Technologia i rozwiązania

Laravel
Tworzenie aplikacji
Receptury

Najlepsze przepisy na aplikację w szkielecie Laravel!

Terry Matula

Helion

[PACKT] PUBLISHING open source community experience distilled

Tytuł oryginału: Laravel Application Development Cookbook

Tłumaczenie: Mirosław Gołda

ISBN: 978-83-283-0302-7

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

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.

Spis treści

ROZDZIAŁ 6

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 routingi 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.*",
   "teepluss/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`:

```
'Teepluss\Asset\AssetServiceProvider',
```

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

```
'Asset' => 'Teepluss\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 routingów. 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 „Filtry 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 załączony.

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 Laravela `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 Laravela.

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 Composera, 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 Laravela.');
   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 przekażemy 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 routingi *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' => 'Szklana 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' => 'Szklana 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'] == 'Szklana 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, musimy im udostępniać zlokalizowane treści. Laravel pozwala w prosty sposób wykonać to zadanie.

Przygotowanie

W bieżącej recepturze będziemy bazować na standardowej instalacji Laravela.

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 routingiem jest strona wyboru języka. Możemy wybrać język polski lub angielski. Zatwierdzony wybór jest przesyłany metodą `POST` do routingu, tworzona jest nowa sesja z dodanym 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 routingi 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 łączami do każdej strony. Aby dodać klasę `active` do elementu odpowiadającego bieżącej stronie, stosujemy metodę Laravela `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()` Laravela, 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">
                O 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>O 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>
        </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 moglibyśmy 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 przekazywaną do widoku.

W naszych trzech routingach 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

E

F

G

H

I

J

K

L

Ł

M

R

S

Ś

T

U

W

Z

Ż

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 rozpoczynał 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. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-0302-7

cena: 47,00 zł