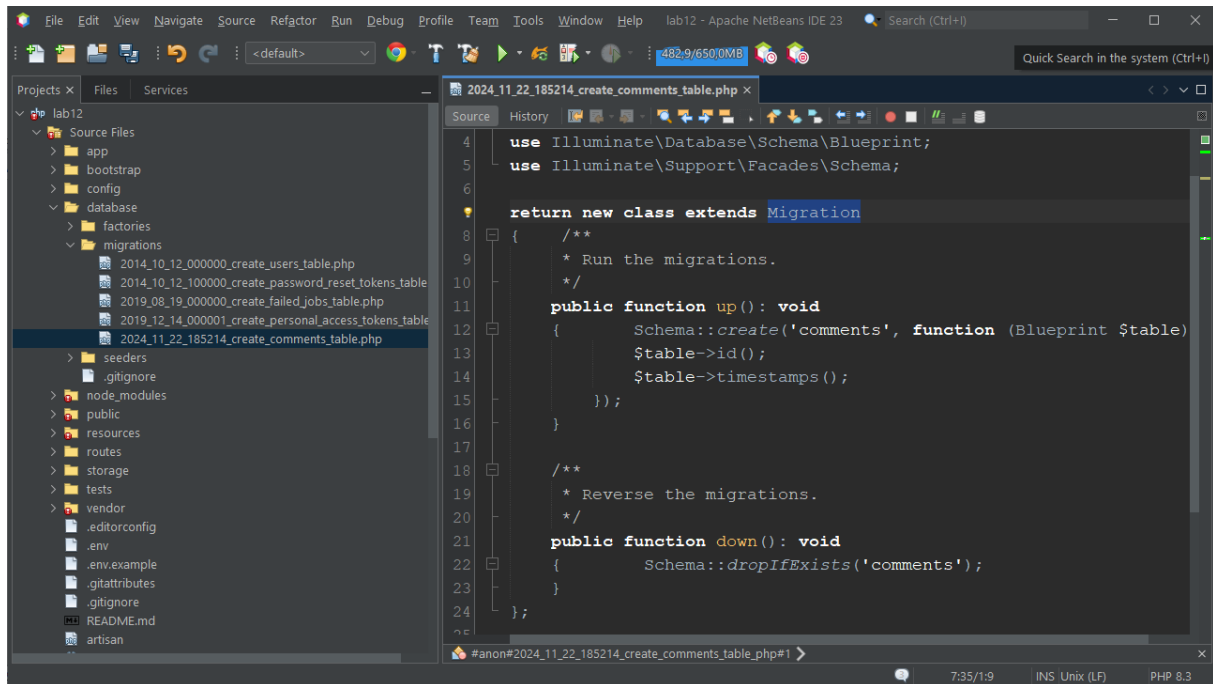
INFO Migration [C:\xampp\htdocs\lab12\database\Migrations\2024_11_22_185214_create_comments_table.php] created successfully.

C:\xampp\htdocs\lab12>
    
```

Rys. 12.15. Polecenie tworzenia pliku migracji dla tabeli komentarzy

Przejrzyj zawartość nowego pliku migracji, który znajduje się w katalogu *database/migrations* (Rys. 12.16). Warto zauważyć, że Laravel „odczytał” słowa kluczowe „create” i „table” z nazwy migracji oraz wykorzystał je do dodania podstawowych

właściwości (klucza głównego **id** oraz daty dodania i aktualizacji komentarza). Na podstawie nazwy migracji, utworzona tabela nazywa się **comments**.



Rys. 12.16. Plik migracji dla dla tabeli comments

W utworzonym pliku migracji dla tabeli **comments** w metodzie **up()** potrzebujemy jeszcze dwóch dodatkowych kolumn - **treści komentarza** oraz **id użytkownika**, które za pomocą relacji 1:1 połączymy z tabelą **users**.

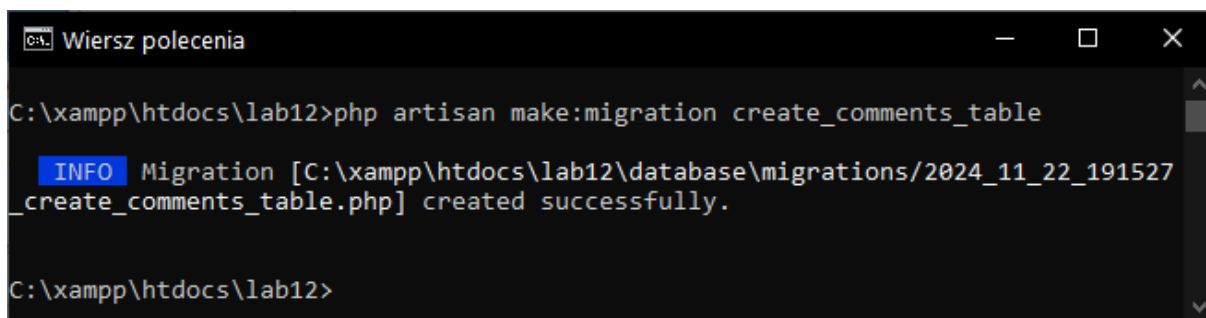
Dodaj najpierw brakujące pola w funkcji **up()** (Listing 12.1 i Rys. 12.17).

Listing 12.1. Dodatkowe kolumny w tabeli comments

```
public function up(): void
{
    Schema::create('comments', function (Blueprint $table) {
        $table->id();
        $table->bigInteger('user_id')->unsigned();
        $table->text('message');
        $table->foreign('user_id')->references('id')->on('user')
            ->onUpdate('cascade')->onDelete('cascade');
        $table->timestamps();
    });
}
```

Rys. 12.17. Dodanie brakujących pól i relacji w metodzie **up()**

I wykonaj ponownie migrację jak na Rys. 12.18.



```
C:\xampp\htdocs\lab12>php artisan make:migration create_comments_table

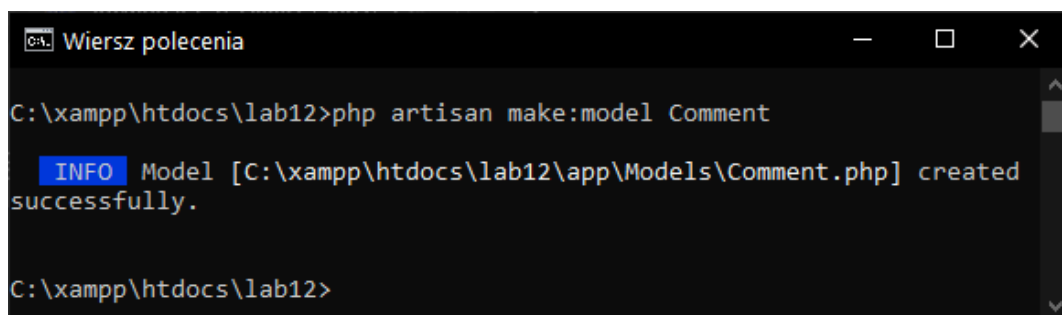
INFO Migration [C:\xampp\htdocs\lab12\database\Migrations\2024_11_22_191527_create_comments_table.php] created successfully.

C:\xampp\htdocs\lab12>
```

Rys. 12.18. Wykonanie migracji

Sprawdź w *PhpMyAdmin* efekt wykonania polecenia z Rys. 12.18. Następnie utwórz klasę modelu ***Comment*** (zwróć uwagę na liczbę pojedynczą w nazwie modelu), który będzie abstrakcyjną reprezentacją bytu bazodanowego dla tabeli komentarzy z poziomu aplikacji (Rys. 12.19):

```
php artisan make:model Comment
```



```
C:\xampp\htdocs\lab12>php artisan make:model Comment

INFO Model [C:\xampp\htdocs\lab12\app\Models\Comment.php] created successfully.

C:\xampp\htdocs\lab12>
```

Rys. 12.19. Polecenie tworzenia modelu

W wyniku wykonania polecenia z Rys. 12.19, w katalogu ***app/Models*** utworzył się plik ***Comment.php*** z definicją klasy ***Comment***. Dodaj do niego fragment (Rys. 12.20) definiujący relację jeden do jednego (jeden komentarz ma jednego autora).

Dodaj także import klasy *User*.

```

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use App\Models\User;

class Comment extends Model {

    public function user() {
        return $this->belongsTo(User::class);
    }

    use HasFactory;
}

```

Rys. 12.20. Dodanie relacji 1:1 do modelu *Comment*

Kolejny etap to powiązanie operacji wykonywanych na danych za pomocą klasy modelu *Comment* z akcjami kontrolera *CommentController* i stronami do prezentacji odpowiednich widoków dla komentarzy. W pliku kontrolera *CommentController.php* zaimportuj klasę modelu *Comment*, tak aby akcje kontrolera mogły wykonywać operacje na tym modelu (Rys. 12.21).

```

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Comment;

class CommentController extends Controller

```

Rys. 12.21. Dodanie importu klasy *Comment* do kontrolera *CommentController*

Zależność ta będzie wykorzystana nieco później. Teraz zajmiemy się utworzeniem formularza dodawania komentarzy.

6. Formularz dodawania nowego komentarza. Akcja *create* w kontrolerze.

Tak jak poprzednio, aby pokazać stronę (widok), należy najpierw w pliku *routes/web.php* dodać odpowiednią regułę routingu. Do obsługi żądania dodania nowego komentarza potrzebne będą dwie reguły routingu (Rys. 12.22), związane z obsługą dwóch metod HTTP:

- *get*, za pomocą której zostanie wyświetlony widok formularza dodawania komentarza (co zrealizuje metoda *create()* kontrolera *CommentController*);
- *post*, która odbierze żądanie z parametrami wysyłane z formularza i przekaże je do metody *store()* kontrolera *CommentController*.

Rys. 12.22 przedstawia dodane reguły routingu w pliku *web.php*. Zauważ, że reguły mają przypisaną nazwę za pomocą *->name('...')*. Nadanie nazw regułom routingu jest bardzo wygodne w późniejszym odwoływaniu się do nich.



```
Route::get('/comments', [CommentController::class, 'index'])->name('comments');
Route::get('/create', [CommentController::class, 'create'])->name('create');
Route::post('/create', [CommentController::class, 'store'])->name('store');
```

Rys. 12.22. Dodatkowe reguły routingu w routes/web.php

Kolejny etap to uzupełnienie metod w kontrolerze **CommentController** do obsługi dodania nowego komentarza do bazy danych. W istniejącej (pustej) metodzie **create()** kontrolera **CommentController**, utwórz nowy obiekt **\$comment** oraz dodaj przekierowanie do widoku z formularzem **commentsForm.blade.php** (Rys. 12.23). Przekazywanie zmiennych z kontrolera do widoków można realizować za pomocą drugiego parametru (typu tablicy asocjacyjnej) funkcji **view()**. W przykładzie przekazywany jest tylko jeden element - obiekt **\$comment**.

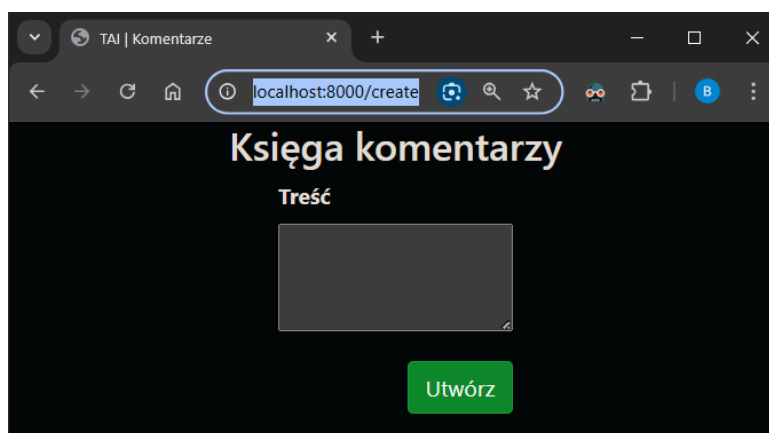
```
/**
 * Show the form for creating a new resource.
 */
public function create()
{
    $comment = new Comment;
    return view('commentsForm', ['comment'=>$comment]);
}
```

Rys. 12.23. Instrukcje w metodzie **create()** kontrolera **CommentsController**.

Następnie, w katalogu **resources/views** utwórz plik widoku **commentsForm.blade.php** z treścią jak w załączonym do ćwiczenia pliku (fragment pliku pokazany jest na Rys. 12.25). Na stronie widoku **commentsForm.blade.php** zwróć uwagę na obsługę błędów (zmienna **\$errors** i **{{ csrf_field() }}**) i inne elementy specyficzne dla silnika szablonów Blade:

- instrukcje sterujące takie jak **@foreach-@endforeach**,
- wartość parametru **action="{{ route('store') }}"** w formularzu, gdzie odwołanie następuje do nazwanej reguły routingu,
- parametr **class="form-group{{ \$errors->has('message')?'has-error':'' }}"** dla elementu **div**.

Widok w przeglądarce pod adresem <http://localhost:8000/create> prezentuje Rys. 12.24.



Rys.12.24. Widok formularza dodawania komentarza


```

<div class="table-container">
  <div class="title"> <h3>Księga komentarzy</h3> </div>
  @if ($errors->any())
  <div class="alert alert-danger">
    <ul>
      @foreach ($errors->all() as $error)
      <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
  @endif
  <div class="box box-primary ">
    <!-- /.box-header -->
    <!-- form start -->
    <form role="form" action="{{ route('store') }}" id="comment-form"
      method="post" enctype="multipart/form-data" >
      {{ csrf_field() }}
      <div class="box">
        <div class="box-body">
          <div class="form-group{{ $errors->has('message')?'has-error':'' }}" id="roles_box">
            <label><b>Treść</b></label> <br>
            <textarea name="message" id="message" cols="20" rows="3" required></textarea>
          </div>
        </div>
        <div class="box-footer"><button type="submit" class="btn btn-success">Utwórz</button>
        </div>
      </form>
    </div>
  </div>

```

Rys. 12.25. Fragment pliku widoku commentsForm.blade.php

Aby ze strony z listą komentarzy (/comments) przejść do strony dodawania nowego komentarza - do pliku *comments.blade.php*, po znaczniku zamykającym tabelę </table>, dodaj blok <div> z hiperłączem: jak na Listingu 12.2.

Listing 12.2. Dodatkowy przycisk-hiperłącze na stronie comments.blade.php

```

<div class="footer-button">
  <a href="{{ route('create') }}" class="btn btn-secondary">Dodaj</a>
</div>

```

7. Walidacja formularza

Przed zapisem do bazy danych, podana treść komentarza powinna zostać sprawdzona. Można skorzystać z walidacji po stronie klienta za pomocą HTML, ale walidację koniecznie trzeba powtórzyć po stronie serwera. Laravel wprowadza bardzo zaawansowany moduł walidacji danych przesyłanych w żądaniach. Aby zrozumieć jak działa ten mechanizm – trzeba wiedzieć co się dzieje z danymi z formularza po ich przesłaniu w wyniku kliknięcia przycisku „Utwórz”. Dane z formularza są wysyłane metodą POST protokołu HTTP i odbierane przez wskazany w routingu kontroler Laravla jako obiekt *Request*. Następnie trafiają do wskazanej w regule routingu metody kontrolera (w naszym przypadku jest to metoda *store()* kontrolera *CommentController*). Metoda *store()*, jako parametr wejściowy otrzymuje właśnie obiekt *Request*. Aby przeprowadzić walidację danych przesłanych z formularza należy wykorzystać funkcję *validate()*, w której można zadeklarować warunki walidacji w postaci tablicy asocjacyjnej.

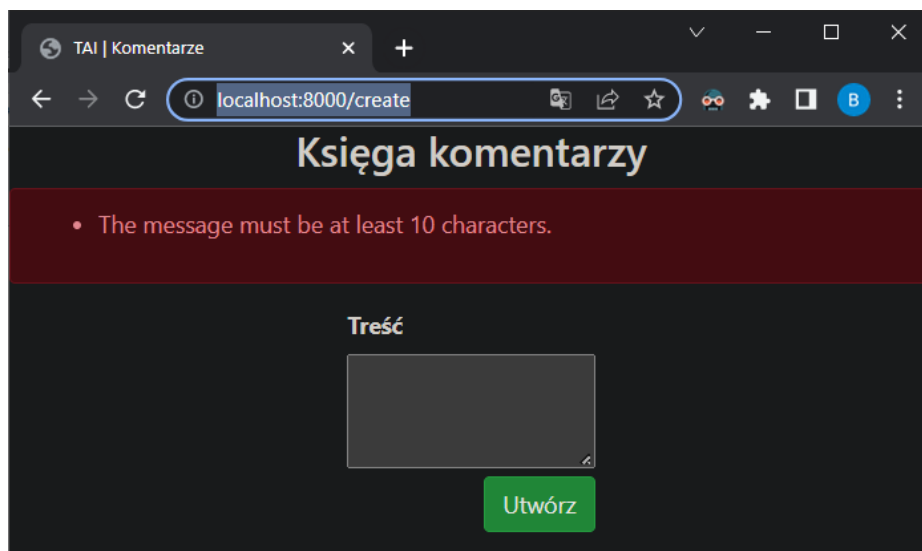


Uzupełnij metodę `store(Request $request)` kontrolera `CommentsController`, kodem jak na Listingu 12.3.

Listing 12.3. Metoda store z walidacją

```
public function store(Request $request)
{
    // Podstawowa walidacja formularza:
    $this->validate($request, [
        'message' => 'required|min:10|max:255',
    ]);
}
```

Przetestuj działanie walidatora (Rys. 12.26).



Rys. 12.26. Działanie walidacji

8. Zapis komentarzy do bazy danych i wyświetlenie danych z bazy

Jeśli dane pobrane z formularza są prawidłowe i użytkownik jest zalogowany, to można jego komentarz dodać do bazy danych. Uzupełnimy naszą metodę `store()` dodatkowymi instrukcjami jak na Rys. 12.27. Zwróć uwagę na komentarze w kodzie, które opisują cały ten proces. Korzystając z narzędzia *Eloquent* nie musimy samodzielnie formułować zapytania typu INSERT. Tzeba tylko wywołać metodę `save()` na obiekcie klasy *Comment*. *Eloquent*, korzystając z pliku migracji dla tabeli *comments*, sam odwzoruje dane obiektu *\$comment* na rekord w tabeli *commentst* w bazie danych.

```

/**
 * Store a newly created resource in storage.
 */
public function store(Request $request) {    // Podstawowa walidacja formularza:
    $this->validate($request, [
        'message' => 'required|min:10|max:255',
    ]);
    if (\Auth::user() == null) {
        redirect()->route('login'); //jeśli użytkownik nie jest zalogowany
    }
    //Użytkownik zalogowany
    //utwórz nowy, pusty obiekt komentarz:
    $comment=new Comment();
    //do pola user_id komentarza przypisz id aktualnie zalogowanego użytkownika
    $comment->user_id = \Auth::user()->id;
    //do pola message przypisz pobrany z formularza (request)
    //i zwalidowaną treść komentarza:
    $comment->message = $request->message;
    //zapisz, korzystając z Eloquent, nowy komentarz
    //do tabeli comments w bazie danych za pomocą polecenia save()
    if ($comment->save()) {
        //jeśli się udało - przekieruj na akcję controlera
        //obsługującą pokaz wszystkich komentarzy po aktualizacji
        return redirect('comments');
    }
    else{
        //jeśli się nie udało - wróć do
        //formularza dodawania komentarza
        return view('commentsForm');
    }
}

```

Rys. 12.27. Zapis komentarza w metodzie *store()* kontrolera *CommentsController*

Przetestuj działanie metody *store()* - za pomocą formularza dodawania, wprowadź nowy komentarz i sprawdź w *PhpMyAdmin*, czy został on prawidłowo dodany do tabeli *comments*.

Ostatni etap to wyświetlenie wszystkich komentarzy pobranych z bazy danych. W tym celu w metodzie *index()* kontrolera *CommentController* zmodyfikuj kod jak na Rys. 12.28. Wykorzystano tu klasę modelu *Comment*, która po stronie aplikacji reprezentuje obiekt komentarza odpowiadający pojedynczemu rekordowi tabeli *comments* w bazie danych. Zmienna *\$comments* jest rezultatem zapytania wykonanego za pomocą metody *get()* narzędzia *Eloquent* i zawiera tablicę obiektów klasy *Comment*, która reprezentuje wszystkie, posortowane alfabetycznie, rekordy tabeli *comments*. Ta tablica zostaje przekazana do widoku, który wyświetli wszystkie komentarze z tabeli na stronie (Rys. 12.28).

```

/**
 * Display a listing of the resource.
 */
public function index() {
    $comments = Comment::orderBy('created_at', 'asc')->get();
    return view('comments', ['comments' => $comments]);
}

```

Rys. 12.28. Pobranie wszystkich komentarzy z bazy danych w *CommentController*

Dostęp do danych w Laravel zrealizowany został za pomocą narzędzia **Eloquent**. **Eloquent** jest warstwą ORM (ang. Object-Relation Mapping) wykorzystywaną do mapowania obiektów aplikacji na rekordy odpowiadających im tabel w bazie danych (i odwrotnie). **Eloquent** tłumaczy operacje na obiektach aplikacji na zapytania **SQL** i nie ogranicza się przy tym do jednego systemu bazodanowego (korzysta z interfejsu PDO). Metoda **store()** kontrolera wywołała metodę **save()** do utrwalenia obiektu **Comment** w tabeli **comments** a teraz w metodzie **index()** (Rysunek 12.28) do pobrania danych z bazy danych wykorzystwała metodę **get()**.

Za pomocą metod narzędzia **Eloquent**, instrukcja:

```
Comment::orderBy('created_at', 'asc')->get();
```

zostanie przekształcona na zapytanie SQL do tabeli **comments** postaci:

```
SELECT * FROM comments ORDER BY created_at ASC
```

Eloquent można wykorzystać do odczytywania wprost wartości **obiektów** (modeli) po stronie aplikacji pozostających ze sobą w relacji, a co więcej nakładać na nie odpowiednie warunki. Oznacza to, że programista może pominąć zapytania SQL oparte o JOIN i warunki w nich zagnieżdżone, i skorzystać z metod **Eloquent** typu: **whereHas**, **whereDoesntHas**, które pozwolą na prostsze dodanie warunków na struktury danych **objektowych**. Przykładowo, jeśli zamiast wszystkich komentarzy, trzeba zwrócić tylko komentarze użytkowników, którzy mają w swoim adresie *email*, np. imię *Karolina* - to dzięki zadeklarowaniu relacji 1:1 w modelu **Comment**, wystarczy, że sformułujemy zapytanie **Eloquent** postaci:

```

$comments = Comment::whereHas('user', function ($query) {
    $query->where('email', 'like', '%karolina%');
})->get();

```

Dla porównania, czyste zapytanie SQL ma postać:

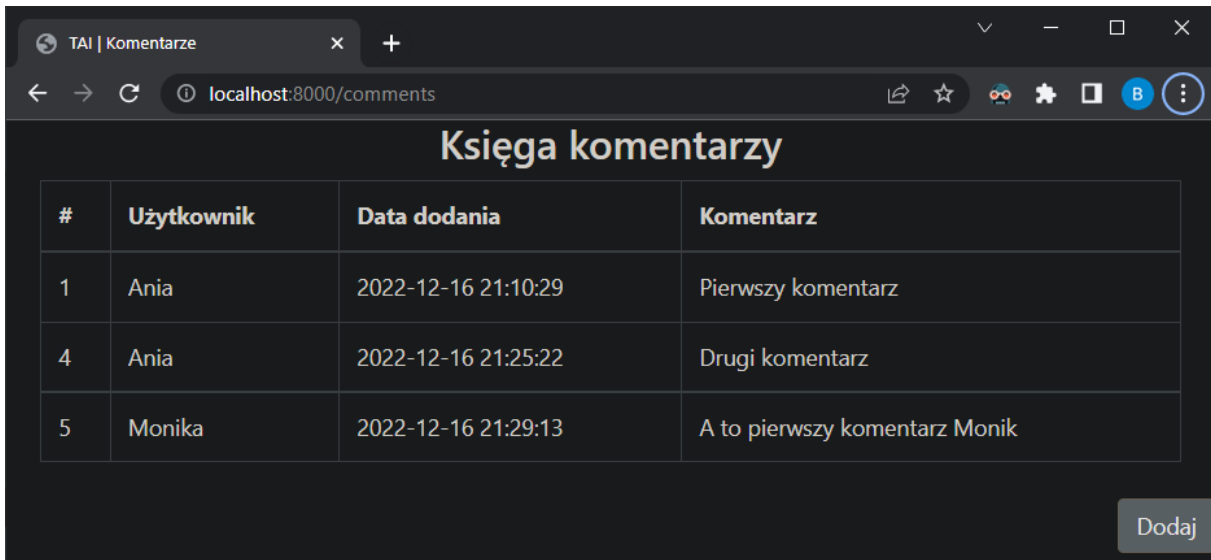
```
"select * from `comments` where exists (select * from `users` where `comments`.`user_id` = `users`.`id` and `email` like '%karolina%')";
```

Odczytanie danych z tablicy komentarzy w widoku **comments.blade.php** można zrealizować za pomocą **zamiany kodu** w bloku **<tbody>**, jak na Rys. 12.29.

```
<tbody>
    @foreach($comments as $comment)
        <tr>
            <td>{{ $comment->id }}</td>
            <td>{{ $comment->user->name }}</td>
            <td>{{ $comment->created_at }}</td>
            <td>{{ $comment->message }}</td>
        </tr>
    @endforeach
</tbody>
```

Rys. 12.29. Zmiana zawartości bloku `<tbody>` w widoku `comments.blade.php` tak aby wyświetlane były dane z bazy a nie statyczna treść tabeli

Widok strony z komentarzami pobranymi z bazy danych przedstawia Rys. 12.30. Widok ten jest dostępny jedynie dla zalogowanych użytkowników.



#	Użytkownik	Data dodania	Komentarz
1	Ania	2022-12-16 21:10:29	Pierwszy komentarz
4	Ania	2022-12-16 21:25:22	Drugi komentarz
5	Monika	2022-12-16 21:29:13	A to pierwszy komentarz Monik

Dodaj

Rys. 12.30. Widok strony z komentarzami

LABORATORIUM 12. CZĘŚĆ 2. IMPLEMENTACJA WZORCA PROJEKTOWEGO MVC W PRZYKŁADOWEJ APLIKACJI TYPU CRUD

Cel laboratorium:

Celem zajęć jest przygotowanie aplikacji w oparciu o wytyczne wzorca projektowego MVC i poznanie podstawowych elementów implementacji tego wzorca w języku PHP na przykładzie szkieletu programistycznego Laravel.

Zakres tematyczny zajęć:

Rozszerzenie funkcjonalności aplikacji utworzonej w zadaniu 12.1. o elementy walidacji danych z formularza oraz obsługa akcji usuwania i modyfikacji komentarzy.

Zadanie 12.2. Dodatkowe funkcjonalności aplikacji ‘Księga komentarzy’

1. Własny styl dla formularza logowania/rejestracji

Widok dla formularzy rejestracji i logowania znajduje się w katalogu **resources/views/auth/** odpowiednio w plikach **login.blade.php** oraz **register.blade.php**. W celu dodania własnego stylu dla formularzy, wystarczy je zmodyfikować. Oba pliki wykorzystują szablon **resources/layouts/app.blade.php**. Naniesione tam zmiany będą wykorzystane na wszystkich stronach korzystających z tego szablonu. Stosowanie takich szablonów pozwala zachować spójny wygląd całego projektu oraz umożliwia łatwe definiowanie kolejnych elementów wyglądu aplikacji.

2. Komunikaty o błędach walidacji w j. polskim

Domyślnie aplikacja Laravel nie zawiera katalogu **lang**. Aby dostosować pliki językowe Laravel lub utworzyć własne, należy najpierw utworzyć katalog **lang** za pomocą polecenia **artisan**:

```
php artisan lang:publish
```

Polecenie to utworzy katalog **lang** i opublikuje domyślny (en) zestaw plików językowych używanych przez Laravel.

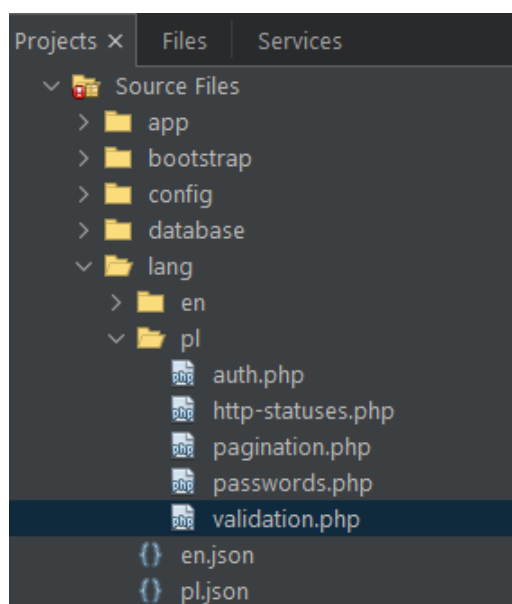
W celu dodania komunikatów w języku polskim, można skorzystać z dodatkowego modułu **Laravel Lang**, który pozwala na szybkie tłumaczenia komunikatów, domyślnie ustawionych na język angielski. W tym celu należy z wiersza poleceń wydać kolejno polecenia, zgodnie z instrukcją instalacji https://laravel-lang.com/basic-usage.html#update_locales:

```
composer require laravel-lang/common
```

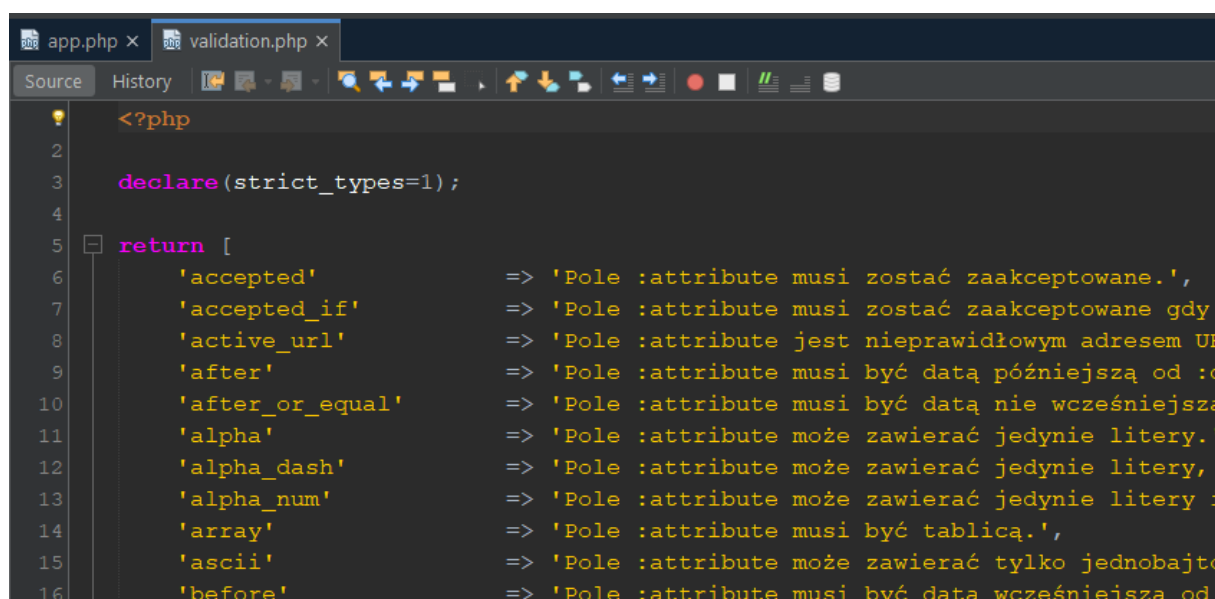
```
php artisan lang:add pl
```

```
php artisan lang:update
```

Po instalacji tego modułu do katalogu **lang** został dodany podkatalog **pl** (Rys. 12.32) zawierający pliki (analogiczne jak w podkatalogu **en**, np. **validation.php**) z komunikatami w języku polskim (Rys. 12.32), które w miarę potrzeby można zmienić.



Rys. 12.32. Zawartość katalogu lang



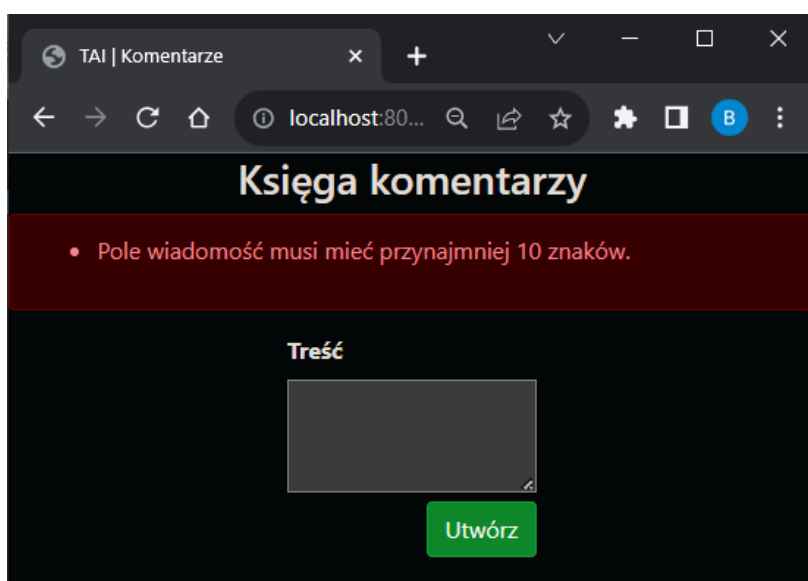
Rys. 12.33. Fragment komunikatów z pliku validation.php

Aby komunikaty o błędach były wyświetlane w języku polskim, należy jeszcze zmodyfikować wartość parametru **locale** w pliku **config/app.php** (Rys. 12.34).


```
/*
|-----|
| Application Locale Configuration |
|-----|
|
| The application locale determines the default locale that will be used
| by the translation service provider. You are free to set this value
| to any of the locales which will be supported by the application.
|
|*/
'locale' => 'pl',
```

Rys. 12.34. Ustawienie w pliku konfiguracyjnym domyślnego języka polskiego

Przetestuj działanie walidatora (po zrestartowaniu serwera i zalogowaniu użytkownika) w połączeniu z modułem Laravel Lang (Rys. 12.35).



Rys. 12.35. Komunikat walidacji w języku polskim

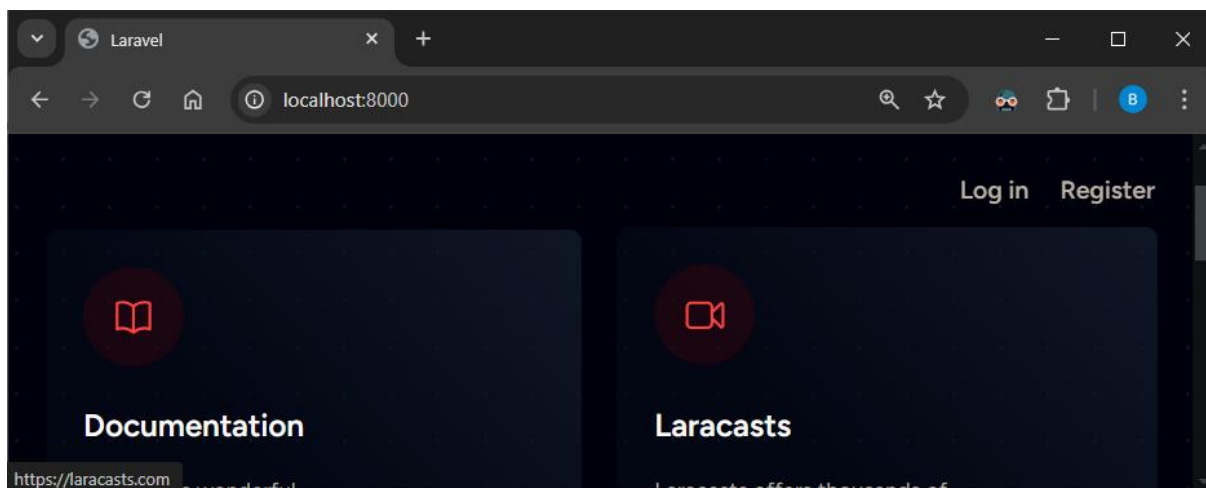
Laravel posiada bardzo prosty, ale jednocześnie dający bardzo duże możliwości, moduł do tłumaczeń elementów strony. Warty uwagi jest fakt, że Laravel sam dba o ewentualny brak tłumaczenia w danym języku. Jeśli aplikacja ma możliwość tłumaczenia na kilka języków, ale nie każde tłumaczenie jest w 100% ukończone, to dzięki opcji:

'fallback_locale' => 'en',

domyślnym językiem jest język angielski. Dlatego w przypadku braku tłumaczenia dla jakiegoś klucza, zostanie automatycznie zastosowany zwrot w języku angielskim.

3. Połączenie ze stroną domową Laravel

W przykładowej aplikacji Laravel, domyślny widok dla strony głównej (*welcome.blade.php*) jest postaci jak na Rys. 12.36.



Rys. 12.36. Domyślny widok strony *welcome.blade.php*

W aplikacji tworzonej na poprzednim laboratorium były definiowane dodatkowe reguły routingu. Sprawdź, czy Twoje reguły w pliku *routes/web.php* są zgodne z tymi na Rys. 12.37.

```
Route::get('/', function () {
    return view('welcome');
});

Route::get('/comments', [CommentController::class, 'index'])->name('comments');
Route::get('/create', [CommentController::class, 'create'])->name('create');
Route::post('/create', [CommentController::class, 'store'])->name('store');

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');

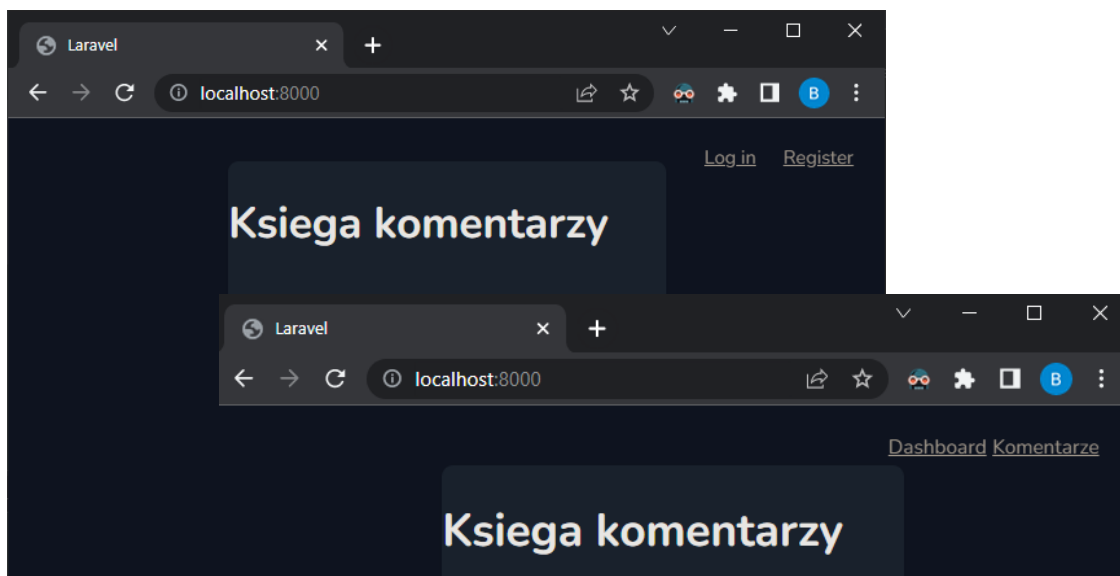
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});

require __DIR__.'/auth.php';
```

Rys. 12.37. Reguły routingu w pliku *routes/web.php*

Punktem wejścia do projektu (*Route::get('/')*), jest widok *welcome.blade.php*. Ponadto plik *routes/web.php* zawiera definicje tras już utworzonych w poprzedniej części laboratorium dla kontrolera *CommentController* i reguły automatycznie dodane do obsługi rejestracji i logowania (dodane regułą grupową *Route::middleware*).

Jeśli reguły routingu są już takie same, uprość i zmodyfikuj stronę *welcome.blade.php* do postaci podobnej jak na rysunku 12.38.



Rys. 12.38. Przykładowy wygląd strony *welcome.blade.php* po modyfikacjach z menu dla niezalogowanego i zalogowanego użytkownika

Kolejnym etapem będzie podłączenie menu pod *Księgę komentarzy*, celem łatwego powrotu do strony głównej. Wykorzystaj możliwości szablonów Blade i w folderze *resources/views/layouts* utwórz element nawigacyjny strony *navbar.blade.php* z zawartością jak na Listingu 12.5.

Listing 12.5. Prosty szablon *navbar.blade.php*

```
<nav class="navbar navbar-expand-md navbar-light shadow-sm">
  <div class="container">
    <a href="{{ url('/') }}"> Home </a>
    <a href="{{ route('comments') }}"> Komentarze </a>

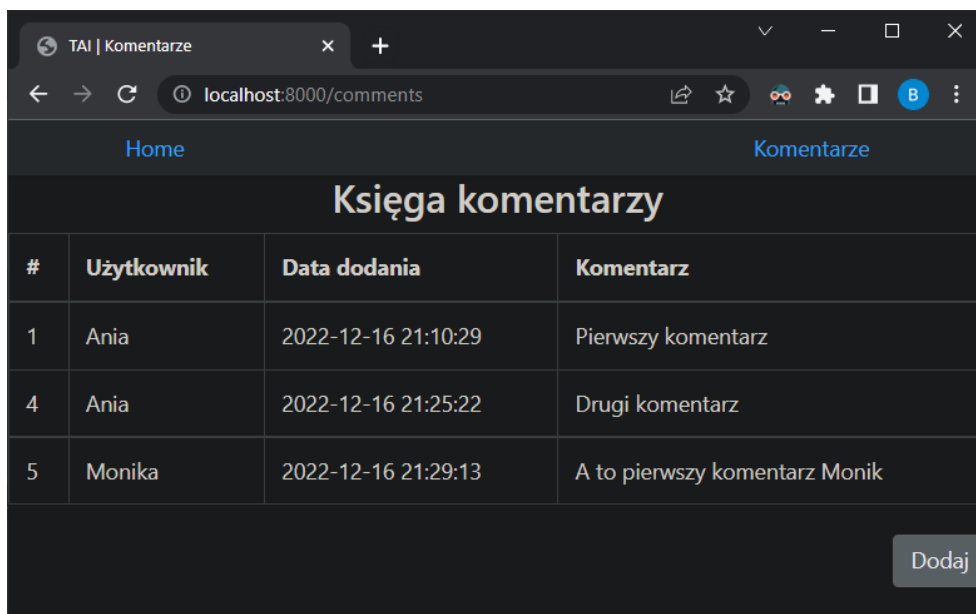
    <!-- Authentication Links -->
    @guest
      <a href="{{ route('login') }}">Login</a>
    @endguest

  </div>
</nav>
```

Element z nawigacją dołącz do pliku *resources/views/comments.blade.php* za pomocą *include* (zaraz po elemencie *body*):

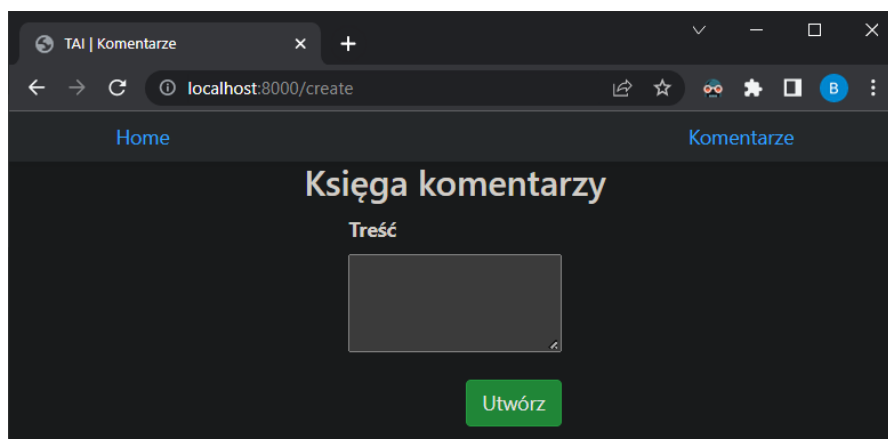
```
@include('layouts.navbar')
```

Pozwoli to załadować zawartość menu z szablonu do strony z komentarzami (Rys. 12.39).



Rys. 12.39. Widok strony komentarzy z menu

W celu zachowania spójności na stronach aplikacji, element nawigacyjny należy w ten sam sposób dodać do *commentsForm.blade.php* (Rys. 12.40).



Rys. 12.40. Widok formularza dodawania komentarzy z menu

4. Usuwanie komentarzy z bazy danych

Kolejnym etapem prac nad projektem jest dodanie możliwości kasowania swoich komentarzy przez autorów wpisów. Dodajmy najpierw kolejną regułę routingu do pliku *routes/web.php*:

```
Route::get('/delete/{id}',
    [CommentController::class, 'destroy']->name('delete');
```

Reguła ta definiuje obsługę URL postaci *delete/id* przez metodę *destroy(\$id)* (z parametrem *id*) kontrolera *CommentController*. Przy usuwaniu komentarza ważne będzie sprawdzenie, czy użytkownik usuwa swój komentarz (czy ma do tego uprawnienia). W pliku kontrolera *app/Http/Controllers/CommentController.php*, uzupełnij istniejącą już tam metodę *destroy* (Listing 12.6).

Listing 12.6. Dodatkowa metoda delete w CommentsController.php

```

public function destroy($id)
{
    ../Znajdź komentarz o danych id:
    $comment = Comment::find($id);
    //Sprawdź czy użytkownik jest autorem komentarza:
    if(\Auth::user()->id != $comment->user_id)
    {
        return back();
    }
    if($comment->delete()){
        return redirect()->route('comments');
    }
    else return back();
}

```

Aby użytkownik mógł wykonać akcję usunięcia **swojego** wpisu z poziomu interfejsu graficznego, dodaj jeszcze przycisk kasowania komentarza, który powinien wyświetlać się tylko gdy dany komentarz może być skasowany. W tym celu, w widoku *resources/views/comments.blade.php*, w pętli wyświetlającej tabelę komentarzy, dodaj, w ostatniej komórce tabeli, instrukcję *@if* jak na Listingu 12.7.

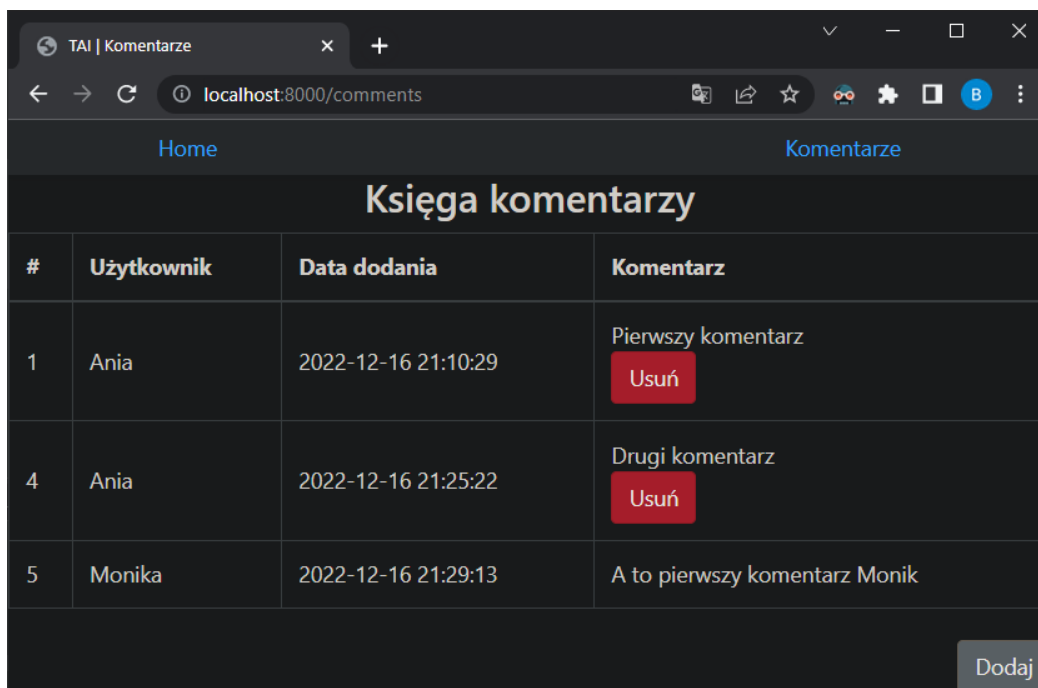
Listing 12.7. Dodatkowy blok @if w pętli widoku comments.blade.php

```

<tbody>
    @foreach($comments as $comment)
        <tr>
            <td>{{$comment->id}}</td>
            <td>{{$comment->user->name}}</td>
            <td>{{$comment->created_at}}</td>
            <td>{{$comment->message}}
                <br /> @if($comment->user_id == \Auth::user()->id)
                    <a href="{{ route('delete', $comment->id) }}"
                        class="btn btn-danger btn-xs"
                        onclick="return confirm('Jesteś pewien?')"
                        title="Skasuj"> Usuń
                </a>
                @endif
            </td>
        </tr>
    @endforeach
</tbody>

```

Wygląd strony po modyfikacjach przedstawia rysunek 12.41. Przetestuj działanie przycisku *Usuń. Sprawdź czy komentarz został usunięty z bazy danych.*



#	Użytkownik	Data dodania	Komentarz
1	Ania	2022-12-16 21:10:29	Pierwszy komentarz Usuń
4	Ania	2022-12-16 21:25:22	Drugi komentarz Usuń
5	Monika	2022-12-16 21:29:13	A to pierwszy komentarz Monik

[Dodaj](#)

Rys. 12.41. Widok strony komentarza z przyciskiem do usuwania

5. Edycja komentarzy

Ostatni etap to dodanie możliwości edycji treści komentarza. Zaczynij od dodania kolejnych reguł routingu koniecznych do obsługi modyfikacji danych, jak na Listingu 12.8.

Listing 12.8. Dodatkowe reguły routingu

```
Route::get('/edit/{id}', [CommentController::class, 'edit'])->name('edit');
Route::put('/update/{id}', [CommentController::class, 'update'])->name('update');
```

W nowych regułach dodano odniesienie do już istniejącej w kontrolerze **CommentController** metody **edit**, odpowiedzialnej za załadowanie formularza do edycji komentarza. Kolejna metoda **update** tego kontrolera, odpowiada za odebranie wysłanych z formularza edycji danych zmodyfikowanego komentarza oraz za zmodyfikowanie ich w bazie danych. W kontrolerze **CommentsController** uzupełnij te istniejące, ale jeszcze puste metody zgodnie z kodem na Listingu 12.9.

Zwróć uwagę w jaki sposób zaimplementowano obie metody (tutaj także wykorzystany został Eloquent do wykonania operacji na bazie danych).

Listing 12.9. Uzupełnione funkcje edit i update

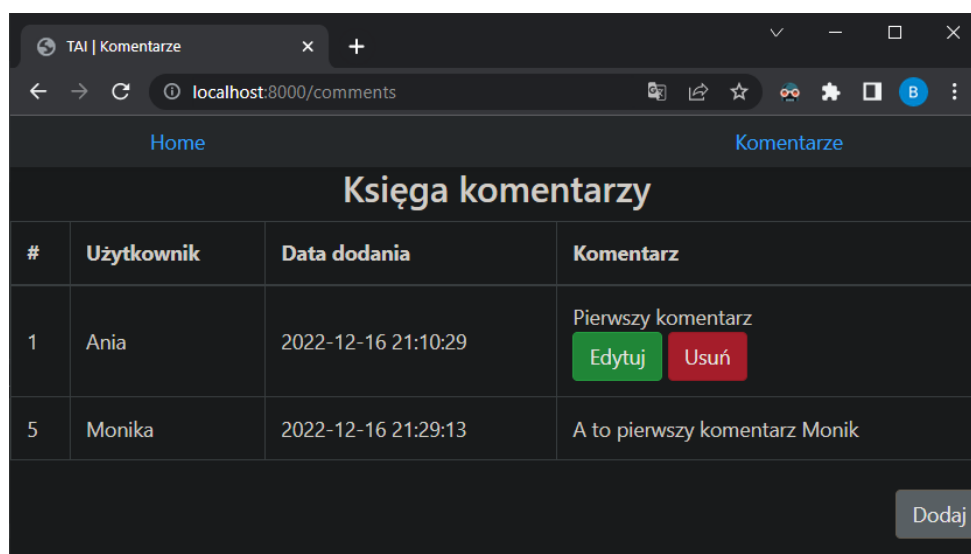
```
public function edit($id) {
    $comment = Comment::find($id);
    //Sprawdzenie czy użytkownik jest autorem komentarza
    if (\Auth::user()->id != $comment->user_id) {
        return back()->with(['success' => false, 'message_type' => 'danger',
            'message' => 'Nie posiadasz uprawnień do przeprowadzenia tej operacji.']);
    }
    return view('commentsEditForm', ['comment'=>$comment]);
}
```

```
public function update(Request $request, $id)
{
    $comment = Comment::find($id);
    //Sprawdzenie czy użytkownik jest autorem komentarza
    if(\Auth::user()->id != $comment->user_id)
    {
        return back()->with(['success' => false, 'message_type' => 'danger',
            'message' => 'Nie posiadasz uprawnień do przeprowadzenia tej operacji.']);
    }
    $comment->message = $request->message;
    if($comment->save()) {
        return redirect()->route('comments');
    }
    return "Wystąpił błąd.";
}
```

Widok listy komentarzy należy uzupełnić dodatkowym przyciskiem **Edytuj**. W pliku *comments.blade.php* w bloku `@if`, przed przyciskiem **Usuń**, dodaj kod:

```
<a href="{{ route('edit', $comment) }}" class="btn btn-success btn-xs"
    title="Edytuj"> Edytuj </a>
```

Po uruchomieniu strony z komentarzami, dla zalogowanego użytkownika, otrzymamy teraz widok jak na Rys. 12.42.



Rys. 12.42. Widok strony komentarza z przyciskami **Edytuj** i **Usuń**

Potrzebujemy jeszcze tylko formularza do modyfikacji wskazanego komentarza - pliku *commentsEditForm.blade.php* (Listing 12.10).

Listing 12.10. Formularz do edycji komentarza - fragment pliku *commentsEditForm.blade.php*

```
<form role="form" id="comment-form" method="post"
    action="{{ route('update', $comment) }}">
    {{ csrf_field() }}
    <input name="_method" type="hidden" value="PUT">
    <div class="box">
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny




```
<div class="box-body">
  <div class="form-group{{ $errors->has('message') ? ' has-error' : '' }}"
    id="roles_box">
    <label><b>Treść</b></label><br>
    <textarea name="message" id="message" cols="30" rows="10" required>
      {{ $comment->message }}
    </textarea>
  </div>
</div>
</div>
<div class="box-footer">
  <button type="submit" class="btn btn-success">Zapisz</button>
</div>
</form>
```

Zauważ, że formularz jest analogiczny do formularza dodawania komentarza.

Są tam 3 różnice (oznaczone kolorem czerwonym na Listingu 12.10):

- wartość atrybutu **action** formularza (dane komentarza przesłane metodą POST mają teraz trafić do metody **update**, zgodnie z regułą routingu o nazwie 'update'),
- do pola **textarea** wprowadzono starą treść wskazanego do edycji komentarza,
- dodano ukryte pole wskazujące metodę PUT protokołu HTTP.

UWAGA!

POST jest używany do tworzenia zasobu, PUT do tworzenia lub aktualizowania zasobu.

Ponieważ formularze HTML obsługują tylko metody POST i GET, metody PUT i DELETE można symulować przez dodanie do formularza ukrytego pola z atrybutem **name="_method"**.

Sprawdź efekt działania przycisku **Edytuj**.

Przy modyfikacji zasobu można też klasycznie zastosować metodę POST – zmień odpowiednio regułę w pliku **web.php**, usuń ukryte pole z formularza i przetestuj ponownie działanie edycji.

Sprawdź działanie całej aplikacji, ewentualnie dokonaj (według uznania) modyfikacji w wyglądzie i w nawigacji pomiędzy stronami.