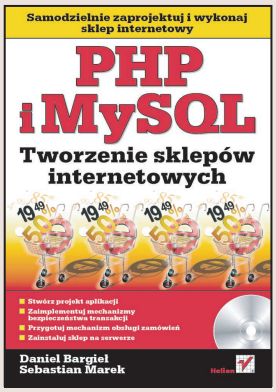
PHP i MySQL. Tworzenie sklepów internetowych



Autorzy: Daniel Bargieł, Sebastian Marek
ISBN: 83-7361-359-5
Format: B5, stron: 244

Coraz więcej firm oferuje swoje towary w internecie. Taka metoda prezentowania oferty umożliwia dotarcie do większej liczby klientów i zredukowanie kosztów wynikających z prowadzenia tradycyjnej działalności handlowej. Rozwój handlu elektronicznego spowodował zwiększenie zainteresowania usługami związanymi z tworzeniem sklepów internetowych.

Do realizacji sklepu internetowego wielu programistów wykorzystuje duet PHP i MySQL. PHP jest najpopularniejszym językiem skryptowym interpretowanym po stronie serwera. Ciągle rozwijany i rozbudowywany PHP jest wykorzystywany przez tysiące autorów dynamicznych aplikacji WWW korzystających z baz danych. Rolę zaplecza bazodanowego doskonale spełni baza MySQL – prosta i wydajna, a co najważniejsze, dostępna nieodpłatnie podobnie, jak PHP. Napisanie efektywnego i bezpiecznego sklepu internetowego to ciekawe wyzwanie dla programisty. Może i Ty spróbujesz się z nim zmierzyć?

Jeśli myślisz o podjęciu tego wyzwania, to książka „PHP i MySQL. Tworzenie sklepów internetowych" jest dla Ciebie idealną lekturą. Zawiera wszystkie informacje, jakich potrzebujesz, by zaprojektować i napisać funkcjonalny, wydajny i bezpieczny sklep internetowy, korzystając z języka PHP i bazy danych MySQL.

- Konfiguracja środowiska projektowego i uruchomieniowego
- Szablony Smarty, biblioteka PEAR i narzędzia kontroli sesji
- Projekt aplikacji z rozbiciem na moduły
- Zagadnienia związane z bezpieczeństwem sklepu i transakcji
- Obsługa formularzy
- Zastosowanie słowników
- Katalog produktów
- Wykonanie modułu koszyka na zakupy
- Moduł administracyjny
- Obsługa zamówień
- Wyszukiwarka towarów

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp **7**

Rozdział 1. Koncepcja sklepu internetowego **11**
Część publiczna 11
Część administracyjna 12

Rozdział 2. Co należy wiedzieć **15**
Środowisko pracy — Windows i Linux 15
System operacyjny 15
Serwer WWW 17
PHP: Hypertext Preprocessor 19
MySQL 19
Podstawowa konfiguracja środowiska 21
Smarty — oddzielenie kodu PHP od HTML-a 24
Instalacja systemu szablonów Smarty 25
Konfiguracja systemu szablonów Smarty 26
Pierwszy szablon 27
Złożone szablony 29
Komunikacja z bazą danych 31
Standaryzacja interfejsu dostępu do danych 31
PEAR oraz pakiet Database 32
Mechanizmy obsługi i raportowania błędów 38
Typy błędów 38
Obsługa błędów poprzez standardowe funkcje języka PHP 40
Obsługa błędów w zbiorze bibliotek PEAR 46
Mechanizmy autoryzacji użytkownika i sesje 50
Identyfikacja użytkownika 50
Mechanizmy sesji 63
XML jako narzędzie konfiguracji aplikacji 69
Cele korzystania z plików konfiguracyjnych 69
Dane konfiguracyjne w dokumentach XML 70
Dane informacyjne w plikach XML 72

Rozdział 3. Projekt aplikacji **75**
Interfejs użytkownika 76
Nagłówek strony 77
Menu główne sklepu 78
Część centralna sklepu 78
Stopka strony 80

Struktura i konfiguracja aplikacji ... 80
Struktura katalogowa ... 80
Konfiguracja serwisu ... 83
Przetwarzanie żądań ... 84
Struktura bazy danych ... 85
Użytkownicy i klienci sklepu ... 86
Produkty ... 87
Kategorie ... 90
Producenci ... 91
Zamówienia ... 92
Słowniki ... 94
Biblioteka zdjęć ... 95
Budowa modułowa aplikacji ... 96
Rdzeń aplikacji ... 97
Przykładowy prosty moduł aplikacji ... 98

Rozdział 4. Bezpieczeństwo ... 103
Bezpieczeństwo systemu operacyjnego oraz serwera WWW ... 103
Cel instalacji serwera ... 104
Tylko potrzebne usługi ... 104
Bezpieczna konfiguracja serwera WWW ... 105
Bezpieczeństwo wykorzystywanego oprogramowania ... 105
Instalacja PHP jako pliku wykonywalnego CGI ... 105
Instalacja PHP jako modułu Apache ... 107
Opcja register_globals ... 108
Raportowanie błędów ... 110
Ukrywanie PHP ... 111
Aktualizacje ... 111
Bezpieczeństwo własnej aplikacji ... 112
Brak walidacji danych ... 112
Nieskuteczne mechanizmy kontroli dostępu i autoryzacji ... 113
Nieprawidłowe zarządzanie kontami oraz sesjami użytkowników ... 115
Ataki typu Cross-Site Scripting (XSS) ... 116
Wstrzykiwanie kodu ... 117
Przechowywanie niezabezpieczonych danych ... 119
Bezpieczeństwo bazy danych ... 119
Zarządzanie hasłami ... 120

Rozdział 5. Obsługa formularzy ... 121
Format dokumentu XML definiującego formularz ... 121
Pola formularza i reguły walidacyjne ... 122
Dokument form.xml ... 124
Moduł formularza ... 128
Metody obiektów klasy Forms ... 128
Konfiguracja obiektu formularza w skryptach PHP ... 130
Wyświetlanie formularza w szablonach Smarty ... 132
Testowanie aplikacji z formularzem ... 134

Rozdział 6. Słowniki i ich zastosowanie ... 139
Object — klasa bazowa dla obiektów ... 140
Dane adresowe — klasa State, Country oraz AddressType ... 141
Słowniki wykorzystywane przy składaniu zamówień — klasa DeliveryType, PaymentType, OrderStatus ... 145
Waluty i stawki podatku VAT — klasa Currency oraz TaxRate ... 146
Parametry asortymentu — klasa Parameter ... 147

Producenci produktów — klasa Producer ... 148
Biblioteka zdjęć — klasa Image ... 148
Przesyłanie zdjęć na serwer — pakiet HTTP_Upload ... 150
Zapisywanie zdjęć w bazie danych ... 153
Pobieranie zdjęć z bazy danych ... 153

Rozdział 7. Kategorie i produkty ... 155
Asortyment i produkt — różnice i zastosowania ... 155
Asortyment sklepu — klasa Item ... 155
Dodawanie nowego asortymentu do sklepu ... 158
Produkty dostępne w ofercie sklepu — klasa Product ... 162
Produkty w promocji ... 167
Obsługa promocji — moduł Special ... 168
Zarządzanie promocjami ... 170
Kategorie produktów ... 171
Struktura katalogowa — klasa Catalog ... 172
Wyświetlanie struktury katalogowej ... 173
Zarządzanie kategoriami ... 174

Rozdział 8. Koszyk ... 177
Sesja jako podstawowy mechanizm realizacji koncepcji koszyka ... 178
Moduł koszyka — klasa Basket ... 179
Operacje na produktach w koszyku ... 180
Operacje na sumarycznych wartościach cen produktów w koszyku ... 182
Składanie zamówienia ... 182
Wyświetlanie koszyka w szablonie TPL ... 184

Rozdział 9. Rejestracja i zarządzanie klientami ... 189
Koncepcja użytkowników aplikacji ... 190
Klasy użytkownika —– User oraz CustomUser ... 190
Rejestracja nowego użytkownika ... 193
Pierwszy etap rejestracji — wypełnienie formularzy rejestracyjnych ... 194
Drugi etap rejestracji — aktywacja konta użytkownika ... 208
Proces gromadzenia danych za pomocą wielu formularzy ... 210
Zarządzanie klientami ... 211

Rozdział 10. Obsługa zamówień ... 213
Warunki złożenia zamówienia ... 213
Moduł zamówienia — klasa Order ... 216
Właściwości i metody obiektów klasy Order ... 216
Zarządzanie zamówieniami ... 218

Rozdział 11. Wyszukiwanie informacji ... 221
Formularz wyszukiwarki ... 221
Analizator danych ... 222
Wyszukiwanie informacji — klasa Search ... 225
Stworzenie i wysłanie zapytania do bazy danych — metoda makeSQLQuery() ... 226
Pobieranie wyników wyszukiwania — metoda fetchQueryResult() ... 229

Rozdział 12. Instalacja sklepu internetowego ... 231
Instalacja sklepu od strony serwera WWW ... 231
Instalacja sklepu od strony bazy danych ... 232
Plik konfiguracyjny sklepu internetowego ... 232

Skorowidz ... 235

Rozdział 8.
Koszyk

Koncepcja koszyka w sklepie internetowym została zapożyczona z rzeczywistości. Podczas wizyty w zwykłym sklepie przeglądamy półki sklepowe w poszukiwaniu interesujących nas towarów. Towar, który znajduje się na półkach, nie jest rozłożony na nich dowolnie, lecz pogrupowany według pewnych kryteriów. Na przykład nabiał może znajdować się w lodowce, a soki owocowe na górnej półce pod ścianą.

W hipermarketach całe działy zorganizowane są w ten sposób. Rozłożenie towaru (odpowiadającego produktom w naszym sklepie internetowym) w odpowiednim miejscu sklepu, które dodatkowo może być stosownie opisane, odpowiada kategoriom w naszym wirtualnym sklepie.

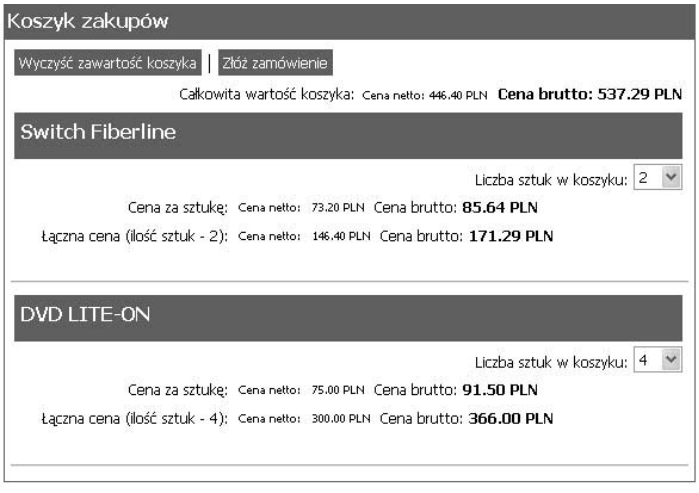
Wyobraźmy sobie teraz, że klient zauważa interesujący go towar, który chce kupić. Chwyta towar, biegnie do kasy, płaci, zostawia za sklepem (daje komuś, ewentualnie wiezie do domu), po czym wraca po następny. Łatwo sobie wyobrazić, że zrobienie nieco większych zakupów w takim przypadku zabrałoby kilka dni lub nawet tygodni.

Dlatego też klienci sklepów i hipermarketów używają koszyków lub wózków, do których mogą powkładać produkty. Gdy już umieszczą w koszykach wszystko, co chcą kupić, niosą je do kasy, gdzie towar jest podliczany i gdzie następuje zapłata (gotówką, kartą, czekiem).

Mechanizm koszyka jest również wykorzystywany w sklepie internetowym. Dzięki takiemu rozwiązaniu internauta odwiedzający sklep internetowy, gdy znajdzie jakiś interesujący go produkt, może dodać go do koszyka, a następnie powrócić do dalszego przeglądania asortymentu sklepu. Koszyk będzie „pamiętał", jakie produkty wybrał klient. Z kolei sam klient może następnie zamówić wszystkie produkty znajdujące się w koszyku.

Koszyk zakupów w sklepie internetowym ma jeszcze tę przewagę nad koszykiem zakupów w rzeczywistym sklepie, że użytkownik może zwiększać lub zmniejszać liczbę sztuk produktów w koszyku bez potrzeby ponownego odwiedzania kategorii, które zawierają te produkty.

Przykładowa zawartość koszyka w aplikacji sklepu internetowego dodanego do książki została pokazana na rysunku 8.1.



Rysunek 8.1. *Zawartość koszyka w aplikacji sklepu internetowego dołączonego do książki*

Sesja jako podstawowy mechanizm realizacji koncepcji koszyka

Koszyk ma to do siebie, że powinien „pamiętać" swoją zawartość niezależnie od tego, na której stronie sklepu w danym momencie znajduje się użytkownik. Co więcej, jeżeli opuści on serwis sklepu internetowego, a po pewnym czasie powróci do niego, to koszyk wciąż powinien zawierać produkty, które użytkownik do niego dodał.

Idealnym rozwiązaniem jest przechowywanie obiektu koszyka w zmiennej sesyjnej. Dzięki temu wszystkie opisane powyżej problemy rozwiązujemy za jednym zamachem.

W sklepie internetowym, który został dołączony do książki, klasę obiektu koszka (`Basket`) można znaleźć w pliku modułu koszyka o nazwie *basket.inc.php*.

Pliki aplikacji sklepu internetowego dołączonego do książki można znaleźć na CD-ROM-ie, w katalogu *sklepinternetowy*. Skrypt modułu koszyka znajduje się w katalogu *sklepinternetowy/modules/basket/*.

W naszym sklepie internetowym moduł koszyka jest jawnie określany w pliku konfiguracyjnym, tak jak zostało to pokazane na listingu 8.1.

Listing 8.1. *Plik module.cfg.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    ...
```

```
    <section id="modules">
        ...
        <param id="Basket" path="../modules/basket/" file="basket.inc.php"/>
        ...
    </section>
</config>
```

W głównym pliku części publicznej (*mindex.php*) moduł koszyka jest ładowany przez metodę `loadModule` rdzenia aplikacji (listing 8.2):

Listing 8.2. *Ładowanie modułu koszyka przez metodę 'loadModule' obiektu rdzenia aplikacji*

```
if (PEAR::isError($err = $appCore->loadModule('Basket'))){
    echo($appCore->getLastErrorMessage()); exit();
}
```

Moduł koszyka powinien być załadowany jeszcze przed rozpoczęciem sesji (`session_start`), ponieważ sesja przechowuje obiekt koszyka, a więc definicja klasy obiektu musi już być znana interpreterowi PHP.

Obiekt koszyka tworzony jest tylko jeden raz w czasie trwania konkretnej sesji (listing 8.3).

Listing 8.3. *Tworzenie obiektu koszyka w skrypcie mindex.php*

```
<?php
...
//Rozpoczęcie sesji
session_start();
//Obiekt koszyka tworzony jest na początku
if (!isset($_SESSION['Basket'])) {
    session_register('Basket');
    $_SESSION['Basket'] = new Basket();
}
...
?>
```

Od tej pory koszyk może być używany przez internautę, który robi zakupy w sklepie.

Moduł koszyka — klasa Basket

Jak większość tego typu obiektów w sklepie internetowym również moduł koszyka nie stanowi samodzielnego modułu. Współpracuje ściśle z modułem produktu (`Product`) oraz z modułem zamówień (`Order`), dlatego też na początku pliku definiującego klasę `Basket` znajduje się kod dołączający pliki obu modułów.

Ponieważ moduł `Order` również korzysta z modułu `Product`, można by pominąć w skrypcie modułu `Basket` fragment kodu ładującego moduł `Product`. Jednak ponieważ w pliku modułu `Basket` mamy bezpośrednie odwołania do obiektów klasy `Product`, dobrze jest jawnie określić, jakie moduły są wykorzystywane przez obiekt.

Moduł zawierający definicję obiektu klasy `Product` został opisany w poprzednim rozdziale, natomiast moduł zawierający definicję obiektu klasy `Order` zostanie opisany w rozdziale „Obsługa zamówień".

Obiekty klasy `Basket` posiadają jedynie dwie właściwości:

- `_products` — jest to tablica, która zawiera obiekty produktów dodanych do koszyka;
- `_maxquantity` — limit sztuk jednego produktu w koszyku (wartość domyślnie jest ustawiana na 10).

Metody obiektów klasy `Basket` można podzielić na trzy grupy:

- Metody bezpośrednio operujące na zawartości koszyka, czyli na produktach, które się w koszyku znajdują: `getBasket`, `addProduct`, `changeProductCount`, `getProductFromBasket`, `clearBasket`.
- Metody operujące na wartościach cen netto i brutto produktów znajdujących się w koszyku: `getTotalPrice`, `getTotalPriceNetto` oraz `getTotalPriceBrutto`.
- Metoda składająca wykonująca proces składania zamówienia: `placeOrder`.

Opis poszczególnych metod obiektów klasy `Basket`, lista ich atrybutów oraz rodzaj zawracanej wartości zostały opisane w pliku modułu koszyka *basket.inc.php*.

Operacje na produktach w koszyku

Korzystając z metod wymienionych powyżej, w pierwszej grupie programista może dodać produkt do koszyka poprzez proste wywołanie metody `addProduct`. Musi jedynie dysponować 32-znakowym identyfikatorem produktu (wynik działania funkcji `md5`).

Jeżeli dodanie produktu do koszyka się nie powiedzie (bo np. produkt o podanym identyfikatorze nie istnieje), metoda `addProduct` zwróci obiekt klas `PEAR_Error`. W skrypcie realizującym operację dodawania produktu do koszyka stosowny fragment kodu wygląda jak na listingu 8.4 (plik *sklepinternetowy/www/public/basket/addtobasket.php*):

Listing 8.4. *Fragment pliku addtobasket.php — dodanie produktu do koszyka*

```
<?php
...
//Sprawdzenie, czy podano zmienną typu GET stanowiącą identyfikator obiektu produktu
if (isset($_GET['prodid'])) {
     if(PEAR::isError($_SESSION['Basket']->addProduct($_GET['prodid']))) {
          //Operacja dodania produktu do koszyka nie powiodła się
```

```
            //Wyświetlenie komunikatu o błędzie
        } else {
            //Operacja dodania produktu do koszyka powiodła się
            //Wyświetlenie informacji potwierdzającej dodanie produktu do koszyka
        }
    }
    ...
    ?>
```

Jak widać, z punktu widzenia programisty korzystającego z interfejsu obiektu koszyka dodanie produktu do koszyka jest naprawdę banalne. Cała praca sprowadza się do wyświetlenia odpowiedniego komunikatu informującego użytkownika o tym, czy operacja dodania produktu do koszyka powiodła się czy też nie.

Jeżeli wystąpił błąd (czyli metoda `addProduct` zwróciła obiekt klasy `PEAR_Error`), kompletną treść komunikatu o tym błędzie można uzyskać poprzez wywołanie `$appCore->getLastErrorMessage()`.

> Metoda `addProduct` sama sprawdzi, czy produkt, który ma być dodany, nie znajduje się już w koszyku i, jeżeli tak jest, zamiast dodawać nowy obiekt do koszyka (czyli tworzyć nowy obiekt klasy `Product`) zwiększy jedynie liczbę sztuk tego produktu. Operacja ta jest powtarzana, dopóki liczba sztuk danego produktu w koszyku nie osiągnie wartości maksymalnej (określonej we właściwości `_maxquantity` obiektu koszyka).

Modyfikacja zawartości koszyka oznacza zmianę liczby produktów znajdujących się koszyku lub też usunięcie ich. Za realizację tego zadania odpowiedzialna jest metoda `changeProductCount` obiektów klasy `Basket`.

Przykładowy kod wykorzystujący tę metodę wygląda jak na listingu 8.5 (patrz plik *sklepinternetowy/www/public/basket/displaybasket.php*):

Listing 8.5. *Fragment pliku displaybasket.php — zmiana liczby sztuk danego produktu w koszyku*

```
<?php
...
if(PEAR::isError($_SESSION['Basket']->changeProductCount($_GET['prodid'],
↪$_GET['act']))) {
    //Wystąpił błąd - nie można zmodyfikować zawartości słownika
}
?>
```

Jak widać w powyższym przykładzie, metoda `changeProductCount` przyjmuje dwa parametry: identyfikator obiektu produktu (obiektu klasy `Product`), którego liczba sztuk ma zostać zmieniona, oraz nowa liczba sztuk wybranego produktu. Jeżeli nowa liczba sztuk produktu wynosi zero, produkt jest usuwany z koszyka.

Jeżeli internauta będzie chciał wyczyścić koszyk zakupów, to programista obsługujący akcje użytkownika może to zrobić, wywołując metodę `clearBasket` obiektu koszyka. Metoda nie przyjmuje żadnych parametrów.

Operacje na sumarycznych wartościach cen produktów w koszyku

Ponieważ wyświetlając internaucie zawartość koszyka, należy również pokazać sumaryczną wartość cen produktów w koszyku, w klasie `Basket` zostały zdefiniowane dwie bardzo przydatne metody.

Te metody to: `getTotalPriceNetto()`oraz `getTotalPrice()`. Pierwsza z nich sumuje wszystkie ceny netto produktów znajdujących się w koszyku przemnożone przez liczbę sztuk każdego z produktów, natomiast druga robi dokładnie to samo z tą różnicą, że bierze pod uwagę ceny brutto produktów, czyli do ceny każdego z produktów dolicza odpowiednią stawkę podatku VAT.

Obie metody uwzględniają fakt, że produkty znajdujące się w koszyku mogą mieć obowiązujące ceny promocyjne. W takim przypadku cena wliczana do sumy cen (zarówno netto, jak i brutto) jest oczywiście ceną promocyjną produktu znajdującego się w koszyku.

Stawki podatku VAT są zdefiniowane w słowniku stawek podatku VAT. Każdy produkt posiada odwołanie do odpowiedniej wartości w słowniku stawek podatku VAT. Mechanizm słowników oraz ich zastosowanie w sklepie internetowym zostało omówione w rozdziale „Słowniki i ich zastosowanie".

Mechanizm promocji został omówiony w rozdziale „Kategorie i produkty" w podrozdziale „Produkty w promocji".

W przypadku sklepu internetowego obie metody są wywoływane jedynie w szablonach *Smarty,* czyli w plikach TPL. O samym szablonie TPL wykorzystywanym do wyświetlania kodu HTML koszyka powiemy w dalszej części rozdziału.

Składanie zamówienia

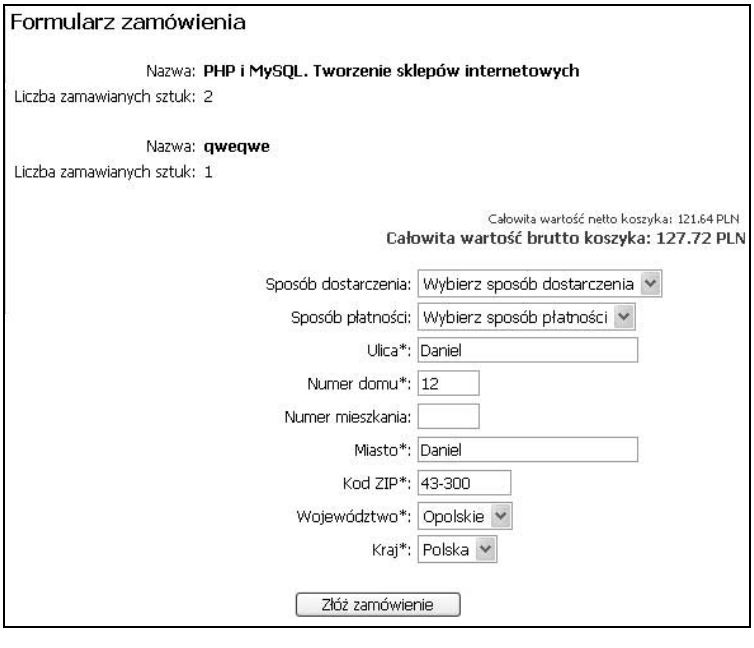
Z punktu widzenia programisty PHP obsługującego moduł koszyka składanie zamówień jest najtrudniejszą procedurą do oprogramowania. Dzieje się tak ponieważ, aby złożyć zamówienie, trzeba zebrać dodatkowe dane od internauty wypełniającego zamówienie.

W przypadku aplikacji sklepu internetowego dołączonego do książki są to dane adresowe klienta, dzięki którym wiadomo, na jaki adres wysłać zamówienie.

Z punktu widzenia obiektu koszyka sposób, w jaki owe dane zostaną zgromadzone, nie jest istotny. Ważny jest sposób dostarczenia tych danych do obiektu koszyka. Operacja składania zamówienia sprowadza się do wywołania metody `placeOrder($ordAddr)` obiektu koszyka. Atrybut `$ordAddr` (adres zamówienia) musi być tablicą asocjacyjną zawierającą następujące pola:

- `dt` (rodzaj dostawy) — identyfikator pola słownikowego `DeliveryType`. Słownik `DeliveryType` zawiera informacje o tym, w jaki sposób zawartość zamówienia powinna być dostarczona do klienta;
- `pt` (rodzaj płatności) — identyfikator pola słownikowego `PaymentType`. Słownik `PaymentType` zawiera informacje o tym, w jaki sposób klient zamierza dokonać płatności w sklepie internetowym;
- `astrnm` (nazwa ulicy) — nazwa ulicy, na którą towar ma zostać dostarczony;
- `astrnum` (numer domu) — numer domu, do którego towar ma zostać dostarczony;
- `ahomenum` (numer lokalu) — opcjonalny numer lokalu, do którego towar ma zostać dostarczony;
- `acity` (miasto) — miasto, do którego towar ma zostać dostarczony;
- `astate` (województwo, stan, okręg) — identyfikator pola słownikowego `State`. Słownik `State` zawiera listę województw (stanów lub obszarów) danego kraju;
- `acountry` (kraj) — identyfikator pola słownikowego `Country`. Słownik `Country` zawiera listę krajów.

W przypadku aplikacji sklepu internetowego dołączonej do książki wszystkie te dane są gromadzone w trakcie składania zamówienia przez formularz (rysunek 8.2).



Rysunek 8.2. *Moment składania zamówienia przez klienta sklepu internetowego*

Ponieważ tylko użytkownicy zarejestrowani w sklepie internetowym mogą złożyć zamówienie, dlatego też pola formularza z rysunku 8.2 wstępnie zostają wypełnione danymi adresowymi, które użytkownik zarejestrowany (klient) podaje w trakcie rejestracji.

O procesie rejestracji użytkownika w systemie i aktywacji konta użytkownika można przeczytać więcej w rozdziale „Rejestracja i zarządzanie klientami”.

Dane te mogą być zmodyfikowane przez użytkownika lub pozostawione bez zmian. Formularz jest wygenerowany i obsługiwany przez moduł `Forms`, który został dokładnie opisany w rozdziale „Obsługa formularzy”.

Po złożeniu zamówienie jest zapisywane w bazie danych z odpowiednim statusem.

Koszyk sklepu internetowego ściśle współpracuje z obiektem zamówień. Żeby jednak użytkownik mógł złożyć zamówienie, musi się wcześniej zarejestrować w sklepie internetowym. Kolejny rozdział „Rejestracja i zarządzanie klientami” prezentuje sposób, w jaki w aplikacji sklepu internetowego dołączonej do książki rozwiązano ten problem.

Po złożeniu zamówienia koszyk jest czyszczony, a użytkownik może ponownie przystąpić do zakupów.

Wyświetlanie koszyka w szablonie TPL

Mimo że do obiektu klasy `Basket` (o nazwie `Basket`), który jest przechowywane w sesji, w samym szablonie TPL można się dostać poprzez konstrukcje `{$smarty.session.Basket}`, to dla wygody w głównym pliku części publicznej (*mindex.php*) obiekt ten jest przekazywany przez referencję do szablonów *Smarty* pod nazwą `Basket: $smarty->assign_by_ref('Basket', $_SESSION['Basket']);`.

Dzięki temu w szablonach TPL do koszyka można się już odwoływać poprzez nazwę `$Basket`.

Wyświetlanie informacji w koszyku możemy podzielić na dwie główne sekcje: podsumowanie oraz szczegółowa zawartość koszyka.

W przytoczonych poniżej listingu szablonu *basket.tpl* (listing 8.6) pominęliśmy większość znaczników kodu HTML, aby nie komplikować samego kodu szablonu *Smarty*. Odpowiednie fragmenty kodu zostały odpowiednio skomentowane.

Listing 8.6. *Kod (wycięto znaczniki HTML) szablonu basket.tpl*

```
{*Całkowita wartość koszyka:*}
{*Cena netto*}{$Basket->getTotalPriceNetto()|string_format:"%.2f"}
{*Cena brutto*} {$Basket->getTotalPriceBrutto()|string_format:"%.2f"}
```

Sekcja podsumowania zawiera sumaryczną wartość cen netto oraz brutto z koszyka. Do ich uzyskania wystarczy skorzystać z metod `getTotalPriceNetto()` oraz `getTotalPriceBrutto()`. Ponieważ jednak nie możemy zagwarantować, że wartości zwrócone

przez obie funkcję będą poprawnymi wartościami walutowymi (precyzja do 2. miejsca po przecinku), szczególnie w przypadku wartości brutto, która jest obliczana przez dodanie do wartości netto stawki podatku VAT, należy skorzystać z modyfikatorów *Smarty*.

Modyfikator zmiennej `string_format` pozwala na określenie formatu wyświetlania wartości, której dotyczy. W powyższym przypadku nakazujemy wyświetlanie wartości zmienno-pozycyjnych z dokładnością do dwóch miejsc po przecinku (po zaokrągleniu).

W sekcji przedstawiającej zawartość koszyka stworzona jest pętla `foreach`, która wykonuje iteracje po tablicy produktów z koszyka. Tablica zwracana jest przez metodę `getBasket` obiektu koszyka.

```
...
{foreach from=$Basket->getBasket() item=product}
     ...
```

Nazwa produktu znajdującego się w koszyku jest jednocześnie łączem do strony prezentacyjnej produktu. W szablonie TPL została zdefiniowana następująco:

```
{* Nazwa produktu w koszyku*}
<a href="?prodid={$product[0]->getId()}">{$product[0]->_Item->getName()}</a>
```

Jako łącze, które tworzy zmienną typu `GET` o nazwie `prodid` zawierającą identyfikator obiektu produktu, jest wyświetlana nazwa obiektu asortymentu powiązanego z produktem.

Następnie definiowana jest lista rozwijana, z której można wybrać liczbę sztuk danego produktu:

```
...
{*Lista rozwijana zawierająca liczbę sztuk produktu*}
<select onChange="changeBasket(this, '{$product[0]->getId()}')">
     {assign var=maxquantity value=$Basket->getMaxQuantity()}
     {section name=licz loop=$maxquantity+1 start=0 max=$maxquantity+1}
          <option value="{$smarty.section.licz.iteration-1}"
          ↳{if $smarty.section.licz.iteration-1 == $product[1]}selected{/if}>
               {$smarty.section.licz.iteration-1}
          </option>
     {/section}
</select>
...
```

Do wygenerowanie listy rozwijanej zawierającej liczbę sztuk, które można wybrać, został wykorzystany element `section`**.** Maksymalna liczba sztuk, która zostanie wygenerowana na liście, jest uzyskiwana przez metodę `getMaxQuantity()` obiektu `koszyka`.

Jak widać element `select` (element formularza HTML) zawiera atrybut `onChange`, który powoduje, że po zmianie elementu wyświetlanego na liście rozwijanej wywoływana jest funkcja JavaScript o nazwie `changeBasket`. Sama funkcja wygląda następująco:

```
<script>
{literal}
     //Funkcja rozpoczynająca proces modyfikacji zawartości koszyka
```

```
        function changeBasket(src, prodid) {
            document.location.href= '?menu=1679e7b1b297c07de53ee3651f4c759&prodid=
            ↳'+prodid+'&act='+src.value;
        }
        ...
        ...
    {/literal}
    </script>
```

Funkcja `literal` języka szablonów *Smarty* powoduje, że kod, który znajduje się między znacznikami `{literal}` oraz `{/literal}`, jest nie jest interpretowany w trakcie kompilacji szablonu. Gdybyśmy nie użyli tego elementu, kompilacja nie przebiegłaby pomyślnie, ponieważ klamrowy nawias { otwierający ciało funkcji `changeBasket` zostałby zinterpretowany jako rozpoczęcie funkcji *Smarty*.

Pozostały fragment kodu szablonu TPL jest odpowiedzialny za wygenerowanie informacji o cenach produktu. Jeżeli produkt jest objęty promocją, należy wyświetlić ceny, tak jak zostało to pokazane na rysunku 8.3.

Rysunek 8.3. *Wyświetlanie ceny detalicznej produktu znajdującego się w koszyku. Produkt jest objęty promocją*

Aby sprawdzić, czy produkt jest objęty promocją, należy skorzystać z metody `hasProductAvalSpecial()` obiektu klasy `Product`. Metoda ta zwraca wartość `TRUE`, jeżeli produkt jest objęty promocją, oraz `FALSE` w przeciwnym razie. Ponieważ obiekty produktów dodane do koszyka są, podobnie jak cały koszyk, przechowywane w zmiennej sesyjnej, podczas wywołania metody `hasProductAvalSpecial` podajemy jako argument wartość `TRUE`. Dzięki temu na obiekcie klasy `Product` zostaje wymuszone odświeżenie informacji o promocjach (informacje te są wtedy pobierane bezpośrednio z bazy danych — uaktualniana jest wartość właściwości `_Special` obiektu klasy `Product`).

Więcej o metodach obiektu klasy `Product` można przeczytać w rozdziale „Kategorie i produkty".

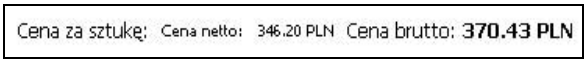
```
{*Cena za sztukę*}
{if $product[0]->hasProductAvalSpecial(TRUE)}
```

Jeżeli produkt jest objęty promocją, wyświetlamy najpierw ceny detaliczne. Wykorzystujemy tutaj metody obiektu klasy `Product`, takie jak: `getPriceNetto()` i `getSpecialPriceNetto()`, aby wyświetlić ceny netto, oraz `getPriceBrutto()` i `getSpecialPriceBrutto()`, aby wyświetlić ceny brutto:

```
{*Ceny promocyjne detaliczne*}

{*Cena netto produktu bez promocji (przekreślona) *}
{$product[0]->getPriceNetto()}
{*Cena promocyjna netto produktu (na czerwono) *}
{$product[0]->getSpecialPriceNetto()}
```

```
{*Cena brutto produktu bez promocji (przekreślona) *}
{$product[0]->getPriceBrutto()|string_format:"%.2f"}
(*Cena promocyjna brutto produktu (na czerwono) *}
{$product[0]->getSpecialPriceBrutto()|string_format:"%.2f"}
```

Jeżeli produkt nie jest objęty promocją, wyświetlane ceny są uzyskiwane tylko przez metody `getPriceNetto()` oraz `getPriceBrutto()` (rysunek 8.4):



Rysunek 8.4. *Wyświetlanie ceny detalicznej produktu znajdującego się w koszyku. Produkt nie jest objęty promocją*

```
{else}
     {*Ceny bez promocji*}

     {*Cena netto produktu *}
     {$product[0]->getPriceNetto()}

     {*Cena brutto produktu*}
     {$product[0]->getPriceBrutto()|string_format:"%.2f"}
```

Następnie w koszyku jest wyświetlana łączna wartość cenowa produktu, która zależy od liczby sztuk danego produktu w koszyku:

```
{*Liczba sztuk produktu *}
{$product[1]}
```

Jeżeli produkt jest objęty promocją, ceny powinny być wyświetlane tak, jak zostało to pokazane na rysunku 8.5.

Rysunek 8.5. *Wyświetlanie łącznej wartości cenowej produktu w koszyku, produkt jest objęty promocją*

```
{if $product[0]->hasProductAvalSpecial()}
```

Jak widać, tym razem metoda `hasProductAvalSpecial()` obiektu klasy `Product` nie posiada żadnego argumentu. Nie trzeba odświeżać informacji o promocji, ponieważ zostało to zrobione wcześniej w tym samym szablonie TPL.

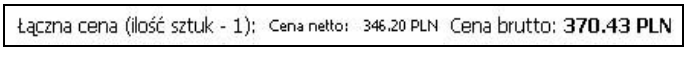
Aby wyświetlić wartość sumaryczną liczby sztuk danego produktu w koszyku, korzystamy ze znanych nam już metod klasy `Product` oraz z funkcji `math` szablonów *Smarty*:

```
{*Łączna wartość produktu w promocji*}

{*Wartość netto wszystkich sztuk produktu bez promocji (przekreślona)*}
{math equation="x*y" x=$product[0]->getPriceNetto() y=$product[1] format="%.2f"}

{*Cena netto wszystkich sztuk produktu w promocji (na czerwono)*}
{math equation="x*y" x=$product[0]->getSpecialPriceNetto() y=$product[1] format="%.2f"}

{*Cena brutto wszystkich sztuk produktu bez promocji (przekreślona)*}
{math equation="x*y" x=$product[0]->getPriceBrutto() y=$product[1] format="%.2f"}
```

```
{*Cena brutto wszystkich sztuk produktu w promocji (na czerwono)*}
{math equation="x*y" x=$product[0]->getSpecialPriceBrutto() y=$product[1] format="%.2f"}
```

Funkcja `math` pozwala wyliczyć dowolne wyrażenie matematyczne, a także sformatować jego wynik. W powyższym przypadku wykonujemy działanie `x*y` (atrybut `equation`), gdzie wartość `x` to odpowiednia cena, a wartość `y` liczba sztuk danego produktu w koszyku. Całość jest formatowana tak, aby wynik był zaokrąglony z dokładnością do dwóch miejsc po przecinku.

Jeżeli produkt nie jest objęty promocją, wyświetlane jest jego podstawowa cena (rysunek 8.6).



Rysunek 8.6. *Wyświetlanie łącznej wartości cenowej produktu w koszyku. Produkt nie jest objęty promocją*

```
{else}
     {*Ceny produktu bez promocji *}

     {*Cena netto wszystkich sztuk produktu*}
     {math equation="x*y" x=$product[0]->getPriceNetto() y=$product[1] format="%.2f"}

     {*Cena brutto wszystkich sztuk produktu*}
     {math equation="x*y" x=$product[0]->getPriceBrutto() y=$product[1] format="%.2f"}

{/if}
{/foreach}
```

W ten oto sposób w szablonie TPL wyświetlana jest zawartość koszyka. Kompletny plik szablonu *basket.tpl* można znaleźć w pliku *sklepinternetowy/www/templates/public/basket.tpl*.