

Informatyka, Aplikacje internetowe i mobilne, semestr 5

Projektowanie serwisów internetowych

Laboratorium nr 7

Tworzenie aplikacji WWW z wykorzystaniem frameworka PHP Laravel

Laravel jest frameworkiem (szkieletem do budowy aplikacji) PHP przeznaczonym do szybkiego programowania aplikacji webowych. Został zbudowany na bazie Symfony czyli innego znanego frameworka dla PHP. Oba są rozwijane od 2011 r.

Laravel jest frameworkiem ogólnego przeznaczenia, co oznacza, że może być wykorzystany do stworzenia dowolnej aplikacji internetowej korzystającej z PHP, takiej jak serwis, portal, forum, CMS, e-sklep czy usługa sieciowa.

Aplikacje budowane za pomocą Laravel są zorganizowane zgodnie ze wzorcem architektonicznym MVC czyli model-widok-kontroler (Model-View-Controller). Modele reprezentują dane, logikę biznesową i reguły; widoki są wyjściową reprezentacją modeli; a kontrolery pobierają dane wejściowe i konwertują je na polecenia dla modeli i widoków.

Kod źródłowy Laravel jest udostępniany w serwisie GitHub, na licencji MIT.

Więcej informacji:

<https://pl.wikipedia.org/wiki/Laravel>

<https://laravel.com/docs/10.x>

1. Instalacja szablonu aplikacji na serwerze foka

Instalacja wymaga pobrania kodów źródłowych oraz utworzenia projektu w postaci szablonu przyszłej aplikacji.

Większość frameworków umożliwia wykorzystanie, popularnego obecnie, narzędzia wspomagającego zarządzanie pakietami i zależnościami w projektach PHP o nazwie Composer. Narzędzie to zostało zainstalowane i jest dostępne na serwerze foka.

Zaloguj się na swoje konto na serwerze foka korzystając z Putty.

Sprawdź czy na Twoim koncie jest katalog `public_html`. Jeśli nie, załóż go korzystając z polecenia `mkdir`. Twój projekt ma być dostępny na serwerze, a więc będzie musiał być założony w katalogu `public_html`.

Na zajęciach wykonamy projekt prostej aplikacji do zarządzania książkami o nazwie `laravel_książki`.

Zainstaluj szablon aplikacji Laravel korzystając z narzędzie Composer. W tym celu przejdź do katalogu `public_html` korzystając z polecenia `cd` (`cd public_html`) i utwórz projekt wykonując polecenie, które pobierze pliki szablonu aplikacji laravel z repozytorium laravel i wgra do podkatalogu `laravel_książki`:

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

Na ekranie powinna pojawić się długa lista komunikatów:

```
Creating a "laravel/laravel" project at "./laravel_książki"
Installing laravel/laravel (v10.2.8)
- Downloading laravel/laravel (v10.2.8)
- Installing laravel/laravel (v10.2.8): Extracting archive
Created project in /home/s<nrAlbumu>/public_html/laravel_książki
> @php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies
Lock file operations: 107 installs, 0 updates, 0 removals
- Locking brick/math (0.10.2)
- Locking dflydev/dot-access-data (v3.0.2)
- Locking doctrine/inflector (2.0.6)
...
> @php artisan package:discover --ansi
INFO Discovering packages.
laravel/sail..... DONE
laravel/sanctum..... DONE
laravel/tinker..... DONE
nesbot/carbon..... DONE
nunomaduro/collision..... DONE
nunomaduro/termwind..... DONE
spatie/laravel-ignition..... DONE
...
> @php artisan vendor:publish --tag=laravel-assets --ansi --force
INFO No publishable resources for tag [laravel-assets].
> @php artisan key:generate --ansi
INFO Application key set successfully.
```

W efekcie zawartość katalogu **laravel_książki** została przygotowana do tworzenia nowego projektu aplikacji laravel. Composer zapisał pliki, utworzył plik konfiguracyjny z rozszerzeniem **.env** oraz wygenerował klucz aplikacji.

```
..
app
artisan
bootstrap
composer.json
composer.lock
config
database
.editorconfig
.env
.env.example
.gitattributes
.gitignore
lang
package.json
phpunit.xml
public
README.md
resources
routes
storage
tests
vendor
vite.config.js
```

Uruchomienie bieżącej wersji wymaga jeszcze ustawienia praw dostępu do katalogu **storage** projektu oraz wszystkich jego podkatalogów. Ustaw pełne prawa dostępu do katalogu storage i jego podkatalogów korzystając z polecenia **chmod** lub klienta FTP np. programu WinSCP ustaw prawa dostępu do katalogu storage oraz wszystkich jego podkatalogów i plików (rekursywnie) na 777. Użycie **chmod** wymaga przejścia do katalogu **laravel_książki** i wykonania polecenia:

\$ chmod -R 777 storage

Uruchom bieżącą wersję aplikacji wpisując w przeglądarce jej adres. W katalogu głównym widać jeszcze listę katalogów i plików. Obejrzyj zawartość podkatalogu **public**, zawiera on plik uruchamiany domyślnie **index.php**.

```
4096 lis 14 16:18 .
4096 lis 20 14:59 ..
0 lis 14 16:18 favicon.ico
603 lis 14 16:18 .htaccess
1710 lis 14 16:18 index.php
24 lis 14 16:18 robots.txt
laravel_książki/public$
```

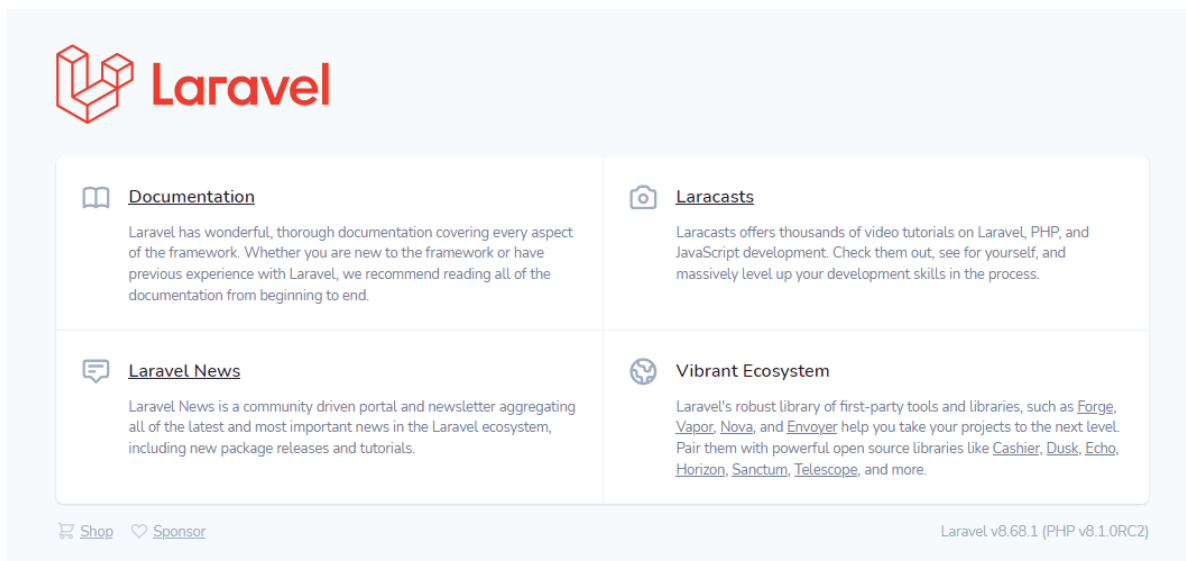
Adres twojej aplikacji ma więc postać:

https://foka.umg.edu.pl/~s<nrAlbumu>/laravel_książki/public/

Przykładowo dla studenta o nr albumu 99999 adres ma postać:

https://foka.umg.edu.pl/~s99999/laravel_książki/public/

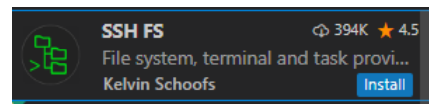
Na ekranie powinna być teraz widoczna strona startowa jak na obrazku poniżej.



Korzystając z WinSCP lub Putty obejrzyj zawartość pliku .env, w którym znajdują się ustawienia aplikacji.

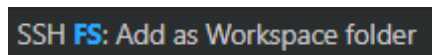
2. Edycja plików projektu

W środowisku MS VS Code, korzystając z odpowiedniego rozszerzenia/pakietu można podłączyć swój katalog domowy z foki co umożliwia łatwiejsze przeglądanie i edycję plików w tworzonym projekcie. Można skorzystać z pakietu SSH FS.

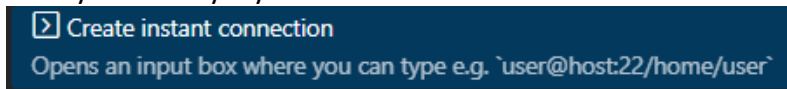


Zainstaluj pakiet SSH FS w edytorze MS VS Code i za jego pomocą podłącz do VS Code katalog **laravel_ksiadzki** ze swojego konta na foka. Kolejne kroki opisano poniżej.

Korzystając z opcji menu View | Command Palette wywołaj komendę SSH FS: Add as Workspace folder.



Z otrzymane listy wybierz:



Podaj nazwę połączenia w formacie:

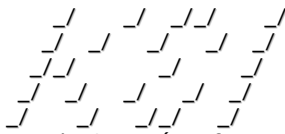
`s[nr_albumu]@foka.umg.edu.pl:22/home/s[nr_albumu]/public_html/laravel_ksiadzki`

Dla studenta o nr indeksu 99999 nazwą połączenia będzie:

[s99999@foka.umg.edu.pl:22/home/s99999/public_html/laravel_ksiadzki](ssh://s99999@foka.umg.edu.pl:22/home/s99999/public_html/laravel_ksiadzki)

Podaj hasło dla swojego konta po pojawieniu się komunikatu password.

Możesz sprawdzić ustawienia połączenia przez zakładkę rozszerzeń. Z menu podręcznego dla rozszerzenia SSH FS wybierz Extension Settings | Workspace, a następnie Edit in settings.json:



Katedra Systemów Informacyjnych
Uniwersytet Morski w Gdyni

Sshfs: Configs

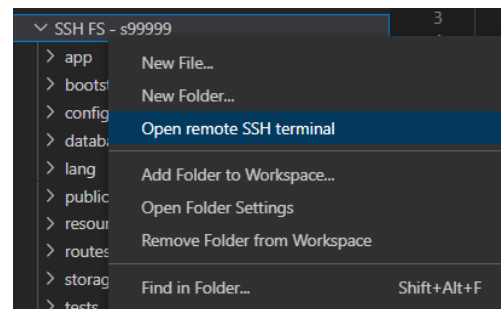
Use the Settings UI to edit configurations (run command *SSH FS: Open settings and edit configurations*)

[Edit in settings.json](#)

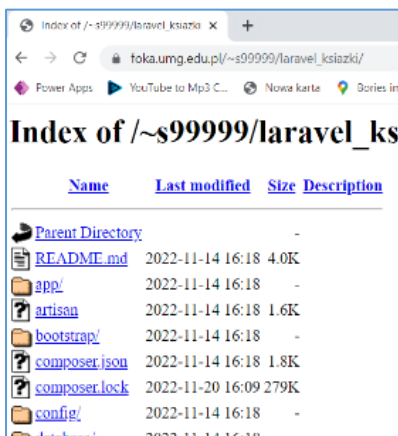
Zmiany w pliku **workspace.json** wpływają na konfigurację aktualnego połączenia. W celu zapewnienia bezpieczeństwa nie wpisuj w tym pliku swojego hasła!

```
{
  "folders": [
    {
      "name": "SSH FS – s<nr_indeksu>@foka.umg.edu.pl:22/home/s<nr_indeksu>/public_html/laravel_ksiazki",
      "uri": "ssh://s<nr_indeksu>@foka.umg.edu.pl:22/home/s<nr_indeksu>/public_html/laravel_ksiazki"
    }
  ],
  "settings": {
    "sshfs.configs": [
      {
        "root": "/home/s<nr_indeksu>/public_html/laravel_ksiazki",
        "host": "foka.umg.edu.pl",
        "port": 22,
        "username": "s<nr_indeksu>",
        "password": "",
        "name": "s<nr_indeksu>"
      }
    ]
  }
}
```

Można też skorzystać z terminala SSH w VSCode – w tym celu należy wskazać w workspace folder: SSH FS – s[*nr_albumu*], a następnie kliknąć prawy klawisz myszy i z lokalnego menu i wybrać opcję Open remote SSH terminal.



3. Podstawowe bezpieczeństwo aplikacji



Podstawowym zabezpieczeniem dla aplikacji jest uniemożliwienie czytania zawartości katalogów. W katalogu głównym projektu **laravel_ksiazki** utwórz plik **.htaccess** zawierający jedną dyrektywę:

Options -Indexes

Sprawdź jak wyglądają strony aplikacji w przeglądarce w katalogu **laravel_ksiazki** i **laravel_ksiazki/public**.

Zauważ, że w katalogu **public** taki plik został utworzony automatycznie podczas działania composera.

4. Współpraca z bazą danych

Podstawą działania większości aplikacji są dane. Najczęściej są one przechowywane w plikach i/lub bazach danych. Aplikacja `laravel_książki` będzie współpracowała z relacyjną bazą danych w systemie PostgreSQL. Dane będą przechowywane w trzech tabelach `ksiazka`, `kategoria` i `wydawnictwo`. Umieścimy je w osobnym schemacie: `laravel_książki`.

Połącz się ze swoją bazą danych PostgreSQL na serwerze `foka` korzystając z aplikacji `pgAdmin4` dostępnej pod adresem: <https://foka.umg.edu.pl/pgadmin4>.

Załącz schemat `laravel_książki`. Umieścimy w nim 3 tabele: `kategoria`, `wydawnictwo` i `ksiazka`. Docelowo powinny mieć postać:

`kategoria`

123 id	ABC opis
1	WWW
2	HTML
3	JavaScript
4	Java

`wydawnictwo`

123 id	ABC nazwa	ABC adres
1	Helion	Gliwice, Polska
2	PWN	Warszawa, Polska
3	OREILLY	Boston, USA

`ksiazka`

123 id	ABC tytuł	123 idkat	123 idwyd
1	Java. Podstawy	4	1
2	Projektownie serwisów WWW. Standardy sieciowe	1	1
3	Zrozumieć JavaScript	3	1
4	Head first Java	4	3
5	HTML5. Komponenty	2	2
6	Wydajny JavaScript	3	2

Tabele można dodać na dwa sposoby, albo bezpośrednio w bazie danych albo w Laravel za pośrednictwem tzw. migracji (migration). W tym drugim przypadku tabele trzeba opisać w PHP tworząc kod z klasami migracji, a następnie uruchomić. Każda klasa migracji musi być dziedziczona z klasy `Migration` zdefiniowanej w Laravelu.

W naszej aplikacji tabele zostaną utworzone właśnie za pomocą migracji z poziomu frameworka.

Przed utworzeniem plików migracji dla aplikacji Laravel warto skonfigurować połączenie z utworzonym schematem w bazie danych. Wymaga to ustawienia odpowiednich parametrów w pliku `.env` oraz `config/database.php`.

W pliku `.env` ustaw następujące wartości:

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1 # lub localhost
DB_PORT=5432
DB_DATABASE=s<nr_albumu> # np. s99999
DB_SCHEMA=laravel_ksiazki
DB_USERNAME=s<nr_albumu> # np. s99999
DB_PASSWORD=secret_password # Twoje hasło
```

Uwaga - w oryginalnym pliku nie ma zmiennej `DB_SCHEMA`, trzeba ją dopisać.

W pliku **config/database.php**, w tablicy return w elemencie z kluczem `pgsql` zmień ustawienie `search_path` z `public` na podany poniżej kod. Funkcja `env` umożliwia pobranie wartości podanej zmiennej środowiskowej, tutaj `DB_SCHEMA` z pliku **.env**: Jeśli takiej zmiennej nie ma, zostanie ustawiona wartość podana jako drugi parametr funkcji. Ustawienie:

```
'search_path' => 'public',
```

zmień na:

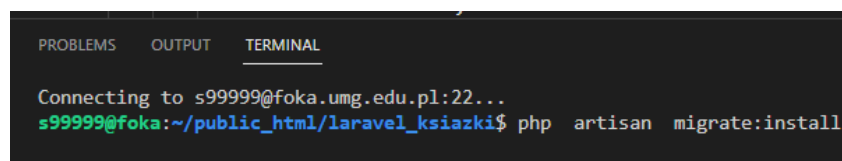
```
'search_path' => env('DB_SCHEMA', 'forge'),
```

5. Połączenie z bazą danych

Wskaż w VS Code zasób/folder SSH FS - `s<nr_albumu>@...` i wciśnij prawy klawisz myszy, aby z menu wybrać terminal poleceń Open remote SSH terminal. Zauważ, że otworzyło się dodatkowe, trzecie okno w prawej dolnej części aplikacji MS VSCode. Z poziomu katalogu `laravel_ksiazki` w terminalu poleceń będą wydawane polecenia ściśle związane z instalowanym frameworkiem Laravel.

Sprawdź poprawność ustawień generując tabelę testową o nazwie `migration` w schemacie `laravel_ksiazki`. W tym celu skorzystaj z narzędzia `artisan`. Polecenie powinno zostać wykonane w terminalu ssh z katalogu projektu `laravel_ksiazki` i mieć postać:

```
php artisan migrate:install
```



```
Connecting to s99999@foka.umg.edu.pl:22...
s99999@foka:~/public_html/laravel_ksiazki$ php artisan migrate:install
```

W efekcie na ekranie terminala powinna pojawić się informacja:

```
INFO Migration table created successfully.
```

Sprawdź czy w Twojej bazie w schemacie `laravel_ksiazki` pojawiła się tabela `migration` (wcześniej odświeżając listę tabel w aplikacji pgAdmin4 lub DBeaver).

Utwórz pliki migracji dla tabel w aplikacji. Pliki takie muszą zawierać klasy o nazwach odpowiadających nazwom tabel dziedziczącymi z klasy `Migration` oraz odpowiednie metody. Przykłady takich klas można obejrzeć w katalogu **database/migrations**, w którym obecnie znajdują się pliki podane poniżej. Sprawdź co zawierają.

```
2014_10_12_000000_create_users_table.php
2014_10_12_100000_create_password_resets_table.php
2019_08_19_000000_create_failed_jobs_table.php
```



2019_12_14_000001_create_personal_access_tokens_table.php

Artisan umożliwia wygenerowanie szablonu pliku migracji. Żeby wygenerować plik migracji dla tabeli `kategoria` użyj polecenia:

```
php artisan make:migration create_kategoria_table
```

```
<?php

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

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

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('kategoria');
    }
};
```

Sprawdź jaki plik został dodany do katalogu **migrations** oraz jaka jest jego zawartość.

Każdy plik z migracją tworzy klasę rozszerzającą klasę bazową `Migration` zawierającą metodę `up` oraz `down`. Metoda `up` uruchamia się za każdym razem gdy wykonujemy migrację (czyli komendę: `php artisan migrate`) a metoda `down` wykonywana jest zawsze, gdy cofamy zmiany dokonane przez migrację (`php artisan migrate:rollback`). Laravel domyślnie tworzy tabelę o nazwie `kategoria` z auto-inkrementowanym polem `id` oraz `timestamps` do utworzenia w tabeli pól na znaczniki czasu (`created_at` i `updated_at`).

Ponieważ w tej aplikacji nie będziemy używać znaczników czasu, natomiast w tabeli `kategoria` jest pole `opis`, zamień `timestamps()` na instrukcję tworzenia pola `string` o nazwie `opis`, czyli: `$table->string('opis');`

Gotowa klasa powinna wyglądać tak jak na listingu poniżej.

```
class Kategoria extends Migration
{
    public function up()
    {
        Schema::create('kategoria', function (Blueprint $table) {
            $table->id();
            $table->string('opis');
        });
    }

    public function down()
    {
        Schema::dropIfExists('kategoria');
    }
}
```

Zauważ, że dziedziczenie oraz wykorzystanie innych klas wymusza dołączenie do plików kodów odpowiadających im plików. Służy do tego instrukcja `use`, natomiast kody są częścią usługi/przestrzeni nazw `Illuminate`.

Podobnie utwórz plik migracji dla tabeli `wydawnictwo` opisanej powyżej.

Podobnie utwórz plik migracji dla tabeli `ksiazka`. Usuń znaczniki czasu. Dodaj pola `tytul` i `idkat` oraz `idwyd`:

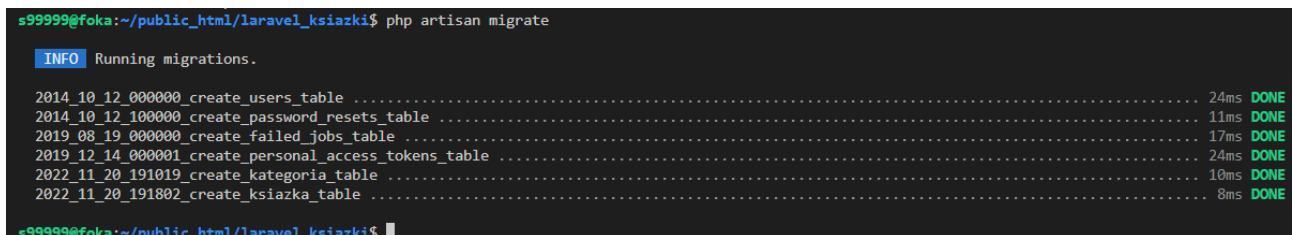
```
$table->string('tytul');  
$table->integer('idkat');  
$table->integer('idwyd');
```

Gotowa klasa dla tabeli `ksiazka` powinna wyglądać tak jak na listingu poniżej.

```
class Ksiazka extends Migration  
{  
    public function up()  
    {  
        Schema::create('ksiazka', function (Blueprint $table) {  
            $table->id();  
            $table->string('tytul');  
            $table->integer('idkat');  
            $table->integer('idwyd');  
        });  
    }  
  
    public function down()  
    {  
        Schema::dropIfExists('ksiazka');  
    }  
}
```

Na podstawie tak przygotowanych plików migracji można utworzyć tabele w bazie danych. W głównym katalogu projektu wykonaj polecenie:

```
$ php artisan migrate
```



```
s99999@foka:~/public_html/laravel_ksiazki$ php artisan migrate  
  
INFO Running migrations.  
  
2014_10_12_000000_create_users_table ..... 24ms DONE  
2014_10_12_100000_create_password_resets_table ..... 11ms DONE  
2019_08_19_000000_create_failed_jobs_table ..... 17ms DONE  
2019_12_14_000001_create_personal_access_tokens_table ..... 24ms DONE  
2022_11_20_191019_create_kategoria_table ..... 10ms DONE  
2022_11_20_191802_create_ksiazka_table ..... 8ms DONE  
  
s99999@foka:~/public_html/laravel_ksiazki$
```

Zauważ, że w bazie zostały utworzone wszystkie tabele, dla których były dostępne pliki migracji. Ważna jest ich kolejność.

Więcej informacji na temat migracji można znaleźć w dokumentacji:

<https://laravel.com/docs/10.x/migrations>

6. Wypełnienie danymi tabel `kategoria` i `ksiazka`

Wypełnij tabele danymi testowymi korzystając z kodów SQL dla tabel `kategoria` i `ksiazka` zapisanych w plikach:

`sql_kategoria_dane.sql`, `sql_wydawnictwo_dane.sql` i `sql_ksiazka_dane.sql`.

7. Uzupełnienie informacji o plikach migracji – opcja!!!

Tego punktu **nie należy wykonywać podczas laboratorium**. Jest to tylko dodatkowa informacja, która może przydać się osobom decydującym się na laravela podczas projektu swojej aplikacji PSI.

UWAGA!

Ogólne polecenie `$ php artisan migrate` wykonuje próbę migracji według kolejności plików po nazwach poprzedzonych prefiksem `TIMESTAMP`. Na przykład, w przypadku kiedy plik migracji `ksiazka: [TimeStamp]_create_ksiazka_table.php` zostałby utworzony wcześniej niż plik migracji `wydawnictwo`, czy też `kategoria` i wskazano by w nim zależności referencyjne do tabel słownikowych `kategoria`, `wydawnictwo`, kod:

```
$table->integer('idkat')->unsigned();  
$table->foreign('idkat')->references('id')->on('kategoria');  
$table->integer('idwyd')->unsigned();  
$table->foreign('idwyd')->references('id')->on('wydawnictwo');
```

W takim przypadku należy przestrzegać kolejności wykonywania migracji i wykonywać je etapami dla konkretnych tabel. Najpierw powinno się wygenerować migrację dla tabel `wydawnictwo` i `kategoria` za pomocą ogólnej postaci polecenia ze wskazaniem ścieżki `path`.

```
php artisan migrate --path  
database/migrations/[TIMESTAMP]_create_[nazwa_tabeli]_table.php
```

a dopiero w ostatniej kolejności zająć się migracją lub wycofaniem dla `ksiazka`. Na przykład:

```
php artisan migrate --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php  
php artisan migrate:rollback --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php
```

Przydatne opcje migracji

Żeby wycofać ostatnią utworzoną tabelę można skorzystać z opcji `--step=1`.

Żeby sprawdzić, którą tabelę/tabele wycofa migracja można użyć opcji `--pretend`, wtedy zamiast wykonania polecenia zostanie wyświetlona informacja na jego temat.

Co w przypadku gdy zapomnieliśmy w projekcie pliku migracji `ksiazka` o jakimś polu?

Metoda 1:

Wycofaj całą tabelę `ksiazka` za pomocą `migrate:rollback`

```
php artisan migrate:rollback --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php
```

Sprawdź, czy tabela `ksiazka` została usunięta z bazy. Dodaj do pliku migracji `ksiazka` w funkcji `up()` pole z informacją o liczbie stron: `$table->integer('liczbastron')->unsigned();`

Ponownie uruchom migrację dla `ksiazka`:

```
php artisan migrate --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php
```

Sprawdź, czy pojawiła się tabela `ksiazka` z polem `liczbastron`. Ponownie wycofaj całą tabelę za pomocą `rollback`:

```
php artisan migrate:rollback --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php
```

Usuń lub ustaw komentarz w pliku migracji `[TIMESTAMP]_create_ksiazka_table.php` dla pola:

```
$table->integer('liczbastron')->unsigned();
```

Ponownie uruchom migrację dla `ksiazka`:

```
php artisan migrate --path  
database/migrations/[TIMESTAMP]_create_ksiazka_table.php
```

Sprawdź, czy w bazie danych pojawiła się tabela `ksiazka` już bez pola `liczbastron`.

Metoda 2:

Można oczywiście wykonać dodanie pola lub jego usunięcie bez wycofywania całej tabeli `ksiazka`.

W tym celu należy utworzyć plik migracji o nazwie
`add_remove_liczbastron_to_ksiazka_table`

```
$ php artisan make:migration add_remove_liczbastron_to_ksiazka_table
```

Do funkcji `up()` dodajemy informację o całkowitym

```
$table->integer('liczbastron')->unsigned();
```

Do funkcji `down()` dodajemy za pomocą `dropColumn` informację jaką kolumnę chcemy usunąć:

```
$table->dropColumn('liczbastron');
```

Wykonaj migrację ze wskazaniem pliku `add_remove_liczbastron_to_ksiazka_table`:

```
$ php artisan migrate --path
```

```
database/migrations/[TimeStamp]_add_remove_liczbastron_to_ksiazka_table.p  
hp
```

Sprawdź, czy w tabeli `ksiazka` pojawiło się dodatkowe pole `liczbastron`!

Wykonaj migrację, z `rollback` ze wskazaniem `add_remove_liczbastron_to_ksiazka_table`:

```
$ php artisan migrate:rollback --path  
database/migrations/[TIMESTAMP]_  
add_remove_liczbastron_to_ksiazka_table:
```

Sprawdź, czy w tabeli `ksiazka` zostało usunięte pole `liczbastron`!

Więcej informacji:

<https://laravel-school.com/posts/how-to-add-new-columns-to-the-existing-table-in-laravel-migration-24>

8. Strona domowa aplikacji – interfejs w widoku

W architekturze MVC interfejs aplikacji zapisywany jest w postaci widoków. W aplikacjach WWW są to zwykle dokumenty HTML, wygenerowane na przykład przez PHP. Laravel ma specjalny

katalog do przechowywania zasobów dla strony klienta, które wymagają wcześniejszej kompilacji po stronie serwera o nazwie `resources`. Zasoby zostały tam jeszcze podzielone na podkatalogi: `css`, `js` i `views`.

Strony aplikacji są często składane z kilku części, np. nagłówka, menu, zawartości i stopki. Dobrą praktyką będzie utworzenie w katalogu `resources/views` nowego podkatalogu o nazwie `partials`. Korzystając z VSCode utwórz katalog `partials`. W podkatalogu `partials` można przechowywać części stron. Więcej informacji na ten temat można znaleźć w dokumentacji:

<https://laravel.com/docs/10.x/views>

Widoki generowane przez PHP wymagają łączenia kodu HTML z PHP co nie jest ani wygodne, ani eleganckie. W większych aplikacjach aby uniknąć takiego zapisu korzysta się z tzw. silników szablonów. Laravel korzysta z silnika szablonów Blade. Pliki szablonów Blade mają zwykle rozszerzenie `.blade.php`. Więcej informacji na temat silnika Blade w Laravelu można znaleźć na przykład tutaj:

<https://laravel.com/docs/10.x/blade>

<https://laravelpolska.com/blog/laravel-krok-po-kroku-czesc-3-blade>

Dla aplikacji `laravel_ksiazki` utworzymy dwa pliki z częściami strony, które będą dołączane do każdej strony aplikacji. Te części to nagłówek i główne menu.

W katalogu `resources/views/partials` utwórz plik **`head.blade.php`** z następującą zawartością:

```
<head>
  <meta charset="UTF-8">
  <title>Aplikacja Książki</title>
  <link rel="stylesheet" href="{{ URL::asset('styles.css') }}" />
</head>
```

Jak widać zostały tu umieszczone wszystkie niezbędne elementy tzn. deklaracja kodowania znaków, tytuł aplikacji oraz podłączenie pliku ze stylem.

W tym samym katalogu utwórz plik `navi.blade.php` z następującą zawartością:

```
<nav>
  <div>
    <div>
      <a>Strona domowa<span>(current)</span></a>
      <a href="/ksiazki">Książki</a>
      <a href="/dodaj_kategorie">Dodaj kategorię</a>
      <a href="/dodaj_ksiazke">Dodaj książkę</a>
      <a href="/loguj">Zaloguj</a>
    </div>
  </div>
</nav>
```

Utworzone części wykorzystamy przy tworzeniu strony domowej aplikacji. W katalogu `resources/views` przyjrzyj się zawartości istniejącego pliku `welcome.blade.php`, a następnie utwórz plik strony domowej np. **`domowa.blade.php`** z następującą zawartością:

```
<!DOCTYPE html>
<html lang="pl">
  @include('partials.head')
```

```
<body>
  @include('partials.navi')
  <div id="zawartosc">
    <h1>Witaj w aplikacji Laravel - Książki</h1>
  </div>
</body>
</html>
```

Zwróć uwagę w jaki sposób następuje odwołanie do zdefiniowanych wcześniej fragmentów head oraz navi (polecenie `@include`). Sprawdź jak co wygląda w przeglądarce.

9. Zdefiniowanie podstawowego kontrolera aplikacji

Żeby uruchomić aplikację w architekturze MVC potrzebny jest kontroler sterujący ruchem w aplikacji. W Laravel jest to klasa dziedzicząca z klasy `Controller`. Szkielet takiej klasy można utworzyć korzystając z polecenia `artisan make:controller` z nazwą klasy, a jednocześnie pliku. Na przykład:

```
$ php artisan make:controller PodstawowyKontroler
```

Wykonaj polecenie, a następnie odzyskaj kontroler w katalogu **app/Http/Controllers** i sprawdź jego zawartość.

Do utworzonej klasy kontrolera dodaj funkcję zwracającą w wyniku widok zapisany w pliku **domowa.blade.php**. Kod podano poniżej. Cała klasa powinna mieć teraz postać:

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

class PodstawowyKontroler extends Controller{
    public function zwrocStroneDomowa(){
        return view('domowa');
    }
}
```

10. Ustawienie odpowiedniego routingu

Każdą taką funkcję trzeba skojarzyć z odpowiednim URL aplikacji (routing oznacza kierowanie żądania do odpowiedniego kontrolera). Tutaj służy do tego plik **web.php** znajdujący się w katalogu **routes**. Otwórz ten plik i dopisz znaki komentarza do istniejącej definicji domyślnego routingu:

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

Na końcu dopisz skierowanie do strony **domowa** po wejściu do głównego katalogu aplikacji:

```
Route::get('/', [PodstawowyKontroler::class, 'zwrocStroneDomowa']);
```

Na początku kodu zadeklaruj użycie pliku kontrolera dopisując:

```
use App\Http\Controllers\PodstawowyKontroler;
```

Odśwież widok aplikacji w przeglądarce. Pamiętaj, że URL powinien wskazywać katalog public. Główna strona aplikacji powinna mieć teraz postać:



Obejrzyj kod źródłowy strony w przeglądarce. Przypomnij sobie, albo sprawdź, że do kodu nagłówka w pliku **navi.blade.php** dodane zostało połączenie z plikiem stylów CSS o nazwie **styles.css**. Pliki przeznaczone dla strony klienta, które nie są przetwarzane po stronie serwera powinny być zapisywane bezpośrednio w katalogu public. Utwórz w tym katalogu plik **styles.css** i umieść w nim następujący kod:

```
@charset "UTF-8";
div#zawartosc {color:#888; margin-left:10em; padding: 1px 0 1px 1em;}
nav {float:left; width:8em; background:#fef3a5; margin:0;}
nav a {display:block; padding:0.2em 0 0.2em 0.5em; text-decoration:none;
margin:0; color:black;}
nav a:hover {background-color:#fdc153;}
```

Strona w przeglądarce powinna mieć teraz postać:



Oczywiście, można również podłączyć do aplikacji framework do CSS, np. Bootstrap.

Więcej informacji na temat budowania układów stron i widoków można znaleźć pod adresem:

<https://laravel.com/docs/10.x/blade#building-layouts>

11. Strona z listą książek i ich kategorii

Utworzenie kolejnych stron wymaga skonstruowania odpowiedniego widoku oraz dodania odpowiedniego kodu do kontrolera PodstawowyKontroler. Większość stron będzie jednak korzystała z bazy danych, trzeba więc połączyć się z bazą. Na początek pobierzemy z bazy danych listę książek i wyświetlimy je na ekranie.

Utwórz nową stronę widoku o nazwie **lista_ksiazek.blade.php** i umieść na niej następujący kod:

```
<!DOCTYPE html>
<html lang="pl">
  @include('partials.head')
  <body>
```

```
@include('partials.navi')
<div id="zawartosc">
  <h2>Lista książek</h2>
  <table>
    <thead>
      <tr> <th>Tytuł</th> <th>Kategoria</th> </tr>
    </thead>
    <tbody>
      @foreach($ksiazki as $el)
        <tr> <td>{{ $el->tytul }}</td> <td>{{ $el->opis }}</td> </tr>
      @endforeach
    </tbody>
  </table>
</div>
</body>
</html>
```

Jak można zauważyć na stronie nie ma kodu PHP. Zamiast niego znajdują się tu znaczniki silnika szablonów Blade @.... Oprócz @include wykorzystana została instrukcja @foreach do przetwarzania kolejnych elementów tabeli książka i wyświetlania ich w strukturze tabeli HTML. Znaczniki Blade w trakcie kompilacji są zastępowane kodem PHP, dlatego oba zapisy są podobne. Więcej informacji na temat instrukcji iteracyjnych i warunkowych Blade nazywanych również dyrektywami można znaleźć w dokumentacji.

<https://laravel.com/docs/10.x/blade#blade-directives>

Użycie na stronie danych z bazy wymaga ich pobrania. W tym przykładzie nie będziemy tworzyć do tego celu modelu jak zakłada MVC. Kod pobierający dane z bazy dodamy po prostu do funkcji kontrolera. Kontroler PodstawowyKontroler będzie też potrzebował biblioteki Laravel do obsługi bazy. Przed definicją klasy kontrolera umieść instrukcję:

```
use Illuminate\Support\Facades\DB;
```

Do klasy dodaj następującą funkcję:

```
public function zwrocListeKsiazek()
{
    $ksiazkiZBazy = DB::table('ksiazka')->leftJoin('kategoria',
        'ksiazka.idkat', '=', 'kategoria.id') -> get();
    return view('lista_ksiazek', ['ksiazki' => $ksiazkiZBazy,]);
}
```

Podana funkcja zawiera dwie instrukcje. W pierwszej, na podstawie podanego kodu, Laravel konstruuje zapytanie SQL `SELECT * FROM ksiazka...` do systemu baz danych i zapisuje kolejne książki w postaci obiektów w tablicy PHP `$ksiazkiZBazy`. W drugiej ta tablica jest kojarzona z nazwą zmiennej, pod którą będzie widoczna w kodzie .html oraz następuje przekazanie tablicy do widoku `lista_ksiazek`. Ponieważ takich danych może być więcej przekazywanie odbywa się przy pomocy kolejnej tablicy [].

Tak jak poprzednio trzeba zdefiniować routing, czyli dodać powiązanie funkcji z odpowiednim URL. Dokonaj edycji pliku `routes/web.php` i dodaj na końcu wiersz:

```
Route::get('/ksiazki', [PodstawowyKontroler::class, 'zwrocListeKsiazek']);
```

12. Dodawanie nowych książek

Zdefiniuj widok z formularzem w pliku **dodaj_ksiazke.blade.php**, który umożliwi dodawanie do bazy nowych książek. W tym celu obejrzyj i wykorzystaj następujący kod:

```
<!DOCTYPE html>
<html>
@include('partials.head')
<body>
  @include('partials.navi')
  <div id="zawartosc">
    <h2>Dodaj książkę</h2>
    <form class="form-inline" action = "../dodaj_ksiazke" method = "post" >
      @csrf
      <p>
        <label for="tytul">Tytuł książki</label>
        <input id="tytul" name="tytul" size="20">
      </p>
      <p>
        @foreach($kategorie as $el)
          <input type="radio" name="idkat" id="idkat" value={{$el->id}}>
          <label for="idkat">{{$el->opis}}</label>
        @endforeach
      </p>
      <p><button type="submit" class="btn btn-primary mb-2">Dodaj</button></p>
    </form>
  </div>
</body>
</html>
```

Zauważ, że formularz jest wysyłany metodą post, natomiast wewnątrz formularza, oprócz standardowych pól, znajduje się znacznik @csrf, który umieści w formularzu ukryte pole. Znacznik ten umożliwia przekierowywanie żądań do walidatora ochrony CSRF (*Cross-site request forgery*).co zapobiega wysyłaniu niebezpiecznych treści.

W klasie kontrolera umieść kolejne dwie funkcje:

```
public function zwrocDodajKsiazke(){
  $kategorieZBazy = DB::table('kategoria')->get();
  return view('dodaj_ksiazke', ['kategorie' => $kategorieZBazy,]);
}

public function dodajKsiazke(Request $request){
  $tytulZFormularza = $request->tytul;
  $idKategoriiZFormularza = $request->idkat;
  DB::table('ksiazka')->insert([
    'tytul' => $tytulZFormularza,
    'idkat' => intval($idKategoriiZFormularza),
    'idwyd' => intval(0),
  ]);
  return redirect('/ksiazki');
}
```


Funkcja `zwrocDodajKsiazke` umożliwia wywołanie widoku `dodaj_ksiazke`, w którym umieszczono formularz.. Tym razem jest tam przekazywana lista kategorii, które zostały odczytane z bazy danych.

Druga funkcja, `dodajKsiazke` zajmuje się samym dodawaniem . Została w niej opisana instrukcja `INSERT` języka `SQL`. Ponieważ nie mamy na razie konstrukcji do pobierania i wyświetlania w formularzu wydawnictw do pola `idwyd` jest wpisywana zawsze wartość `0`.

W pliku `web.php` umieść następujące instrukcje:

```
Route::get('/dodaj_ksiazke', [PodstawowyKontroler::class, 'zwrocDodajKsiazke']);
Route::post('/dodaj_ksiazke', [PodstawowyKontroler::class, 'dodajKsiazke']);
```

Sprawdź działanie aplikacji.

13. Zadania do wykonania

- Zmodyfikuj link do strony głównej w menu, tak żeby działa poprawnie.
- Dodaj do aplikacji strony wyświetlające listy kategorii i wydawnictw.
- Dodaj do aplikacji strony do dodawania nowych kategorii i wydawnictw.
- Dodaj do aplikacji możliwość wyboru wydawnictwa przy dodawaniu książek za pomocą listy rozwijalnej / pola kombi.
- Zdefiniuj klucze obce w tabeli `ksiazka` (patrz Uwaga w punkcie 7), a następnie dodaj do aplikacji możliwość usuwania danych. Pamiętaj o zależnościach pomiędzy tabelami.
- Dodaj do aplikacji kod do walidacji danych w formularzach: nie można wysłać formularza, jeśli wszystkie pola nie zostały uzupełnione, w polach tekstowych muszą być wpisane co najmniej 3 znaki i nie więcej niż 50 znaków.
- Zmodyfikuj inne elementy zgodnie z własnymi pomysłami.

14. Uwierzytelnianie użytkowników

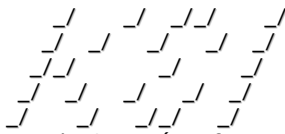
W menu aplikacji `laravel_ksiazki` znajduje się opcja `Zaloguj`, która nie została do tej pory oprogramowana. Zauważ, że `Laravel` wyświetla w takim przypadku automatycznie stronę z informacją o błędzie `404 - Not found`.

Podstawowa biblioteka (pakiet) wspierająca uwierzytelnianie użytkowników to `Breeze`. Dostarcza takich możliwości jak logowanie, rejestracja, zmiana hasła oraz weryfikacja i potwierdzanie hasła poprzez email. Umożliwia również autoryzację poprzez kontrolę dostępu do poszczególnych stron.

UWAGA: Instalacja nowego pakietu spowoduje napisanie pliku **web.php**. Wykonaj w terminalu `ssh` kopię pliku **routes/web.php** w celu skopiowania do nowego pliku wcześniej dopisanych tras routingu. Powinien pojawić się np. plik **routes/web_copy.php**.

Dodanie pakietu `Breeze` wymaga pobrania przez `composer` oraz umieszczenia odpowiedniego zapisu w pliku `.composer.json` w katalogu głównym aplikacji `laravel_ksiazki`. Korzystając z `putty` lub terminala przejdź do katalogu aplikacji i wywołaj polecenie:

```
composer require laravel/breeze --dev
```



```
82 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force

INFO No publishable resources for tag [laravel-assets].

No security vulnerability advisories found
Using version ^1.26 for laravel/breeze
```

Następnie, trzeba zainstalować pakiet breeze w aplikacji. Spowoduje to wygenerowanie plików i kodu: widoków, kontrolerów, tras (routes) i innych zasobów. W trakcie instalacji trzeba odpowiedzieć na pytania o preferowany frontend i Framework do testowania. Wybierz domyślne opcje: Blade with Alpine and PHPUnit. Wykonaj polecenie:

```
php artisan breeze:install
```

Sprawdź zmiany. W katalogu **views** pojawił się nowy widok **dashboard.blade.php** oraz nowe podkatalogi z widokami stron do uwierzytelniania użytkowników. Sprawdź również zmiany w zawartości pliku **routes/web.php**.

Przejdź do utworzonego katalogu **views/auth** i otwórz do edycji pliki **login.blade.php** oraz **register.blade.php**. Żeby korzystać z własnego szablonu trzeba usunąć dodane przez breeze znaczniki `<x-guest-layout>`. Nie będą również potrzebne znaczniki `<x-auth-card>`, `<x-slot>`, ani logo.

Znaczniki rozpoczynające się od `<x-` oznaczają komponenty silnika Blade. Ponieważ można je definiować Breeze dodał własne. Ich kody znajdują się w katalogu **resources/views/components**.

Usuń kod z pierwszej i ostatniej linii w plikach do logowania i rejestracji ze znacznikiem `</x-guest-layout>`.

Dodaj znaczniki struktury HTML oraz zdefiniowany dla naszej aplikacji nagłówek i menu na początku obu plików:

```
<!DOCTYPE html>
<html>
@include('partials.head')
<body>
    @include('partials.navi')
```

Zamknij znaczniki na końcu pliku.

```
</body>
</html>
```

Za instrukcją `@include('partials.navi')` umieść znacznik `<div>` oraz nagłówek. W pliku **login.blade.php**:

```
<div id="zawartosc">
    <h2>Logowanie</h2>
```

Natomiast w pliku **register.blade.php**:

```
<div id="zawartosc">
    <h2>Rejestracja</h2>
```

Nie zapomnij zamknąć `div` na końcu obu plików.

Pliki są teraz widoczne w przeglądarce po dopisaniu do URL, odpowiednio: /login lub /register. Potrzebna będzie jeszcze strona, z informacją o wylogowaniu. W katalogu views utwórz nowy plik widoku o nazwie **wylogowano.blade.php**. Umieść w nim tylko nagłówek h2 z informacją o wylogowaniu:

```
<!DOCTYPE html>
<html id="pl">
@include('partials.head')
<body>
    @include('partials.navi')
    <div id="zawartosc">
        <h2>Wylogowano</h2>
    </div>
</body>
</html>
```

Pozostaje jeszcze zdefiniowanie funkcji w kontrolerze oraz tras w pliku **web.php**.

W pliku **PodstawowyKontroler.php** dodaj funkcję o nazwie **zmienStanUwierzytelnienia**, która sprawdzi, czy użytkownik jest uwierzytelniony, jeśli tak to wywoła funkcję **logout**, jeśli nie, przekieruje do rejestracji. Kod takiej funkcji może wyglądać następująco:

```
public function zmienStanUwierzytelnienia()
{
    if(auth()->check()){
        $user = auth()->user();
        Auth::logout();
        return view('wylogowano');
    }
    else{
        return redirect('/register');
    }
}
```

Aby podany fragment kodu działał poprawnie trzeba dodać informację o wykorzystaniu klasy tzw. fasady Auth na początku pliku:

```
use Illuminate\Support\Facades\Auth;
```

W pliku **web.php** dodaj znaki komentarza do trasy Route, wygenerowanej przez Breeze, przekierowującej do widoku dashboard. Zakomentuj też, tak jak poprzednio, domyślną trasę do widoku welcome.

W przypadku jeśli instalator breeze nadpisał wcześniej dopisane trasy w pliku **web.php** skopiuj je z pliku kopii **web_copy.php** (trasy: /, książki, dodaj_książke (get i post)).

Nie zapomnij dołączyć do pliku **web.php** informacji o użyciu podstawowego kontrolera

```
use App\Http\Controllers\PodstawowyKontroler;
```

Sprawdź, czy w kodzie istnieje ostatni wiersz w pliku **web.php** oraz czy jest to wymaganie dołączenia pliku **auth.php** w postaci:

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

Dopisz do pliku **web.php** obsługę opcji Zaloguj i Wyloguj z menu jako funkcję `zmienStanUwierzytelnienia`. Zdefiniuj trasę:

```
Route::get('/loguj',
[PodstawowyKontroler::class,'zmienStanUwierzytelnienia']);
Route::get('/wyloguj',[PodstawowyKontroler::class,'zmienStanUwierzytelnien
ia']);
Sprawdź działanie aplikacji.
```

Uwaga!

Pakiet Breeze generuje URL HOME z domyślną trasą `/dashboard`, z której nie korzystamy. Naszą domyślną trasą powinien być katalog główny czyli `/`.
W pliku: `App/Providers/RouteServiceProvider.php`
dodaj komentarz do wiersza `//public const HOME = '/dashboard';`
natomiast w zamian wprowadź własny: `public const HOME = '/';`

Sprawdź ponownie działanie rejestracji i logowania.

Zauważ, że można się zarejestrować i zalogować, ale zawsze widoczne są wszystkie strony. Zabroń otwierać strony z dodawaniem danych do bazy użytkownikom, którzy nie są uwierzytelnieni. Rezultat taki można osiągnąć dodając do żądania `get` pośrednika za pomocą metody `middleware('auth')`. Dla strony `dodaj_ksiazke` trasa powinna mieć postać:

```
Route::get('/dodaj_ksiazke',
[PodstawowyKontroler::class, 'zwrocDodajKsiazke' ]) -> middleware('auth');
```

Po poprawnym dopisaniu tras w pliku `web.php` mogą one wyglądać tak jak przedstawiono poniżej.

```
Route::get('/', [PodstawowyKontroler::class,'zwrocStroneDomowa']);
Route::get('/ksiazki', [PodstawowyKontroler::class,'zwrocListeKsiazek']);

Route::get('/dodaj_ksiazke', [PodstawowyKontroler::class,'zwrocDodajKsiazke'])->
middleware('auth');
Route::post('/dodaj_ksiazke', [PodstawowyKontroler::class,'dodajKsiazke']);
Route::get('/loguj',[PodstawowyKontroler::class,'zmienStanUwierzytelniania']);
Route::get('/wyloguj',[PodstawowyKontroler::class,'zmienStanUwierzytelniania'])
;

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

15. Testowanie autoryzacji

Próba wybrania opcji Dodaj książkę z menu powinna kończyć się teraz przekierowaniem do rejestracji. Zarejestruj nowego użytkownika i sprawdź co się zmieniło w tabeli `users` w bazie danych. Tabela ta wygenerowała się podczas uruchamiania kodów z plików migracji.

Opcja Zaloguj w menu powinna zmienić nazwę po zalogowaniu użytkownika na Wyloguj. W pliku **navi.blade.php** dodaj do opcji Zaloguj następujący kod:

```
@if(Auth::check())
<a href="/wyloguj">Wyloguj</a>
```

```
@else  
    <a href="./loguj">Zaloguj</a>  
@endif
```

Jak widać sprawdzanie zalogowania jest możliwe dzięki wykorzystaniu fasady Auth i metody `check()`. Linki będą miały różne nazwy, ale przekierują użytkownika do tej samej funkcji. Fasada jest klasą o specyficznych możliwościach. W Laravel oznacza statyczny interfejs do obiektów innych klas. Zapis w przykładzie nie jest więc odwołaniem do metody statycznej, tylko konkretnego obiektu poprzez statyczny interfejs. Laravel dostarcza wielu fasad, które znacznie ułatwiają korzystanie z obiektów w aplikacji. Dostarcza także tzw. funkcji pomocniczych ułatwiających korzystanie z fasad. Więcej informacji na ten temat można znaleźć między innymi w dokumentacji.

<https://laravel.com/docs/10.x/facades>

Oprócz pakietu Breeze, Laravel oferuje jeszcze dwa pakiety wspomagające uwierzytelnianie: Jetstream i Fortify. Możliwe jest również napisanie kodu własnego uwierzytelniania. Więcej informacji na ten temat można znaleźć w dokumentacji.

<https://laravel.com/docs/10.x/authentication#authentication-quickstart>

16. Zadania do samodzielnego wykonania

- Zablokuj niezalogowanemu użytkownikowi otwierać wszystkie strony dodające dane do bazy danych.
- Strony dodające dane do bazy wyświetl w menu tylko dla zalogowanego użytkownika.
- Zmień opisy na stronach logowania i rejestracji na język polski.
- Dodaj stronę z informacją o poprawnym zalogowaniu

Sprawozdanie

W **Sprawerze** wyślij **link** do swojej aplikacji `laravel_ksiazki` działającej na serwerze foka.

Uzupełnij wiadomości na temat poleceń CRUD oraz poprawnej struktury aplikacji w architekturze MVC. Link do filmiku na temat CRUD i MVC w Laravel:

<https://www.youtube.com/watch?v=CtkNd00RQTO>