



Technologia i rozwiązania

Laravel

Tworzenie aplikacji Receptury

Najlepsze przepisy na aplikację w szkieletcie Laravel!



Terry Matula



Tytuł oryginału: Laravel Application Development Cookbook

Tłumaczenie: Mirosław Gołda

ISBN: 978-83-283-0302-7

Copyright © Packt Publishing 2013.

First published in the English language under the title:
„Laravel Application Development Cookbook”.

Polish edition copyright © 2015 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/larare>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorze	7
O recenzentach	8
Wstęp	9
Rozdział 1. Instalowanie Laravela	13
Wprowadzenie	13
Instalowanie Laravela w postaci modułu zależnego repozytorium git	14
Konfigurowanie hostów wirtualnych i środowiska deweloperskiego na serwerze Apache	15
Tworzenie „czystych” adresów URL	17
Konfigurowanie Laravela	18
Praca z Laravelem w edytorze Sublime Text 2	19
Konfigurowanie środowiska IDE pozwalające włączyć autouzupełnianie przestrzeni nazw dla Laravela	21
Wykorzystywanie mechanizmu autoloadera do mapowania pomiędzy nazwą klasy a plikiem z jej definicją	24
Tworzenie zaawansowanych autoloaderów z przestrzeniami nazw i katalogami	25
Rozdział 2. Pobieranie danych wejściowych	29
Wprowadzenie	29
Tworzenie prostego formularza	30
Pobieranie danych z formularza i wyświetlenie ich na innej stronie	31
Walidacja danych wysłanych przez użytkownika	33
Tworzenie mechanizmu przesyłania plików	35
Walidacja przesyłanych plików	37
Tworzenie własnego komunikatu o błędzie	39
Dodawanie „wabika” do formularza	42
Przesyłanie obrazów za pomocą biblioteki Redactor	44
Przycinanie obrazu za pomocą biblioteki Jcrop	47
Tworzenie pola tekstowego z autouzupełnianiem	50
Tworzenie mechanizmu przechwytywania spamu w stylu CAPTCHA	53

Rozdział 3. Uwierzytelnianie w Twojej aplikacji	57
Wprowadzenie	57
Instalowanie i konfigurowanie biblioteki Auth	58
Tworzenie systemu uwierzytelniania	60
Pobieranie i aktualizowanie danych o użytkowniku po zalogowaniu	64
Ograniczanie dostępu do wybranych stron	67
Konfigurowanie uwierzytelniania OAuth z użyciem pakietu HybridAuth	69
Wykorzystywanie OpenID do logowania	70
Logowanie z poświadczeniami Facebooka	72
Logowanie z poświadczeniami Twittera	74
Logowanie z poświadczeniami LinkedIn	76
Rozdział 4. Przechowywanie i wykorzystywanie danych	79
Wprowadzenie	80
Tworzenie tabel w bazie danych z zastosowaniem migracji i schematów	80
Tworzenie zapytań w języku SQL	83
Tworzenie zapytań z użyciem konstruktora Fluent	85
Tworzenie zapytań z wykorzystaniem systemu ORM Eloquent	88
Automatyczna walidacja w modelach	90
Relacje i zaawansowane funkcje systemu Eloquent	93
Tworzenie systemu CRUD	95
Importowanie plików CSV za pomocą systemu Eloquent	99
Kanały RSS jako źródła danych	101
Atrybuty do zmiany nazw kolumn w tabeli	102
Zastąpienie systemu Eloquent innym systemem ORM	105
Rozdział 5. Wykorzystywanie kontrolerów i routingu do obsługi adresów URL i tworzenia API	109
Wprowadzenie	110
Tworzenie prostych kontrolerów	110
Tworzenie routingu z wykorzystaniem domknięcia	111
Tworzenie kontrolera REST-owego	112
Zaawansowane opcje routingu	113
Filtry w routingu	115
Grupy reguł routingu	116
Tworzenie REST-owego API z wykorzystaniem routingu	118
Nazwany routing	124
Nazwa subdomeny w routingu	125
Rozdział 6. Wyświetlanie widoków	129
Wprowadzenie	129
Tworzenie i wykorzystywanie prostego widoku	130
Przekazywanie danych do widoku	131
Wczytywanie widoku do innego (zagnieżdżonego) widoku	133
Dodawanie zasobów	136

Tworzenie widoków z zastosowaniem systemu szablonów Blade	138
System szablonów Twig	140
Zaawansowane możliwości systemu Blade	142
Tworzenie zlokalizowanej zawartości	145
Tworzenie menu w Laravelu	148
Integracja z Bootstrapem	151
Nazwane widoki i kompozytory widoków	153
Rozdział 7. Tworzenie i wykorzystywanie pakietów Composer'a	157
Wprowadzenie	157
Pobieranie i instalowanie pakietów	158
Pakiet Generators do tworzenia szkieletu aplikacji	161
Tworzenie pakietu Composer'a w Laravelu	165
Dodawanie pakietów Composer'a do Packagista	169
Dodawanie pakietu spoza Packagista do Composer'a	171
Tworzenie własnego polecenia dla Artisana	173
Rozdział 8. Ajax i jQuery	177
Wprowadzenie	177
Pobieranie danych z innej strony	178
Konfigurowanie kontrolera tak, aby zwracał dane w formacie JSON	181
Tworzenie funkcji wyszukiwania z wykorzystaniem techniki Ajax	183
Tworzenie i walidowanie użytkownika z wykorzystaniem techniki Ajax	185
Filtrowanie danych na podstawie pól wyboru	188
Tworzenie okna rejestracji do newslettera z użyciem techniki Ajax	191
Wysyłanie wiadomości e-mail z zastosowaniem Laravela i biblioteki jQuery	194
Tworzenie tabeli z możliwością sortowania przy użyciu Laravela i biblioteki jQuery	197
Rozdział 9. Efektywne wykorzystywanie mechanizmów bezpieczeństwa i sesji	201
Wprowadzenie	201
Szyfrowanie i odszyfrowywanie danych	202
Hashowanie haseł i innych danych	205
Tokeny CSRF i filtry w formularzach	208
Zaawansowana walidacja w formularzach	210
Tworzenie koszyka zakupowego	213
Wykorzystywanie Redisa do przechowywania sesji	216
Podstawowe zabezpieczenie sesji i ciasteczek	218
Tworzenie bezpiecznego serwera API	221
Rozdział 10. Testowanie i debugowanie aplikacji	225
Wprowadzenie	225
Instalowanie i konfigurowanie biblioteki PHPUnit	226
Tworzenie i uruchamianie testów	227
Wykorzystywanie biblioteki Mockery do testowania kontrolerów	228
Pisanie testów akceptacyjnych z użyciem biblioteki Codeception	231
Debugowanie i profilowanie aplikacji	233

Rozdział 11. Wdrażanie i integrowanie aplikacji z usługami firm trzecich	237
Wprowadzenie	237
Tworzenie kolejek i wykorzystywanie Artisana do ich uruchamiania	238
Wdrażanie aplikacji Laravela na platformę Pagoda Box	240
Używanie bramki płatności Stripe z frameworkiem Laravel	244
Przeszukiwanie bazy GeoIP i konfiguracja własnego routingu	247
Gromadzenie adresów e-mail i ich wykorzystywanie w usługach pocztowych firm trzecich	248
Przechowywanie i pobieranie zawartości zapisanej w chmurze Amazon S3	251
Skorowidz	255

Wyświetlanie widoków

W tym rozdziale omówimy:

- Tworzenie i wykorzystywanie prostego widoku
- Przekazywanie danych do widoków
- Wczytywanie widoku do innego (zagnieżdżonego) widoku
- Dodawanie zasobów
- Tworzenie widoków z zastosowaniem systemu szablonów Blade
- System szablonów Twig
- Zaawansowane możliwości systemu Blade
- Tworzenie zlokalizowanej treści
- Tworzenie menu w Laravelu
- Integracja z Bootstrapem
- Nazwane widoki i kompozytory widoków

Wprowadzenie

Zgodnie ze wzorcem *Model-View-Controller* **widoki** przechowują cały kod HTML strony wraz ze stylami i wykorzystywane są do wyświetlania przekazanych danych. W Laravelu widokami mogą być zwykłe pliki PHP lub pliki systemu szablonów frameworka Laravel, systemu Blade. Laravel jest równocześnie na tyle rozszerzalny, że umożliwia stosowanie dowolnego wybranego systemu szablonów.

Tworzenie i wykorzystywanie prostego widoku

W tej recepturze poznamy podstawowe funkcjonalności **widoków** i sposoby załączania widoków w aplikacji.

Przygotowanie

Będziemy bazować na standardowej instalacji Laravela.

Jak to zrobić...

Postępuj zgodnie z poniższymi wskazówkami, aby wykonać wszystkie kroki receptury:

1. W katalogu `app/views` utwórz folder `myviews`.
2. W katalogu `myviews` utwórz dwa pliki: `home.php` oraz `second.php`.
3. Otwórz plik `home.php` i wprowadź następujący kod HTML:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Strona domowa</title>
  </head>
  <body>
    <h1>Witamy na stronie domowej!</h1>
    <p>
      <a href="second">Przejdź na drugą stronę</a>
    </p>
  </body>
</html>
```

4. Otwórz plik `second.php` i wprowadź następujący kod HTML:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Druga strona</title>
  </head>
  <body>
    <h1>Witamy na drugiej stronie</h1>
    <p>
      <a href="home">Przejdź na stronę domową</a>
    </p>
  </body>
</html>
```


5. W pliku *app/routes.php* dodaj routingi, które zwrócą oba widoki:

```
Route::get('home', function()
{
    return View::make('myviews.home');
});
Route::get('second', function()
{
    return View::make('myviews.second');
});
```

6. Sprawdź działanie widoków, odwiedzając adres *http://{twój-serwer}/home* (gdzie *twój-serwer* jest adresem URL serwera), a następnie kliknij łącze.

Jak to działa...

Wszystkie widoki w Laravelu przechowywane są w katalogu *app/views*. Rozpoczynamy od utworzenia dwóch plików, w których zamieszczamy kod HTML. W naszym przykładzie tworzymy statyczne strony, a każdy utworzony widok przechowuje pełny kod HTML strony.

W pliku routingu zwracamy metodę `View::make()` z parametrem będącym nazwą widoku. Ze względu na fakt, że pliki widoków znajdują się w podkatalogach katalogu widoków, wykorzystujemy notację kropkową.

Przekazywanie danych do widoku

W typowej aplikacji musimy wyświetlać różnego rodzaju dane, pochodzące z bazy danych lub innego źródła. W Laravelu przekazywanie danych do widoków jest bardzo proste.

Przygotowanie

W bieżącej recepturze bazujemy na kodzie powstałym po ukończeniu receptury „Tworzenie i wykorzystywanie prostego widoku”.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. Otwórz plik *routes.php* i zastąp kod strony domowej i drugiej strony następującym kodem:

```
Route::get('home', function()
{
    $page_title = 'Tytuł strony domowej';
```

```

        return View::make('myviews.home')->with('title',
            $page_title);
    });
Route::get('second', function()
{
    $view = View::make('myviews.second');
    $view->my_name = 'Jan Kowalski';
    $view->my_city = 'Kraków';
    return $view;
});

```

2. W katalogu `views/myviews` otwórz plik `home.php` i zastąp jego zawartość poniższym kodem:

```

<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Strona domowa : <?= $title ?></title>
  </head>
  <body>
    <h1>Witamy na stronie domowej!</h1>
    <h2><?= $title ?></h2>
    <p>
      <a href="second">Przejdź na drugą stronę</a>
    </p>
  </body>
</html>

```

3. W katalogu `views/myviews` otwórz plik `second.php` i zastąp jego zawartość poniższym kodem:

```

<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Druga strona</title>
  </head>
  <body>
    <h1>Witamy na drugiej stronie</h1>
    <p> Nazywasz się <?= $my_name ?> i pochodzisz z miasta <?= $my_city ?>
    </p>
    <p>
      <a href="home">Przejdź na stronę domową</a>
    </p>
  </body>
</html>

```

4. Sprawdź działanie widoków, odwiedzając adres `http://{twój-serwer}/home` (gdzie `twój-serwer` jest adresem URL serwera), a następnie kliknij łącze.

Jak to działa...

Laravel udostępnia kilka sposobów przekazywania danych do widoku. Rozpoczynamy od aktualizacji pierwszego routingu, tak aby wykorzystując kaskadowe wywołanie (ang. *chaining*) metody `with()` z metodą `View::make()`, przekazać do widoku jedną zmienną. Następnie w pliku widoku możemy używać tej zmiennej za pomocą wybranej nazwy.

W następnym routingu przypisujemy wynik metody `View::make()` do zmiennej i przekazywane wartości do właściwości obiektu. Teraz możemy w widoku stosować zmienne odpowiadające nazwom tych właściwości. Aby wyświetlić widok, zwracamy po prostu zmienną, do której przypisany jest obiekt.

To nie wszystko...

Inna metoda przekazania danych do widoku przypomina sposób zaprezentowany w drugim routingu, ale z użyciem tablicy, a nie obiektu. Nasz kod mógłby wyglądać podobnie jak widoczny poniżej:

```
$view = View::make('myviews.second');
$view['my_name'] = 'Jan Kowalski';
$view['my_city'] = 'Kraków';
return $view;
```

Wczytywanie widoku do innego (zagnieżdżonego) widoku

Bardzo często strony internetowe mają jednolity układ i zbliżoną do siebie strukturę kodu HTML. Możemy wyodrębnić powtarzający się kod HTML za pomocą mechanizmu **zagnieżdżonych widoków**.

Przygotowanie

Przed przystąpieniem do tej receptury konieczne jest wykonanie czynności opisanych w recepturze „Tworzenie i wykorzystywanie prostego widoku”.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. W katalogu *app/view* dodaj nowy folder o nazwie *common*.
2. W katalogu *common* utwórz plik *header.php* i wprowadź następujący kod:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Moja strona</title>
  </head>
</body>
```

3. W katalogu *common* utwórz plik *footer.php* i wprowadź następujący kod:

```
<footer>&copy; 2014 MojaFirma</footer>
</body>
</html>
```

4. W katalogu *common* utwórz plik *userinfo.php* i wprowadź następujący kod:

```
<p>Nazywasz się <?= $my_name ?> i pochodzisz z miasta <?= $my_city ?></p>
```

5. W pliku *routes.php* zaktualizuj routingsi strony głównej i drugiej strony tak, aby korzystały z zagnieżdżonych widoków:

```
Route::get('home', function()
{
    return View::make('myviews.home')
        ->nest('header', 'common.header')
        ->nest('footer', 'common.footer');
});
Route::get('second', function()
{
    $view = View::make('myviews.second');
    $view->nest('header', 'common.header')->nest('footer',
        'common.footer');
    $view->nest('userinfo', 'common.userinfo',
        array('my_name' => 'Jan Kowalski', 'my_city' => 'Kraków'));
    return $view;
});
```

6. W katalogu *views/myviews* otwórz plik *home.php* i wprowadź następujący kod:

```
<?= $header ?>
<h1>Witamy na stronie domowej!</h1>
<p>
  <a href="second">Przejdź na drugą stronę</a>
</p>
<?= $footer ?>
```

7. W katalogu *views/myviews* otwórz plik *second.php* i wprowadź następujący kod:

```
<?= $header ?>
<h1>Witamy na drugiej stronie</h1>
<?= $userinfo ?>
<p>
  <a href="home">Przejdź na stronę domową</a>
</p>
<?= $footer ?>
```

8. Sprawdź działanie widoków, odwiedzając adres *http://{twój-serwer}/home* (gdzie *twój-serwer* jest adresem URL serwera), a następnie kliknij łącze.

Jak to działa...

Na początek wydzielamy z naszych widoków kod nagłówka i stopki. Ponieważ na wszystkich stronach są one takie same, tworzymy podkatalog w folderze *views*, w którym przechowywać będziemy te wspólne dla wszystkich plików widoki. Pierwszym plikiem jest nagłówek, który zawiera cały kod HTML aż do tagu `<body>`. Drugim plikiem jest stopka, która zawiera kod HTML znajdujący się na dole strony.

Trzecim plikiem jest widok *userinfo* zawierający informacje o użytkowniku. W aplikacjach, w których konta użytkowników posiadają profile, potrzebujemy często załączyć dostępne informacje o użytkowniku w bocznej kolumnie lub w nagłówku. Aby przechowywać ten plik w oddzielnym widoku, tworzymy widok *userinfo* i wykorzystujemy przekazane do niego dane.

Do obsługi routingu *home* stosujemy widok *home*, w którym załączamy nagłówek i stopkę. Pierwszym parametrem metody `nest()` jest nazwa, używana w głównym widoku, a drugim parametrem jest lokalizacja widoku. W tym przykładzie widoki zlokalizowane są w niestandardowym podkatalogu, używamy więc notacji kropkowej, aby się do nich odwołać.

W widoku strony głównej wypisujemy nazwy zmiennych utworzone w routingu, aby wyświetlić zagnieżdżone widoki.

W drugim routingu oprócz nagłówka i stopki załączamy dodatkowo widok *userinfo*. W tym celu przekazujemy trzeci parametr do metody `nest()`, którym jest tablica danych przesłanych do widoku. Następnie, gdy widok *userinfo* jest włączany do głównego widoku, zmienne zostają do niego automatycznie przekazane „z góry”.

Zobacz także

- Receptura „Przekazywanie danych do widoku”.

Dodawanie zasobów

Aby dynamiczne strony internetowe mogły działać, potrzebują stylów CSS i skryptów JavaScript. Pakiet Asset frameworka Laravel umożliwia proste zarządzanie tymi zasobami i załączanie ich w widoku.

Przygotowanie

W tej recepturze będziemy bazować na kodzie utworzonym w recepturze „Wczytywanie widoku do innego (zagnieżdżonego) widoku”.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. Otwórz plik *composer.json* i w sekcji `require` dodaj pakiet Asset w podany niżej sposób:

```
"require": {
    "laravel/framework": "4.0.*",
    "tepluss/asset": "dev-master"
},
```

2. W linii poleceń zaktualizuj Composera, aby pobrać pakiet:

```
php composer.phar update
```

3. Otwórz plik *app/config/app.php* file i dodaj ServiceProvider na koniec tablicy w sekcji `providers`:

```
'Tepluss\Asset\AssetServiceProvider',
```

4. W tym samym pliku w tablicy aliasów dodaj skróconą nazwę pakietu:

```
'Asset' => 'Tepluss\Asset\Facades\Asset'
```

5. W pliku *app/filters.php* dodaj własny filtr do zasobów:

```
Route::filter('assets', function()
{
    Asset::add('jqueryui', 'http://ajax.googleapis.com/ajax
    /libs/jqueryui/1.10.2/jquery-ui.min.js', 'jquery');
    Asset::add('jquery', 'http://ajax.googleapis.com/ajax
    /libs/jquery/1.10.2/jquery.min.js');
    Asset::add('bootstrap', 'http://netdna.bootstrapcdn.com
    /twitter-bootstrap/2.3.2/css/
    bootstrap-combined.min.css');
});
```

6. Zaktualizuj routingi strony głównej i drugiej strony tak, aby korzystały z utworzonego filtra:

```
Route::get('home', array('before' => 'assets', function()
{
    return View::make('myviews.home')
        ->nest('header', 'common.header')
        ->nest('footer', 'common.footer');
}));
Route::get('second', array('before' => 'assets', function()
{
    $view = View::make('myviews.second');
    $view->nest('header', 'common.header')->nest
        ('footer', 'common.footer');
    $view->nest('userinfo', 'common.userinfo', array
        ('my_name' => 'Jan Kowalski', 'my_city' => 'Kraków'));
    return $view;
}));
```

7. W katalogu *views/common* otwórz plik *header.php* i wprowadź następujący kod:

```
<!doctype html>
<html lang="pl">
<head>
<meta charset="utf-8">
<title>Moja strona</title>
<?= Asset::styles() ?>
</head>
<body>
```

8. W katalogu *views/common* otwórz plik *footer.php* i wprowadź następujący kod:

```
<footer>&copy; 2014 MojaFirma</footer>
<?= Asset::scripts() ?>
</body>
</html>
```

9. Sprawdź działanie widoków, odwiedzając adres *http://{twój-serwer}/home* (gdzie *twój-serwer* jest adresem URL serwera), kliknij łącze i przejrzyj kod źródłowy strony, aby zobaczyć, że zasoby zostały załączone.

Jak to działa...

Pakiet Asset ułatwia dodawanie plików CSS i JavaScript do kodu HTML. Rozpoczynamy od „zarejestrowania” każdego zasobu w routingu. Aby uprościć ten proces, dodajemy filtr *assets*, wykorzystywany przed wywołaniem routingu. Dzięki temu kod odpowiedzialny za załączanie zasobów znajduje się w jednym miejscu i łatwo jest w nim wprowadzać wymagane zmiany. W naszym przykładzie dodajemy biblioteki jQuery, jQueryUI i plik CSS Bootstrapa z serwera CDN.

Pierwszym parametrem metody `add()` jest nazwa, którą nadajemy zasobowi. Drugim parametrem jest adres URL zasobu. Może nim być ścieżka względna lub pełen adres URL. Trzecim, opcjonalnym parametrem jest zależność zasobu. Do działania biblioteki jQueryUI wymagane jest wcześniejsze wczytanie biblioteki jQuery, podajemy więc jako trzeci parametr nazwę zasobu jQuery.

Następnie aktualizujemy routing i dodajemy filtr. Jeśli dodamy lub usuniemy jakiś zasób w filtrze, będzie to automatycznie odzwierciedlone w każdym routingu.

Korzystamy z zagnieżdżonych widoków, zasoby dodajemy więc tylko do nagłówka i stopki. Pliki CSS wywoływane są przez metodę `styles()`, a pliki JavaScript przez metodę `scripts()`. Laravel sprawdza rozszerzenia plików zasobów i umieszcza je automatycznie we właściwym miejscu. Sprawdzenie kodu źródłowego strony pozwala nam stwierdzić, że Laravel zadbał o to, aby skrypt jQuery znalazł się przed skryptem jQueryUI, ponieważ w ten sposób ustawiliśmy zależności.

Zobacz także

- Receptury „Filtr w routingu” w rozdziale 5., „Wykorzystywanie kontrolerów i routingu do obsługi adresów URL i tworzenia API”

Tworzenie widoków z zastosowaniem systemu szablonów Blade

W języku PHP przygotowano wiele systemów szablonów, a dostępny w Laravelu system Blade należy do najlepszych. W tej recepturze pokażemy, jak w szybki sposób rozpocząć pracę z systemem szablonów Blade.

Przygotowanie

W tej recepturze będziemy bazować na standardowej instalacji Laravela.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. W pliku `routes.php` utwórz nowy routing dla tworzonych stron:

```
Route::get('blade-home', function()
{
    return View::make('blade.home');
});
```



```
});
Route::get('blade-second', function()
{
    return View::make('blade.second');
});
```

2. W katalogu *views* utwórz folder o nazwie *layout*.

3. W katalogu *views/layout* utwórz plik *index.blade.php* o następującej zawartości:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Moja strona</title>
  </head>
  <body>
    <h1>
      @section('page_title')
        Witamy na
      @show
    </h1>
    @yield('content')
  </body>
</html>
```

4. W katalogu *views* utwórz folder *blade*.

5. W katalogu *views/blade* utwórz plik *home.blade.php* i wprowadź następujący kod:

```
@extends('layout.index')

@section('page_title')
  @parent
  Strona domowa utworzona z użyciem systemu Blade
@endsection

@section('content')
  <p>
    Przejdź na {{ HTML::link('blade-second',
      'drugą stronę.') }}
  </p>
@endsection
```

6. W katalogu *views/blade* utwórz plik *second.blade.php* o następującej zawartości:

```
@extends('layout.index')

@section('page_title')
  @parent
  Druga strona utworzona z użyciem systemu Blade
@endsection
```

```

@section('content')
    <p>
        Przejdź na {{ HTML::link('blade-home', 'stronę główną.')
        }}
    </p>
@endsection

```

7. Sprawdź działanie widoków, odwiedzając adres `http://{twój-serwer}/blade-home` (gdzie `twój-serwer` jest adresem URL serwera), i kliknij łącze oraz wyświetl kod źródłowy strony, aby zobaczyć, że szablon Blade został załadowany.

Jak to działa...

Rozpoczynamy od utworzenia dwóch prostych routingów, które zwracają widoki Blade. Używając notacji kropkowej, ustawiamy lokalizację plików szablonów na podkatalog `blade` w folderze `views`.

Następnym krokiem jest utworzenie pliku układu strony w systemie Blade. Plik ten stanowi szkielet strony i znajduje się w podkatalogu `layout`, w folderze `views`. Musimy mu nadać rozszerzenie `blade.php`. Widok jest prostym plikiem HTML z wyjątkiem obszarów `@section()` oraz `@yield()`. Ich zawartość zostanie zamieniona lub dodana do widoków.

Widoki dla routingów rozpoczynamy od deklaracji układu strony, z którego chcemy korzystać. W naszym przypadku jest to wywołanie `@extends('layout.index')`. Następnie dodajemy lub zmieniamy zawartość sekcji zadeklarowanych w układzie. W sekcji `page_title` chcemy wyświetlić tekst zapisany w układzie, ale uzupełniony o dodatkową treść. W tym celu wywołujemy `@parent` jako pierwszy element w obszarze, a następnie dopisujemy pożądaną zawartość.

W obszarze `@section('content')` nie był zadeklarowany żaden domyślny tekst, dodana zawartość będzie więc tworzyć całą treść obszaru. System Blade pozwala nam również stosować nawiasy `{{ }}` do zamieszczenia dowolnego kodu PHP. Używamy metody Larela `HTML::link()` do wyświetlenia łącza. Teraz po przejściu na stronę cała jej zawartość znajdzie się we właściwym miejscu w strukturze układu strony.

System szablonów Twig

Chociaż praca z systemem szablonów Blade jest bardzo przyjemna, w niektórych przypadkach musimy zastosować inną bibliotekę. Popularnym rozwiązaniem jest Twig. W bieżącej recepturze pokażemy, jak włączyć system szablonów Twig do aplikacji stworzonej w Laravelu.

Przygotowanie

W tej recepturze wykorzystamy standardową instalację Larela.

Jak to zrobić...

Postępuj zgodnie z poniższymi wskazówkami, aby wykonać wszystkie kroki receptury:

1. Otwórz plik *composer.json* i w sekcji *require* dodaj następujące linie:

```
"rcrowe/twigbridge": "0.6.*"
```

2. W linii poleceń zaktualizuj Comosera, aby zainstalować dodany pakiet:

```
php composer.phar update
```

3. Otwórz plik *app/config/app.php* i na końcu tablicy *providers* dodaj *TwigServiceProvider*:

```
'TwigBridge\ServiceProvider'
```

4. Aby utworzyć plik konfiguracyjny, w linii poleceń wykonaj:

```
php artisan config:publish rcrowe/twigbridge
```

5. W pliku *routes.php* utwórz następujący routing:

```
Route::get('twigview', function()
{
    $link = HTML::link('http://laravel.com',
        'strona Laraveła.');
```

```
    return View::make('twig')->with('link', $link);
});
```

6. W katalogu *views* utwórz plik *twiglayout.twig* i wprowadź następujący kod:

```
<!doctype html>
<html lang="pl">
    <head>
        <meta charset="utf-8">
        <title>Moja strona</title>
    </head>
    <body>
        <h1>
            {% block page_title %}
                Witamy na
            {% endblock %}
        </h1>
        {% block content %}{% endblock %}
    </body>
</html>
```

7. W katalogu *views* utwórz plik *twig.twig* i wprowadź następujący kod:

```
{% extends "twiglayout.twig" %}
{% block page_title %}
    {{ parent() }}
    Moja strona z utworzona użyciem systemu Twig
```

```
{% endblock %}
{% block content %}
  <p>
    Przejdź na adres {{ link|raw }}
  </p>
{% endblock %}
```

8. Sprawdź działanie widoków, odwiedzając adres `http://twój-serwer/twigview` (gdzie `twój-serwer` jest adresem URL serwera), i wyświetl źródło strony, aby zobaczyć, czy szablon Twig został pomyślnie załączony.

Jak to działa...

Na początek instalujemy w naszej aplikacji pakiet TwigBranch. Pakiet ten instaluje również bibliotekę Twig. Po zainstalowaniu pakietu tworzymy za pomocą Artisana jego plik konfiguracyjny i ustawiamy dostawcę usługi.

W routingu wykorzystujemy taką samą składnię jak w dostępnej w Laravelu bibliotece widoków, a następnie wywołujemy widok. Tworzymy również proste łącze, zapisujemy je do zmiennej i przekazujemy tę zmienną do widoku.

Następnie tworzymy układ. Wszystkie pliki widoków w systemie Twig posiadają rozszerzenie `.twig`, nadajemy więc układowi nazwę `twiglayout.twig`. Wewnątrz układu strony znajduje się standardowy kod HTML, dodatkowo dodajemy też jednak dwa bloki Twig. W bloku `page_title` ustawiamy domyślną zawartość, podczas gdy blok `content` jest pusty.

Widok dla routingu rozpoczynamy od rozszerzenia układu strony. Na początku bloku `page_title` wypisujemy domyślną zawartość za pomocą wyrażenia `{{ parent() }}` i dodajemy własną treść. Następnie dodajemy blok `content`, a w jego wnętrzu wyświetlamy przekazane w postaci zmiennej łącze.

Korzystając z systemu Twig, nie musimy podawać znaku `$` w nazwach zmiennych, a zmienne zawierające kod HTML są automatycznie „eskejpowane” (ang. *escaping*). Tak więc, aby wyświetlić łącze w naszym widoku, musimy pamiętać o dodaniu parametru `raw`.

Teraz, gdy wejdziemy na stronę w oknie przeglądarki, zobaczymy całą zawartość we właściwym miejscu.

Zaawansowane możliwości systemu Blade

Dzięki systemowi szablonów Blade mamy dostęp do potężnych funkcji, które zwiększają wydajność tworzenia aplikacji. W tej recepturze prześlemy dane do widoków Blade i wyświetlimy je w pętli, filtrując je dodatkowo według pewnych kryteriów.

Przygotowanie

W tej recepturze wykorzystamy kod utworzony w recepturze „Tworzenie widoków z zastosowaniem systemu szablonów Blade”.

Jak to zrobić...

Postępuj zgodnie z poniższymi wskazówkami, aby wykonać wszystkie kroki receptury:

1. Otwórz plik *routes.php* i zaktualizuj routings *blade-home* oraz *blade-second* w następujący sposób:

```
Route::get('blade-home', function()
{
    $movies = array(
        array('name' => 'Gwiezdne wojny', 'year' => '1977', 'slug'
            => 'star-wars'),
        array('name' => 'Matrix', 'year' => '1999',
            'slug' => 'matrix'),
        array('name' => 'Szkłana pułapka', 'year' => '1988', 'slug'
            => 'die-hard'),
        array('name' => 'Sprzedawcy', 'year' => '1994', 'slug'
            => 'clerks')
    );
    return View::make('blade.home')->with('movies',
        $movies);
});
Route::get('blade-second/{:any}', function($slug)
{
    $movies = array(
        'star-wars' => array('name' => 'Gwiezdne wojny', 'year'
            => '1977', 'genre' => 'Science-fiction'),
        'matrix' => array('name' => 'Matrix', 'year'
            => '1999', 'genre' => 'Science-fiction'),
        'die-hard' => array('name' => 'Szkłana pułapka', 'year'
            => '1988', 'genre' => 'Sensacyjny'),
        'clerks' => array('name' => 'Sprzedawcy', 'year'
            => '1994', 'genre' => 'Komedia')
    );
    return View::make('blade.second')->with('movie'
        , $movies[$slug]);
});
```

2. W katalogu *views/blade* zaktualizuj plik *home.blade.php* i wprowadź poniższy kod:

```
@extends('layout.index')

@section('page_title')
    @parent
```

```

    Nasza lista filmów
@endsection

@section('content')
    <ul>
        @foreach ($movies as $movie)
            <li>{{ HTML::link('blade-second/' . $movie['slug'],
                $movie['name']) }} ( {{ $movie['year'] }} )</li>
            @if ($movie['name'] == 'Szkłana pułapka')
                <ul>
                    <li>Główny charakter: John McClane</li>
                </ul>
            @endif
        @endforeach
    </ul>
@endsection

```

3. W katalogu *views/blade* zaktualizuj plik *second.blade.php* i wprowadź poniższy kod:

```

@extends('layout.index')

@section('page_title')
    @parent
    Strona filmu {{ $movie['name'] }}
@endsection

@section('content')
    @include('blade.info')
    <p>
        Przejdź na {{ HTML::link('blade-home', 'stronę główną.') }}
    </p>
@endsection

```

4. W katalogu *views/blade* utwórz plik *info.blade.php* o następującej zawartości:

```

<h1>{{ $movie['name'] }}</h1>
<p>Rok: {{ $movie['year'] }}</p>
<p>Gatunek: {{ $movie['genre'] }}</p>

```

5. Sprawdź działanie widoków — odwiedź adres <http://{twój-serwer}/blade-home> (gdzie *twój-serwer* jest adresem URL serwera), a następnie kliknij łącze.

Jak to działa...

W tej recepturze przekazujemy dane do widoków Blade i przechodzimy po nich w pętli, uwzględniając dodatkowe warunki. W rzeczywistych aplikacjach pobralibyśmy dane z bazy, ale w naszym przykładzie korzystamy z danych zapisanych w tablicy.

Pierwszy routing zawiera tablicę filmów, a przekazane dane zawierają rok i uproszczoną nazwę (ang. *slug*), których można użyć do przygotowania adresu URL. Drugi routing przyjmuje uproszczoną nazwę w adresie URL i tworzy tablicę, której kluczami są uproszczone nazwy. Następnie przekazujemy szczegóły wybranego filmu do widoku, wykorzystując do tego nazwę z adresu URL.

W pierwszym widoku tworzymy pętlę `@foreach`, aby wyświetlić w pętli dane zapisane w tablicy. Użyliśmy także prostej instrukcji `@if` do wyświetlenia dodatkowych informacji o określonym filmie. W każdym przebiegu pętli wyświetlamy łącza do drugiego routingu uzupełnione o uproszczoną nazwę.

W drugim widoku wyświetlamy nazwę filmu, ale dodatkowo w bloku `content` załączamy inny widok `Blade` za pomocą instrukcji `@include()`. Dzięki temu wszystkie dane, które są widoczne w głównym widoku, są również widoczne w widoku załączonym. W widoku *info* możemy zatem użyć wszystkich zmiennych ustawionych w routingu.

Tworzenie zlokalizowanej zawartości

Jeśli z naszej aplikacji mają korzystać osoby z innych krajów i mówiące w innych językach, musi im udostępniać zlokalizowane treści. Laravel pozwala w prosty sposób wykonać to zadanie.

Przygotowanie

W bieżącej recepturze będziemy bazować na standardowej instalacji Laravla.

Jak to zrobić...

W tej recepturze wykonaj następujące czynności:

1. W katalogu *app/lang* dodaj trzy nowe katalogi (jeśli jeszcze nie istnieją): *pl*, *en* i *de*.
2. W katalogu *pl* utwórz plik *localized.php* i wprowadź następujący kod:

```
<?php
return array(
    'greeting' => 'Dzień dobry, :name',
    'meetyou' => 'Miło Cię poznać!',
    'goodbye' => 'Do zobaczenia jutro.',
);
```

3. W katalogu *en* utwórz plik *localized.php* i dodaj następujący kod:

```
<?php
return array(
    'greeting' => 'Buenos días :name',
```

```

        'meetyou' => 'Nice to meet you!',
        'goodbye' => 'Adiós, hasta mañana.',
    );

```

4. W katalogu *de* utwórz plik *localized.php* i wprowadź następujący kod:

```

<?php
    return array(
        'greeting' => 'Guten morgen :name',
        'meetyou' => 'Es freut mich!',
        'goodbye' => 'Tag. Bis bald.',
    );

```

5. W pliku *routes.php* utwórz cztery routingi:

```

Route::get('choose', function()
{
    return View::make('language.choose');
});
Route::post('choose', function()
{
    Session::put('lang', Input::get('language'));
    return Redirect::to('localized');
});
Route::get('localized', function()
{
    $lang = Session::get('lang', function() { return 'pl';
});
App::setLocale($lang);
return View::make('language.localized');
});
Route::get('localized-german', function()
{
    App::setLocale('de');
return View::make('language.localized-german');
});

```

6. W katalogu *views* utwórz folder o nazwie *language*.

7. W katalogu *views/language* utwórz plik *choose.php* i dodaj do niego następujący kod:

```

<h2>Wybierz język:</h2>
<?= Form::open() ?>
<?= Form::select('language', array('pl' => 'polski', 'en' =>
    'angielski')) ?>
<?= Form::submit() ?>
<?= Form::close() ?>

```

8. W katalogu *views/language* utwórz plik *localized.php* o następującej zawartości:

```

<h2>
    <?= Lang::get('localized.greeting', array('name' =>
        'Katarzyna Malinowska')) ?>

```



```

</h2>
<p>
  <?= Lang::get('localized.meetyou') ?>
</p>
<p>
  <?= Lang::get('localized.goodbye') ?>
</p>
<p>
  <?= HTML::link('localized-german', 'Page 2') ?>
</p>

```

9. W katalogu `views/language` utwórz plik `localized-german.php` i wprowadź następujący kod:

```

<h2>
  <?= Lang::get('localized.greeting', array('name' =>
    'Katarzyna Malinowska')) ?>
</h2>
<p>
  <?= Lang::get('localized.meetyou') ?>
</p>
<p>
  <?= Lang::get('localized.goodbye') ?>
</p>

```

10. W oknie przeglądarki odwiedź adres `http://{twój-serwer}/choose` (gdzie `twój-serwer` jest adresem URL serwera), zatwierdź formularz i sprawdź działanie lokalizacji.

Jak to działa...

W tej recepturze zaczynamy od utworzenia katalogów dla poszczególnych języków w folderze `app/lang`. Tworzymy katalog `pl` dla plików z polskim tłumaczeniem, `en` z angielskim oraz `de` z niemieckim. Wewnątrz każdego z tych katalogów tworzymy plik o takiej samej nazwie i zawierający tablicę o identycznych kluczach.

Pierwszym routinguem jest strona wyboru języka. Możemy wybrać język polski lub angielski. Zatwierdzony wybór jest przesyłany metodą POST do routingu, tworzona jest nowa sesja z danym wyborem i następuje przekierowanie na stronę z tekstem w wybranym języku.

Zlokalizowany routing pobiera dane z sesji i przekazuje wybór do metody `App::setLocale()`. Jeśli w sesji nie jest ustawiona żadna wartość, domyślnie wybierany jest język polski.

W zlokalizowanym widoku wypisujemy tekst za pomocą metody `Lang::get()`. W pierwszej linii pliku z tłumaczeniami zamieściliśmy miejsce na nazwę `:name`, które zostanie wypełnione przez odpowiednie dane z tablicy zwróconej w pliku z tłumaczeniem.

W ostatnim routingu pokazaliśmy, jak ustawić język domyślny.

Tworzenie menu w Laravelu

Menu jest elementem wykorzystywanym w większości witryn internetowych. W tej recepturze utworzymy menu za pomocą zagnieżdżonych widoków i zmienimy domyślny „stan” jego elementów na podstawie bieżącej strony.

Przygotowanie

Do przygotowania menu użyjemy standardowej instalacji Laravela.

Jak to zrobić...

Postępuj zgodnie z poniższymi wskazówkami, aby wykonać wszystkie kroki receptury:

1. W pliku *routes.php* utwórz trzy routingi:

```
Route::get('menu-one', function()
{
    return View::make('menu-layout')
        ->nest('menu', 'menu-menu')
        ->nest('content', 'menu-one');
});
Route::get('menu-two', function()
{
    return View::make('menu-layout')
        ->nest('menu', 'menu-menu')
        ->nest('content', 'menu-two');
});
Route::get('menu-three', function()
{
    return View::make('menu-layout')
        ->nest('menu', 'menu-menu')
        ->nest('content', 'menu-three');
});
```

2. W katalogu *views* utwórz plik *menu-layout.php* i dodaj następujący kod:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Przykład menu</title>
    <style>
      #container {
        width: 1024px;
        margin: 0 auto;
```

```

        border-left: 2px solid #ddd;
        border-right: 2px solid #ddd;
        padding: 20px;
    }
    #menu { padding: 0 }
    #menu li {
        display: inline-block;
        border: 1px solid #ddf;
        border-radius: 6px;
        margin-right: 12px;
        padding: 4px 12px;
    }
    #menu li a {
        text-decoration: none;
        color: #069;
    }
    #menu li a:hover { text-decoration: underline }
    #menu li.active { background: #069 }
    #menu li.active a { color: #fff }
</style>
</head>
<body>
    <div id="container">
        <?= $menu ?>
        <?= $content ?>
    </div>
</body>
</html>

```

3. W katalogu *views* utwórz plik *menu-menu.php* i dodaj następujący kod:

```

<ul id="menu">
    <li class="<?= Request::segment(1) == 'menu-one' ?
        'active' : '' ?>">
        <?= HTML::link('menu-one', 'Pierwsza strona') ?>
    </li>
    <li class="<?= Request::segment(1) == 'menu-two' ?
        'active' : '' ?>">
        <?= HTML::link('menu-two', 'Druga strona') ?>
    </li>
    <li class="<?= Request::segment(1) == 'menu-three' ?
        'active' : '' ?>">
        <?= HTML::link('menu-three', 'Trzecia strona') ?>
    </li>
</ul>

```

4. W katalogu *views* utwórz trzy pliki widoków: *menu-one.php*, *menu-two.php* i *menu-three.php*.

5. W pliku *menu-one.php* wprowadź następujący kod:

```
<h2>Pierwsza strona</h2>
<p>
  Lorem ipsum dolor sit amet.
</p>
```

6. W pliku *menu-two.php* wprowadź następujący kod:

```
<h2>Druga strona</h2>
<p>
  Suspendisse eu porta turpis
</p>
```

7. W pliku *menu-three.php* wprowadź następujący kod:

```
<h2>Trzecia strona</h2>
<p>
  Nullam varius ultrices varius.
</p>
```

8. W oknie przeglądarki odwiedź adres *http://{twój-serwer}/menu-one* (gdzie *twój-serwer* jest adresem URL Twojego serwera), a następnie odwiedź łącza dostępne w menu.

Jak to działa...

Rozpoczynamy od utworzenia trzech routingów dla trzech stron. Wszystkie routings wykorzystują domyślny układ strony oraz zagnieżdżone widoki menu i treści, która jest unikalna dla każdego routingu.

Układ strony tworzy prosty szkielet HTML z zagnieżdżonym kodem CSS. Pragniemy podświetlić element menu odpowiadający bieżącej stronie, dodajemy więc do niego klasę `css active`.

Następnie tworzymy widok menu. Wykorzystujemy do tego listę nieuporządkowaną z linkami do każdej strony. Aby dodać klasę `active` do elementu odpowiadającego bieżącej stronie, stosujemy metodę `Laravel::request::segment(1)` do pobrania aktualnego routingu. Jeśli routing jest taki sam jak element na liście, dodajemy klasę `active`, w przeciwnym wypadku zostawiamy ją pustą. Następnie używamy metody `HTML::link()` Laravla, aby dodać łącza do stron.

Kolejne trzy widoki są tworzone przez nagłówek i prostą treść, złożoną z kilku słów. Teraz po wejściu na stronę w oknie przeglądarki zobaczymy, że element odpowiadający bieżącej stronie jest podświetlony, a pozostałe elementy nie są. Po kliknięciu kolejnych łączy podświetlony będzie zawsze tylko aktualnie wybrany element.

Integracja z Bootstrapem

Framework CSS Bootstrap stał się ostatnio bardzo popularny. W tej recepturze pokażemy, jak użyć tego frameworka w projektach opartych na Laravelu.

Przygotowanie

W tej recepturze wykorzystamy standardową instalację Laravela. Potrzebny nam też będzie pakiet Asset zainstalowany według receptury „Dodawanie zasobów”. Możemy też pobrać pliki Bootstrapa i zapisać je lokalnie.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. W pliku *routes.php* utwórz nowy routing:

```
Route::any('boot', function()
{
    Asset::add('jquery', 'http://ajax.googleapis.com/ajax
        /libs/jquery/1.10.2/jquery.min.js');
    Asset::add('bootstrap-js', 'http://
        netdna.bootstrapcdn.com/twitter-
        bootstrap/2.3.2/js/bootstrap.min.js', 'jquery');
    Asset::add('bootstrap-css', 'http://
        netdna.bootstrapcdn.com/twitter-
        bootstrap/2.3.2/css/bootstrap-combined.min.css');
    $superheroes = array('Batman', 'Superman', 'Wolverine',
        'Deadpool', 'Iron Man');
    return View::make('boot')->with('superheroes',
        $superheroes);
});
```

2. W katalogu *views* utwórz plik *boot.php* o następującej zawartości:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Strona wykorzystująca Bootstrap</title>
    <? = Asset::styles() ?>
  </head>
  <body>
    <div class="container">
      <h1>Wykorzystywanie Bootstrapa z Laravelem</h1>
      <ul class="nav nav-tabs">
        <li class="active"><a href="#welcome" data-
```

```

        toggle="tab">Zapraszamy</a></li>
<li><a href="#about" data-toggle="tab">
    0 nas</a></li>
<li><a href="#contact" data-toggle="tab">
    Kontakt</a></li>
</ul>
<div class="tab-content">
    <div class="tab-pane active" id="welcome">
        <h4>Witamy na naszej stronie</h4>
        <p>Lista superbohaterów:</p>
        <ul>
            <?php foreach($superheroes as $hero): ?>
                <li class="badge badge-info">
                    <?= $hero ?></li>
            <?php endforeach; ?>
        </ul>
    </div>
    <div class="tab-pane" id="about">
        <h4>0 nas</h4>
        <p>Cras at dui eros. Ut imperdiet
            pellentesque mi faucibus dapibus.
            Phasellus vitae lacus at massa viverra
            condimentum quis quis augue. Etiam
            pharetra erat id sem pretium egestas.
            Suspendisse mollis, dolor a sagittis
            hendrerit, urna velit commodo dui, id
            adipiscing magna magna ac ligula. Nunc
            in ligula nunc.</p>
    </div>
    <div class="tab-pane" id="contact">
        <h3>Formularz kontaktowy</h3>
        <?= Form::open('boot', 'POST') ?>
        <?= Form::label('name', 'Twoje imię') ?>
        <?= Form::text('name') ?>
        <?= Form::label('email', 'Twój adres e-mail') ?>
        <?= Form::text('email') ?>
        <br>
        <?= Form::button('Wyślij', array('class' =>
            'btn btn-primary')) ?>
        <?= Form::close() ?>
    </div>
</div>
<?= Asset::scripts() ?>
</body>
</html>

```

3. W oknie przeglądarki odwiedź adres *http://twój-serwer/boot* (gdzie *twój-serwer* jest adresem URL serwera), a następnie odwiedź dostępne karty.

Jak to działa...

W bieżącej recepturze tworzymy pojedynczy routing i zmieniamy zawartość za pomocą kart Bootstrapa. Aby routing odpowiadał na wszystkie żądania, używamy metody `Route::any()` i przekazujemy do niej domknięcie. Pliki CSS i JavaScript mogliśmy dodać w sposób opisany w recepturze „Dodawanie zasobów”; ale dla pojedynczego routingu załączamy je po prostu w domknięciu. Nie musimy więc pobierać tych plików — wykorzystujemy wersje Bootstrapa i jQuery dostępne na serwerze CDN.

Następnie dodajemy w routingu jakieś dane. Moglibyśmy w tym miejscu podłączyć bazę danych, ale na potrzeby naszego przykładu wystarczająca będzie zwykła tablica zawierająca nazwy superbohaterów. Następnie przekazujemy tę tablicę do widoku.

Widok rozpoczyna kod HTML, w którym załączamy style w nagłówku, a skrypty przed końcowym znacznikiem `</body>`. W górnej części strony wykorzystujemy style nawigacji Bootstrapa oraz atrybuty danych do tworzenia łączy kart. Następnie w ciele strony używamy trzech paneli kart z identyfikatorami odpowiadającymi znacznikowi `<a href>` w menu.

Gdy wchodzimy na stronę, pierwszy panel jest widoczny, a wszystkie pozostałe panele są ukryte. Kliknięcie innej karty powoduje przełączenie widocznego panelu.

Zobacz także

- Receptura „Dodawanie zasobów”.

Nazwane widoki i kompozytory widoków

W tej recepturze zobaczymy, jak uprościć kod routingu za pomocą nazwanych widoków i kompozytorów widoku.

Przygotowanie

W tej recepturze wykorzystamy kod utworzony w recepturze „Tworzenie menu w Laravelu”.

Potrzebny nam też będzie pakiet Asset zainstalowany według receptury „Dodawanie zasobów”.

Jak to zrobić...

Aby wykonać kroki tej receptury, postępuj zgodnie z poniższymi wskazówkami:

1. W pliku *routes.php* file dodaj następującą definicję widoku:

```
View::name('menu-layout', 'layout');
```

2. W pliku *routes.php* dodaj kompozytora widoku:

```
View::composer('menu-layout', function($view)
{
    Asset::add('bootstrap-css',
        'http://netdna.bootstrapcdn.com/twitter-
        bootstrap/2.2.2/css/bootstrap-combined.min.css');
    $view->nest('menu', 'menu-menu');
    $view->with('page_title', 'Tytuł kompozytora widoku');
});
```

3. W pliku *routes.php* zaktualizuj routingi dla menu:

```
Route::get('menu-one', function()
{
    return View::of('layout')->nest('content', 'menu-one');
});
Route::get('menu-two', function()
{
    return View::of('layout')->nest('content', 'menu-two');
});
Route::get('menu-three', function()
{
    return View::of('layout')->nest('content', 'menu-three');
});
```

4. W katalogu *views* zaktualizuj plik *menu-layout.php*:

```
<!doctype html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title><?= $page_title ?></title>
    <?= Asset::styles() ?>
    <style>
      #container {
        width: 1024px;
        margin: 0 auto;
        border-left: 2px solid #ddd;
        border-right: 2px solid #ddd;
        padding: 20px;
      }
      #menu { padding: 0 }
      #menu li {
        display: inline-block;
        border: 1px solid #ddf;
        border-radius: 6px;
        margin-right: 12px;
```



```

        padding: 4px 12px;
    }
    #menu li a {
        text-decoration: none;
        color: #069;
    }
    #menu li a:hover { text-decoration: underline }
    #menu li.active { background: #069 }
    #menu li.active a { color: #fff }
</style>
</head>
<body>
    <div id="container">
        <?= $menu ?>
        <?= $content ?>
    </div>
</body>
</html>

```

5. W oknie przeglądarki odwiedź adres `http://{twój-serwer}/menu-one` (gdzie `twój-serwer` jest adresem URL Twojego serwera), a następnie odwiedź łącza dostępne w menu.

Jak to działa...

Rozpoczynamy od utworzenia nazwy dla jednego z widoków. Pozwala nam to skrócić nazwy widoków w sytuacjach, gdy są one długie lub skomplikowane albo gdy widoki znajdują się w złożonej strukturze katalogów. Równocześnie umożliwia to późniejszą szybką zmianę nazwy pliku widoku. Nawet gdy widok jest wykorzystywany w wielu miejscach, zmiana dotyczy tylko jednej linii.

Następnie tworzymy kompozytor widoku. Kod znajdujący się w kompozytorze jest automatycznie wywoływany po utworzeniu widoków. Za każdym razem, gdy tworzymy widok, załączamy trzy zasoby: plik CSS frameworka Bootstrap, zagnieżdżony widok oraz zmienną przeznaczoną do widoku.

W naszych trzech routinach zamiast metody `View::make('menu-layout')` stosujemy utworzoną nazwę, wywołujemy metodę `View::of('layout')` i załączamy ją w treści strony. Ponieważ nasz układ korzysta z kompozytora, menu, plik CSS i tytuł strony zostaną automatycznie załączone.

Zobacz także

- Receptura „Tworzenie menu w Laravelu”.

Skorowidz

A

adres

- e-mail, 248
- IP, 248
- localhost, 74, 76
- URL, 17, 109

Ajax, 177, 185, 191

aktualizowanie

- Composer, 161, 173, 229
- wpisu, 121
- danych o użytkowniku, 64

API, 118, 168, 221

aplikacje

- debugowanie, 233
- profilowanie, 233
- testowanie, 225

atak typu

- CSRF, 41, 208
- XSS, 202

autoloader, 24, 107

autoloader zaawansowany, 25

automatyczna walidacja, 90

automatyczne eskejpowanie, 142

autouzupełnianie, 50

autouzupełnianie przestrzeni nazw, 21

B

baza danych

- MySQL, 118, 205
- GeoIP, 247
- Redis, 217
- typu klucz-wartość, 216

bezpieczeństwo, 201

biblioteka

- Auth, 58
- Codeception, 231
- GD, 53
- HybridAuth, 70
- Jcrop, 47
- jQuery, 46, 137, 177, 194
- jQueryUI, 137
- MailChimp, 251
- Mockery, 228, 230
- PHPUnit, 226
- Redactor, 44
- Stripe, 246
- Twig, 142

błędy walidacji, 38

bramka płatności Stripe, 244

C

CAPTCHA, 53

chmura, 240

chmura Amazon S3, 251

ciasteczko, 218

ciąg dopasowania, 223

CRUD, create, read, update, delete, 95

CSRF, Cross-Site Request Forgery, 41, 208

czas odwiedzin routing, 116

D

dane

- autouzupełniania, 51
- o użytkowniku, 64
- testowe, 119, 123, 163
- wejściowe, 29
- z kanału RSS, 101

debugowanie aplikacji, 225, 233, 236
 dodatek DataTables, 199
 dodawanie
 pakietów do Packagista, 169
 pakietów do Composeera, 171
 plików CSS, 137
 zasobów, 136
 dokumentacja, 24, 87, 95
 domknięcie, 111
 dostawca usług, 161
 dostawca usługi AWS, 252
 dostęp do
 bazy, 205
 danych, 181
 dystrybucja pakietów, 169

E

e-commerce, 213, 244
 edytor
 Sublime Text 2, 19
 WYSIWYG, 44
 edytowanie
 danych, 65
 programów, 121
 element
 div, 182
 textarea, 46

F

Facebook, 72
 fasada AWSFacade, 254
 filtr, 224
 assets, 137
 auth, 68
 auth_admin, 68
 before, 116
 csrf, 41, 63, 209, 210
 FILTER_VALIDATE_EMAIL, 35
 filtrowanie danych, 188
 filtry w routingu, 115
 flaga --resources, 168
 format JSON, 120, 181
 formularz
 automatyczne podpowiedzi, 50
 biblioteka Redactor, 44
 edycja danych, 65
 e-mail, 196

filtr, 208
 logowanie, 62, 71, 207
 obraz CAPTCHA, 53
 pobieranie danych, 31
 przesyłanie obrazu, 47
 przycinanie obrazu, 48
 rejestracja, 60, 193, 207
 token CSRF, 208
 tworzenie, 30
 wabik, 42
 walidacja danych, 33
 zaawansowana walidacja, 210

framework
 CSS Bootstrap, 151
 Laravel, 9

funkcja

dd(), 87, 169
 fgetcsv(), 101
 fopen(), 101
 get_headers(), 102
 imagecreatefrompng(), 50
 imagepng(), 50
 orWhere(), 87
 sadd(), 218
 simplexml_load_string(), 102
 updateCoords, 50
 var_dump(), 35
 where(), 87

funkcje

anonimowe, 111
 systemu Eloquent, 93
 wyszukiwania, 183

G

grupa reguł routingu, 116
 grupa routingu, 128

H

hashowanie haseł, 205, 207
 hasło, 205
 HTML5, 173

I

IDE, Integrated Development Environment, 21
 identyfikator, 64
 aplikacji, 73

- book-list, 182
- list, 251
- results, 185, 187
- Secret ID, 252
- importowanie plików CSV, 99
- informacje
 - o karcie kredytowej, 246
 - o logowaniu, 73
 - o odwiedzinach routingu, 116
 - o pakiecie, 159
 - o pliku, 36
 - o użytkownika, 60, 64, 66, 94
 - z modelu, 89
- instalowanie
 - biblioteki Auth, 58
 - biblioteki PHPUnit, 226
 - Laravela, 13, 14
 - pakietów, 158
 - pakietu Mockery, 230
 - Sublime, 19
- instrukcja
 - @if, 145
 - @include(), 145
- integracja z Bootstrpem, 151

J

- język
 - JavaScript, 177
 - PHP, 251
 - SQL, 83

K

- kanal RSS, 101
- katalog
 - acceptance, 232
 - app, 18, 25
 - blade, 139, 143
 - common, 134
 - controllers, 181
 - lang, 145
 - language, 146
 - libraries, 55, 105, 249
 - logs, 116, 240
 - migrations, 59, 80
 - models, 84, 190, 222
 - myviews, 130
 - public, 17

- search, 184
- storage, 18, 19
- test, 168
- tests, 227, 232
- vendor, 226
- views, 131
- vimeolist, 169
- workbench, 168
- klasa
 - Auth, 57
 - BaseController, 182
 - Captcha, 53, 55
 - Eloquent, 92, 101
 - Filesystem, 176
 - Form, 30
 - Input, 32
 - MailChimp, 251
 - MyAppTest, 228
 - Spaceship, 229, 240
 - SpaceshipTest, 230
 - TestCase, 228
 - Validator, 33, 37, 92
- klonowanie repozytorium Git, 243
- klucz
 - Access Key ID, 252
 - API, 191, 221, 246
 - API Key, 76
 - aplikacji, 19, 204
 - Consumer Key, 75
 - Consumer Secret, 75
 - Secret Key, 76
 - SSH, 240
- kod CVV karty, 246
- kolejka, 238
- kompozytory widoków, 153
- komunikat o błędzie, 38–41, 197, 212
- komunikat o błędzie HTML 401, 224
- konfigurowanie
 - bazy danych, 107
 - biblioteki Auth, 58
 - biblioteki PHPUnit, 226
 - DNS, 126
 - hostów wirtualnych, 15
 - klienta e-mail, 194
 - kontrolera, 181
 - Laravela, 18
 - środowiska deweloperskiego, 15
 - środowiska IDE, 21
 - uwierzytelniania OAuth, 69
 - własnego routingu, 247

konstruktor
 Fluent, 85
 zapytań Fluent, 87
 kontener div, 180
 konto Amazon AWS, 252
 kontroler, 181
 books, 181
 REST-owy, 112, 185
 User, 111, 112
 zasobów, resourceful controller, 124
 koszyk zakupowy, 213
 kubelek, bucket, 252

L

Laravel, 9
 liczniki czasu, 236
 LinkedIn, 76
 lista
 kubelków, 253
 mailingowa, 248
 pakietów, 161
 logowanie, 62, 70, 71
 z poświadczeniami Facebooka, 72
 z poświadczeniami LinkedIn, 76
 z poświadczeniami Twittera, 74
 lokalizacja, 19

Ł

łączenie tabel, 95

M

maska, 202
 mechanizm
 ClassLoader, 24
 hashowania haseł, 207
 IoC, 27
 OpenID, 70, 72
 przechwytywania spamu, 53
 przesyłania plików, 35
 menu, 148
 metoda
 add(), 138
 ajax(), 180
 all(), 90
 allShows(), 85, 87
 attach(), 95

attempt(), 64, 208
 autocomplete, 52
 call, 228
 check(), 64, 208
 connection(), 218
 controller(), 113
 dataTable, 199
 deleteRecord(), 98
 down, 87
 encrypt, 205
 file(), 254
 find(), 67
 fire(), 176, 240
 first, 230
 flashOnly(), 32
 forget(), 220
 get(), 95, 147
 getCreate(), 98
 getIndex(), 113
 getRecord(), 98
 getUsernameAttribute(), 105
 guessExtension(), 36
 input(), 31
 insert(), 85
 intended(), 64
 json(), 46, 182
 link(), 140
 listBuckets(), 254
 listSubscribe(), 251
 loginUsingId(), 64
 make(), 55, 63, 131, 207
 move(), 36
 nest(), 135
 of(), 155
 old(), 32, 67
 open(), 36, 38, 41
 postBooks(), 190
 postIndex(), 113
 postRegister(), 187
 postSearch(), 185
 push(), 239
 putObject(), 254
 putRecord(), 98
 queue(), 240
 random(), 55
 reflash(), 220
 route(), 125
 scripts(), 138
 segment(), 150

- select, 31
- send, 197
- setLocale(), 147
- setLayout(), 111
- shows(), 95
- token(), 210
- user(), 67
- users(), 95
- validate, 92
- with(), 133
- metody
 - HTTP, 123
 - konstruktora zapytań, 87
 - statyczne, 169
 - typu getter, 104
- migracja, 80, 104, 127
- model
 - Eloquent, 199
 - Show, 87
 - Spaceship, 230
 - User, 60, 92
- moduł
 - mod_rewrite, 17
 - zależny, 14, 15
- MVC, Model-View-Controller, 110

N

- nadawanie uprawnień administratora, 63
- nagłówek, 135
- narzędzie
 - Artisan, 59, 69, 80, 123
 - Composer, 14, 24
 - curl, 123
 - Workbench, 165
- nazwa
 - kubelka, 254
 - routingu, 124
 - subdomeny, 125
- newsletter, 191, 248

O

- obsługa
 - adresów URL, 109
 - danych, 34
 - formularza rejestracji, 60
 - koszyka zakupów, 213
 - obrazów, 44

- plików konfiguracyjnych, 16
- przesyłania obrazów, 45
- Redisa, 216
- relacji, 95
- tabeli, 162
- transakcji, 244
- wysyłki formularza, 34
- odszyfrowywanie danych, 202
- okno rejestracji, 191
- opcje routingu, 113
- oprogramowanie LAMP, 10

P

- Packagist, 169
- pakiet, 158
 - Amazon SDK, 252
 - Ardent, 92
 - Asset, 136
 - Composer, 157, 165, 169
 - DataTables, 199
 - Generators, 161, 165
 - HybridAuth, 69
 - image, 161
 - IronMQ, 239
 - Laravel 4 Snippets, 21
 - Laravel-Blade, 21
 - MailChimp, 193
 - Mockery, 228
 - Package Control, 19
 - TwigBranch, 142
 - Universal Forms, 171
- parametr
 - blade, 176
 - raw, 142
- pętla @foreach, 145
- platforma Pagoda Box, 240, 243
- plik
 - .htaccess, 17
 - accounts.php, 204
 - add_users_data, 127
 - ajaxemail.php, 195
 - Api.php, 222
 - api-key.php, 222
 - app.php, 18, 252
 - artisan.php, 176
 - auth.php, 58
 - autocomplete.php, 51
 - AviatorCept.php, 231

plik

- aws.php, 252
 - BaseController.php, 111
 - Book.php, 190
 - Bookprices.php, 198
 - BooksController.php, 181, 189
 - boot.php, 151
 - Captcha.php, 53
 - cart.php, 215
 - choose.php, 146
 - cloud.blade.php, 253
 - composer.json, 26, 53, 69, 105, 136, 159–161, 171, 191, 226, 238, 244, 247–252
 - create.php, 97
 - create_users_table, 126
 - cross-site.php, 209
 - database.php, 18, 58
 - DatabaseSeeder.php, 234
 - emailform.php, 194
 - fb_auth.php, 73
 - fileform.php, 35
 - filters.php, 68, 117, 136, 209, 224, 247
 - footer.php, 134, 137
 - geoip, 247
 - getting-data.php, 179
 - global.php, 25
 - header.php, 134, 137
 - home.blade.php, 139, 143
 - home.php, 130, 132
 - httpd.conf, 16, 126
 - httpd-vhosts.conf, 16
 - imageform.php, 47
 - index.blade.php, 139
 - index.php, 15, 186, 189
 - info.blade.php, 144
 - item-detail.php, 215
 - items.php, 214
 - jpeg.php, 48
 - li_auth.php, 76
 - localized.php, 145
 - login.php, 62, 71
 - mailchimp.php, 191, 249
 - menu-layout.php, 148, 154
 - menu-menu.php, 149
 - myapp.php, 42
 - MyAppTest.php, 227
 - myform.php, 39
 - MyShapes.php, 24, 26
 - oauth.php, 69
 - Odd.php, 104
 - openid_auth.php, 70
 - pay.blade.php, 245
 - profile.php, 65
 - profile_edit.php, 65
 - queue.php, 238
 - record.php, 98
 - redactor.php, 45
 - redis-login.php, 217
 - register.php, 206
 - registration.php, 60
 - routes.php, 30, 35, 37
 - scifi.csv, 99
 - Scifi.php, 100
 - SearchController.php, 183
 - second.blade.php, 139, 144
 - second.php, 130, 132
 - session.php, 58
 - session-one.php, 218
 - ships.blade.php, 235
 - ShipsController.php, 229, 235
 - Show.php, 84, 86, 89
 - show-delete.php, 122
 - signup.php, 191
 - SkeletonCommand.php, 174
 - Spaceship.php, 235, 239
 - SpaceshipSeeder.php, 234
 - SpaceshipTest.php, 229
 - stripe.php, 245
 - subscribe.blade.php, 250
 - SuperheroesTableSeeder.php, 163
 - table.php, 198
 - tw_auth.php, 75
 - twig.twig, 141
 - twiglayout.twig, 141
 - User.php, 91
 - userform.php, 30, 33
 - userinfo.php, 134
 - UsersController.php, 96, 110, 185
 - valid.php, 211
 - validation.php, 40
 - validator.php, 212
 - Vimeolist.php, 166
 - VimeolistServiceProvider.php, 167
 - workbench.php, 165
- pliki
- .jpg, 39
 - .png, 50
 - .twig, 142

- autoloadingu, 251
- Bootstrapa, 151
- CSS, 137, 153
- CSV, 99
- frameworka, 15
- JavaScript, 153
- migracji, 59, 80, 233
- z szablonami, 21
- pobieranie
 - danych, 178
 - danych o użytkowniku, 64
 - danych wejściowych, 29
 - listy, 84, 86
 - pakietów, 158
- podłączanie frameworka, 15
- pole
 - tekstowe, 50
 - typu checkbox, 63
 - ukryte, 98
 - wyboru, 188
- połączenia SQL, 188, 197, 202
- połączenie
 - artisan, 233, 236
 - bootstrap, 232
 - composer install, 15
 - dump-autoload, 26, 169, 251
 - git clone, 15, 243
 - phpunit, 226, 227
- poświadczenia
 - bezpieczeństwa, 252
 - Facebooka, 72
 - LinkedIn, 76
 - serwisu Amazon, 254
 - Twittera, 74
- prefiks profile, 118
- proces migracji, 123, 127
- profiler, 236
- profilowanie aplikacji, 233
- przechowywanie
 - danych, 79
 - danych sesji, 202
 - kluczy API, 193, 221
 - listy, 82
 - sesji, 60, 216
- przechwytywanie spamu, 53
- przekazywanie danych do widoku, 131
- przekierowanie, 32, 55, 64, 72, 216
- przestrzeń nazw, 25

- przesyłanie
 - kluczy, 224
 - obrazów, 44, 47
 - pakietów, 170
 - plików, 35
- przeszukiwanie bazy GeoIP, 247
- przycinanie obrazu, 47, 48

R

- Redis, 216
- reguła walidacji, 212
- reguły routingu, 116
- rejestrowanie
 - klucza api, 222
 - kontrolera, 186
 - routingu, 113
- relacja wiele do wielu, 95
- relacje, 95
- repozytorium Git, 14, 170, 243
- routing, 32, 45
 - admin-only, 116
 - blade-home, 143
 - blade-second, 143
 - cloud, 254
 - dla administratorów, 68
 - domyślny, 232
 - e-mail-form, 197
 - email-send, 196
 - fbauth, 74
 - liauth, 78
 - link, 125
 - myapp, 228
 - named, 125
 - nazwany, 124
 - pay, 246
 - redirect, 125
 - REST-owy, 112
 - ship, 231
 - show-delete, 123
 - show-form, 123
 - shows, 85, 87, 89
 - signup-submit, 193
 - z domknięciem, 111
- rusztowanie, 162

S

schemat, 80
 schemat shows, 82
 serializacja danych, 193
 serwer
 Apache, 15
 Apache 2, 10
 API, 221
 CDN, 137, 180
 deweloperski, 15
 HTTP, 15
 MAMP, 10
 MySQL 5.6, 10
 Redis, 216
 WAMP, 10, 16
 XAMMP, 10
 serwis
 Amazon S3, 251
 GitHub, 169
 MailChimp, 191, 249
 Pagoda Box, 240
 sesja, 201, 218
 skrypt
 autocomplete, 52
 composer, 54
 sortowanie, 197
 spam, 53
 sprawdzanie adresu IP, 248
 standard
 PSR-0, 25
 W3C, 30
 sterownik bazy danych
 mysql, 19
 pgsql, 19
 sqlite, 19
 sqlsrv, 19
 sterownik native, 60
 stopka, 135
 strona profilu użytkownika, 62
 subdomena, 125, 128
 symbole wieloznaczne, 114, 126
 system
 CRUD, 95
 Eloquent, 99
 IronMQ, 238
 ORM Eloquent, 60, 88, 93, 105, 205, 230
 RedBeanPHP, 107
 szablonów Blade, 21, 138, 142
 szablonów Twig, 140

szablon
 Blade, 140, 142
 Twig, 140, 142
 szablony kodu, 21
 szkielet
 aplikacji, 161
 kodu HTML5, 173
 szyfrowanie danych, 202

Ś

środowisko uruchomieniowe, 15

T

tabela, 80
 accounts, 205
 books, 190
 cities, 162
 items, 213
 migracji, 80, 90
 shows, 89, 123
 superheroes, 107, 164
 users, 92
 tablica
 \$cities, 162, 165
 \$rules, 39
 aliases, 236
 aliasów, 136
 attributes, 41
 cart, 216
 providers, 141, 161
 technika
 Ajax, 185, 191
 scaffoldingu, 162
 test
 akceptacyjny, 231
 jednostkowy, 227
 testowanie
 aplikacji, 225
 kontrolerów, 228
 kubeków, 253
 token CSRF, 208, 210
 Twitter, 74
 tworzenie
 adresów URL, 17
 filtra, 118, 137, 224
 formularza, 30
 funkcji wyszukiwania, 183

- grup reguł, 117
- grupy routingu, 128
- kolejek, 238
- komunikatu o błędzie, 39
- kontrolera REST-owego, 112
- kontrolerów, 110
- koszyka zakupowego, 213
- mechanizmu przesyłania plików, 35
- menu, 148
- obrazu CAPTCHA, 53
- okna rejestracji, 191
- pakietu Composer, 165
- pliku migracji, 82, 233
- poła tekstowego, 50
- reguł, 44
- reguły walidacji, 212
- repozytorium Git, 170
- REST-owego API, 118
- routingu, 21, 25, 32, 111
- schematu dla tabeli, 118
- serwera API, 221
- systemu CRUD, 95
- systemu uwierzytelniania, 57, 60
- szkieletu aplikacji, 161
- tabeli, 80, 197
- tabeli łączącej, 95
- tabeli migracji, 82
- testów, 168, 227
- unikalnego klucza, 19
- użytkownika, 185
- walidatora, 35, 43
- widoku, 130, 138
- własnego polecenia, 173
- zaawansowanych autoloaderów, 25
- zapytań, 83, 85, 88
- zlokalizowanej zawartości, 145
- znaczników czasu, 82
- typ MIME, 38

U

- uruchamianie testów, 227
 - Codeception, 232
 - PHPUnit, 228
- usługa
 - hostingu, 240, 244
 - MailChimp, 248
 - Stripe, 246

- usługi firm trzecich, 237, 248
- ustawienia konfiguracyjne, 19
- usuwanie wpisu, 121
- uwierzytelnianie, 58, 60, 71
 - OAuth, 69
 - procesu logowania, 63

W

- wabik, honey pot, 42
- walidacja, 40, 43
 - automatyczna, 90
 - danych, 33
 - długości tekstu, 35
 - przesyłanych plików, 37
 - użytkownika, 185
 - zaawansowana, 210
- wczytywanie widoku, 30, 133
- wiadomości e-mail, 194
- widok, 129 *Patrz także* plik
 - autocomplete.php, 51
 - captcha.php, 54
 - fileform.php, 35
 - home, 135
 - info, 145
 - myapp.php, 42
 - myform.php, 39
 - profile.php, 65
 - redactor.php, 45
 - ships.blade.php, 235
 - userform.php, 30, 33
 - userinfo, 135
 - zagnieżdżony, 133, 138
- widoki nazwane, 153
- włączanie uwierzytelniania, 71
- wyjątek TokenMismatchException, 210
- wykorzystywanie danych, 79
- wysyłanie wiadomości e-mail, 194
- wyszukiwanie, 183
- wyświetlanie
 - błędów, 34
 - danych, 32, 168
 - formularza, 39
 - informacji, 87, 89, 203
 - listy, 122
 - łącza, 124
 - obrazu, 49
 - widoków, 129

wywołanie
kaskadowe, 133
routingu, 43
zwrotne, 74, 76
wzorzec MVC, 110

Z

zabezpieczanie informacji, 204
zapisywanie informacji, 203
zapytania, 83, 85, 88
zastosowanie sesji, 218
zlokalizowane treści, 145
zmiana nazw kolumn, 102
zmienna
\$app, 169
\$error, 38
\$table, 101
sesyjna typu flash, 220

znacznik
<a href>, 153
<form>, 31
, 46
znak \$, 142
rzut danych autoloadera, 106

Ż

żądania typu AJAX, 52
żądanie
GET, 113, 228
POST, 31, 224

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

Laravel

Tworzenie aplikacji

Receptury

Laravel to szkielet aplikacji dla języka PHP. W ostatnim czasie podbija on serca programistów, głównie dzięki przejrzystej dokumentacji, a także dzięki temu, że tworzenie aplikacji za jego pomocą jest wyjątkowo łatwe i przyjemne. Błyskawiczna konfiguracja i tak samo szybkie uruchomienie środowiska oraz przyjazny system szablonów to tylko niektóre z zalet tego szkieletu.

Książka, którą trzymasz w rękach, zawiera omówienie ponad 90 zagadnień dotyczących Laravela. Sięgnij po nią i przekonaj się, jak błyskawicznie rozpocząć pracę, uwierzytelnić użytkowników, przetestować aplikację, a następnie wdrożyć ją w środowisku produkcyjnym. W kolejnych rozdziałach znajdziesz kompletny kod służący do pobierania danych wejściowych, przeszukiwania baz danych, tworzenia REST-owego API; jest tu także system szablonów Blade. Ponadto dowiesz się, jak sprytnie używać Composera, stworzyć Autoloader oraz zapewnić sprawny routing. Książka ta jest genialną lekturą dla wszystkich programistów korzystających ze szkieletu Laravel w codziennej pracy!

Sprawdź, jak przyjemne może być tworzenie aplikacji!

Dzięki tej książce:

- błyskawicznie zainstalujesz szkielet Laravel
- stworzysz REST-owe API
- odbierzesz dane z przeglądarki
- przygotujesz testy automatyczne Twojej aplikacji
- docenisz możliwości, elastyczność oraz łatwość zastosowania szkieletu Laravel

Terry Matula — twórca aplikacji sieciowych. Swoją przygodę z programowaniem rozpoczął od języka BASIC na komputerze Commodore VIC-20. Pracował jako programista w językach Flash/ActionScript, ASP.NET oraz PHP. Prowadzi bloga poświęconego programowaniu aplikacji internetowych. Spośród wielu frameworków dla platformy PHP wybrał Laravela.

[PACKT] open source 
PUBLISHING community experience distilled

Helion 

31846 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

● <http://helion.pl/promocje>

Książki najchętniej czytane:

● <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

● <http://helion.pl/nowosci>

Helion SA
ul. Koszłuski 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-0302-7



9 788328 303027

Informatyka w najlepszym wydaniu

cena: 47,00 zł