Play Framework. Tutorial.

Alicja Bodys 183682
Marcin Sokołowski 183718

23 stycznia 2013

Streszczenie

W tutorialu omówiono podstawy architektury MVC w odniesieniu do frameworka Play (Sekcje 1-2). Poruszono tematykę rozwiązań specyficznych dla Play w zakresie tworzenie klas modelu, kontrolera i widoku (Sekcja 3), pokazano przykładową konfigurację bazy danych (Sekcja 4) i pokazano realizację dwóch przypadków użycia: logowania oraz złożenia zamówienia, zaimplementowanych w systemie księgarni internetowej Libro (Sekcja 5).

Spis treści

1 Play

Play jest nowoczesnym frameworkiem służącym do tworzenia aplikacji webowych w języku *Java* i *Scala*. Filozofia Playa zakłada produkt lekki, ale rozszerzalny, sam framework nie jest monolitem, lecz integruje komponenty i biblioteki służące choćby do budowania projektu, mapowania obiektowo relacyjnego (*ORM*), integracji z technologią *AJAX* oraz autentykacji i autoryzacji. Play ma także wsparcie dla frameworka *CSS3* i *HTML5 Boostrap*.

2 Architektura

Play jest zbudowane w oparciu o architekturę model – widok – kontroler (MVC).

2.1 Warstwa modelu

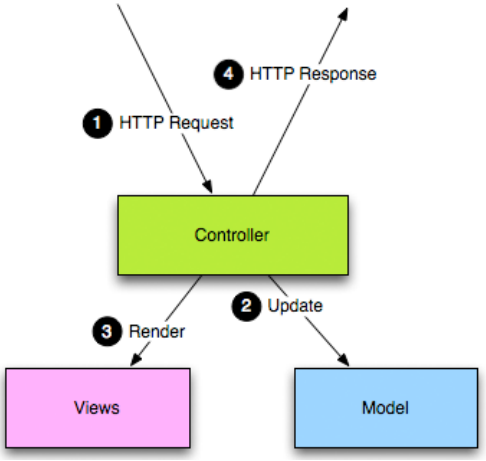
Reprezentuje obiekty z dziedziny problemu, na których użytkownik/system operuje. Przykładem klasy modelowej może być np. Książka, Płatność, Dostawa, Użytkownik. Klasy z modelu posiadają również stosowne metody, np. metodę BigDecimal Zamówienie.obliczCenę(), która wywołana na rzecz obiektu modelowego reprezentującego zamówienie, zwróci całkowitą cenę zamówienia. Podsumowując: w modelu zawiera się cała logika biznesowa.

2.2 Warstwa widoku

Jest pomostem pomiędzy bytami z logiki biznesowej, a użytkownikiem. Odpowiada za prezentowanie treści i umożliwienie interakcji z oprogramowaniem. Widoki z reguły przyjmują obiekty z modelu i na tej podstawie generują użytkownikowi stronę *html*. Przykładowo – widok Prezentacja_koszyka(Koszyk k) jako parametr przyjmuje obiekt z dziedziny – koszyk – a generuje dokument *html*, zawierający listę produktów z koszyka w formie listy numerowanej *
*.

2.3 Warstwa kontrolera

Zarządza żądaniami od użytkownika i odpowiedziami do niego skierowanymi, odpowiada za wyświetlenie odpowiedniego widoku i wywoływanie metod dla obiektów z warstwy modelu.



Rysunek 1: Przepływ danych w architekturze MVC.

3 MVC we frameworku Play

3.1 Warstwa modelu

Jeśli ma zostać zapewniona persystencja danych dla obiektów z dziedziny, klasa modelowa musi zostać poprzedzona adnotacją @Entity oraz musi ona rozszerzać klasę Model. Spowoduje to automatyczne stworzenie tabeli w bazie danych na podstawie opisu klasy! Rozszerzenie o Model dostarczy gotowych metod, upraszczających komunikację z bazą danych w zakresie operacji *CRUD (Create, Read, Update, Delete)* encji związanych z obiektem klasy rozszerzającej Model. Pola klasy zostaną odzwierciedlone automatycznie jako atrybuty w bazie danych. W celu oznaczenia klucza czy zachowaniu większej kontroli nad typami atrybutów, należy wykorzystać mechanizm adnotacji. Na tym etapie nie określamy jeszcze jaki system zarządzania bazą danych będziemy wykorzystywali – nie jest ważne czy to będzie **MySQL**, **PostgreSQL** czy **H2**.

Listing 1: Przykładowy kod klasy modelowej Platnosci

```
1  package models;
2  @Entity
3  public class Platnosci extends Model {
4      @Id
5      public Long idP;
6      public String nazwaP;
7      @Column(scale = 2)
8      public BigDecimal koszt;
9      public static Finder<Long, Platnosci> find = new Finder<Long, ...
           Platnosci>(Long.class, Platnosci.class);
10     public static List<Platnosci> all() { return find.all(); }
11     public static void create(Platnosci platnosci) { platnosci.save(); }
12     public static void delete(Long id) { find.ref(id).delete(); }
13 }
```

3.2 Warstwa widoku

Play do generowania odpowiedzi wykorzystuje *Scalę*, język kompilujący się do bytecodu i działający na wirtualnej maszynie *Javy* (**JVM**). *Scala*, podobnie jak *Java*, jest językiem statycznie typizowanym. Jako język wieloparadygmatowy, jest językiem funkcyjnym, obiektowym i imperatywnym. Kod *Scali* osadzony jest w kodzie *html*.

Listing 2: Przykładowy kod widoku **Pokaz_szczegoly.scala.html**

```
1  @(Ksiazki: List[Ksiazka])
2  <table class="table">
3      @for(ksiazka <- Ksiazki) {
4          <tr>
5          <td><img src='@routes.Assets.at("img/"+ ksiazka.idK + ".jpg")'></td>
6          <td><a href="@routes.Application.szczegoly(ksiazka.id)"> @ksiazka.tytul</a>
7          @for(autor <- ksiazka.autorzy) { @autor.nazwaA, } <br>
8          @ksiazka.wyd,
9          @ksiazka.rokWydania</td>
10         <td>@ksiazka.cena zl</td>
11         <td><a href="@routes.Menadzer.dodaj(ksiazka.idK)"> do koszyka</a></td>
12         </tr>
13     }
14 </table>
15 }
```

3.3 Warstwa kontrolera i routes.conf

Mapowanie żądań użytkowników na właściwy kontroler zachodzi dzięki plikowi routes.conf

Listing 3: Przykładowy wpis w pliku routes.conf

```
1 GET  /cart/add/:id        controllers.MenadzerSesji.dodajDoKoszyka(id: Long)
2 POST /cart/save           controllers.MenadzerZamowienia.przeliczKoszyk()
```

Pierwsza kolumna określa metodę protokołu **HTTP**, po czym występuje względna ścieżka, a na końcu wyspecyfikowana jest metoda kontrolera, która ma się wykonać gdy użytkownik będzie chciał uzyskać dostęp do tejże ścieżki. Metody kontrolera są statyczne i zwracają obiekt typu Result. W Play zdefiniowane są odpowiednie metody, dzięki którym łatwo zwrócić zasób o odpowiednim kodzie odpowiedzi **HTTP**.

Listing 4: Przykładowy kod klasy kontrolera Application

```
1  package controllers;
2  import models.Ksiazka;
3  import play.mvc.Controller;
4  import play.mvc.Result;
5
6  public class Application extends Controller {
7      public static Result pokazWszystkieKsiazki() {
8          return ok(views.html.ksiazkiLista.render(Ksiazka.all()));
9      }
10 }
```

Na pokazanym przykładzie metody pokazWszystkieKsiazki() kontroler zwraca zasób z odpowiedzią 200/OK (kod odpowiedzi protokołu **HTTP**), a stroną zwróconą jest widok ksiazkiLista.scala.html, przyjmujący jako parametr listę książek, które należy wyświetlić użytkownikowi.

3.4 Rozwiązania charakterystyczne

Zanim przejdziemy do realizacji konkretnych przypadków użycia, należy wspomnieć o ważnych cechach Play

3.4.1 Baza danych

Określenie parametrów wykorzystywanego *Systemu Zarządzania Bazą Danych* następuje w pliku konfiguracyjnym conf/application.conf. Jeśli nie mamy lub nie chcemy instalować dodatkowego oprogramowania, *Play* wspiera system zarządzania bazą danych **H2**. **H2** zostało napisane w *Javie* i jest to rozwiązanie, które może zostać wbudowane w aplikację. Przy nowym projekcie domyślnym silnikiem bazy danych jest właśnie baza **H2**.

3.4.2 Mechanizm sesji oraz cache

W *Play* sesja, w odróżnieniu od sesji natywnej z **JEE**, jest przechowywana po stronie użytkownika, a zawartość sesji doklejana jest do kolejnych żądań za pomocą mechanizmu ciasteczek. Z tego powodu rozmiar danych sesji jest ograniczony, tylko **4 kb** i co najważniejsze, w sesji można przechowywać tylko łańcuchy znaków. Ciasteczka są podpisane kluczem i ich modyfikacja po stronie klienta oznacza unieważnienie danych. W *Play* istnieje też mechanizm *cache*, zbliżony do sesji znanej z **JEE**, przechowywanej po stronie serwera. Jednak są pewne różnice, które powodują, że

cache ma ograniczone zastosowanie. Przede wszystkim *cache* jest dostępny globalnie dla wszystkich użytkowników oraz dane przechowywane w *cache* mogą bez ostrzeżenia zniknąć, gdyż jak nazwa wskazuje – *cache* ma być używany do optymalizacji i przechowywania bieżących danych.

4 Przygotowanie środowiska

Zanim przejdziemy do prezentacji kodu, omówimy konfigurację bazy danych. W całym projekcie wykorzystuje się jedne źródło danych i jest nim baza **MySQL**. By komunikacja z zewnętrzną bazą mogła zajść, należy do projektu dołączyć sterownik odpowiedzialny za komunikację z bazą danych. W tym celu, w pliku project/Build.scala należy dodać odpowiednią zależność:

Listing 5: Zawartość pliku Build.scala

```
1  import sbt._
2  import Keys._
3  import PlayProject._
4
5  object ApplicationBuild extends Build {
6      val appName         = "todolist"
7      val appVersion      = "1.0-SNAPSHOT"
8
9      val appDependencies = Seq(
10             "mysql" % "mysql-connector-java" % "5.1.18"
11     )
12     val main = PlayProject(appName, appVersion, appDependencies, mainLang = ...
           JAVA).settings(
13       // Add your own project settings here
14     )
15 }
```

Dzięki temu wpisowi narzędzie **Sbt** do automatycznego budowania projektu (takim narzędziem jest np. **Maven** czy **Ant**) samo dołączy odpowiednie biblioteki. Należy teraz w głównych ustawieniach aplikacji wskazać źródło danych i podać parametry dostępowe do serwera. W pliku applications.conf szukamy sekcji odpowiedzialnej za bazę danych, zaczynającej się od komentarza: # Database configuration. Komentujemy domyślne wpisy *Play* i wprowadzamy własne:

Listing 6: Zawartość pliku applications.conf

```
1  # Database configuration
2  # You can declare as many datasources as you want.
3  # By convention, the default datasource is named `default`
4  #db.default.driver=org.h2.Driver
5  #db.default.url="jdbc:h2:mem:play"
6  db=mysql
7    db.default.driver=com.mysql.jdbc.Driver
8    db.default.url="jdbc:mysql://localhost/lab?characterEncoding=UTF-8"
9    db.default.user=[nazwa uzytkownika]
10   db.default.password=[haslo]
11 # db.default.user=sa
12 # db.default.password=
13 #
14 # You can expose this datasource via JNDI if needed (Useful for JPA)
15 # db.default.jndiName=DefaultDS
```

Oczywiście w miejsce [nazwa użytkownika] oraz [hasło] wprowadzamy własne dane dostępowe. Jeśli nasz serwer nie jest zainstalowany na localhoście, należy również edytować pole db.default.url.

5 Realizacja przypadków użycia

5.1 Logowanie

Przedstawiony zostanie sposób realizacji logowania. Definicje klas ograniczone zostaną do metod i pól wymaganych dla danego przypadku.

5.1.1 Stworzenie modelu

By mogła zachodzić identyfikacja użytkownika w systemie, wprowadzimy klasę modelową Użytkownik, która będzie przechowywała login i hasło. W celu zapewnienia persystencji, klasa będzie rozszerzała klasę Model. Zacznijmy od stworzenia szkieletu klasy:

Listing 7: Kod klasy modelowej Uzytkownicy

```
1  package models;
2
3  import play.db.ebean.*;
4  import javax.persistence.*;
5  import play.data.validation.Constraints.Required;
6
7  public class Uzytkownicy{
8      public Long idU;
9      public String haslo;
10     public String email;
11 }
```

Taka definicja klasy mówi nam, że każda instancja klasy Użytkownicy ma pola idU, haslo oraz email. IdU będzie nam odzwierciedlał identyfikator, będący kluczem głównym, użytkownika w bazie danych. Haslo oraz email posłuży do autentykacji. Ponieważ chcemy zapewnić persystencję danych, uczyńmy z naszej klasy prawdziwą klasę modelową.

Listing 8: Poprawiony kod klasy modelowej Uzytkownicy

```
1  package models;
2  import play.db.ebean.*;
3  import javax.persistence.*;
4  import play.data.validation.Constraints.Required;
5
6  @Entity
7  public class Uzytkownicy extends Model {
8      @Id
9      public Long idU;
10     @Required
11     public String haslo;
12     @Required
13     public String email;
14 }
```

Dzięki dodaniu @Entity, Play automatycznie stworzy nam odpowiednią tabelę w bazie danych, która będzie reprezentowała użytkowników. W celu ułatwienia zapytań do bazy i operacji *CRUD*, niech nasza klasa rozszerza Model. Adnotacje właściwe dla specyfikacji *JPA* określają nam pole, które jako atrybut będzie kluczem głównym (adnotacja @Id). Ponieważ każdy użytkownik musi mieć email oraz haslo do logowania, wprowadźmy przy nich adnotację @Required (wymagane). Wprowadźmy metody, obsługujące *CRUD*.

Listing 9: Dodatkowe metody modelu.

```
1       public static Finder<Long, Uzytkownicy> find = new Finder<Long, ...
            Uzytkownicy>(Long.class, Uzytkownicy.class);
2
3       public static List<Uzytkownicy> all() {
4           return find.all();
5       }
6
7       public static void create(Uzytkownicy user) {
8           user.save();
9       }
10
11      public static void delete(Long id) {
12          find.ref(id).delete();
13      }
14  }
```

Dzięki temu, że nasza klasa rozszerza Model możemy stworzyć właściwy dla naszej klasy obiekt typu Finder, dzięki któremu będziemy mogli pobierać dane z tabeli użytkownicy, bez formułowania zapytań w języku *SQL*. Podobnie odbywać się będzie tworzenie nowego użytkownika oraz jego usuwanie. Kontrakt, który zawarliśmy rozszerzając klasę Model, powoduje, że encje będą tworzone automatycznie, po wywołaniu metody save() na obiekcie. Nie musimy sami pisać ciała tych metod.

Dodajmy jeszcze metodę obsługującą autentykację użytkownika.

Listing 10: Metoda służąca do autentykacji

```
1       public static Uzytkownicy authenticate(String email, String password) {
2           return find.where().eq("email", email).eq("haslo", password).findUnique();
3       }
```

Widać prostotę formułowania zapytań, które umożliwia mapowanie obiektowo relacyjne. Metoda przyjmuje haslo oraz email w postaci łańcucha znaków, a następnie, korzystając z obiektu find wyszuka encję z tabeli użytkownik, której hasło oraz email odpowiadają argumentom metody. Ponieważ w wyniku spodziewamy się unikalnego użytkownika, zakańczamy zapytanie metodą findUnique(), która powoduje, że zwrócona zostanie tylko jedna encja. Gdy podane zostaną nieprawidłowe dane (błędne haslo, nieistniejący email w systemie), metoda nie znajdzie odpowiedniej encji i zostanie zwrócony null.

5.1.2 Obsługa interakcji z użytkownikiem – Kontroler

Mamy już reprezentację użytkownika w systemie. Należy określić sposób interakcji z użytkownikiem. Stwórzmy klasę MenadzerSesji, która przy poprawnym logowaniu zapisze ten fakt w danych sesji.

Listing 11: Kod klasy MenadzerSesji

```
1  package controllers;
2  public class MenadzerSesji extends Controller{
3      public static void zaloguj(String email)
4      {
5      session("email", email);
6      }
```

Sesję należy traktować jako mapę, a metoda session(klucz, wartość) zapisuje w danych sesji wartość wartość pod kluczem klucz. Należy pamiętać informacje ze wstępu – w danych sesji można przechowywać jedynie łańcuchy znaków. Następna klasą, jaką stworzymy będzie MenadzerLogowania, rozszerzający Controller, który będzie kontrolował przepływ danych oraz autentykację. MenadzerLogowania będzie wyposażony w metody zaloguj() - wyświetlanie formularza logowania, sprawdzPoprawnoscDanych() - sprawdzanie poprawności danych, reagowanie na błędne dane oraz przekazywanie sterowania do MenadzeraSesji, który zapisze fakt poprawnej autentykacji oraz wyloguj() - unieważnienie logowanie w bieżącej sesji. By zaprezentować kolejny ciekawy mechanizm wspomagający tworzenie aplikacji webowych w *Play*, pokażemy jak mogą być generowane formularze automatycznie, na podstawie pól klasy. Stwórzmy wewnętrzną klasę w MenadzerzeLogowania, której obiekty będą reprezentowały pojedyncze logowanie.

Listing 12: Kod zagnieżdżonej w MenadzerzeLogowania klasy Login

```
1  public static class Login {
2
3  public String email;
4  public String password;
5
6      public String validate() {
7          if (Uzytkownicy.authenticate(email, password) == null) {
8              return "Wprowadzono niepoprawne dane. Rozwaz skorzystania z ...
                   przypomnienia hasla.";
9          } else{
10             return null;
11         }
12     }
13 }
```

Klasa ma dwa pola – email oraz password, przechowujące odpowiednio wprowadzony adres e-mail użytkownika oraz wprowadzone hasło. Ponadto stworzyliśmy metodę validate(), określającą czy formularz został wypełniony poprawnie – innymi słowy – czy podane dane są prawidłowe i użytkownik zostanie zalogowany.

Do wyświetlenia formularza stworzymy kolejną metodę w klasie MenadzerLogowania – zaloguj(). Będzie ona odpowiedzialna za wyświetlenie odpowiedniego widoku z formularzem logowania. Ponieważ widokami na razie nie będziemy się zajmować, przyjmijmy, że mamy zdefiniowany widok o nazwie login.scala.html, przyjmujący jako argument formularz, który ma zostać wyświetlony.

Wykorzystujemy to w kontrolerze w następujący sposób:

Listing 13: Kod metody zaloguj() klasy kontrolera MenadzerLogowania

```
1      public static Result zaloguj() {
2          return ok(login.render(form(Login.class)));
3      }
```

Gdy widok wygeneruje odpowiednią treść na podstawie listy argumentów, kontroler zwróci efekt w postaci obiektu typu Result, który już bezpośrednio zostanie wyświetlony w przeglądarce użytkownika. Widać tutaj cel stworzonej wcześniej klasy Login.java. Play posiada klasę Form, której metoda form(Class klasa) generuje formularz na podstawie pól tej klasy i przeprowadza walidację na podstawie odpowiednich adnotacji (np. @Required).

Załóżmy, że formularz po wypełnieniu zostanie wysłany do serwera jako żądanie **POST**. Stwórzmy metodę kontrolera, która odbierze wprowadzone dane.

Listing 14: Metoda służąca do autentykacji

```
1      public static Result sprawdzPoprawnoscDanych() {
2          Form<Login> loginForm = form(Login.class).bindFromRequest();
3          if (loginForm.hasErrors()) {
4              return badRequest(login.render(loginForm));
5          } else {
6              MenadzerSesji.zaloguj(loginForm.get().email);
7              return redirect(routes.Application.pokazWszystkieKsiazki());
8          }
9      }
```

Możemy ponownie skorzystać z metody **Form.form()**, która na podstawie danych przesłanych z formularza stworzy obiekt typu Form<Login>. Ponieważ w klasie Login stworzyliśmy metodę validate(), wywołanie metody hasErrors() spowoduje sprawdzenia formularza pod kątem poprawności danych właśnie za pomocą metody validate() (hasErrors() sprawdza też formularz pod kątem wypełnienia wszystkich pól lub też czy dane wprowadzone są zgodnie z opisanym typem w klasie Login). W ciele metody validate() zawarliśmy autentykację, więc brak błędów (hasErrors() zwróci false) oznacza poprawność danych i przekażemy wtedy do MenadzeraSesji polecenie, by zarejestrował on poprawność logowania użytkownika o podanym e-mailu, oraz wyświetlimy użytkownikowi pełną ofertę.

Widać, że w parametrach redirect() nie musimy wprowadzać widoku, wystarczy inna metoda z klasy kontrolera, która ten widok będzie wywoływać. Jest to dobra praktyka, ponieważ gdy zmienimy widok, wyświetlany jako rezultat wykonania metody z kontrolera, to zmianę należy wykonać jedynie w metodzie – pokazWszystkieKsiazki() w klasie kontrolera Application. Jeśli dane zostały wprowadzone błędnie, zwrócimy ponownie stronę logowania, jednak tym razem kod odpowiedzi to 400 Bad Request.

5.1.3 Warstwa prezentacji. Widok.

Listing 15: Kod widoku login.scala.html

```
1  @(form: Form[MenadzerLogowania.Login])
2  @glowna("Zaloguj sie") {
3      @helper.form(routes.MenadzerLogowania.sprawdzPoprawnoscDanych) {
4      @if(form.hasGlobalErrors) {
5          <p class="error">@form.globalError.message</p>
6      }
7      <p> <input type="email" name="email" placeholder="Email" ...
           value="@form("email").value"> </p>
8      <p> <input type="password" name="password" placeholder="Password"> </p>
9      <p> <button type="submit">Login</button> </p>
10     }
11 }
```

Widok przyjmuje jako parametr formularz logowania Form<Login>. W drugiej linii widzimy wywołanie głównego widoku, który jako parametr przyjmuje łańcuch znaków. Zajrzyjmy do widoku głównego:

Listing 16: Kod głównego widoku glowna.scala.html

```
1  @(title: String)(content: Html)
2  <!DOCTYPE html>
3  <html>
4  <head>
5  <title>@title</title>
6  </head>
7  <body>
8      <div class="navbar">
9          <ul class="nav">
10         <li><a href="@routes.Application.pokazWszystkieKsiazki">Oferta</a></li>
11         <li><a href="@routes.MenadzerZamowienia.pokazKoszyk">Koszyk</a></li>
12         <li><a href="@routes.MenadzerLogowania.zaloguj">Zaloguj</a></li>
13         <li><a href="@routes.MenadzerLogowania.wyloguj">Wyloguj</a></li>
14         </ul>
15     </div>
16     <div class="container">
17         <ul>
18         <li><a href="@routes.Application.pokazKsiazkiZKategorii(1)"> Proza ...
               </a></li>
19         <li><a href="@routes.Application.pokazKsiazkiZKategorii(2)"> Literatura ...
               obyczajowa </a></li>
20         <li><a href="@routes.Application.pokazKsiazkiZKategorii(3)"> Biografia ...
               i dokument</a></li>
21         <li><a href="@routes.Application.pokazKsiazkiZKategorii(4)"> ...
               Fantastyka</a></li>
22         </ul>
23     </div>
24     <div> @content</div>
25 </body>
26 </html>
```

Widok główny odpowiada za stworzenie strony głównej, z tytułem przekazanym jako parametr. W ciele widoku następuje stworzenie menu bocznego i menu górnego. @content (linia 24.) zostanie zastąpione przez zawartość widoku, w którym odwołano się do menu głównego. W naszym przypadku będzie to zawartość widoku logowania.

Wróćmy do widoku logowania. Scala ma wsparcie dla generowania formularzy, w postaci pakietu helper. Metoda form na podstawie argumentów generuje formularz *html*. Jako argument przekazaliśmy funkcję walidującą formularz, która ma wykonać się po przesłaniu formularza przez użytkownika. Blok

Listing 17: Sprawdzanie błędu formularza.

```
1  if(form.hasGlobalErrors) {
2      <p class="error">@form.globalError.message</p>
3      }
```

Sprawdza czy formularz, jaki otrzymaliśmy jako argument widoku, zawiera błędy - czy użytkownik podał w poprzednim wywołaniu strony błędne dane logowania. Jeśli tak, to możemy odwołać się do komunikatu błędu, który pochodzi z metody validate(), jaką zdefiniowaliśmy w zagnieżdżonej klasie Login. Niezależnie czy użytkownik ma za sobą błędne próby logowania, czy też wywołuje stronę pierwszy raz, mają mu zostać wyświetlone pola do wprowadzania danych. Wartości parametrów name pól muszą być takie same jak nazwy pól w klasie Login.

Na zakończenie należy umieścić wpisy w pliku routes.conf, które na podstawie adresu wpisanego przez użytkownika, będą wyświetlały odpowiednie zasoby:

Listing 18: Wpisy z pliku routes.conf

```
1 GET  /login              controllers.MenadzerLogowania.zaloguj()
2 POST /login              controllers.MenadzerLogowania.sprawdzPoprawnoscDanych()
```

W ten sposób uzyskaliśmy działający przypadek użycia: Logowanie.

5.2 Złożenie zamówienia

Zajmijmy się realizacją bardziej skomplikowanego przypadku użycia – składania zamówienia.

5.2.1 Prerekwizyty

Złożenie zamówienia może nastąpić wtedy, gdy użytkownik jest zalogowany. By móc to kontrolować, stworzymy metodę w klasie MenadzerSesji zwracającą zalogowanego użytkownika.

Listing 19: Metoda zwracająca aktualnie zalogowanego użytkownika

```
1 public static Uzytkownicy getUser() {
2     String email = session("email");
3     return email == null ? null : Uzytkownicy.findByEmail(email);
4 }
```

W klasach kontrolera będziemy sprawdzać fakt zalogowania poprzez wywołanie tej metody. Jeśli funkcja getUser() zwróci null, będziemy przekierowywali użytkownika do strony logowania. Linie te będą występować w każdej metodzie kontrolera obsługującego zamówienie. Rozpoczęcie składania zamówienia może zajść w trakcie przeglądania koszyka, który musi być niepusty. Realizację obsługi koszyka pominiemy i skupimy się na realizacji samego złożenia zamówienia.

5.2.2 Pierwszy formularz

Stwórzmy najpierw klasę, która będzie reprezentowała adres dostawy, wybrany przez użytkownika. Posłuży on do wygenerowania formularza wprowadzania dostawy oraz do sprawdzenia poprawności danych. Skorzystamy już z własności *Play* znanej z omawiania klasy Logowanie w poprzednim przypadku użycia.

Listing 20: Klasa Adres.java

```
1  package models;
2  import java.util.*;
3  import play.data.validation.Constraints.*;
4
5  public class Adres {
6      @Required
7      public String imie;
8      @Required
9      public String nazwisko;
10     @Required
11     public String miasto;
12     @Required
13     public String kodPocztowy;
14     @Required
15     public String ulica;
16     @Required
17     public String numerDomu;
18     public String numerLokalu;
```

```
19
20      public Adres()
21      {
22      }
23
24      public Adres(String imie, String nazwisko, String miasto, String ...
            kodPocztowy, String ulica, String numerDomu, String numerLokalu) {
25          this.imie = imie;
26          this.nazwisko = nazwisko;
27          this.miasto = miasto;
28          this.kodPocztowy = kodPocztowy;
29          this.ulica = ulica;
30          this.numerDomu = numerDomu;
31          this.numerLokalu = numerLokalu;
32      }
33
34      public String validate() {
35          Pattern p = Pattern.compile("[0-9]{2}-[0-9]{3}");
36          if (kodPocztowy==null || !p.matcher(kodPocztowy).matches()) {
37              return "zly kod pocztowy";
38          }
39          return null;
40      }
41  }
```

Specyfikacja *Javy* mówi, że kiedy w klasie nie zostanie napisany konstruktor, automatycznie jest tworzony pusty konstruktor domyślny. Mechanizm **ORM** wykorzystywany w *Play* wymaga istnienia konstruktora domyślnego, który musimy dopisać (linia 23.), gdyż z powodu istnienia konstruktora w linii 26., konstruktor domyślny nie zostanie wygenerowany automatycznie.

Walidacja będzie odbywała się tylko na podstawie postaci kodu pocztowego (metoda valdiate() - linia 35.) oraz wypełnienia pól wymaganych (adnotacja @Reuqired - linia 9., 11., ...).

5.2.3 Przegląd koszyka

Przyjrzyjmy się pierwszemu widokowi z naszego przypadku użycia – prezentacji koszyka. Wypisywanie odpowiednich towarów pominiemy, gdyż nasz przypadek użycia tym się nie zajmuje.

Listing 21: Widok podgladKoszyka.scala.html

```
1  @(koszyk: Koszyk)
2  @import helper._
3  @glowna("Koszyk") {
4
5      <h1>Zatwierdz zawartosc koszyka</h1>
6      <br>
7      <form method="post" name="formCartConfirm" id="fcc"
8      action="@routes.MenadzerZamowienia.przeliczKoszyk()">
9
10     @if(koszyk.getSize()!=0){
11         ...
12         <input type="submit" name="zatwierdz" value="Zatwierdz">
13         <input type="submit" name="przelicz" value="Przelicz">
14         </form>
15     } else {
16         <h1>koszyk pusty!</h1>
17         <a href="/books">Przejdz do przegladania</a>
18     }
19 }
```

Widok koszyka za parametr przyjmie obiekt klasy Koszyk. W ciele widoku, przy niepustym koszyku (linia 10.), nastąpi prezentacja jego zawartości, oraz dwa przyciski (linia 12. i 13.) – jeden służący do potwierdzenia zawartości koszyka (przejścia do realizacji zamówienia) oraz drugi – służący do przeliczenia zawartość koszyka, przy zmianie ilości egzemplarzy książek. Obie akcje będą obsługiwane przez tę samą metodę kontrolera - MenadzerZamowienia.przeliczKoszyk().

5.2.4 Rozpoznawanie akcji użytkownika

Listing 22: Metoda MenadzerZamowienia.przeliczKoszyk()

```
1  public static Result przeliczKoszyk() {
2      Map<String, String[]> mapa = request().body().asFormUrlEncoded();
3      if (mapa.containsKey("zatwierdz")) {
4          return redirect(routes.MenadzerZamowienia.wybierzOpcjeZamowienia());
5      }
6      ...
7  }
```

Po odczytaniu wysłanego formularza sprawdzimy, czy w jego ciele jest element o nazwie „zatwierdź" (linia 3.). Jego istnienie znaczy, że użytkownik nacisnął przycisk odpowiedzialny za zatwierdzenie towaru w koszyku, a nie za jego przeliczenie. Sterowanie w takim przypadku przekażemy do kontrolera MenadzerZamowienia i metody wybierzOpcjeZamowienia() - linia 4..

5.2.5 Wybór opcji zamówienia.

Listing 23: Metoda MenadzerZamowienia.wybierzOpcjeZamowienia()

```
1  public static Result wybierzOpcjeZamowienia() {
2      SessionKoszyk sessionKoszyk = MenadzerSesji.pobierzKoszykSesji();
3      Koszyk koszyk = new Koszyk(sessionKoszyk);
4
5      List<Dostawa> dostawy = Dostawa.find.all();
6      List<Platnosci> platnosci = Platnosci.find.all();
7      Adres adres = new Adres(uzytkownik.imie, uzytkownik.nazwisko, ...
           uzytkownik.miasto, uzytkownik.kodPocztowy,
8              uzytkownik.ulica, uzytkownik.numerDomu, uzytkownik.numerLokalu);
9      Form<Adres> adresForm = form(Adres.class);
10     adresForm = adresForm.fill(adres);
11
12     return ok(views.html.zamowienieParametry.render(koszyk, adres, dostawy, ...
           platnosci, adresForm));
13 }
```

Metoda wybierzOpcjeZamówienia() ma zwrócić stronę zawierającą formularz do wyboru opcji zamówienia – alternatywnego adresu dostawy, metody płatności oraz rodzaju przesyłki. Na początku pobieramy koszyk z bieżącej sesji (linia 2. i 3.). Ciała tych metod nie zostaną przytoczone, jednak jako wskazówkę do rozwiązania problemu ograniczenia sesji w Play warto wspomnieć, że użyto dwóch klas - Koszyk i SessionKoszyk. SessionKoszyk korzysta z klasy JSONObject do przechowywania koszyka. Klasa ta posiada metody serializacji do łańcucha znaków i tworzenia obiektu na podstawie takiego łańcucha. Klasa Koszyk przechowuje taką samą informację w postaci HashMapy i ma zaimplementowane metody i konstruktory tworzące instancje klasy Koszyk na podstawie instancji klasy SessionKoszyk. Kolejne dwie linie służą do pobrania z bazy danych

wszystkich rodzajów płatności oraz dostawy i stworzenia z nich listy obiektów (linia 5., 6.). Pokazuje to jak mapowanie obiektowo relacyjne upraszcza komunikację z bazą danych i persystencję danych.

Domyślnie użytkownikowi ma być prezentowany jego adres z bazy jako adres dostawy, w tym celu tworzymy instancję klasy Adres za pomocą konstruktora zawierającego wartości wszystkich pól klasy (linia 7.). Następnie na podstawie klasy Adres tworzymy obiekt typu Form<Adres> reprezentujący formularz właściwy dla tej klasy i wypełniamy go wartościami ze stworzonego obiektu (linia 9., 10.).

Po zakończeniu tych instrukcji zwracamy widok z obliczonymi parametrami – aktualną zawartością koszyka, formularzem adresu, listą wszystkich dostaw, listą wszystkich płatności oraz adresem.

5.2.6 Widok opcji zamówienia

Przyjrzyjmy się teraz widokowi.

Listing 24: Widok zamowienieParametry.scala.html

```
1  @(koszyk: Koszyk)(adres: Adres)(dostawy: ...
       List[Dostawa])(platnosci:List[Platnosci])(adresForm:Form[Adres])
2  @import helper._
3  @glowna("Parametry zamowienia") {
4
5  <h3>Wybierz adres dostawy.</h3>
6  @helper.form(routes.MenadzerZamowienia.potwierdzZawartosc) {
7      @if(adresForm.hasGlobalErrors) {
8          <p class="error">@adresForm.globalError.message</p>
9      }
10     @inputText(adresForm("imie"), args = '_help -> "", '_label -> "Imie")
11     @inputText(adresForm("nazwisko"), args = '_help -> "", '_label -> "Nazwisko")
12     @inputText(adresForm("ulica"), args = '_help -> "", '_label -> "Ulica")
13     @inputText(adresForm("numerDomu"), args = '_help -> "", '_label -> "Numer ...
           domu")
14     @inputText(adresForm("numerLokalu"), args = '_help -> "", '_label -> "Numer ...
           lokalu")
15     @inputText(adresForm("kodPocztowy"), args = '_help -> "", '_label -> "Kod ...
           pocztowy")
16     @inputText(adresForm("miasto"), args = '_help -> "", '_label -> "Miasto")
17
18     <h3>Wybierz rodzaj dostawy:</h3>
19     <p>
20     @for(dostawa <- dostawy) { <input type="radio" name="dostawa" ...
           value="@dostawa.idD">
21     @dostawa.nazwaD @dostawa.kosztD zl <br> }
22     </p>
23     <p></p>
24     <h3>Wybierz rodzaj platnosci:</h3>
25     <p>
26     @for(platnosc <- platnosci) { <input type="radio" name="platnosc" ...
           value="@platnosc.idP">
27     @platnosc.nazwaP +@platnosc.koszt zl <br> }
28     </p>
29     <input type="submit" name="zatwierdz" value="Zatwierdz">
30     }
31 }
```

Widok wyświetla formularz wyboru parametrów zamówienia. Metoda która będzie obsługiwała wysłany formularz to MenadzerZamowienia.potwierdzZawartosc() (linia 6.). W dalszej części widoku widać generowanie pól formularza (10., ..., 16.). Zapis

```
1  \textsf{@inputText(adresForm("imie"), args = '_help -> "", '_label -> "Imie")}
```

oznacza wygenerowanie pola typu inputText, o atrybucie name = imie, należącego do formularza adresForm. Przy polu nie będą wyświetlane informacje o jego obligatoryjności oraz pojawi się etykieta o treści „Imię". Konieczne jest określenie przynależności pola do formularza adresForm, w przeciwnym wypadku odtworzenie obiektu po przesłaniu formularza z danymi nie będzie możliwe. W dalszej części tworzymy radiobuttony prezentujące opcje dostawy oraz płatności (20., 27.).

5.2.7 Obsługa formularza

Przejdźmy teraz do metody kontrolera, która zajmie się obsługą wysłanego formularza.

Listing 25: Metoda MenadzerZamowienia.potwierdzZawartosc()

```
1  public static Result potwierdzZawartosc() {
2      Uzytkownicy uzytkownik =MenadzerSesji.getUser();
3      if (uzytkownik == null) {
4          return ok(login.render(form(Login.class)));
5      }
6      Form<Adres> filledForm = form(Adres.class).bindFromRequest();
7
8      if (filledForm.hasErrors()) {
9          return wybierzOpcjeZamowienia();
10     }
11     if (request().body().asFormUrlEncoded().get("dostawa") == null
12         || request().body().asFormUrlEncoded().get("platnosc") == null) {
13         return wybierzOpcjeZamowienia();
14     }
15     Adres adres = filledForm.bindFromRequest().get();
16     Long dostawaId = ...
           Long.parseLong(request().body().asFormUrlEncoded().get("dostawa")[0]);
17     Long platnoscId = ...
           Long.parseLong(request().body().asFormUrlEncoded().get("platnosc")[0]);
18     Koszyk koszyk = new Koszyk(MenadzerSesji.pobierzKoszykSesji());
19     Dostawa dostawa = Dostawa.find.byId(dostawaId);
20     Platnosci platnosc = Platnosci.find.byId(platnoscId);
21     Uzytkownicy user = MenadzerSesji.getUser();
22
23     MenadzerSesji.zapiszAdresWSesji(adres.toString());
24     MenadzerSesji.zapiszPlatnoscWSesji(platnoscId);
25     MenadzerSesji.zapiszDostaweWSesji(dostawaId);
26
27     return ok(views.html.zamowieniePotwierdzTowar.render(koszyk, adres, ...
           dostawa, platnosc));
28     }
```

W pierwszej części metody pobieramy wysłane dane i tworzymy z nich formularz (linia 6.). Sprawdzamy poprawność wprowadzanych danych (linia 7.). Zgodnie z metodą Adres.validate() jeśli kod pocztowy nie przeszedł walidacji albo rodzaj dostawy lub płatności nie zostały wybrane (linia 11.), prezentujemy użytkownikowi ponownie stronę z wyborem opcji zamówienia.

Jeśli natomiast wszystkie pola zostały wypełnione poprawnie, na podstawie wartości z formularza tworzymy obiekty reprezentujące wybory użytkownika (linia 15.-20.). Nazwy pól formularza

adresForm w widoku zgadzają się z polami klasy, do stworzenia obiektu adres wystarczy polecenie:

```
1   Adres adres = filledForm.bindFromRequest().get();
```

Do pobrania dostawy oraz płatności wykorzystujemy ich id i korzystamy z mapowania obiektowo relacyjnego. Wybory użytkownika zapamiętujemy w sesji.

5.2.8 Prezentacja potwierdzenia zamówienia. Widok

Zaprezentujemy kod widoku, potwierdzającego zamówienie.

Listing 26: Widok zamowieniePotwierdzTowar.scala.html

```
1  @(koszyk: Koszyk)(adres: Adres)(dostawa: Dostawa)(platnosc: Platnosci)
2  @import helper._
3  @glowna("Potwierdz towary") {
4
5      <table>
6           <tr>
7               <td>Calkowita cena towaru </td>
8               <td> @koszyk.calcTotalPrice() zl</td>
9           </tr>
10          <tr>
11              <td>Dostawa: <small> @dostawa.nazwaD</small> </td>
12              <td> @dostawa.kosztD zl</td>
13          </tr>
14          <tr>
15              <td>Platnosc: <small>@platnosc.nazwaP </small></td>
16              <td> @platnosc.koszt zl</td>
17          </tr>
18       <tfoot>
19          <th>Razem</th>
20              <td><strong>@(koszyk.calcTotalPrice().add(platnosc.koszt).add(dostawa.kosztD)) ...
                    zl </strong></td>
21          </tr>
22      </tfoot>
23      </table>
24  <div>
25  Wybrany adres:
26      <address >
27        <strong> @adres.imie @adres.nazwisko </strong><br>
28         @adres.ulica @adres.numerDomu  @(if ((adres.numerLokalu == null) || ...
               (adres.numerLokalu=="")) "" else "/" + adres.numerLokalu)<br>
29         @adres.kodPocztowy @adres.miasto<br>
30      </address>
31  </div>
32      @form(routes.MenadzerZamowienia.finalizujZamowienie)    {
33        <input type="submit" name="zatwierdz" value="Zatwierdz zamowienie do ...
              realizacji">
34      }
35  }
```

Widok dokonuje podsumowania zamówienia – wyświetla całkowitą cenę towaru (linia 8.), adres wysyłki (linia 28., ..., 30.) i wybrany rodzaj płatności/dostawy i dopłatę za wybrane opcje (linia 11., ..., 16.). Zatwierdzenie zamówienia powoduje przejście do metody MenadzerZamowienia.finalizujZamowienie().

Ostatnim ogniwem łańcucha czynności należących do składania zamówienia jest finalizacja, obsługiwana metodą finalizujZamowienie().

5.2.9 Finalizacja zamówienia

Listing 27: Obsługa finalizacji zamówienia w metodzie MenadzerZamowienia.finalizujZamowienie()

```
1  public static Result finalizujZamowienie() {
2      Uzytkownicy uzytkownik = MenadzerSesji.getUser();
3      if (uzytkownik == null) {
4          return ok(login.render(form(Login.class)));
5      }
6      Dostawa dostawa = ...
           Dostawa.find.byId(Long.parseLong(MenadzerSesji.pobierzDostaweWSesji()));
7      Platnosci platnosc = ...
           Platnosci.find.byId(Long.parseLong(MenadzerSesji.pobierzPlatnoscWSesji()));
8      String adresWybrany = MenadzerSesji.pobierzAdresWSesji();
9      String adresUzytkownika = MenadzerSesji.getUser().getAdres().toString();
10     String adresDoZamowienia = adresWybrany.equals(adresUzytkownika) ? null : ...
           adresWybrany;
11     Koszyk koszyk = new Koszyk(MenadzerSesji.pobierzKoszykSesji());
12     Zamowienie zamowienie = new Zamowienie(koszyk, adresDoZamowienia, ...
           MenadzerSesji.getUser(), dostawa, platnosc);
13
14     zamowienie.save();
15     MenadzerEmail.wyslijPotwierdzenieZamowienia(zamowienie);
16     return ok(views.html.zamowienieFinalizuj.render());
17 }
```

W tej metodzie kontrolera pobieramy z sesji wszystkie wybory użytkownika – koszyk (linia 12.), wybrany adres dostawy (linia 8.), adres użytkownika (linia 9.) oraz identyfikatory płatności i dostawy (linia 6., 7.). Zebrane dane służą do stworzenia zamówienia (linia 14.). Mając stworzony obiekt wystarczy wywołać na nim metodę save() by został on automatycznie zapisany w bazie danych (linia 15.). Oprócz tego automatycznie zostaną wykonane wpisy w tabelach zależnych – co zostanie przedstawione w kolejnej części tutoriala, podczas omawiania klasy modelowej Zamówienie. Zanim przejdziemy do klas modelowych zwróćmy uwagę na metodę MenadzerEmail.wyslijPotwierdzenieZamowienia(). Za argument przyjmuje ona zamówienie a w jej ciele następuje wysłanie do użytkownika potwierdzenia złożenia zamówienia (linia 17.).

5.2.10 Powiadomienia wiadomością e-mail.

Listing 28: Klasa MenadzerEmaili, odpowiadająca za wysyłanie powiadomień

```
1  import com.typesafe.plugin.*;
2
3  public class MenadzerEmail extends Controller {
4
5      public static void wyslijPotwierdzenieZamowienia(Zamowienie z) {
6          Uzytkownicy u = z.uzytkownik;
7          MailerAPI mail = ...
               play.Play.application().plugin(MailerPlugin.class).email();
8          mail.setSubject("Zamowienie przyjete do realizacji");
9          mail.addRecipient(u.imie + " " + u.nazwisko + " <" + u.email + ">", ...
               u.email);
10         mail.addFrom(Ksiegarnia libro <noreply@email.com>");
11         mail.send(z.drukujZamowienie());
12     }
```

Do stworzenia i wysłania zamówienia wystarczy stworzenie obiektu typu MailerApi. Taką funkcjonalność wprowadza dodatkowa biblioteka z pakietu com.typesafe.plugin.*, którą oczywiście należy zaimportować. Play dzięki narzędziu **sbt** do automatycznego budowania projektu jest w stanie sam ją pobrać. W tym celu należy dodać jedynie zależność w pliku project/Build.scala.

Cały plik Build.scala prezentuje się następująco:

Listing 29: Zawartość pliku project/Build.scala

```
1  import sbt._
2  import Keys._
3  import PlayProject._
4
5  object ApplicationBuild extends Build {
6      val appName         = "todolist"
7      val appVersion      = "1.0-SNAPSHOT"
8
9      val appDependencies = Seq(
10         "mysql" % "mysql-connector-java" % "5.1.18",
11         "com.typesafe" %% "play-plugins-mailer" % "2.0.4"
12     )
13
14     val main = PlayProject(appName, appVersion, appDependencies, mainLang = ...
           JAVA).settings(
15       // Add your own project settings here
16     )
17 }
```

W podanym przykładzie w zależnościach jest także sterownik do systemu zarządzania bazą danych MySQL, gdyż jest on wykorzystywany w projekcie. Za dołączenie biblioteki do wysyłania e-mail odpowiada jedynie wpis:

```
1  "com.typesafe" %% "play-plugins-mailer" % "2.0.4"
```

Ostatnią rzeczą do konfiguracji będzie serwer SMTP. Konfigurację należy umieścić w pliku application.conf. Należy dodać na końcu pliku:

Listing 30: Konfiguracja serwera SMTP w pliku application.conf

```
1  smtp.host=[adres SMTP serwera]
2  smtp.ssl=[yes/no]
3  smtp.user = [user name]
4  smtp.password = [password]
```

Oczywiście pola [adres SMTP serwera], [yes/no], [user name] i [password] zastępujemy danymi właściwymi dla naszej konfiguracji serwera poczty.

Na tym etapie implementacja przypadku użycia jest skończona.

5.3 Użyte klasy modelowe

Przyjrzyjmy się klasie modelowej Zamówienie:

Listing 31: Klasa Zamowienie

```
1  package models;
2
3  import play.db.ebean.*;
4  import javax.persistence.*;
5
6  @Entity
7  public class Zamowienie extends Model {
8      @Id
9      public Long idZ;
10     @Required
11     public Date dataZlozenia;
12     public Date dataPlatnosci;
13     public Date dataWyslania;
14     public boolean zaplacono;
15     public String idListuPrzewozowego;
16     public String innyAdresDostawy;
17     public BigDecimal koszt;
18
19     @ManyToOne
20     public Uzytkownicy uzytkownik;
21
22     @ManyToOne
23     public Dostawa dostawa;
24
25     @ManyToOne
26     public Platnosci platnosc;
27
28     @ManyToOne
29     public Statusy status;
30
31     @OneToMany(mappedBy = "zamowienie", cascade = CascadeType.ALL)
32     public Set<LiniaZamowienia> liniaZamowienia;
33
34     public static Finder<Long, Zamowienie> find = new Finder<Long, ...
           Zamowienie>(Long.class, Zamowienie.class);
35
36     public Zamowienie() {
37     }
38
39     public Zamowienie(Koszyk k, String adres, Uzytkownicy user, Dostawa ...
           dostawa, Platnosci platnosc) {
40         this.dataZlozenia = new Date();
41         this.uzytkownik = user;
42         this.innyAdresDostawy = adres;
43         this.dostawa = dostawa;
44         this.platnosc = platnosc;
45         this.koszt = k.calcTotalPrice().add(dostawa.kosztD).add(platnosc.koszt);
46         this.liniaZamowienia = new HashSet<LiniaZamowienia>();
47
48         Iterator<LiniaKoszyka> koszykIterator = k.iteratorValues();
49         while(koszykIterator.hasNext())
50         {
51             liniaZamowienia.add(new LiniaZamowienia(koszykIterator.next()));
52         }
53     }
```

Ciało klasy wygląda znajomo – Zamówienie podobnie jak Użytkownicy wykorzystuje adnotację @Entity (linia 6.) oraz rozszerza klasę Model (linia 7.). Organizacja danych zakłada, że każde zamówienie składa się ze zbioru LiniiZamówienia (linia 33.). Każda LiniaZamówienia przechowuje książkę oraz jej ilość. By zapewnić persystencję, należy wyspecyfikować w postaci adnotacji związek jeden-do-wielu (linia 32.):

```
1 @OneToMany(mappedBy = "zamowienie", cascade = CascadeType.ALL)
2 public Set<LiniaZamowienia> liniaZamowienia;
```

Mechanizm **JPA** zagwarantuje, że zostanie stworzona nie tylko tabela Zamówienia oraz LiniaZamowienia, ale także trzecia tabela – zamowienie - która będzie przechowywała identyfikator zamówienia oraz identyfikator LiniiZamówienia. Zapis cascade=Cascade.All oznacza, że przy wywołaniu metody save() na instancji klasy Zamówienie, zostanie zapisane nie tylko zamówienie, ale wszystkie obiekty powiązane kluczem obcym – w naszym przypadku do tabeli LinieZamówienia dodane zostaną odpowiednie wpisy.