Paweł Kamiński

Laravel

Wstęp do programowania aplikacji internetowych

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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/larwpa
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5130-1

Printed in Poland.

Spis treści

Rozdział 5.

Formularze

Wyjątkowo tytuł rozpoczynającego się rozdziału nie jest nazwą technologii czy przykładem terminu zarezerwowanego tylko i wyłącznie dla branży informatycznej. Wręcz przeciwnie, formularze są elementem otaczającym człowieka z każdej strony, towarzyszą zwykłej wizycie u lekarza, uzupełnianiu dziennika lekcyjnego, podpisywaniu listy obecności w pracy, o formularzach możemy nawet mówić w momencie tworzenia trywialnej listy zakupów — można więc powiedzieć, że są codziennością człowieka.

Wszystkie przypadki tworzenia tego typu struktur są przykładem uzupełniania, podawania porcji danych, które są następnie obsługiwane w różny sposób — lista zakupów jest odczytywana w supermarkecie, lista obecności w pracy na pewno przyda się kierownikowi, natomiast dziennik lekcyjny jest niezbędny do podsumowania wyników ucznia. Każde podawanie danych, uzupełnianie formularzy ma konkretny cel i znaczenie.

Nie inaczej jest w przypadku formularzy na stronach internetowych — służą one do pobierania danych bezpośrednio od użytkownika, są elementem interfejsu graficznego, umożliwiają interakcję pomiędzy użytkownikiem a projektowanym systemem.

Mimo że rozpoczynający się rozdział nie jest jednym z najdłuższych, to samo jego istnienie świadczy o randze formularzy — elementy tego typu są powszechnie używane we współczesnych aplikacjach internetowych, a ich obsługa (przygotowanie, zapisanie pobranych danych, walidacja) jest zagadnieniem ważnym i wartym omówienia.

W poszczególnych podrozdziałach spróbujemy stworzyć i obsłużyć kilka formularzy różnego typu — od formularzy pobierających dane w celu stworzenia nowych rekordów po formularze edycji. Przedstawimy również sposoby walidowania pobranych danych.

5.1. Dodawanie danych za pomocą formularzy

Jedną z elementarnych funkcjonalności formularzy jest możliwość obsługi danych, które przeznaczone są do bezpośredniego zapisania w bazie. W naszym systemie przykładem takich danych mogą być obiekty książek i autorów. Są to proste obiekty, niewymagające skomplikowanej logiki, i świetnie nadadzą się jako pierwszy przykład formularza.

Prace rozpoczynamy od stworzenia nowego widoku, oczywiście umieścimy w nim kod HTML formularza, a jako nazwy użyjemy standardowej etykiety operacji, którą implementujemy. Cała ścieżka do pliku widoku będzie więc prezentowała się następująco: */resources/views/books/create.blade.php*, a zawartość będzie zgodna z poniższym listingiem:

```
@extends('template')
@section('title')
         Lista książek
@endsection

@section('content')
<div class="container">

    <h2>Dodawanie książki</h2>
    <form action="{{ action('BookController@store')}}" method="POST" role="form">
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
      <div class="form-group">
        <label for="name">Tytuł książki</label>
        <input type="text" class="form-control" name="name" />
      </div>
      <div class="form-group">
        <label for="name">Rok publikacji</label>
        <input type="text" class="form-control" name="year" />
      </div>
      <div class="form-group">
        <label for="name">Miejsce wydania</label>
        <input type="text" class="form-control" name="publication_place" />
      </div>
      <div class="form-group">
        <label for="name">Liczba stron</label>
        <input type="text" class="form-control" name="pages" />
      </div>
      <div class="form-group">
        <label for="name">Cena</label>
        <input type="text" class="form-control" name="price" />
      </div>

      <input type="submit" value="Dodaj" class="btn btn-primary"/>
    </form>
</div>
@endsection('content')
```

Analizując powyższy kod, a właściwie jego centralną część, można wywnioskować, że:

- W celu stworzenia nowego formularza użyto standardowego znacznika języka HTML o nazwie `form` — w znaczniku tym użyto atrybutu `action`, w którym zadeklarowano, który kontroler (`BookController`) i metoda (`store`) ma obsługiwać zapis. Dalej w atrybucie `method` zdefiniowano typ żądania na `POST`.
- Jako pierwszego elementu formularza użyto ukrytego pola (zapis `type=hidden`), którego wartość definiowana jest przez metodę `csrf_token()` — jest to metoda Laravela, która dodaje do formularza token. Dzięki niemu w momencie obsługi żądania Laravel może sprawdzić, czy żądanie faktycznie ma źródło w formularzu. Cała ta funkcjonalność to ochrona przed atakami i próbami wysyłania zmodyfikowanych żądań HTTP.

- Kolejne pola formularzy to standardowe formatki typu `input`. Co interesujące, nazwy pól są zgodne z ich odpowiednikami z bazy danych, co przyda nam się w kodzie obsługi formularza.
- Formularz zatwierdzany jest przyciskiem stworzonym za pomocą znacznika `input` o atrybucie `type` równym `submit`.

Wszystko to pokazuje, że praktycznie jedyną różnicą przy budowie formularzy obsługiwanych przez Laravela jest zdefiniowanie kodu, który będzie go obsługiwał, a także dodanie kodu CSRF, który zabezpieczy nasz kod obsługi przed przesyłaniem żądań spoza formularza — oczywiście w przypadku braku dodania tego ukrytego pola będzie to oznaczało przerwanie operacji i zakończenie pracy skryptu.

Przejdźmy do kodu obsługi, czyli kontrolera `BookController`. Jeszcze niedawno, gdy zajmowaliśmy się modelem i tworzeniem nowych książek, w metodzie `create` tymczasowo dodaliśmy kod, który tworzył nowy obiekt książki. Obiekt ten miał na twardo zapisane parametry dodawanej pozycji. Teraz, gdy mamy już do dyspozycji formularze, możemy zmodyfikować nasz kod i sprawić, by metoda `create` nie robiła nic innego jak tylko wyświetlała widok, który niedawno stworzyliśmy:

```
class BookController extends Controller
{
(….)
    public function create(BookRepository $bookRepo)
    {
      return view('books/create');
    }
```

Obsługę samego zapisywania danych musimy za to przenieść do metody `store`, której kod prezentuje się następująco:

```
public function store(Request $request, BookRepository $bookRepo)
{
  $data = [
        "name" => $request->input('name'),
        "year" => $request->input('year'),
        "publication_place" =>  $request->input('publication_place'),
        "pages" => $request->input('publication_place'),
        "price" => $request->input('publication_place'),
          ];
   $booksList = $bookRepo->create($data);

   return redirect('books');
}
```

Metoda ta za argumenty przyjmuje dwa obiekty — `Request` i `BookRepository`. Oba są wstrzykiwane bezpośrednio z kontenera, nie musimy ich jawnie tworzyć. Ciało opisywanej metody to przede wszystkim zmienna tablicowa. Tak jak przed modyfikacjami przechowuje ona wartości poszczególnych pól tworzonego obiektu — w naszym przypadku oczywiście książki. Każdemu z pól przypisywane są wartości pobierane z obiektu żądania (w tym celu wykorzystywana jest metoda `input`).

Obiekt jest zapisywany w bazie za pomocą metody `create` repozytorium — tu już nic nie ulega zmianie. Warto również przypomnieć, że w momencie tworzenia widoku formularza zwrócono baczną uwagę na nadawanie polom nazw identycznych jak ich odpowiednikom w bazie danych. Dzięki temu można teraz delikatnie przebudować kod metody `store` i zamiast ręcznie przypisywać kolejnym polom obiektu ich wartości z formatek, użyć metody `all`, która zwróci całą, gotową tablicę danych:

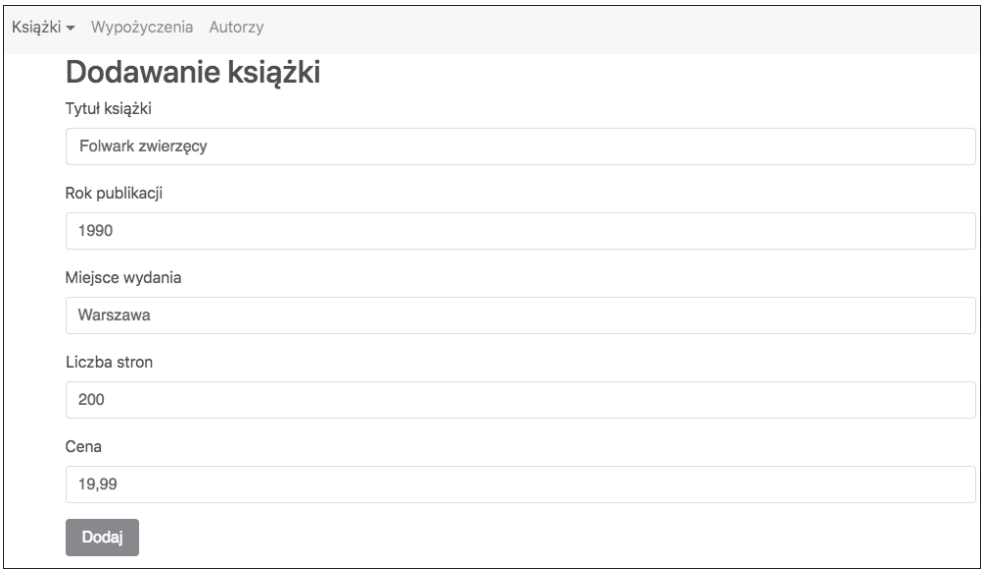
```
public function store(Request $request, BookRepository $bookRepo)
{
   $data =  $request->all();
   $booksList = $bookRepo->create($data);

   return redirect('books');
}
```

Powyższy kod zrealizuje dokładnie tę samą funkcjonalność, co poprzedni — jest jednak zdecydowanie krótszy i bardziej zwięzły. W celu przetestowania nowego kodu należy otworzyć stronę:

```
http://biblioteka.local/books/create
```

Na ekranie powinien wyświetlić się formularz nowej książki, uzupełniony danymi. Przykład widoczny jest na rysunku 5.1.



RYSUNEK 5.1. *Widok formularza nowej książki*

Zatwierdzenie formularza powinno zaowocować dodaniem wpisu do bazy i przekierowaniem na listę wszystkich pozycji, która już powinna zawierać nowy rekord (rysunek 5.2).

RYSUNEK 5.2. *Lista książek z widoczną nową pozycją*

Tę samą czynność — czyli przebudowę dodawania nowego rekordu — spróbujmy wykonać dla tabeli z listą autorów. Analogicznie czynności rozpoczynamy od stworzenia pliku widoku */resources/views/authors/create.blade.php* z zawartością:

```
@extends('template')
@section('title')
         Lista książek
@endsection

@section('content')
<div class="container">
    <h2>Dodawanie autora</h2>
    <form action="{{ action('AuthorController@store')}}" method="POST" role="form">
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
      <div class="form-group">
        <label for="name">Nazwisko</label>
        <input type="text" class="form-control" name="lastname" />
      </div>
      <div class="form-group">
        <label for="name">Imię</label>
        <input type="text" class="form-control" name="firstname" />
      </div>
      <div class="form-group">
        <label for="name">Data urodzenia</label>
        <input type="text" class="form-control" name="birthday" />
      </div>
      <div class="form-group">
        <label for="name">Gatunki</label>
        <input type="text" class="form-control" name="genres" />
      </div>

      <input type="submit" value="Dodaj" class="btn btn-primary"/>
    </form>
</div>
@endsection('content')
```

Zmian jest niewiele — dodane zostały odpowiednie pola (zgodne z bazą danych), a także odpowiedni nagłówek i oczywiście zadeklarowano kod, który będzie służył do obsługi. Podobnie jak w przypadku tabeli książek, modyfikujemy metodę `create` kontrolera `AuthorController`, tak by jej jedyną funkcją było zwracanie widoku formularza.

Zmianie ulega oczywiście również metoda `store`. Ponieważ w tym kontrolerze i modelu autora nie zaimplementowaliśmy jeszcze wzorca repozytorium, odwołujemy się bezpośrednio do Eloquent ORM — tworzymy nowy obiekt, za pomocą metody `fill` uzupełniamy jego pola danymi pobranymi bezpośrednio z formularza i zapisujemy go w bazie, używając przy tym metody `save`:

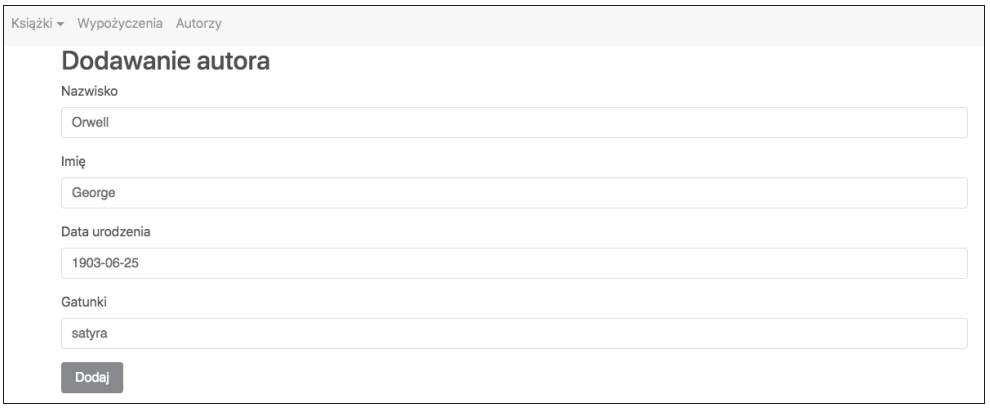
```
class AuthorController extends Controller
{
(…)
  public function create()
    {
        return view('authors/create');
    }
    public function store(Request $request)
    {
      $data = $request->all();
      $author = new Author();
      $author->fill($data);
      $author->save();

      return redirect('authors');
    }
```

Efekt działania powyższego kodu można łatwo przetestować. Wywołanie poniższego adresu URL:

```
http://biblioteka.local/authors/create
```

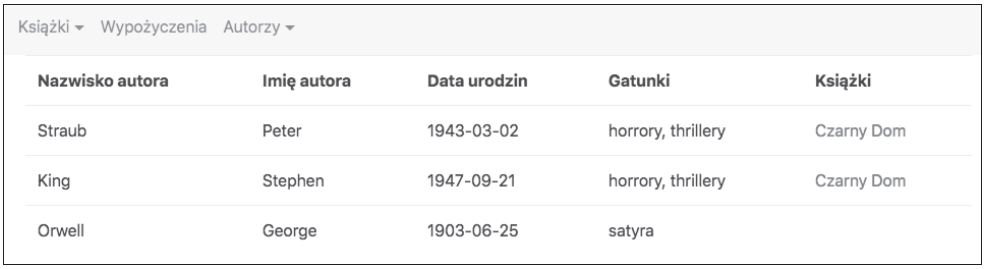
wyświetli formularz nowego autora (rysunek 5.3), którego uzupełnienie i zatwierdzenie powinno zapisać zmiany w bazie.



RYSUNEK 5.3. *Widok formularza nowego autora*

Potwierdzeniem zapisania autora w bazie będzie umiejscowienie go na liście (rysunek 5.4).

Jak widać, dodawanie nowych rekordów nie jest wyjątkowo trudne, gdyż w praktyce używamy lekko zmodyfikowanych klasycznych formularzy zbudowanych w kodzie HTML. Możemy



RYSUNEK 5.4. *Lista autorów z widocznym nowym wpisem*

więc korzystać z wszystkich funkcjonalności i technik tego języka, co powoduje, że ewentualna migracja już istniejącego kodu do kodu, który spełniałby swoją rolę we współpracy z Laravelem, nie jest czynnością trudną.

Na tym etapie ostatnią modyfikacją projektu będzie dodanie nowych pozycji do paska nawigacji. W tym celu należy dodać do pliku */resources/views/template.blade.php* nowe odnośniki, dla książek:

```
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" data-toggle="dropdown"
    ↪href="{{ URL::to('books') }}">Książki</a>
    <div class="dropdown-menu">
        <a class="dropdown-item" href="{{ URL::to('books/cheapest') }}">Top 3 najtańszych</a>
         <a class="dropdown-item" href="{{ URL::to('books/longest') }}">Top 3 najdłuższych</a>
        <a class="dropdown-item" href="{{ URL::to('books') }}">Wszystkie</a>
        <a class="dropdown-item" href="{{ URL::to('books/create') }}">Dodaj nową</a>
</li>
```

a także dla autorów:

```
<li class="nav-item dropdown">
   <a class="nav-link dropdown-toggle" data-toggle="dropdown"
   ↪href="{{ URL::to('authors') }}">Autorzy</a>
   <div class="dropdown-menu">
      <a class="dropdown-item" href="{{ URL::to('authors') }}">Wszyscy</a>
      <a class="dropdown-item" href="{{ URL::to('authors/create') }}">Dodaj nowego</a>
</li>
```

Spróbujmy teraz przyjrzeć się, jak można stworzyć bardziej rozbudowane formularze, w tym z danymi, które są połączone relacją.

5.2. Dodawanie danych powiązanych relacją

Jakkolwiek dodawanie danych okazało się niezbyt trudnym zadaniem, w celu dodania nowych rekordów, które są w relacji z innymi obiektami, należy poczynić pewne przygotowania i nieco dopasować kod napisany w poprzednim rozdziale. Spróbujmy stworzyć formularz, który doda do systemu funkcjonalność tworzenia nowych wypożyczeń.

Gdy zerkniemy do bazy danych projektowanego systemu, a szczególnie do tabeli `loans`, to szybko przypomnimy sobie, że w jej rekordach widnieje klucz obcy o nazwie `book_id`. Jest to oczywiście klucz wykorzystywany do stworzenia wiązania z tabelą `books`, czyli obiektem książki. Tworząc formularz nowego wypożyczenia, musimy pamiętać, że użytkownik nie jest w stanie operować na kluczach — nasz interfejs powinien pozwolić na wybór odpowiedniej książki, na przykład za pomocą tytułu, wykorzystując wybraną formatkę kodu HTML — zapewne w przypadku wyboru książki lista rozwijana będzie najtrafniejszym wyborem. Idąc dalej tym tropem, dochodzimy do wniosku, że widok formularza powinien zawierać listę rozwijaną z wszystkimi dostępnymi w systemie książkami, które oczywiście muszą zostać pobrane również z bazy danych. Zajrzyjmy więc do kontrolera `LoanController`:

```
class LoanController extends Controller
{
(...)
    public function create()
    {
      $books = Book::all();
      return view('loans/create',['books' => $books]);
    }
```

Jak widać na załączonym listingu, modyfikacji uległa metoda `create`, która teraz, podobnie jak w poprzednich przypadkach książek i autorów, jako główne zadanie ma wywołanie widoku `loans/create`. Dodatkowo, w celu realizacji listy rozwijanej z książkami w opisywanej metodzie pobierana jest z bazy lista wszystkich dostępnych pozycji, a następnie przekazywana bezpośrednio do widoku.

Dzięki temu możliwe jest stworzenie samego pliku widoku (*/resources/views/loans/create.blade.php*), którego zawartość wygląda następująco:

```
@extends('template')
@section('title')
         Lista książek
@endsection
@section('content')
<div class="container">

<h2>Dodawanie wypożyczenia</h2>
<form action="{{ action('LoanController@store')}}" method="POST" role="form">
<input type="hidden" name="_token" value="{{ csrf_token() }}" />
<div class="form-group">
        <label for="name">Tytuł książki</label>
        <select type="text" class="form-control" name="book_id">
          @foreach ($books as $book)
            <option value="{{ $book->id }}">{{ $book->name }}</option>
          @endforeach
        </select>
      </div>
      <div class="form-group">
        <label for="name">Dane wypożyczającego</label>
        <input type="text" class="form-control" name="client" />
      </div>
      <div class="form-group">
        <label for="name">Data wypożyczenia</label>
```

```
        <input type="text" class="form-control" name="loaned_on" />
      </div>
      <div class="form-group">
        <label for="name">Przewidywany zwrot</label>
        <input type="text" class="form-control" name="estimated_on" />
      </div>
      <input type="submit" value="Dodaj" class="btn btn-primary"/>
    </form>
</div>
@endsection('content')
```

Zmiany dotyczą oczywiście dodania nowego pola, stworzonego za pomocą znacznika `select`, któremu nadano nazwę `book_id` (podobnie jak klucz obcy tabeli `loans`). W celu wyświetlenia listy użyto pętli `foreach`, która w każdym swoim przebiegu tworzy jeden element `option` o atrybucie `value` równym identyfikatorowi książki, a także etykiecie równej jej nazwie. Dzięki temu wyświetlona lista będzie uzupełniona pełnymi danymi w postaci nazwy książki, natomiast serwer otrzyma wartości klucza.

Kolejny etap to implementacja metody `store` kontrolera książek:

```
public function store(Request $request)
    {
      $book = Book::find($request->input('book_id'));
      $data = $request->all();

      $loan = new Loan();
      $loan->fill($data);
      $book->loans()->save($loan);
      $loan->save();

      return redirect('loans');
    }
```

Jak widać, w pierwszej kolejności na podstawie identyfikatora pobranego z listy rozwijanej wyszukiwany jest obiekt książki, natomiast reszta kodu jest już analogiczna do wersji standardowej — tworzony jest nowy obiekt wypożyczenia (co warte uwagi, nie skorzystano tu z wzorca repozytorium), do którego pól przypisywane są wartości pobrane z formularza. Następnie za pomocą metod `loans` i `save` zapisujemy dane relacyjne, a następnie sam obiekt wypożyczenia.

W celu potwierdzenia skuteczności napisanego kodu warto teraz przejść do adresu:

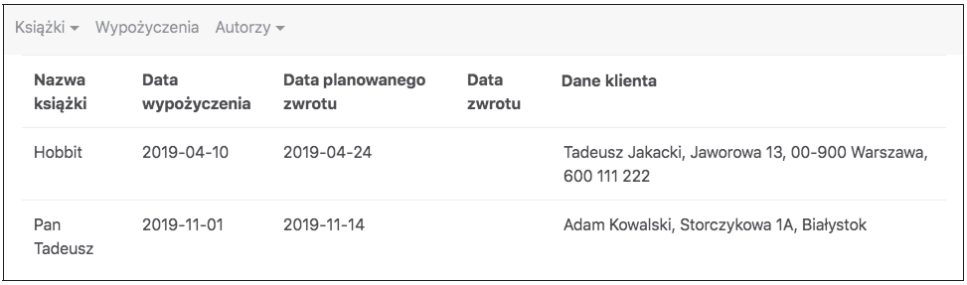
```
http://biblioteka.local/loans/create
```

i wpisać przykładowe dane wypożyczenia. Na rysunku 5.5 umieszczono widok uzupełnionego formularza, gotowego do wysyłki.

Jego zatwierdzenie powinno uruchomić proces zapisywania danych w bazie, czego dowodem będzie zaktualizowana lista, widoczna na rysunku 5.6.

W podobny sposób należy zaktualizować również formularz dodawania nowej książki. Zgodnie z logiką, przy jej dodawaniu chcielibyśmy wybrać autorów — co ważne, może ich być kilku.

RYSUNEK 5.5. *Uzupełniony formularz dodawania nowego wypożyczenia*



RYSUNEK 5.6. *Lista wypożyczeń z widocznym nowym wypożyczeniem*

W tym celu musimy dobrać odpowiedni element interfejsu, który umożliwiałby wybór kilku pozycji — może to być lista checkboxów czy też lista wyboru z dodanym atrybutem `multiple`.

W pierwszej kolejności modyfikacji ulega sam kontroler, a dokładniej metoda `create`:

```
 use App\Models\Author;
(…)
class BookController extends Controller
{
(…)

public function create(BookRepository $bookRepo)
{
  $authors = Author::all();
  return view('books/create',['authors' => $authors]);
}
```

Podobnie jak w przypadku wypożyczeń, tak i tutaj należy pamiętać, że przy tworzeniu formularza niezbędna będzie lista — tym razem autorów. Dane pobieramy oczywiście z bazy i przekazujemy do odpowiedniego widoku formularza. Pamiętamy o dodaniu przestrzeni nazw modelu autora — nie był on jeszcze wykorzystywany w tym kontrolerze.

Kolejne zmiany to modyfikacja widoku nowej książki (plik */resources/views/books/create.blade.php*):

```
(…)
<h2>Dodawanie książki</h2>
    <form action="{{ action('BookController@store')}}" method="POST" role="form">
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />

<div class="form-group">
         <label for="name">Autor</label>
         <select type="text" class="form-control" name="author_id[]" multiple>
                  @foreach ($authors as $author)
                  <option value="{{ $author->id }}">{{ $author->lastname }} {{ $author-
>firstname }}</option>
                  @endforeach
         </select>
</div>
(…)
```

Podobnie jak w przypadku wypożyczeń, także i tu zastosowano:

- Listę rozwijaną `select`, którą zatytułowano `author_id[]`. Użyte nawiasy kwadratowe świadczą o tym, że aplikacja oczekuje tablicy wyników, a nie jednej konkretnej wartości.
- Pętlę typu `foreach`, która dodaje kolejne znaczniki `option` reprezentujące jednego autora.
- Użycie znacznika `multiple` w znaczniku `select` umożliwia wybór więcej niż jednego autora.
- Każde z pól listy rozwijanej ma wartość równą identyfikatorowi autora, a także etykiety w postaci nazwiska i imienia.

Ostatni element układanki to oczywiście zmiany w sposobie obsługi zapisywania formularza. Szybki podgląd kontrolera `BookController` i metody `store` uświadamia, że modyfikacji należy dokonać w repozytorium — wszak właśnie użyliśmy tego wzorca w celu oddzielenia działań na bazie od samego kodu kontrolera.

Zaktualizowany kod repozytorium książki prezentuje się następująco:

```
(…)
class BookRepository extends BaseRepository{

public function __construct(Book $model){
    $this->model = $model;
  }

public function create(array $data) {
  $book = Book::create($data);

  if(isset($data['author_id'])) {
    $book->authors()->sync($data['author_id']);
  }
  return $book;
}
(…)
```

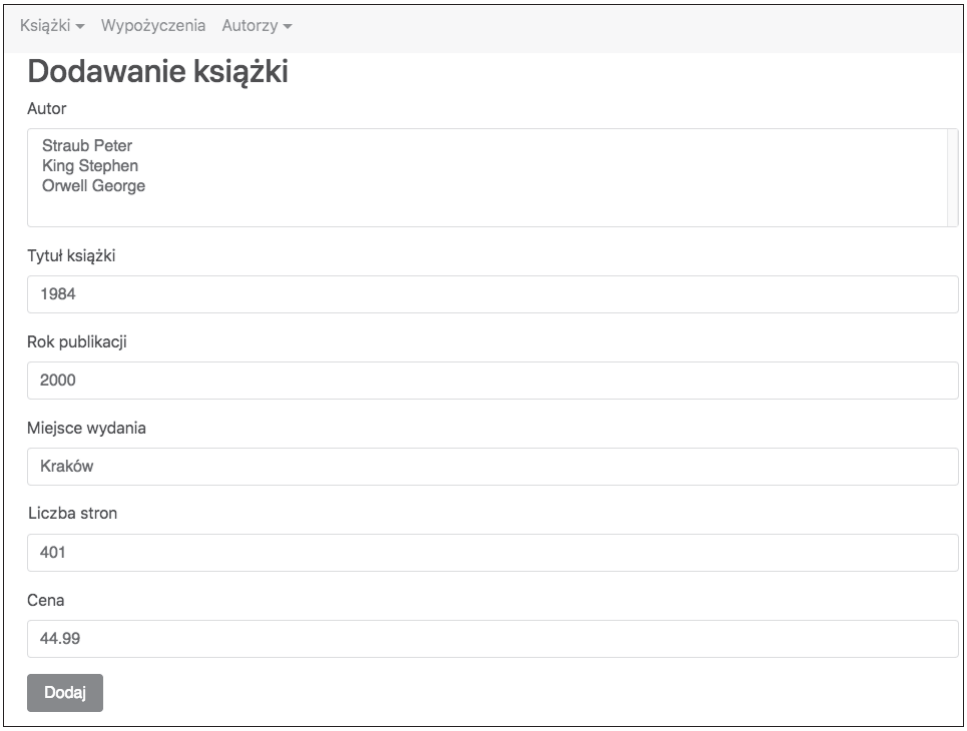
Jak widać, metoda `create` zyskała nowy kod, w którym sprawdzamy, czy dane otrzymane z formularza (czyli żądania HTTP) zawierają informacje o autorach w postaci tablicy identyfikatorów. W tym celu użyta została instrukcja `isset` interpretera. W przypadku istnienia takiej tablicy następuje odwołanie obiektu książki do relacji z autorami, na której wykonywana jest metoda `sync`. Przypomnijmy, że metoda ta przyjmuje tablicę identyfikatorów, które mają zostać dopisane do wybranego obiektu. W naszym przypadku oznacza to, że do wybranej książki zostaną przypisani autorzy o podanych identyfikatorach.

W tym miejscu warto również przypomnieć, że gdyby była potrzeba dodania do relacji tylko jednego rekordu, to możliwe byłoby użycie metody `attach`.

Niemniej jednak powyższe modyfikacje powinny być już gotowe do przetestowania. W tym celu należy otworzyć formularz nowej książki:

```
http://biblioteka.local/books/create
```

a także uzupełnić go przykładowymi danymi (warto skorzystać z opcji kilku autorów). Na rysunku 5.7 przedstawiono formularz z wypełnionymi polami.



RYSUNEK 5.7. *Formularz dodawania nowej książki z możliwością wyboru kilku autorów*

Potwierdzenie poprawnego zapisania nowej pozycji można uzyskać poprzez przejście na listę książek i podgląd nowej. W przypadku wyboru dwóch lub więcej autorów wszyscy oni powinni pojawić się na samym dole panelu (rysunek 5.8).

RYSUNEK 5.8. *Panel podglądu książki z widoczną listą autorów*

Pozostaje już tylko dopisanie w panelu nawigacji opcji dodania nowego wypożyczenia (plik */resources/views/template.blade.php*):

```
   <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" data-toggle="dropdown" href="{{ URL::to('loans')
}}">Wypożyczenia</a>
        <div class="dropdown-menu">
           <a class="dropdown-item" href="{{ URL::to('loans') }}">Wszystkie</a>
           <a class="dropdown-item" href="{{ URL::to('loans/create') }}">Dodaj nową</a>
        </div>
  </li>
```

W tym momencie warto również zapisać zmiany w repozytorium GIT:

```
git add .
git commit -m "Formularze dodawania nowych rekordow"
git push origin master
```

Po poprawnym zapisaniu danych na zdalnym serwerze można przejść do kolejnego rozdziału, czyli próby stworzenia formularzy edycji danych.

5.3. Formularze edycji danych

Oprócz standardowego dodawania nowych rekordów nierzadko istnieje potrzeba aktualizacji już istniejących danych. W niniejszym rozdziale spróbujemy stworzyć nowy formularz, który będzie służył do modyfikacji zapisanych wcześniej rekordów.

W rozdziale opisującym warstwę modelu stworzyliśmy już kod, który odpowiada za realizację samego zapisu — wtedy też na twardo zapisaliśmy dane zaktualizowanego obiektu. Teraz przyszła pora na implementację interfejsu, który zapewni nam dużo większą swobodę, gdyż nowe wartości pól będą pobierane od użytkownika.

Prace rozpoczynają się od modyfikacji kontrolera `BookController` i metody `edit` — w przypadku tworzenia formularza edytującego musimy pamiętać, że w odróżnieniu od tworzenia nowego rekordu, musi być on wypełniony danymi pobranymi z bazy. W tym celu na podstawie identyfikatora uzyskanego ze sparametryzowanego odnośnika listy książek wywołujemy metodę `find` i odnajdujemy rekord, który ma zostać zmodyfikowany. Ponieważ formularz będzie również zawierał listę autorów, ich też pobieramy z bazy i oba stworzone obiekty przekazujemy do widoku edycji. Cała metoda `edit` widoczna jest na poniższym listingu:

```
    public function edit(BookRepository $bookRepo, $id)
    {
      $book = $bookRepo->find($id);
      $authors = Author::all();
      return view('books/edit',['book' => $book,
                                'authors' => $authors]);
    }
```

Przechodząc do pliku widoku edycji (*/resources/views/template.blade.php*), mamy więc do dyspozycji dwa obiekty: książkę, która ma zostać wyedytowana, a także listę wszystkich dostępnych w systemie autorów:

```
@extends('template')
@section('title')
         Edycja książki
@endsection
@section('content')
<div class="container">

<h2>Edycja książki</h2>
<form action="{{ action('BookController@update',[$book->id] )}}" method="POST" role="form">
<input type="hidden" name="_token" value="{{ csrf_token() }}" />

<div class="form-group">
         <label for="name">Autor</label>
         <select type="text" class="form-control" name="author_id[]" multiple>
                  @foreach ($authors as $author)
                           @if(in_array($author->id,$book->authors->pluck('id')->toArray()))
                  <option value="{{ $author->id }}" selected>{{ $author->lastname }} {{
$author->firstname }}</option>
                  @else
                   <option value="{{ $author->id }}">{{ $author->lastname }} {{ $author-
>firstname }}</option>
                  @endif
                  @endforeach
         </select>
</div>
 <input type="hidden" name="book_id" value="{{ $book->id }}" />
<div class="form-group">
         <label for="name">Tytuł książki</label>
         <input type="text" class="form-control" name="name" value="{{ $book->name }}"/>
</div>
<div class="form-group">
         <label for="name">Rok publikacji</label>
         <input type="text" class="form-control" name="year" value="{{ $book->year }}"/>
```

```
</div>
<div class="form-group">
        <label for="name">Miejsce wydania</label>
        <input type="text" class="form-control" name="publication_place" value="
        {{ $book->publication_place }}"/>
</div>
<div class="form-group">
        <label for="name">Liczba stron</label>
        <input type="text" class="form-control" name="pages" value="{{ $book->pages }}"/>
</div>
<div class="form-group">
        <label for="name">Cena</label>
        <input type="text" class="form-control" name="price" value="{{ $book->price }}"/>
</div>
<input type="submit" value="Aktualizuj" class="btn btn-primary"/>
</form>
</div>
@endsection('content')
```

Widok edycji formularza jest bardzo zbliżony do wcześniejszej wersji służącej do tworzenia rekordów. Wprowadzone zmiany to:

- Nagłówek informacyjny.
- Zmodyfikowany atrybut `action` znacznika formularza — podana metoda i kontroler obsługujący, a także w parametrze żądania przekazany identyfikator książki (tak by kod obsługujący wiedział, którą z książek należy zaktualizować).
- Dodanie instrukcji warunkowej w znaczniku `select`, który tworzy listę rozwijaną wszystkich autorów dostępnych w systemie. Instrukcja warunkowa sprawdza, czy identyfikator kolejnego pobranego z listy autora znajduje się na liście autorów przypisanych do relacji z książką. Jeśli identyfikatory są zgodne, oznacza to, że autor z danej iteracji pętli powinien zostać zaznaczony (jako znajdujący się w relacji z książką). Zostanie to zrealizowane poprzez dodanie atrybutu `selected` w znaczniku `option`. W przeciwnym wypadku system ma oczywiście stworzyć zwykłą pozycję listy rozwijanej — niezaznaczoną.
- Kolejne pola formularza posiadają zdefiniowany i przypisany atrybut `value`. Wartości pochodzą z pól obiektu książki, która została przekazana z kontrolera do widoku. Zabieg ten oznacza, że wyświetlone na stronie formatki będą posiadały predefiniowane dane.
- Zmiana etykiety przycisku zatwierdzającego przesłanie formularza do obsługi.

Kolejny etap prac to delikatna modyfikacja tablicy routingu. Co prawda na skutek użycia `resources` mamy domyślnie utworzoną trasę do metody `update` z żądaniem typu `PATCH`, ale zdefiniowany przez nas formularz przesyła dane za pomocą żądania typu POST, więc nad istniejącą już trasę usuwania książki dopisujemy nowy routing do metody `update`:

```
Route::post('/books/{id}/update', 'BookController@update');
Route::get('/books/{id}/delete', 'BookController@destroy');
```

Tak stworzoną trasę warto teraz oprogramować — w tym celu edytujemy kontroler `BookController` i metodę `update`:

```
public function update(Request $request, BookRepository $bookRepo, $id)
{
  $data = $request->all();
  $booksList = $bookRepo->update($data, $id);

  return redirect('books');
}
```

Ciało opisywanej metody składa się z pobrania wszystkich pól formularza do tablicy `$data`, a następnie przekazania jej do metody `update` repozytorium `BookRepository`, którą spróbujemy zaimplementować:

```
public function update(array $data, $id) {
    $book = Book::find($id);
    $book->fill($data);
    $book->save();

    if(isset($data['author_id'])) {
      $book->authors()->sync($data['author_id']);
    }
    return $book;
}
```

Modyfikacje to przede wszystkim odnalezienie odpowiedniej książki (z użyciem identyfikatora uzyskanego z żądania), a także wypełnienie jej pól danymi otrzymanymi z kontrolera (metoda `fill`). Tak zaktualizowany obiekt jest następnie zapisywany, a także wykonywane są czynności związane z przypisywaniem relacji — użycie metod `authors` i `sync` w celu zdefiniowania danych relacyjnych. Cała grupa powyższych czynności jest bliźniaczo podobna do kodu realizującego dodawanie nowego rekordu — należy jedynie pamiętać, że operujemy na już istniejących danych, a nie całkowicie nowych.

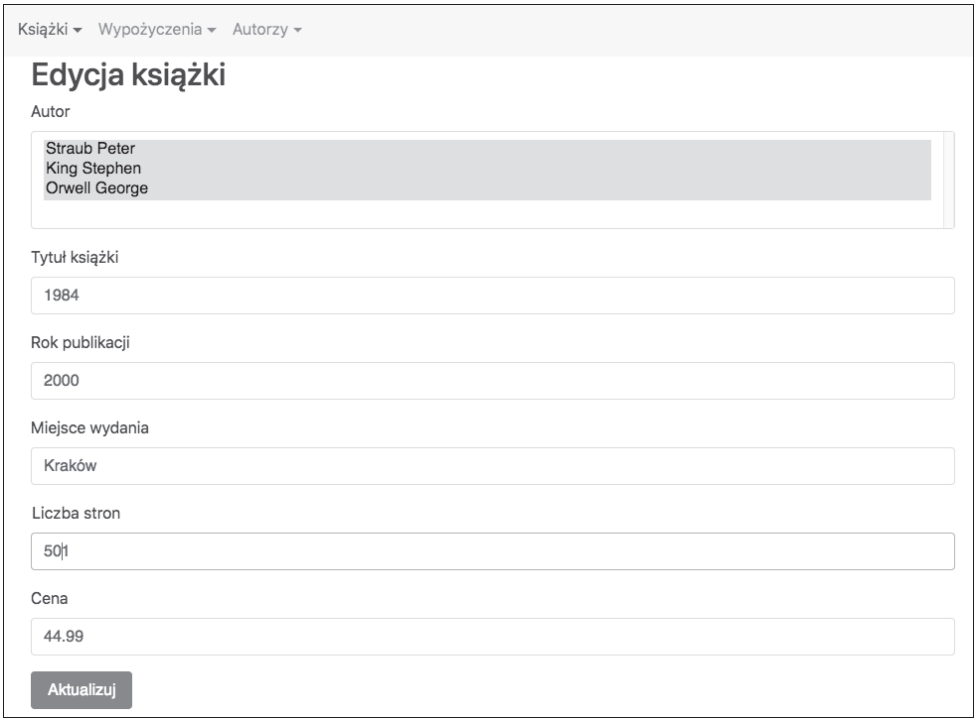
W celu przetestowania nowej funkcjonalności można teraz wybrać opcję edycji wybranej książki, która jest dostępna na stronie listy. Formularz edycyjny powinien zawierać wszystkie dane zgodne z podanymi podczas tworzenia obiektu (przykład na rysunku 5.9).

Wybranie przycisku aktualizacji powinno uruchomić odpowiednią logikę, a po przejściu do podglądu edytowanego rekordu powinny wyświetlić się nowe, nadpisane wartości (rysunek 5.10).

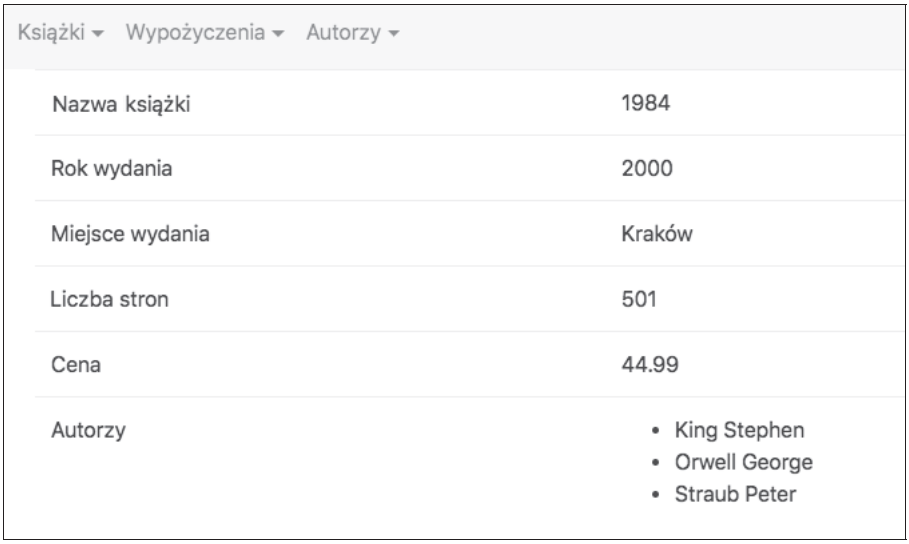
Na zakończenie klasycznie zapisujemy zmiany w repozytorium GIT:

```
git add .
git commit -m "Formularz edycji ksiazki"
git push origin master
```

Podsumowując rozważania na temat edycji, można dojść do wniosku, że najważniejszym jej elementem jest pamięć o tym, by razem z edytowanymi danymi przesyłać również identyfikator obiektu, który podlega edycji. Jest to kluczowe w kwestii zrozumienia idei tej funkcjonalności. Cała reszta, łącznie z odnoszeniem do relacji i samym zapisem, jest bliźniaczo podobna do wcześniej tworzonych formularzy dodających nowe rekordy tabel.



RYSUNEK 5.9. *Formularz edycji książki*



RYSUNEK 5.10. *Podgląd książki z widocznym nowym autorem*

5.4. Walidacja formularzy

W wszystkich testach związanych z dodawaniem i edycją danych zakładaliśmy, że użytkownik świadomie lub nie poprawnie wypełnił wszystkie pola — żadnego nie zostawił pustego, a format danych zgadzał się zakładanym przez nas typem. W rzeczywistych aplikacjach działających produkcyjnie programista nie ma takiego komfortu — wręcz przeciwnie, żelazną zasadą tworzenia kodu obsługujące formularze jest obsługa wszystkich sytuacji wyjątkowych: braku danych, problemów z formatem czy zależności pomiędzy polami (jeśli jedno z pól jest uzupełnione, to inne nie jest obowiązkowe).

Laravel oczywiście posiada zaimplementowane techniki realizujące funkcjonalność walidacji, które od razu wykorzystamy. W tym celu w pierwszym etapie prac należy wybrać miejsce, w którym umieścimy samą logikę testującą dane. W naszym systemie biblioteki dane otrzymywane i obsługiwane są w kontrolerach, a konkretnie w przypadku książek w metodach `store` i `update`. Spróbujmy wyedytować pierwszą z nich:

```
public function store(Request $request, BookRepository $bookRepo)
{
       $validatedData = $request->validate([
           'name' => 'required|max:255',
           'year' => 'required|integer',
           'publication_place' => 'required|string',
           'pages' => 'required|integer',
           'price' => 'required|numeric',
       ]);
       $data = $request->all();
       $book = $bookRepo->create($data);

       return redirect('books');
}
```

Dokonane zmiany to wywołanie metody `validate` na obiekcie żądania (`$request`), które dotarło do serwera. Opisywana metoda jako argument przyjmuje tablicę. Jej kluczami są poszczególne elementy formularza, które mają być testowane pod kątem poprawności. Wartościami są natomiast etykiety, nazwy reguł, które konkretne pola muszą spełnić. Jedno pole może posiadać kilka różnych reguł, odseparowanych od siebie za pomocą symbolu „|”. Co więcej, reguły mogą mieć parametry, tak jak reguła `max`, która za parametr przyjmuje liczbę 255, co oznacza, że długość pola nie może być większa niż ta wartość.

Wśród reguł zaimplementowanych w kodzie powyżej są:

- pole `name` — wymagane, maksymalnie 255 znaków,
- pole `year` — wymagane, liczba całkowita,
- pole `publication_place` — wymagane, ciąg znaków,
- pole `pages` — wymagane, liczba całkowita,
- pole `price` — wymagane, wartość liczbowa.

Przedstawione reguły to oczywiście przykłady pasujące do sytuacji. Tabela 5.1 zawiera najpopularniejsze z zasad walidacyjnych.

TABELA 5.1. *Lista możliwych do użycia reguł*

Nazwa reguły	Testowane pole musi
`accepted`	Posiadać jedną z wartości: `yes`, `on`, `1` lub `true`.
`alpha`	Być ciągiem cyfr i liter.
`after:date`	Być datą po dacie `date`.
`different:field`	Posiadać inną wartość niż pole `field`.
`email`	Być poprawnym adresem poczty e-mail.
`exists:table, column`	Posiadać wartość, która znajduje się wśród danych kolumny `column` tabeli `table`, np. `'miasto' => 'exists:miasta,nazwa'` oznacza, że pole formularza `miasto` musi posiadać wartość, która znajduje się w tabeli `miasta` w kolumnie `nazwa`.
`file`	Posiadać odwołanie do poprawnie wgranego pliku.
`gt:field`	Posiadać wartość większą od wartości pola `field`.
`gt:field`	Posiadać wartość równą wartości pola `field` lub większą.
`file`	Posiadać odwołanie do poprawnie wgranego pliku graficznego (*jpg*, *png*, *bmp*, *gif* lub *svg*).
`in:value1, value2,..`	Posiadać wartość należącą do listy (`value1`, `value2` itd.).
`integer`	Być typu całkowitego.
`ip`	Posiadać wartość będącą poprawnym adresem IP.
`json`	Posiadać wartość będącą poprawnym tekstem reprezentującym dane typu JSON.
`lt:field`	Posiadać wartość mniejszą od wartości pola `field`.
`lte:field`	Posiadać wartość równą wartości pola `field` lub mniejszą.
`max:field`	Posiadać wartość równą polu `field` lub mniejszą.
`min:field`	Posiadać wartość równą polu `field` lub większą.
`numeric`	Być wartością numeryczną.
`required`	Być wypełnione, nie może być puste (włączając pusty ciąg znaków).
`required_if:field,val`	Być wypełnione, jeśli pole `field` jest równe wartości `val`.
`required_with:field1, field2`	Być wypełnione, jeśli któreś z pól `field1`, `field2`… jest wypełnione.

TABELA 5.1. *Lista możliwych do użycia reguł (ciąg dalszy)*

Nazwa reguły	Testowane pole musi
`required_with:field1, field2`	Być wypełnione, jeśli wszystkie z pól `field1`, `field2`… są wypełnione.
`string`	Być ciągiem znaków.
`unique:table,column,except, idColumn`	Posiadać wartość unikalną pośród danych z tabeli `table` i kolumny `column`; za pomocą `except` możliwe jest wyłączenie konkretnych danych z tego warunku.
`url`	Posiadać wartość będącą poprawnym adresem URL.

Tak stworzony kod, posiadający wybrane reguły walidacyjne, możemy teraz przetestować poprzez celowe pominięcie kilku pól i próbę dodania nowej książki. W teorii książka nie powinna zostać dodana — tak też się dzieje, jednak w obecnej chwili po wysłaniu formularza do obsługi jesteśmy przekierowywani ponownie do jego widoku. Zdecydowanie brakuje komentarza czy jakiejkolwiek informacji, gdzie został popełniony błąd i co użytkownik musi zrobić, by system zaakceptował jego żądanie.

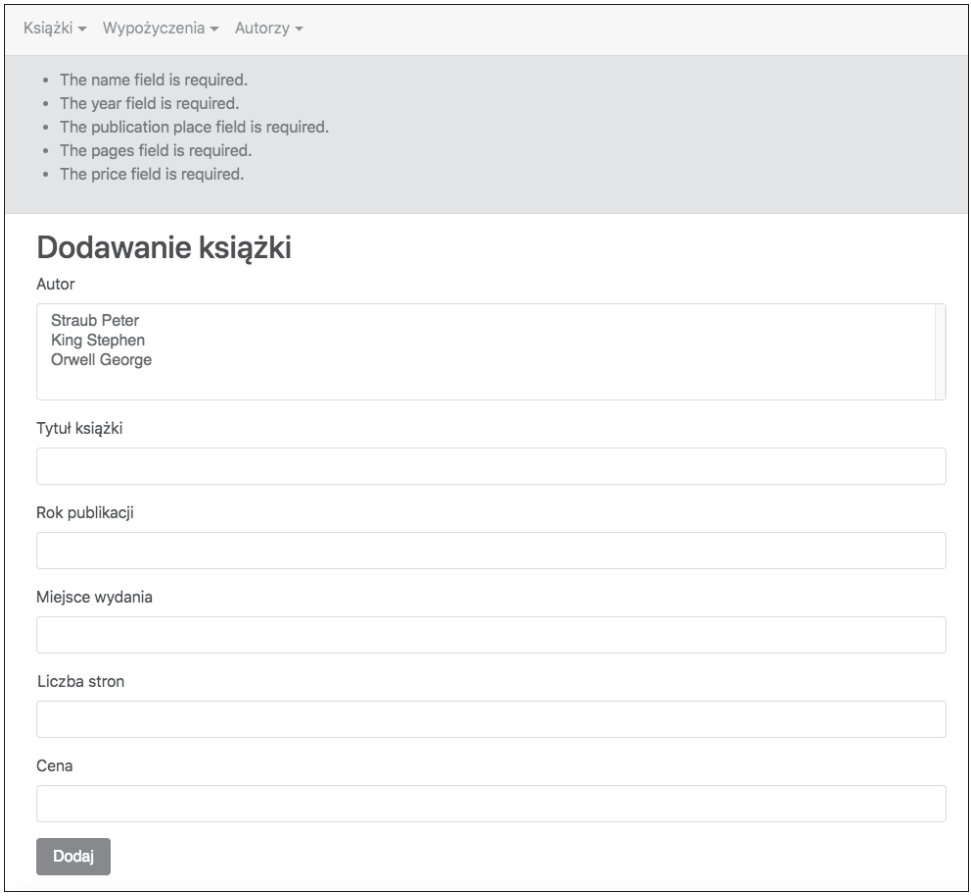
Laravel zapisuje ewentualne błędy walidacyjne w obiekcie sesji przeglądarki. W rezultacie samo przekierowanie, czyli powrót na stronę formularza, nie jest wystarczające — musimy dodać kod, który wyświetli odpowiednie komunikaty. Najbardziej trywialnym miejscem umieszczenia tego typu informacji może być widok formularza — my jednak zastosujemy zdecydowanie bardziej globalne podejście i umieścimy ją bezpośrednio w szablonie strony (plik */resources/views/template.blade.php*, pod znacznikiem zamknięcia nawigacji `nav` i przed poleceniem `@yield('content')`):

```
</nav>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

@yield('content')
```

Tak dodany kod powinien wygenerować odpowiednie komunikaty — oczywiście tylko wtedy, gdy faktycznie poszczególne reguły nie będą spełnione. Przykładowy komunikat walidacji widoczny jest na rysunku 5.11.



Rysunek 5.11. *Przykład błędu walidacji w formularzu dodawania nowej książki*

Dobrym podsumowaniem funkcjonalności walidacji może być pewne przemyślenie: mimo że technicznie jest to nietrudna i szybka funkcjonalność, to spełnia niebagatelne funkcje — powoduje, że logika biznesowa aplikacji, w tym wszelkie odwołania do bazy danych, posiada poprawne i zwalidowane dane. Jest to niezwykle ważne i kluczowe w każdym nowoczesnym systemie aplikacyjnym.

Form Request

Mimo że walidacja umieszczona w kontrolerze w stu procentach spełni swoje funkcje, to Laravel pokusił się o dostarczenie bardziej rozbudowanej techniki kontroli danych, której idea polega na odseparowaniu warstwy walidacji do osobnej klasy i utworzeniu tzw. Form Request. Nierzadko każda z metod kontrolera będzie posiadała osobny zestaw reguł, a co za tym idzie, osobną klasę żądania służącą do walidacji.

I tak kontroler książek w naszym systemie mógłby posiadać osobne żądania dla metody `store` i `updated` — obie one służą do zapisywania danych pobranych z formularza, a co za tym idzie, wymagana jest w nich walidacja poprawności otrzymanych danych. Z pewnością rozważyć można potrzebę tworzenia dwóch oddzielnych klas dla obu tych metod. Być może reguły poprawności danych jednej i drugiej funkcji będą takie same — wtedy można użyć jednej klasy.

Podstawowy obiekt `Form Request` tworzony jest za pomocą linii poleceń Artisan, co spróbujemy przetestować, tworząc obiekt żądania dla kontrolera książek:

```
php artisan make:request StoreBook
```

Efektem wywołania polecenia będzie utworzenie pliku *StoreBook.php* w katalogu *app/Http/Requests/*, który będzie zawierał klasę `StoreBook` dziedziczącą po klasie `FormRequest`. W środku odnaleźć można dwie metody, `authorize` i `rules`. Pierwsza z nich to sprawdzenie, czy dane żądanie w ogóle może się wykonać, czyli czy wywołujący je użytkownik ma możliwość jego wykonania.

Przykładem może być sprawdzenie, czy użytkownik edytuje książkę, do której jest przypisany. Jeśli użytkownik nie ma prawa do wykonania żądania, to funkcja zwróci wartość *false*; w praktyce każda inna odpowiedź da frameworkowi sygnał, że żądanie może zostać dokończone. Warto nadmienić, że w tym przypadku rzeczywistość pokrywa się z teorią — metoda `authorize` to świetne miejsce do dodania specjalnego warunku, który musi zostać spełniony dla konkretnego użytkownika. Zazwyczaj dotyczy jego praw dostępu (edycji nie swoich danych relacyjnych — postów, książek, list itp.).

Druga z metod to `rules`, która — jak sama nazwa wskazuje — przechowuje reguły dotyczące walidacji danego żądania. Możemy w niej używać wszystkich reguł przedstawionych wcześniej. Od razu spróbujmy przebudować nasz ostatni kod walidacji danych z metody `store` kontrolera `BookController` tak, by korzystać z Form Request. W tym celu edytujemy plik *StoreBook.php*, którego zawartość będzie prezentowała się następująco:

```
<?php
namespace App\Http\Requests;
use Illuminate\Foundation\Http\FormRequest;

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

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
```

```
    public function rules()
    {
        return [
             'name' => 'required|max:255',
             'year' => 'required|integer',
             'publication_place' => 'required|string',
             'pages' => 'required|integer',
             'price' => 'required|numeric',
         ];
    }
}
```

Jakie wnioski można wysnuć po analizie kodu?

- Metoda `authorize` zwróci w każdym przypadku logiczną prawdę. Oznacza to, że nie będzie działało żadne ograniczenie dotyczące użytkowników — wszyscy będą mogli wywołać wybrane żądanie.
- Metoda `rules` zdefiniuje szereg reguł i zasad dotyczących pól — jest to dokładna kopia zasad przeniesiona z kontrolera książek.
- Zdefiniowane reguły zwracane są w formie tablicy jako wynik działania metody `rules`.

W celu użycia tak stworzonego żądania musimy zmodyfikować kontroler książek — a dokładniej wskazać miejsce, w którym chcemy go użyć. W tym celu dokonujemy kosmetycznych zmian w metodzie `store` (usunięcie walidacji dodanej w poprzednim rozdziale, dodanie użycia przestrzeni nazw żądań HTTP, usunięcie `use Illuminate\Http\Request;`, a przede wszystkim podmiana żądania typu `Request` na bardziej szczegółowe, dodane przez nas `StoreBook`), której kod po zmianach będzie zgodny z poniższym:

```
(…)
use Illuminate\Http\Request;
use App\Http\Requests\StoreBook;
(…)
public function store(StoreBook $request, BookRepository $bookRepo)
{
   $data = $request->all();
   $book = $bookRepo->create($data);

   return redirect('books');
}
```

Tak dokonane zmiany nie powinny w tej chwili wpłynąć w jakikolwiek sposób na samo działanie aplikacji. Walidacja formularza dodawania nowej książki powinna działać tak jak do tej pory; zmiany dotyczą tylko samej struktury kodu.

Odseparowanie walidacji od kontrolera, mimo odrobinę większego wysiłku ze strony programisty, niesie też ze sobą dużo korzyści:

- możliwość wielokrotnego użycia tych samych reguł i zasad,
- umiejscowienie tych samych reguł w jednym miejscu, przez co ich podmiana czy aktualizacja są łatwiejsze,

- możliwość definiowania dodatkowych reguł dotyczących sprawdzania uprawnień konkretnego użytkownika (metoda `authorize`),
- większa elastyczność, odseparowanie kontrolera od logiki walidacji.

Powyższe zmiany dodajemy do repozytorium:

```
git add .
git commit -m "Form Request"
git push origin master
```

Dobrym ćwiczeniem podsumowującym pracę z Form Requests jest dodanie walidacji do formularzy tabeli wypożyczeń i autorów, do czego oczywiście gorąco zachęcam.

5.5. Internacjonalizacja

Ostatnim tematem poruszanym w rozdziale poświęconym formularzom będzie internacjonalizacja, czyli umiędzynarodowienie projektu. Nie jest to być może temat całościowo i wyłącznie powiązany z formularzami, ale stanowi ogromnie ważną część każdej współczesnej aplikacji internetowej.

Do tematu obsługi wielu języków Laravel podszedł w klasyczny sposób — każdy z języków posiada swój osobny plik, w którym przechowywane są tłumaczenia poszczególnych sentencji, umieszczone w jednej dużej tablicy. Pliki z tłumaczeniami przechowywane są w katalogu zgodnym z formatem: */resources/lang/{nazwa_jezyka}/{nazwa_pliku}.php*

Przykładowy plik z tłumaczeniem może wyglądać następująco (plik *translations.php*):

```
<?php
return [
     'etykieta' => 'Tlumaczenie',
     'powitanie' => 'Witaj, :name',
 ];
```

Pierwszy element tablicy to standardowe tłumaczenie, gdzie pod etykietą o nazwie `etykieta` ukryta jest wartość w postaci napisu `Tlumaczenie`. Drugi wpis to etykieta `powitanie` wraz ze sparametryzowaną wartością.

Warto zauważyć, że domyślnie wraz z zainstalowanym frameworkiem programista otrzyma cztery pliki tłumaczeń:

- *auth.php* — tłumaczenia związane z autentykacją użytkownika,
- *pagination.php* — tłumaczenia związane ze stronicowaniem,
- *passwords.php* — tłumaczenia związane z hasłami,
- *validation.php* — tłumaczenia związane z walidacją.

Oczywiście, w aplikacji jednorazowo ustawiony może być tylko jeden aktualny język. Laravel dokonuje tego wyboru na podstawie ustawień systemowych i samych zmian w kodzie. Domyślny język aplikacji ustawiany jest w 83. linii pliku konfiguracyjnego */app/config/app.php*:

```
'locale' => 'en',
```

Dynamiczna podmiana języka na inny odbywa się za pomocą metody `setLocale()` fasady `App`:

```
App::setLocale("pl"); //ustawi język na pl
```

Powyższe polecenie ustawi język systemu na pl i takiego też pliku będzie wyszukiwać w katalogu tłumaczeń. Ciekawostką jest fakt, że w przypadku nieznalezienia pliku z tłumaczeniami dla wybranego języka wybierany jest język awaryjny. Definiowany jest on w tym samym pliku konfiguracyjnym, co język domyślny, w linii 96.:

```
'fallback_locale' => 'en',
```

Nierzadko programiście przyda się również opcja podglądu, który z języków jest aktualnie używany. W tym celu używana jest funkcja `getLocale()`, również z fasady `App`:

```
$lang = App::getLocale(); //zwróci aktualny język aplikacji
```

W tym miejscu warto zaznaczyć wcale nieoczywisty fakt, że instrukcje ustawiające język działają tylko i wyłącznie w obrębie jednego żądania. Oznacza to, że w celu ustawienia tego samego języka dla całej aplikacji należy użyć dodatkowego kodu, który zapisze aktualnie wybrany język w sesji lub ciasteczkach.

Pozostaje jeszcze jedno pytanie: jak wywołać same tłumaczenia? Jeśli będziemy kontynuować pracę z przykładowym tłumaczeniem przedstawionym kilka linii wyżej, to możemy je wywołać następująco:

```
echo __('translations.etykieta'); //wyświetli napis Tlumaczenie
echo __('translations.powitanie', ['name' => "Adam"]); //wyświetli napis Witaj, Adam
```

W przypadku widoków Blade, użyjemy oczywiście:

```
{{ __('translations.etykieta') }} //wyświetli napis Tlumaczenie
{{ __('translations.powitanie', ['name' => "Adam"]) }} //wyświetli napis Witaj, Adam
```

W celu przećwiczenia nabytej wiedzy spróbujmy teraz zaimplementować internacjonalizację w aplikacji biblioteki, a także przetłumaczyć formularz dodawania nowej książki.

Pierwsza czynność to stworzenie nowej trasy, która będzie odpowiadała za faktyczną zmianę języka. W tym celu w pliku *web.php* dodajemy na samym końcu nowy wpis:

```
Route::get('language/{locale}', function ($locale) {
    session(['locale' => $locale]);
    return redirect()->action('BookController@index');
});
```

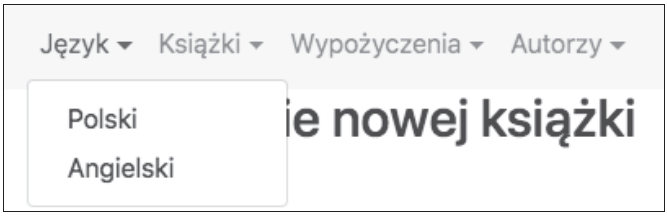
Wpis ten nie jest przypisany do żadnego z kontrolerów, ale od razu wykonuje pewną logikę — przede wszystkim w sesji, pod kluczem `locale`, zapisuje wartość języka pobraną z parametru żądania. I tak żądanie:

- `http://biblioteka.local/language/pl` — ustawi język polski,
- `http://biblioteka.local/language/pl` — ustawi język angielski.

Posiadając zdefiniowane trasy, możemy je teraz dodać do pliku szablonu strony (*resources/views/template.blade.php*) jako oddzielne linki do zmiany języka:

```
(…)
<ul class="nav navbar-nav navbar-right">
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" data-toggle="dropdown" href="#">Język</a>
            <div class="dropdown-menu">
              <a class="dropdown-item" href="{{ URL::to('language/pl') }}">Polski</a>
              <a class="dropdown-item" href="{{ URL::to('language/en') }}">Angielski</a>
            </div>
        </li>
```

Zmiany powinny być już widoczne jako część panelu nawigacji (rysunek 5.12).



Rysunek 5.12. *Menu zmiany języka*

Kolejne kroki to oczywiście stworzenie plików języka. W tym celu tworzymy podwójną strukturę plików i katalogów:

- katalog */resources/lang/en* z plikami: *auth.php*, *forms.php*, *pagination.php*, *passwords.php* i *validation.php*,
- katalog */resources/lang/pl* z plikami: *auth.php*, *forms.php*, *pagination.php*, *passwords.php* i *validation.php*.

Jak łatwo zauważyć, nasze nowe tłumaczenia będą znajdowały się w pliku *forms.php*. Warto jednak przekopiować zawartości pozostałych plików do katalogu z polskimi tłumaczeniami i je przetłumaczyć.

Wracając do samego pliku *forms.php*, możemy teraz uzupełnić go zgodnie z tym, czego faktycznie potrzebujemy w widoku formularza nowej książki. Poniżej znajduje się listing pliku z polskim tłumaczeniem (`/resources/lang/pl/forms.php`):

```
<?php
return [
    'author' => 'Autor',
    'book_title' => 'Tytuł książki',
    'publication_year' => 'Rok wydania',
```

```
    'publication_place' => 'Miejsce wydania',
    'pages' => 'Liczba stron',
    'price' => 'Cena',
    'add' => 'Dodaj',
    'new_book' => 'Dodawanie nowej książki'
];
```

Druga, angielska wersja, trafia oczywiście do *resources/lang/en/forms.php*:

```
<?php

return [
    'author' => 'Author',
    'book_title' => 'Book Title',
    'publication_year' => 'Publication Year',
    'publication_place' => 'Publication Place',
    'pages' => 'Pages',
    'price' => 'Price',
    'add' => 'Add',
    'new_book' => 'Adding a new book'
];
```

Kolejne zmiany to odczytanie zapisanego w sesji języka bezpośrednio w metodzie kontrolera — w naszym wypadku w metodzie `create`, gdyż to ona tworzy formularz dodawania nowej książki. Odczytany z sesji język jest następnie ustawiany jako aktywny przy użyciu metody `setLocale`. Warto przypomnieć, że czynność tę należy wykonać w każdej metodzie, która ma obsługiwać wiele języków. W przyszłości poznamy bardziej ogólne miejsce do jednokrotnego odczytania kodu języka, na razie jednak musi nam wystarczyć takie podejście:

```
public function create(BookRepository $bookRepo)
    {
      \App::setLocale(session('locale'));
      $authors = Author::all();
      return view('books/create',['authors' => $authors]);
    }
```

Ostatnie zmiany to użycie zadeklarowanych etykiet bezpośrednio w formularzu nowej książki (plik */resources/views/books/create.blade.php*):

```
@extends('template')

@section('title')
         Lista książek
@endsection

@section('content')
<div class="container">

    <h2>{{ __('forms.new_book') }}</h2>
    <form action="{{ action('BookController@store')}}" method="POST" role="form">
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />

                          <div class="form-group">
                                  <label for="name">{{ __('forms.author') }}</label>
```

```
                                    <select type="text" class="form-control"
name="author_id[]" multiple>
                                            @foreach ($authors as $author)
                                                <option value="{{ $author->id }}">{{
$author->lastname }} {{ $author->firstname }}</option>
                                            @endforeach
                                    </select>
                            </div>

      <div class="form-group">
        <label for="name">{{ __('forms.book_title') }}</label>
        <input type="text" class="form-control" name="name" />
      </div>

      <div class="form-group">
        <label for="name">{{ __('forms.publication_year') }}</label>
        <input type="text" class="form-control" name="year" />
      </div>

      <div class="form-group">
        <label for="name">{{ __('forms.publication_place') }}</label>
        <input type="text" class="form-control" name="publication_place" />
      </div>

      <div class="form-group">
        <label for="name">{{ __('forms.pages') }}</label>
        <input type="text" class="form-control" name="pages" />
      </div>

      <div class="form-group">
        <label for="name">{{ __('forms.price') }}</label>
        <input type="text" class="form-control" name="price" />
      </div>

      <input type="submit" value="{{ __('forms.add') }}" class="btn btn-primary"/>
    </form>
</div>
@endsection('content')
```

Tak dokonane zmiany powinny zaowocować możliwością przełączenia języka angielskiego na polski — oczywiście na razie tylko w zakresie jednego formularza. W celu modyfikacji reszty kodu należy stworzyć odpowiednie etykiety i przypisać im wartości, dla każdego języka z osobna. Nie można również zapomnieć o ręcznym wywołaniu języka w każdej metodzie kontrolera, lecz ten problem zostanie rozwiązany w jednym z nadchodzących rozdziałów.

Tymczasem zapisujemy zmiany w repozytorium:

```
git add .
git commit -m "Internacjonalizacja"
git push origin master
```

Wysłanie kodu do zewnętrznego repozytorium zakończy nie tylko pracę z umiędzynarodowieniem aplikacji, ale także pracę z formularzami. Co prawda w następnych rozdziałach z pewnością będziemy się z nimi spotykać i z nich korzystać, lecz będą dla nas już tylko narzędziem do

wykonania określonych zadań. Formularze i ich obsługa to jeden z najważniejszych elementów współczesnych aplikacji. Wykorzystujemy je na każdym kroku i nie wygląda na to, by wkrótce miało się to zmienić.

Warto jednak pamiętać o kilku podstawowych zasadach ich tworzenia. Istotne jest pamiętanie o tym, że dane powinny być walidowane i sprawdzane. Ciekawym zagadnieniem jest także badanie UX, czyli tego, jak budować formularze, by były przyjazne, nie męczyły użytkownika, a przy tym wykonywały swoje zadania. Ten temat wykracza już jednak poza ramy niniejszej książki.

Skorowidz

Odkryj możliwości Laravela

- Poznaj nowoczesny framework do tworzenia aplikacji WWW
- Naucz się wykorzystywać jego możliwości w praktyce
- Twórz kod, testuj go i sprawdzaj, czy działa zgodnie z założeniami

Laravel to wydajny, nowoczesny i dostępny za darmo framework do tworzenia aplikacji internetowych, napisany w języku PHP i bazujący na wzorcu architektonicznym Model-View-Controller. Zalety tego rozwiązania ceni coraz więcej programistów tworzących różne aplikacje webowe. Swoją popularność framework zawdzięcza stałemu rozwojowi, spójnemu i przystępnemu kodowi, świetnej dokumentacji oraz aktywnej społeczności, zapewniającej wsparcie również mniej doświadczonym programistom.

Laravel. Wstęp do programowania aplikacji internetowych to świetna książka dla osób, które pragną poznać framework od podstaw. Autor podręcznika postawił sobie za cel przekazanie wiedzy w sposób przyjazny i przystępny. Skoncentrował się na aspekcie praktycznego zastosowania prezentowanych informacji. Dzięki temu czytelnik krok po kroku zagłębia się w kolejne zagadnienia. Tworzy działającą i użyteczną aplikację webową, nie tracąc przy tym czasu na zbędną teorię, którą bez trudu można znaleźć w dokumentacji.

- Podstawowe informacje o Laravelu
- Instalacja i konfiguracja środowiska pracy
- Routing, kontrolery i szablony Blade
- Konfiguracja i używanie bazy danych
- Korzystanie z formularzy i walidacja danych
- Zaawansowane możliwości frameworka
- Uwierzytelnianie użytkowników i tworzenie API
- Publikowanie aplikacji w sieci

Twórz nowoczesne aplikacje przy użyciu doskonałego frameworka!

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5130-1

Cena: 49,00 zł