

Laboratorium zostało przygotowane w wersji Laravel 6. Do wykonania przykładowej aplikacji potrzebny będzie **XAMPP** oraz menedżer pakietów **Composer**. Bardziej szczegółowe wprowadzenie do instalacji i prezentacja możliwości frameworka dostępne jest na platformie Moodle w postaci materiałów wykładowych.

*Celem laboratorium jest wprowadzenie do tworzenia prostej aplikacji typu **CRUD** z autoryzacją użytkowników. Tworzona aplikacja ma za zadanie zarządzanie komentarzami (ang. **Comments**) tworzonymi przez użytkowników.*

1. Utworzenie pierwszej aplikacji w Laravel

W wierszu poleceń wydaj komendę, która pobierze instalator Laravel:

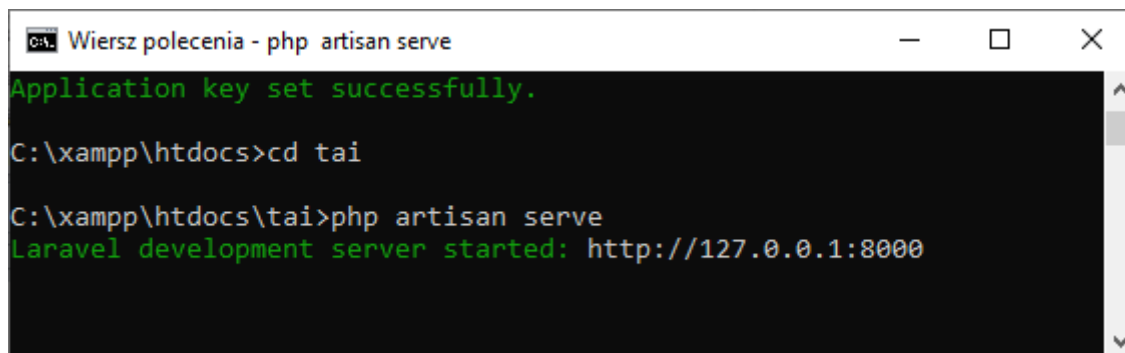
```
composer global require laravel/installer
```

Następnie utwórz nowy projekt Laravel w lokalizacji np. `C:\xampp\htdocs`. Stworzenie nowego projektu o nazwie *tai* możesz zrealizować za pomocą polecenia (z poziomu katalogu, w którym ma się znaleźć projekt):

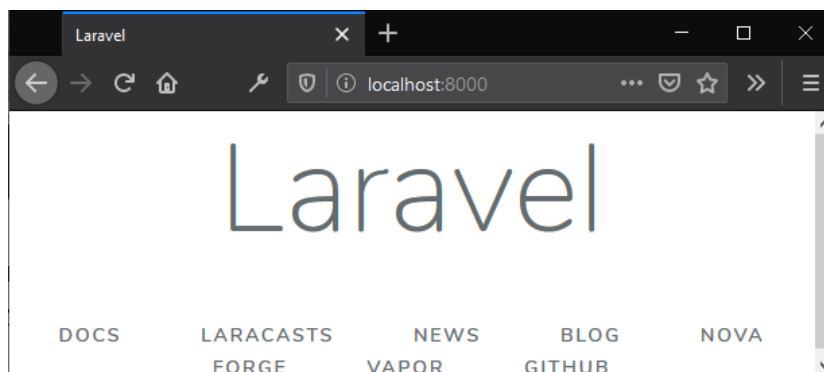
```
composer create-project --prefer-dist laravel/laravel tai
```

Przejdź do folderu *tai* utworzonego projektu (będzie to nasz **główny katalog**, z którego będziemy wydawać kolejne polecenia z wiersza poleceń) a następnie za pomocą narzędzia *artisan* (interfejs wiersza poleceń dołączony do programu Laravel) uruchom serwer developerski, za pomocą którego można testować tworzoną aplikację:

```
php artisan serve
```



Uruchom przeglądarkę i przejdź pod adres **localhost:8000**. Widok jak na rys. 1 oznacza, że instalacja przebiegła pomyślnie i utworzony został pierwszy projekt Laravel o nazwie *tai*.

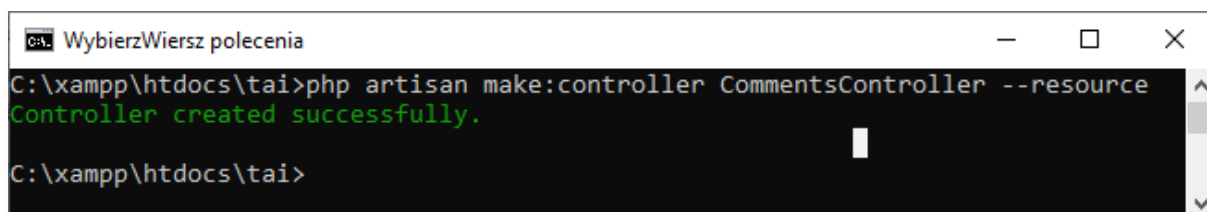


Rys. 1. Widok po stworzeniu projektu laravel w przeglądarce

2. Pierwszy kontroler, routing i widok

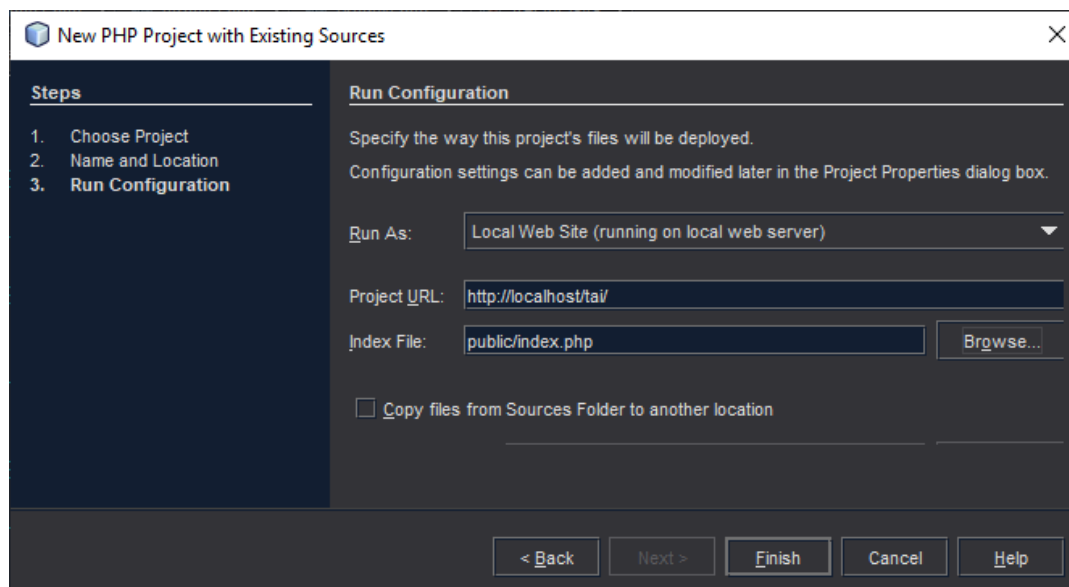
W programowaniu webowym routing (po polsku trasowanie) jest silnikiem napędowym każdej witryny czy aplikacji. Podstawowy routing służy do kierowania żądania do odpowiedniego kontrolera. W celu zobrazowania jak działa routing, utwórz pierwszy kontroler. W tym celu przejdź do wiersza poleceń i ponownie korzystając z narzędzia *artisan* w katalogu z projektem wydaj polecenie (najpierw zatrzymaj serwer – *Ctrl+C*):

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



```
C:\xampp\htdocs\tai>php artisan make:controller CommentsController --resource
Controller created successfully.
C:\xampp\htdocs\tai>
```

Do dalszej pracy z projektem dobrze jest skorzystać ze środowiska programistycznego. Na przykład, korzystając z NetBeans, utwórz nowy projekt *PHP Application (with existing sources)* jak pokazano na rysunku 2 i ustaw startowy plik na *index.php* z katalogu *public* projektu *tai*.



Rys. 2. Utworzenie i konfiguracja projektu *tai* w NetBeans

Przejrzyj zawartość folderu *app\Http\Controllers*, gdzie został utworzony plik *CommentsController.php*, który zawiera automatycznie wygenerowany kod klasy kontrolera i kilka użytecznych (na razie pustych), odpowiednio nazwanych metod, które będziemy uzupełniać w kolejnych punktach. Na początek w wygenerowanej metodzie *index* dodaj instrukcję:

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

jak pokazano na Listingu 1.

```
class CommentsController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return "Hello world";
    }
}
```

Listing 1. Metoda *index* w utworzonym kontrolerze *CommentsController*

Kolejnym krokiem jest dodanie dodatkowej reguły routingu:

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

do pliku *routes/web.php* (Listing 2). Jej zadaniem jest obsłużenie żądania postaci: *localhost:8000/comments* przez metodę *index()* kontrolera *CommentsController*.

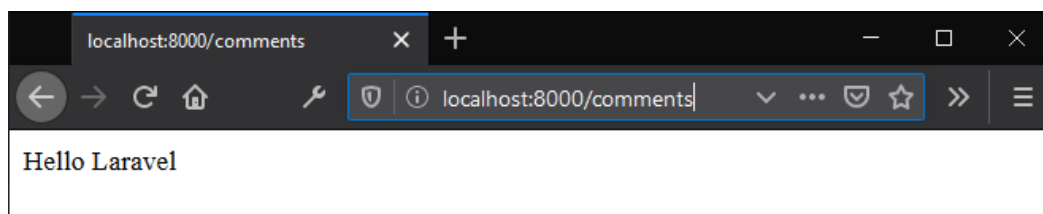
```
<?php
/*
|-----
| Web Routes
|-----
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

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

Listing 2. Reguły routingu w pliku *routes/web.php*

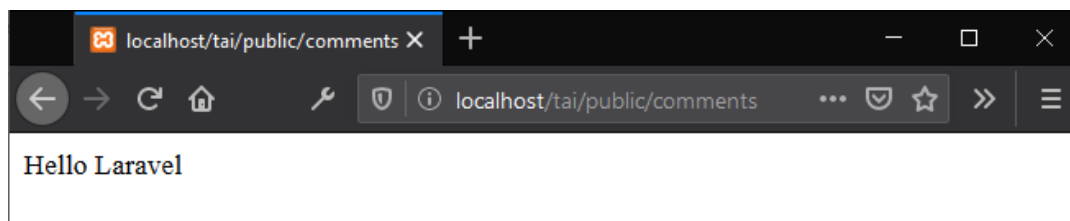
Przetestuj działanie nowej reguły routingu w przeglądarce - uruchom stronę *http:localhost:8000/comments* (Rys. 3). **Pamiętaj o ponownym uruchomieniu serwera developerskiego za pomocą: php artisan serve**



Rys. 3. Efekt działania metody *index* kontrolera *CommentsController*

Do testowania tworzonej aplikacji można również skorzystać ze środowiska XAMPP. Bez dodatkowych konfiguracji serwera Apache należy w tym celu uruchomić stronę pod adresem: ***http://localhost/tai/public/comments*** (Rys. 4).

W tym przypadku nie potrzebujemy uruchamiać serwera developerskiego jak poprzednio a z wiersza poleceń będziemy korzystać tylko do wydawania kolejnych komend.

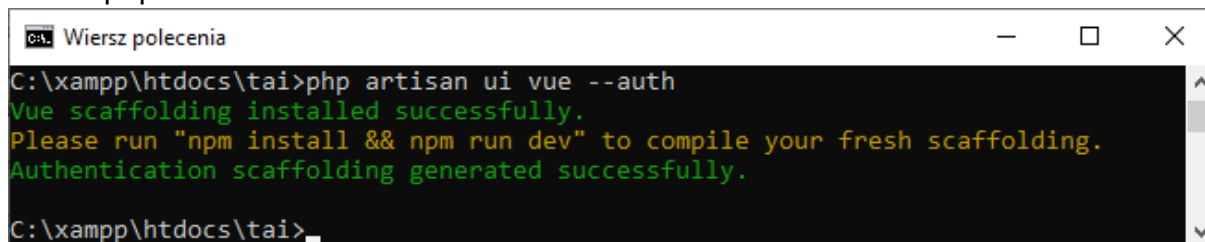


Rys. 4. Uruchomienie strony z XAMPP

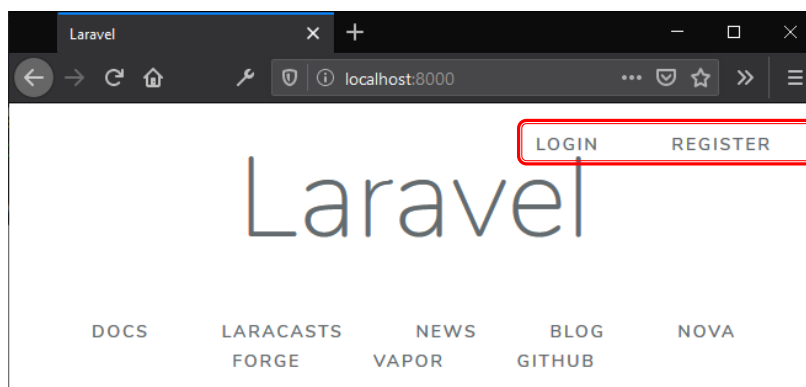
3. Autoryzacja użytkowników i pierwsza migracja

Laravel pozwala na bardzo prostą implementację autoryzacji użytkowników. W tym celu w wierszu poleceń w katalogu projektu należy wydać kolejne dwa polecenia:

```
composer require laravel/ui --dev
php artisan ui vue --auth
```



Po wykonaniu tych poleceń, na stronie głównej pojawiły się opcje rejestracji i logowania jak na rys. 5.



Rys.5. Przyciski do rejestracji i logowania użytkownika

Aby poprawnie działało uwierzytelnienie użytkownika, należy wykorzystać (utworzone już w tym celu przez Laravel) elementy związane z utrwaleniem danych użytkownika w bazie danych. W celu ustawienia połączenia do bazy danych w pliku **.env** (w głównym folderze projektu) należy podać dane autoryzujące dostęp do bazy danych. W przypadku MySQL domyślne ustawienia w pliku **.env** są wystarczające (Listing 3).

```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:DBx6bky4tIGyxZYvFPuV+YZoUTzr5su37irhMCQCYQg=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8
9 DB_CONNECTION=mysql
10 DB_HOST=127.0.0.1
11 DB_PORT=3306
12 DB_DATABASE=laravel
13 DB_USERNAME=root
14 DB_PASSWORD=
```

Listing 3. Domyślne ustawienia połączenia do bazy o nazwie *laravel* na serwerze MySQL

Korzystając z narzędzia *phpMyAdmin* utwórz nową bazę danych o nazwie *laravel* (taka nazwa bazy podana jest w domyślnym ustawieniu – listing 3).

Do pracy z bazą danych wykorzystamy teraz 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 dla użytkowników, która ma być utworzona w bazie danych. Gotowe pliki migracji można podejrzeć w projekcie (Listing 4). Migracja z listingu 4 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 CreateUserTable 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();
        });
    }
}
```

Listing 4. Gotowa klasa migracji dla tabeli *Users*

W celu fizycznego utworzenia tabeli *users* w bazie danych *laravel*, wystarczy już tylko wywołać komendę migracji z poziomu wiersza poleceń:

```
php artisan migrate
```

Sprawdź teraz w **phpMyAdmin** jakie tabele zostały utworzone na podstawie gotowych migracji (Rys. 6).

laravel migrations	
id	int(10) unsigned
migration	varchar(191)
batch	int(11)

laravel password_resets	
email	varchar(191)
token	varchar(191)
created_at	timestamp

laravel users	
id	bigint(20) unsigned
name	varchar(191)
email	varchar(191)
email_verified_at	timestamp
password	varchar(191)
remember_token	varchar(100)
created_at	timestamp
updated_at	timestamp

laravel posts	
id	int(10) unsigned
title	varchar(191)
body	text
created_at	timestamp
updated_at	timestamp

Rys. 6. Tabele w bazie danych po uruchomieniu migracji

4. Widoki dla komentarzy, formularz rejestracji

Kolejny etap to utworzenie widoków, które będą wyświetlane w odpowiedzi na żądanie użytkownika. Widoki takie są zwracane zwykle jako wynik działania metody (akcji) kontrolera. W przykładzie utworzymy stronę, która będzie służyła jako formularz do wprowadzenia komentarza przez użytkownika do księgi gości.

Zadanie podzielimy na dwa etapy:

- utworzenie widoku dla księgi gości.
- utworzenie logiki dodawania komentarzy przez użytkowników.

Zacznijmy od modyfikacji utworzonego wcześniej kontrolera **CommentsController**, w którym zwracany był prosty napis „*Hello Laravel*”. Teraz zmienimy to tak, aby zwracany był specjalny **obiekt** (**widok**) za pomocą funkcji **view()**. W tym celu zmień instrukcję **return** w metodzie **index()** kontrolera **CommentsController** jak pokazano na Listingu 5.

```
class CommentsController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return view('comments');
    }
}
```

Listing 5. 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 (fragment pliku na Listingu 6). W Laravel widoki (dokumenty o strukturze HTML) są obsługiwane przez silnik Blade.

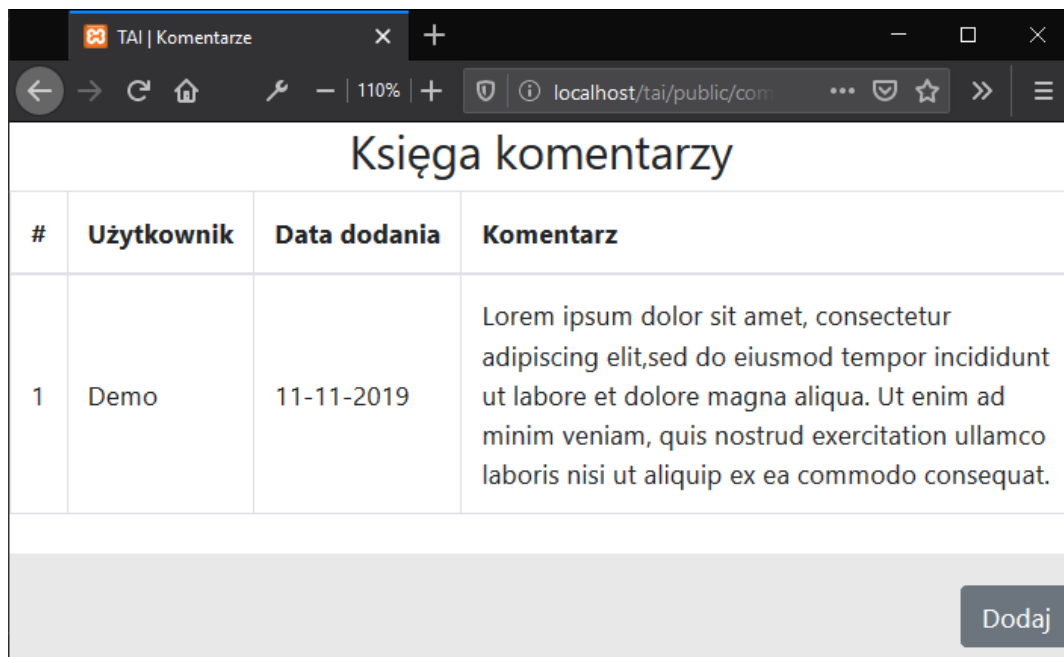
```

<body>
  <div class="table-container">
    <div class="title">
      <h3>Księga komentarzy</h3>
    </div>
    <table data-toggle="table">
      <thead>
        <tr>
          <th>#</th>
          <th>Użytkownik</th>
          <th>Data dodania</th>
          <th>Komentarz</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>1</td>
          <td>Demo</td>
          <td>11-11-2019</td>
          <td>Lorem ipsum dolor sit amet, consectetur adipiscing e<
          </td>
        </tr>
      </tbody>
    </table>
    <br>
    <div class="footer-button">
      <a href="#" class="btn btn-secondary">Dodaj</a>
    </div>
  </div>
</body>
</html>

```

Listing 6. Fragment strony *comments.blade.php*

Po tych zmianach - uruchom ponownie widok dla *'comments'* w przeglądarce (Rys. 7).

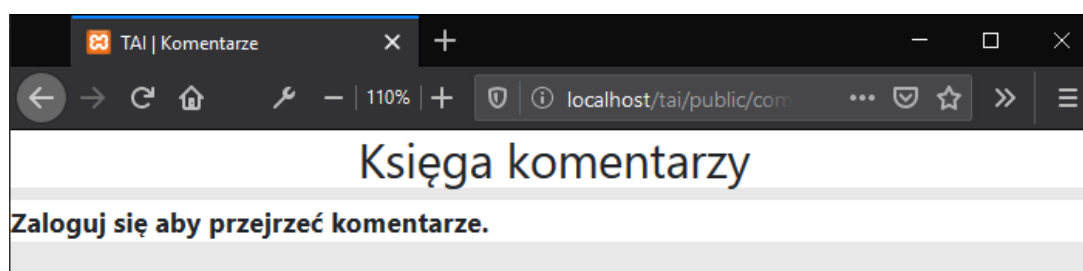
Rys. 7. Widok w przeglądarce strony *comments.blade.php*

Laravel pozwala łatwo określić sekcje na stronie, które mają być dostępne **tylko dla zalogowanych użytkowników**. Zmodyfikujemy teraz kod widoku, tak, aby komentarze były widoczne tylko dla zalogowanych użytkowników. W tym celu ustaw adnotację `@auth` tuż przed widokiem dla zalogowanych użytkowników oraz `@endauth` tuż po tej sekcji. W adnotacjach `@guest` i `@endguest` można z kolei ustawić treść widoczną dla wszystkich użytkowników, np. z informacją o konieczności zalogowania (Listing 7, Rys. 8).

```
<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>
```

Listing 7. Autoryzacja dostępu do elementów strony



Rys. 8. Widok po dodaniu autoryzacji

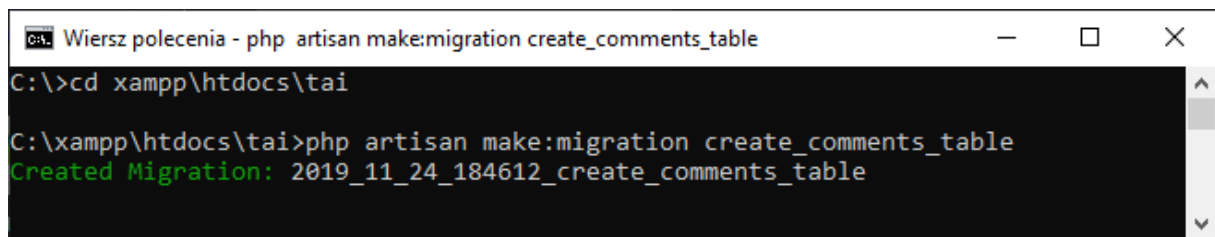
W celu założenia konta użytkownika można skorzystać z gotowego formularza rejestracji, który w naszym przykładzie jest dostępny pod adresem <http://localhost/tai/public/register> (można nadać mu własny styl w odpowiednim pliku Blade). Dokonaj rejestracji nowego użytkownika za pomocą domyślnego formularza a następnie przetestuj ponownie działanie strony *comments*. Tym razem treść z komentarzami będzie widoczna tylko dla zalogowanych użytkowników. Sprawdź za pomocą *phpMyAdmin* czy w bazie danych *laravel* i w tabeli *users* pojawił się nowy użytkownik.

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

5. Migracja tabeli dla komentarzy. Model Comments.

W kolejnym kroku należy utworzyć tabelę w bazie danych, w której będą przechowywane komentarze użytkowników. W tym celu ponownie skorzystamy z odpowiedniej migracji. W wierszu poleceń, w katalogu projektu wpisz komendę:

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



```
Wiersz polecenia - php artisan make:migration create_comments_table
C:\>cd xampp\htdocs\tai
C:\xampp\htdocs\tai>php artisan make:migration create_comments_table
Created Migration: 2019_11_24_184612_create_comments_table
```

Przejrzyj zawartość klasy z nową migracją, która znajduje się w katalogu *database/migrations* (Listing 8). 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, stworzona tabela nazywać się będzie *comments*.

```
class CreateCommentsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('comments', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }
}
```

Listing 8. Klasa migracji dla tabeli *comments*

W klasie *CreateCommentsTable*, w metodzie *up()* potrzebujemy jeszcze dwóch kolumn dla tworzonej tabeli - *treści komentarza* oraz *id użytkownika*, które za pomocą relacji 1:1 połączymy z tabelą *users* w modelu komentarza. W Laravel można to bardzo prosto zrealizować.

Dodaj najpierw brakujące pola w funkcji *up()* (Listing 9).

```

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

```

Listing 9. Dodanie brakujących pól w metodzie up()

I wykonaj migrację:

```

C:\xampp\htdocs\tai>php artisan migrate
Migrating: 2019_11_24_184612_create_comments_table
Migrated: 2019_11_24_184612_create_comments_table (0.48 seconds)

C:\xampp\htdocs\tai>

```

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

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

```

C:\xampp\htdocs\tai>php artisan make:model Comment
Model created successfully.

C:\xampp\htdocs\tai>

```

W katalogu **app** utworzył się plik **Comment.php** z definicją klasy **Comment**. Dodaj do niego fragment (Listing 10) definiujący relację jeden do jednego (jeden komentarz może mieć jednego autora).

```

<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    // Jeden Comment jest napisany przez jednego User
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}

```

Listing 10. Dodanie relacji 1:1 do modelu Comments

Zaimportuj też klasę modelu **Comment** w kontrolerze **CommentsController** (Listing 11) – w nagłówku pliku **CommentsController.php**, zaraz po deklaracji przestrzeni nazw **namespace**, dodaj kod: `use App\Comment;`

```
<?php
namespace App\Http\Controllers;
use App\Comment;
use Illuminate\Http\Request;

class CommentsController extends Controller
{
```

Listing 11. Dodanie importu klasy **Comment** do kontrolera **CommentsController**

Zależność ta będzie wykorzystana nieco później.

6. Formularz dodawania komentarza. Akcja create w kontrolerze.

Kolejny etap to utworzenie widoku formularza dodawania nowego komentarza. Tak jak poprzednio, by pokazać stronę (widok), należy najpierw w pliku **routes/web.php** dodać odpowiednią regułę routingu. Do obsługi żądań dodania komentarzy potrzebne będą dwie reguły routingu (Listing 12), związane z obsługą metod:

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

Listing 12 przedstawia zaktualizowane reguły routingu w pliku **web.php**. Zauważ, że niektóre reguły mają przypisaną **nazwę** za pomocą **->name('store')**. Nadanie nazw regułom routingu jest bardzo wygodne w późniejszym odwoływaniu się do nich.

```
Route::get('/', 'CommentsController@index');
Route::get('/comments', 'CommentsController@index')->name('comments');
Route::get('/create', 'CommentsController@create')->name('create');
Route::post('/create', 'CommentsController@store')->name('store');
```

Listing 12. Dodatkowe reguły routingu w **routes/web.php**

Po zaktualizowaniu reguł routingu, w metodzie **create()** kontrolera **CommentsController**, utwórz nowy obiekt **\$comment** oraz dodaj przekierowanie do widoku z formularzem **commentsForm.blade.php** (Listing 13). Nieużywana wcześniej funkcja **compact** służy do przekazywania zmiennych z kontrolera do widoków (w tym przypadku przekazywany jest obiekt **\$comment = new Comment()** - nowy komentarz).

```
public function create()
{
    $comment = new Comment();
    return view('commentsForm', compact('comment'));
}
```

Listing 13. Instrukcje w metodzie **create()** kontrolera **CommentsController**.

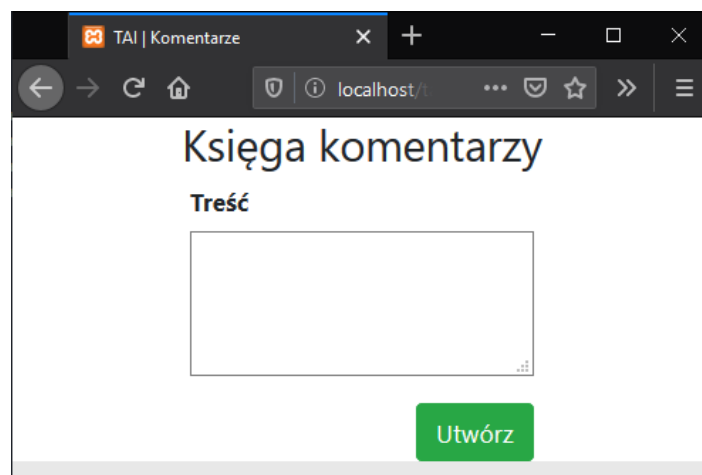
Następnie, w katalogu *resources/views* utwórz plik widoku *commentsForm.blade.php* z treścią jak w załączonym do ćwiczenia pliku (fragment pliku znajduje się na Listingu 14). Na stronie widoku *commentsForm.blade.php* zwróć uwagę na elementy specyficzne dla silnika szablonów Blade:

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

```
<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="5" required></textarea>
          </div>
        </div>
        <div class="box-footer"><button type="submit" class="btn btn-success">Utwórz</button>
      </div>
    </form>
```

Listing 14. Fragment pliku widoku *commentsForm.blade.php*

Sprawdź widok w przeglądarce pod adresem <http://local/tai/public/create> (Rys. 9).



Rys.9. Widok formularza dodawania komentarza

Następnie w widoku z komentarzami – w pliku *comments.blade.php* w hiperłączu „Dodaj” utwórz odwołanie do strony z formularzem dodawania komentarza. W tym celu ustaw atrybut *href* jak na listingu 15.

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

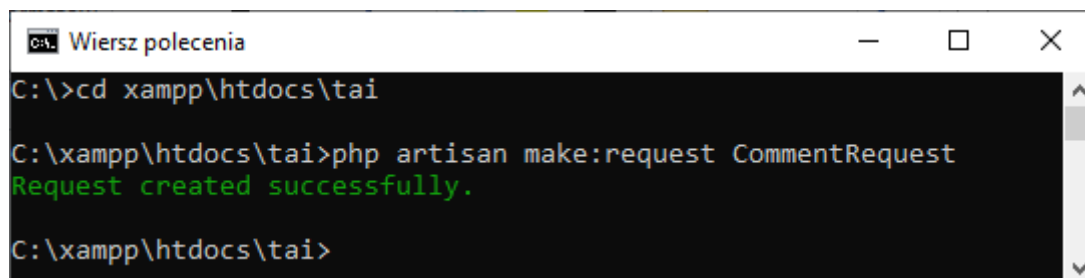
Listing 15. Uzupełnienie atrybutu *href* w *comments.blade.php*

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 również należy walidację 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”. W pierwszej kolejności dane z formularza są wysyłane metodą POST przez protokół HTTP i odbierane przez odpowiedni kontroler Laravla w postaci obiektu *Request*. Następnie trafiają do wskazanej w regule routingu metody kontrolera (w naszym przypadku jest to metoda *store()*). Metoda *store()*, jako parametr wejściowy otrzymuje obiekt klasy *Request*. Aby przeprowadzić walidację danych przesłanych w żądaniu należy utworzyć klasę dziedziczącą po klasie *Request*, w której można będzie zadeklarować warunki walidacji.

W tym celu w wierszu poleceń wydaj komendę:

```
php artisan make:request CommentRequest
```



```
Wiersz poleceń
C:\>cd xampp\htdocs\tai
C:\xampp\htdocs\tai>php artisan make:request CommentRequest
Request created successfully.
C:\xampp\htdocs\tai>
```

Utworzona w ten sposób klasa *CommentRequest.php* znajduje się w katalogu *app\Http\Requests*. Klasa zawiera dwie metody: *authorize()* i *rules()*. W celu wymuszenia autoryzacji użytkownika, ustaw wynik zwracany przez metodę *authorize()* na *true*. Z kolei w metodzie *rules()* zdefiniuj warunki walidacji w formie tablicy asocjacyjnej (kluczem jest nazwa pola formularza) (Listing 16).

Konieczne jest też dodanie klasy walidatora *CommentRequest* do kontrolera *CommentsController*, aby kontroler podczas zapisu nie korzystał z domyślnej klasy *Request*. W tym celu dodaj w nagłówku pliku *CommentsController.php*, tuż po deklaracji przestrzeni nazw, wiersz: `use App\Http\Requests\CommentRequest;` oraz zmień parametr metody *store()* z *Request* na *CommentRequest* (Listing 17).

Przetestuj działanie walidatora, wpisując nieprawidłową liczbę znaków w komentarzu (Rys. 10).

```
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

class CommentRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'message' => 'string|min:100|max:300'
        ];
    }
}
```

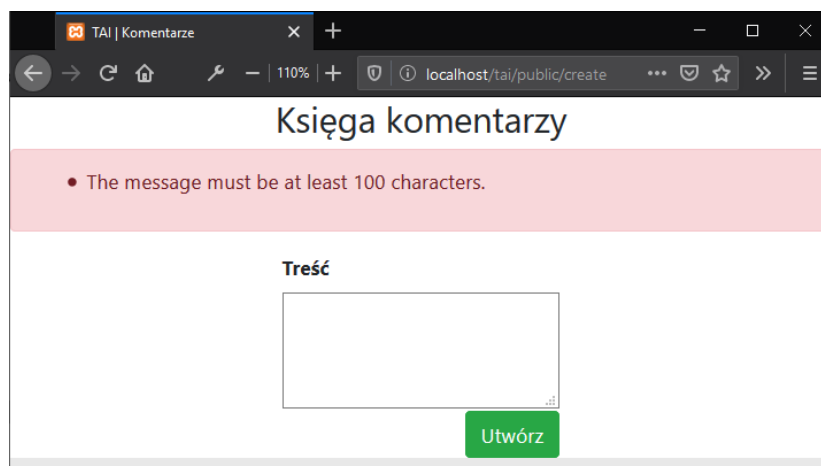
Listing 16. Klasa *CommentRequest* z regułami walidacji

```
<?php
namespace App\Http\Controllers;

use App\Comment;
use App\Http\Requests\CommentRequest;
use Illuminate\Http\Request;

class CommentsController extends Controller
{
    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(CommentRequest $request)
    {
        //
    }
}
```

Listing 17. Klasa *CommentsController* po modyfikacjach



The screenshot shows a web browser window with the title "Księga komentarzy". The address bar displays "localhost/tai/public/create". A red error message is shown: "The message must be at least 100 characters." Below the message is a text input field labeled "Treść" and a green button labeled "Utwórz".

Rys. 10. Działanie walidacji

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

Poprawne dane pobrane z formularza, powinny być zapisane w bazie danych. Wykorzystamy w tym celu metodę *store()* kontrolera *CommentsController* (Listing 18).

```
public function store(CommentRequest $request) {
    $comment = new Comment();
    $comment->user_id = \Auth::user()->id; //ID aktualnie zalogowanego
    $comment->message = $request->message; //Nazwa pola z walidatora
    if ($comment->save()) {
        return redirect()->route('comments');
    }
    return "Wystąpił błąd.";
}
```

Listing 18. 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 bazy danych.

Ostatni etap to wyświetlenie w widoku głównym komentarzy pobranych z bazy danych. W tym celu w metodzie *index()* kontrolera *CommentsController* zmodyfikuj kod jak na listingu 19. Wykorzystano tu klasę modelu *Comment*, która jest abstrakcyjnym bytem odwzorowującym zbiór bazodanowy komentarzy na kolekcję obiektów klasy *Comment*.

Dostęp do danych w Laravel jest najczęściej realizowany za pomocą narzędzia *Eloquent*. *Eloquent* jest warstwą ORM wykorzystywaną do mapowania obiektów aplikacji Laravel, na rekordy tabel w bazie danych (i odwrotnie). *Eloquent* pozwala przekształcić żądania obsługi danych modeli w odpowiednie zapytania i co najważniejsze, nie ogranicza się przy tym do jednego systemu bazodanowego. Metoda *index()* wykorzystuje *Eloquent* do pobrania danych w postaci posortowanej kolekcji obiektów klasy (modelu) *Comment*.

```
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $comments = Comment::orderBy('created_at', 'asc')->get();
    return view('comments', compact('comments'));
}
```

Listing 19. Pobranie wszystkich komentarzy z bazy danych metodą *get()*

Za pomocą metod *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
```


Jedną z zalet **Eloquent** jest też to, że można go wykorzystać do odczytywania wprost wartości modeli pozostających ze sobą w relacji, a co więcej nakładać na nie odpowiednie warunki. Oznacza to, że programista może pominąć żmudne i długie zapytania SQL oparte o JOIN i warunki w nich zagnieżdżone, i może skorzystać z metod **Eloquent** typu: **whereHas**, **whereDoesntHas**, które pozwolą na prostsze dodanie warunków na struktury danych obiektowych. Przykładowo, jeśli zamiast wszystkich komentarzy, należy zwrócić tylko komentarze użytkowników, którzy mają w swoim adresie *e-mail*, 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:

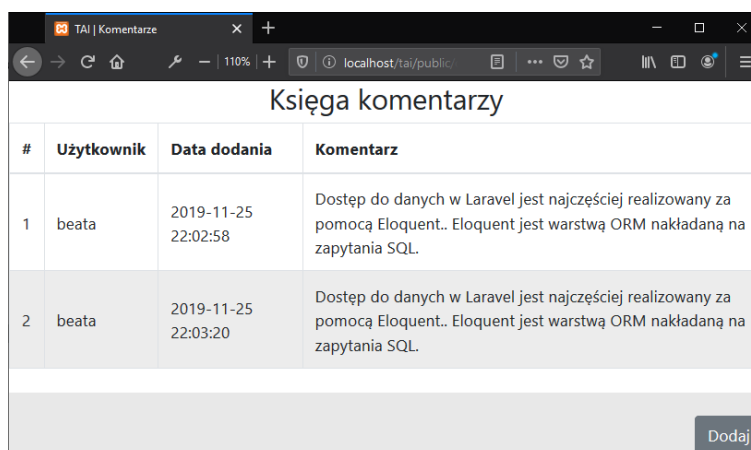
```
"select * from `comments` where exists (select * from `users`
where `comments`.`user_id` = `users`.`id` and `email` like
'%karolina%')";
```

Odczytanie danych z kolekcji komentarzy w widoku **comments.blade.php** można zrealizować za pomocą modyfikacji kodu w bloku **<tbody>**, jak na Listingu 20.

```
<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>
```

Listing 20. Wyświetlanie wszystkich komentarzy w widoku **comments.blade.php**

Widok strony z komentarzami pobranymi z bazy danych przedstawia rysunek 11. Widok ten jest dostępny jedynie dla zalogowanych użytkowników.



Księga komentarzy			
#	Użytkownik	Data dodania	Komentarz
1	beata	2019-11-25 22:02:58	Dostęp do danych w Laravel jest najczęściej realizowany za pomocą Eloquent.. Eloquent jest warstwą ORM nakładaną na zapytania SQL.
2	beata	2019-11-25 22:03:20	Dostęp do danych w Laravel jest najczęściej realizowany za pomocą Eloquent.. Eloquent jest warstwą ORM nakładaną na zapytania SQL.

Dodaj

Rys. 11. Widok strony z komentarzami