helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

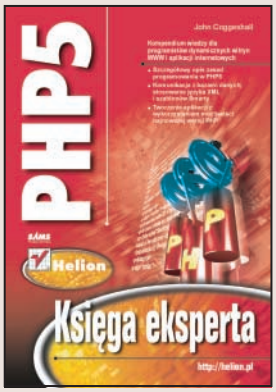
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

PHP5. Księga eksperta



Autor: John Coggeshall
Tłumaczenie: Paweł Gonera, Piotr Rajca
ISBN: 83-7361-889-9
Tytuł oryginału: PHP 5 Unleashed
Format: B5, stron: 744

Kompendium wiedzy dla programistów dynamicznych witryn WWW i aplikacji internetowych

- Szczegółowy opis zasad programowania w PHP5
- Komunikacja z bazami danych, stosowanie języka XML i szablonów Smarty
- Tworzenie aplikacji z wykorzystaniem możliwości najnowszej wersji PHP

PHP jest najpopularniejszym obecnie językiem skryptowym, wykorzystywanym do tworzenia dynamicznych witryn WWW i aplikacji internetowych. W połączeniu z bazą danych MySQL tworzy potężną platformę, której zalety docenili twórcy ponad 14 milionów witryn WWW. Stabilność, wydajność i szybkość działania, a przede wszystkim – nieodpłatny dostęp na zasadach licencji open-source, to cechy, które zadecydowały o ogromnej popularności tej technologii. Każda kolejna wersja języka PHP pozwala na tworzenie coraz bardziej zaawansowanych i rozbudowanych aplikacji. Najnowsza wersja – PHP5 to w pełni obiektowy język programowania umożliwiający stosowanie najnowszych technologii takich, jak XML i SOAP.

Książka „PHP5. Księga eksperta" zawiera opis wszystkich zagadnień związanych z tworzeniem aplikacji w języku PHP. Czytając ją poznasz zasady programowania w PHP5 zarówno te podstawowe jak i bardziej zaawansowane. Dowiesz się, w jaki sposób łączyć aplikację WWW z bazą danych i jak zapewnić jej bezpieczeństwo. Nauczysz się wykorzystywać mechanizmy sesji i cookies do zapamiętywania danych użytkowników i napiszesz skrypty generujące elementy graficzne i dokumenty przeznaczone do wydruku. Przeczytasz także o tworzeniu stron WWW przeznaczonych do wyświetlania na wyświetlaczach urządzeń mobilnych oraz o stosowaniu języka XML w aplikacjach.

- Podstawy programowania w PHP
- Stosowanie wyrażeń regularnych
- Obsługa formularzy
- Mechanizmy obsługi sesji i plików cookies
- Szablony Smarty i biblioteka PEAR
- Korzystanie z XML i XSLT
- Uwierzytelnianie użytkowników
- Komunikacja z bazami danych
- Operacje wejścia/wyjścia
- Generowanie grafiki i dokumentów PDF

Po przeczytaniu tej książki zostaniem prawdziwym ekspertem w dziedzinie tworzenia aplikacji internetowych w PHP5.

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

Spis treści

Autor prowadzący **17**

Wprowadzenie **19**

Część I Zastosowanie PHP do ogólnych prac związanych z tworzeniem aplikacji internetowych **21**

Rozdział 1. Podstawy zastosowania PHP **23**

Sposób działania skryptów PHP 23
Podstawowa składnia PHP 24
Proste typy danych PHP 26
Stosowanie zmiennych 28
Struktury sterujące 31
Logiczne struktury sterujące 31
Powtarzające struktury sterujące 37
Osadzanie kodu HTML w skryptach 39
Definiowanie funkcji 40
Zmienne i funkcje dynamiczne 44
Zmienne dynamiczne 44
Dynamiczne funkcje 45
Skrypty zapisane w wielu plikach 46
Referencje 48
Referencje zmiennych 48
Referencje używane w funkcjach 50
Łańcuchy znaków w PHP 52
Szybkość i wydajność wyrażeń łańcuchowych 52
Porównywanie łańcuchów znaków 52
Zaawansowane porównywanie łańcuchów znaków 54
Porównywanie fraz 55
Wyszukiwanie i zastępowanie 56
Zastępowanie łańcuchów znaków 58
Formatowanie łańcuchów znaków 59
Alternatywy dla funkcji printf() 61
Łańcuchy znaków i ustawienia lokalne 62
Formatowanie wartości walutowych 63
Formatowanie dat i godzin 66
Podsumowanie 67

Rozdział 2. Tablice 69
Proste tablice 69
Składnia tablic 70
Posługiwanie się tablicami 73
Implementacja tablic 78
Zastosowanie tablic jako list 78
Zastosowanie tablic jako dających się sortować tabel 79
Tworzenie tabel przeglądowych przy użyciu tablic 82
Konwersja łańcucha znaków na tablicę i na odwrót 85
Więcej informacji o tablicach 87

Rozdział 3. Wyrażenia regularne 89
Podstawy wyrażeń regularnych 89
Ograniczenia podstawowej składni 91
Wyrażenia regularne standardu POSIX 94
Wyrażenia regularne zgodne z wyrażeniami języka Perl (PCRE) 97
Wzorce nazwane 102
Modyfikatory PCRE 103
Kilka uwag na zakończenie 105

Rozdział 4. Obsługa formularzy 107
Kompendium wiedzy o formularzach HTML 107
Sposoby tworzenia formularzy HTML 108
Elementy formularzy 108
Obsługa formularzy w PHP 114
Odczytywanie zawartości formularza 114
Zastosowanie tablic w nazwach pól formularzy 117
Obsługa przesyłania plików na serwer 118
Podsumowanie 120

Rozdział 5. Zaawansowane techniki obsługi formularzy 121
Operacje na danych oraz ich konwersja 121
Magiczne cudzysłowy 121
Konwersja i kodowanie danych 123
Serializacja 125
Integralność danych formularzy 127
Zabezpieczanie wartości pól ukrytych 128
Funkcja protect() 129
Funkcja validate() 130
Praktyczne zastosowanie funkcji protect() i validate() 133
Przetwarzanie formularzy 135
Proste metody przetwarzania i weryfikacji formularzy 135
Mechanizm weryfikacji formularzy ogólnego przeznaczenia 136
Oddzielenie prezentacji danych od ich weryfikacji 144
Podsumowanie 145

Rozdział 6. Zapewnianie trwałości danych przy użyciu sesji i cookies 147
Cookies 147
Cechy cookies oraz ich ograniczenia 148
Sposoby implementacji cookies 149
Stosowanie cookies w skryptach PHP 151
Sesje PHP 154
Podstawowe sposoby stosowania sesji 154
Propagacja sesji 158
Zaawansowane sposoby obsługi sesji 160
Niestandardowa obsługa sesji 160
Dostosowywanie obsługi sesji do własnych potrzeb 161
Podsumowanie 162

Rozdział 7. Stosowanie szablonów 163
Czym są szablony i dlaczego warto je stosować? 163
Separacja najczęściej używanych elementów 163
Prosty przykład systemu szablonów 165
Mechanizm obsługi szablonów Smarty 171
Instalacja pakietu Smarty 172
Podstawy działania mechanizmu Smarty: zmienne i modyfikatory 174
Pliki i funkcje konfiguracyjne 179
Podsumowanie 189

Część II Zaawansowane programowanie dla WWW 191

Rozdział 8. PEAR 193
Co to jest PEAR? 193
Biblioteka kodu 194
Standard kodowania 194
System dystrybucji i utrzymania 194
Podstawowe klasy PHP 194
PEAR Package Manager 195
Różnorodna społeczność 195
Pobieranie i instalowanie PEAR 195
Systemy *NIX 196
Systemy Windows 196
Przeglądarka WWW 197
Użycie PEAR Package Manager 197
Wyświetlanie listy pakietów 197
Wyszukiwanie pakietów 198
Instalowanie i aktualizacja pakietów 199
Usuwanie pakietów 200
Alternatywne metody instalacji 200
Użycie strony WWW PEAR 201
Przeglądanie listy pakietów 202
Wyszukiwanie pakietu 202
Pobieranie i instalowanie pakietów 202

Zastosowanie pakietów PEAR w aplikacjach ... 203
Konfiguracja php.ini ... 204
Dołączanie pakietów ... 204
Stosowanie pakietów nieinstalowanych poprzez pear ... 204
Podsumowanie ... 206
Informator ... 206
Listy wysyłkowe i dyskusyjne ... 207
WWW ... 207
Pozostałe ... 207

Rozdział 9. XSLT oraz inne mechanizmy związane z XML ... 209

Związki XML z HTML ... 210
Zastosowanie XSLT do opisywania danych wyjściowych HTML na podstawie wejściowych danych XML ... 210
Arkusze stylu XSL ... 211
Podstawy formatu plików XSLT ... 211
Często wykorzystywane instrukcje XSLT ... 212
Użycie elementów instrukcji XSLT z wzorcami XSLT ... 214
Przykładowa transformacja XML na HTML z zastosowaniem XSLT ... 215
Wykorzystanie XSLT w PHP za pomocą modułu DOM XML ... 220
Przykładowe transformacje przy użyciu PHP 4 oraz DOM XML ... 220
Funkcje i właściwości DOM XML przydatne dla użytkowników XSLT ... 221
Dołączanie do PHP 4 obsługi XSLT za pomocą DOM XML ... 222
Wykorzystanie XSLT w PHP za pomocą modułu XSLT ... 223
Przykładowa transformacja z wykorzystaniem PHP 4 i XSLT ... 223
Funkcje oraz właściwości XSLT ... 224
Dołączanie do PHP 4 obsługi XSLT za pomocą XSLT ... 225
PHP 5 i XSLT ... 225
Przykładowa transformacja w PHP 5 ... 225
Funkcje i właściwości PHP 5 przydatne dla użytkowników XSLT ... 227
Dołączanie obsługi XSL w PHP 5 ... 228
Korzystanie z danych XML z użyciem SimpleXML ... 228
Użycie SimpleXML w skryptach PHP ... 229
Dodatkowe informacje na temat wykorzystania SimpleXML w skryptach PHP ... 230
Generowanie dokumentów XML przy użyciu PHP ... 230
Funkcje i właściwości służące do zapisywania obiektów XML w plikach ... 231
Podsumowanie ... 231
Odnośniki ... 232

Rozdział 10. Debugowanie i optymalizacja ... 233

Debugowanie skryptów PHP ... 233
Błędy składniowe ... 234
Błędy logiczne ... 234
Optymalizacja skryptów PHP ... 240
Sekret dobrej optymalizacji — profilowanie ... 240
Najczęstsze wąskie gardła w PHP i ich usuwanie ... 242
Podsumowanie ... 250

Rozdział 11. Uwierzytelnianie użytkowników 251
Uwierzytelnianie użytkowników w PHP 251
Dlaczego? 252
Wykorzystanie autoryzacji HTTP w Apache 252
Zastosowanie uwierzytelniania HTTP 255
Wykorzystanie sesji PHP 263
Zabezpieczanie kodu PHP 270
Register_Globals 270
Maksymalny poziom raportowania błędów 271
Nie ufaj nikomu 272
Wyświetlanie danych użytkownika 273
Operacje na plikach 273
Operacje na bazie danych 274
Podsumowanie 275

Rozdział 12. Szyfrowanie danych 277
Tajne hasło kontra klucz publiczny 277
Algorytmy tajnego hasła 278
Podstawianie frazy 278
Podstawianie znaków 279
Rozszerzanie algorytmu 279
Silniejsze algorytmy szyfrowania 280
Kryptografia klucza publicznego 281
Algorytm RSA 282
Podpisywanie a zabezpieczanie 284
Problem pośrednika 285
Zastosowanie kluczy publicznych w PHP 286
Strumienie SSL 287
Generowanie certyfikatu klucza publicznego i klucza prywatnego 287
Szyfrowanie i odszyfrowywanie danych 288
Szyfrowanie i wysyłanie bezpiecznych przesyłek e-mail z użyciem S/MIME 289
Podsumowanie 290

Rozdział 13. Programowanie obiektowe w PHP 291
Dlaczego obiekty? 291
Tworzenie prostych klas 291
Modyfikatory private, protected i public 292
Konstruktory i destruktory 297
Stałe klasowe 298
Metody statyczne 298
Dziedziczenie klas 299
Zaawansowane klasy 301
Klasy i metody abstrakcyjne 301
Interfejsy 302
Klasy i metody finalne 304
Metody specjalne 305
Metody odczytujące i zapisujące 305

Metoda __call() ... 306
Metoda __toString() ... 307
Automatyczne ładowanie klas ... 307
Serializacja obiektów ... 308
Wyjątki ... 309
Przedstawiamy stos wywołań ... 310
Klasa Exception ... 310
Zgłaszanie i przechwytywanie wyjątków ... 311
Iteratory ... 314
Podsumowanie ... 316

Rozdział 14. Obsługa błędów ... 317

Model obsługi błędów w PHP ... 317
Typy błędów ... 317
Co robić w przypadku błędu ... 319
Domyślna obsługa błędów ... 319
Zapobieganie powstawaniu błędów ... 323
Własny moduł obsługi błędów ... 323
Powodowanie błędów ... 326
Łączymy wszystko ... 326
Podsumowanie ... 329

Rozdział 15. Operacje na HTML i XHTML z użyciem Tidy ... 331

Wstęp ... 331
Podstawowe zastosowania Tidy ... 331
Analizowanie danych wejściowych i odczytywanie danych wyjściowych ... 331
Naprawa i czyszczenie dokumentów ... 333
Identyfikowanie problemów w dokumentach ... 334
Opcje konfiguracji Tidy ... 334
Zmiana opcji Tidy w czasie działania skryptu ... 335
Pliki konfiguracyjne Tidy ... 336
Użycie parsera Tidy ... 337
Sposób przechowywania dokumentów przez Tidy ... 337
Węzeł Tidy ... 338
Zastosowania Tidy ... 341
Tidy jako bufor danych wyjściowych ... 341
Konwersja dokumentów na CSS ... 341
Zmniejszenie wykorzystania pasma ... 342
Upiększanie dokumentów ... 343
Wycinanie adresów URL z dokumentów ... 343
Podsumowanie ... 344

Rozdział 16. Tworzenie poczty elektronicznej w PHP ... 345

Protokół MIME ... 345
Tworzenie poczty elektronicznej MIME w PHP ... 350
Klasy MIMEContainer oraz MIMESubcontainer ... 352
Klasy MIMEAttachment, MIMEContent oraz MIMEMessage ... 355
Podsumowanie ... 359

Część III Tworzenie aplikacji w PHP 361

Rozdział 17. Zastosowanie PHP w skryptach konsoli 363

Podstawowe różnice między standardowym PHP a wersją CLI 363
Wykorzystanie PHP w wersji CLI 365
Argumenty wiersza poleceń oraz zwracane wartości 366
Narzędzia i rozszerzenia wersji CLI 367
Rozszerzenie Readline 367
Tworzenie interfejsu użytkownika 371
Podsumowanie 376

Rozdział 18. SOAP i PHP 379

Czym są usługi sieciowe? 379
Transport z użyciem SOAP 380
Opis za pomocą WSDL 381
Przeszukiwanie katalogu za pomocą UDDI 383
Instalacja 384
Tworzenie usług sieciowych 385
Korzystanie z usług sieciowych 386
Wyszukiwanie usług sieciowych 388
Podsumowanie 390

Rozdział 19. Tworzenie witryn dla WAP 391

Czym jest WAP? 391
Wymagania systemowe 392
Nokia Mobile Internet Toolkit 392
Ericsson 393
Openwave SDK 393
Motorola Wireless IDE/SDK 394
Wprowadzenie do WML 395
Struktura WML 396
Tekst 397
Łącza 399
Grafika 401
Formularze WML 403
Udostępnianie treści WAP 412
Typy MIME 412
Konfiguracja serwera WWW 412
Ustawianie typu MIME z PHP 414
Rozpoznawanie klienta 414
Wyświetlanie grafiki 416
Przykładowe zastosowania 416
Przetwarzanie danych formularza w serwerze 417
Kinowy system rezerwacji biletów 418
Podsumowanie 425

Część IV Wejście-wyjście, wywołania systemowe i PHP 427

Rozdział 20. Operacje na systemie plików 429

Operacje na systemie plików w PHP 429
Odczyt i zapis plików tekstowych 431
Odczyt i zapis plików binarnych 436
Operacje na katalogach 440
Prawa dostępu do plików 442
Działanie praw dostępu do plików w systemie Unix 443
Określanie praw dostępu w skryptach PHP 446
Funkcje wspomagające operacje na plikach 448
Funkcje logiczne 448
Operacje na plikach 449
Specjalizowane operacje dostępu do plików 451
Podsumowanie 454

Rozdział 21. Sieciowe operacje wejścia-wyjścia 455

Przeszukania i wsteczne przeszukania DNS 455
Pobieranie rekordu DNS na podstawie adresu IP 455
Pobieranie adresu IP na podstawie nazwy komputera 456
Pobieranie informacji z rekordów DNS 457
Programowanie gniazd 460
Podstawy stosowania gniazd 460
Tworzenie nowego gniazda 461
Obsługa błędów w przypadku stosowania gniazd 462
Tworzenie gniazd klienta 463
Tworzenie gniazd serwera 464
Jednoczesne korzystanie z wielu gniazd 466
Funkcje pomocnicze związane z operacjami sieciowymi 468
Podsumowanie 470

Rozdział 22. Korzystanie z możliwości systemu operacyjnego w skryptach PHP 471

Wprowadzenie 471
Możliwości funkcjonalne charakterystyczne dla systemów Unix 471
Bezpośrednie wejście-wyjście 472
Funkcje standardu POSIX 478
Kontrola procesów w systemie Unix 485
Funkcje niezależne od używanego systemu operacyjnego 493
Wykonywanie aplikacji z poziomu PHP 493
Podstawowe sposoby wykonywania aplikacji 493
Jednokierunkowe potoki poleceń zewnętrznych 495
Operacje związane ze środowiskiem systemowym 496
Krótkie informacje na temat bezpieczeństwa 496
Podsumowanie 498

Część V Stosowanie baz danych w PHP 499

Rozdział 23. Wprowadzenie do zagadnień baz danych 501

Korzystanie z klienta MySQL 501
Podstawy zastosowania MySQL 502
Podstawowe informacje o systemach RDBMS 502
Wykonywanie zapytań przy użyciu języka SQL 503
Podsumowanie 516

Rozdział 24. Stosowanie bazy danych MySQL w skryptach PHP 517

Wykonywanie zapytań w skryptach PHP 519
Podstawy MySQLi 519
Wykonywanie wielu zapytań 525
Tworzenie systemu śledzenia odwiedzin 527
Polecenia przygotowane 534
Transakcje 540
Zastosowanie funkcji MySQLi do obsługi sesji 542
Czym są niestandardowe procedury obsługi sesji? 542
Definiowanie własnej procedury obsługi sesji 542
Procedura obsługi sesji wykorzystująca funkcje MySQLi 544
Podsumowanie 552

Rozdział 25. Stosowanie bazy danych SQLite w skryptach PHP 553

Co sprawia, że baza SQLite jest tak unikalna? 553
Ogólne różnice pomiędzy SQLite oraz MySQL 554
W jaki sposób SQLite obsługuje tekstowe i liczbowe typy danych? 555
Sposób obsługi wartości NULL w SQLite 556
Dostęp do bazy danych z wielu procesów 556
Podstawowe możliwości bazy SQLite 557
Otwieranie i zamykanie baz danych 558
Wykonywanie poleceń SQL 559
Pobieranie wyników 561
Obsługa błędów 565
Poruszanie się po zbiorze wyników 566
Stosowanie funkcji definiowanych przez użytkownika 568
Wywoływanie funkcji PHP w poleceniach SQL 572
Pozostałe zagadnienia 573
Podsumowanie 574

Rozdział 26. Funkcje dba 575

Przygotowania i ustawienia 575
Tworzenie bazy danych działającej w oparciu o pliki 577
Zapis danych 578
Odczyt danych 580
Przykładowa aplikacja 582
Podsumowanie 586

Część VI Tworzenie grafiki w PHP 587

Rozdział 27. Operacje na rysunkach 589

Podstawy tworzenia rysunków z użyciem GD 589
Odczytywanie informacji o rysunku 591
Użycie funkcji rysunkowych PHP i GD 593
Rysowanie figur składających się z linii 593
Rysowanie linii zakrzywionych 595
Rysowanie wypełnionych figur oraz funkcje rysunków 597
Operacje na kolorach i pędzlach 602
Operacje na palecie kolorów 602
Rysowanie z wykorzystaniem pędzli 607
Zastosowanie czcionek i drukowanie ciągów 613
Zastosowanie wbudowanych czcionek GD 614
Użycie czcionek TrueType 616
Użycie czcionek Postscript Type 1 619
Manipulacje obrazem 623
Kopiowanie jednego rysunku na drugi 623
Inne funkcje graficzne 627
Funkcje EXIF 629
Podsumowanie 630

Rozdział 28. Generowanie dokumentów drukowanych 631

Uwaga dotycząca przykładów w tym rozdziale 632
Dynamiczne generowanie dokumentów RTF 632
Dynamiczne generowanie dokumentów PDF 636
System współrzędnych PDFLib 636
Użycie parametrów konfiguracyjnych PDFLib 637
Generowanie dokumentów PDF od podstaw 637
Zasoby dodatkowe 653

Dodatki 655

Dodatek A Instalowanie PHP 5 oraz MySQL 657

Instalowanie PHP 5 657
Linux 658
Windows 660
Mac OS X 663
Instalowanie MySQL oraz modułów PHP 664
Linux 664
Windows 668
Instalowanie PEAR 670

Dodatek B Protokół HTTP — kompendium 673

Czym jest HTTP? 673
Biblioteki PHP przeznaczone do pracy z HTTP 673
Poznajemy transakcje HTTP 674
Metody klienckie HTTP 676

To, co wraca: kody odpowiedzi serwera ... 677
Nagłówki HTTP ... 678
Kodowanie ... 679
Identyfikacja klientów i serwerów ... 679
Nagłówek Referer ... 680
Pobieranie danych ze źródła HTTP ... 681
Typy mediów ... 681
Cookies — zapamiętywanie stanu ... 682
Bezpieczeństwo i autoryzacja ... 684
Buforowanie treści HTTP po stronie klienta ... 685

Dodatek C Migracja aplikacji z PHP 4 do PHP 5 ... 687
Konfiguracja ... 687
Programowanie obiektowe ... 689
Nowe działanie funkcji ... 690
Propozycje dalszych lektur ... 690

Dodatek D Dobre techniki programowania i zagadnienia wydajności ... 693
Częste błędy stylu ... 693
Dyrektywy konfiguracji ... 693
PHP wiele wybacza, również błędy ... 694
Ponowne wynajdywanie koła ... 695
Zmienne — używaj ich, ale nie nadużywaj ... 696
Najczęstsze problemy z bezpieczeństwem ... 698
Nieoczekiwane konsekwencje ... 698
Wywołania systemowe ... 700
Zabezpieczanie przed atakami na wywołania systemowe ... 702
Zabezpieczanie przesyłania plików ... 703
Styl i bezpieczeństwo — rejestracja ... 704
Rejestrowanie własnych komunikatów błędów ... 705
Podsumowanie ... 706

Dodatek E Zasoby i listy dyskusyjne ... 707
Ważne witryny ... 707
Listy wysyłkowe i dyskusyjne ... 708

Skorowidz ... 711

Rozdział 7.
Stosowanie szablonów

Gdy PHP w coraz to większym stopniu stawał się centralnym komponentem witryn internetowych, rosło także znaczenie odpowiedniego zarządzania jego kodem. Zagadnienie to jest szczególnie istotne w sytuacjach, gdy nad witryną jednocześnie pracuje większa ilość osób. Jednym z najlepszych sposobów ułatwienia zarządzania witryną jest oddzielenie kodu HTML stron od kodu PHP, który obsługuje te strony. Rozwiązanie takie nosi nazwę separacji *prezentacji* oraz *logiki aplikacji*. W niniejszym rozdziale przedstawię niektóre spośród najczęściej stosowanych metod takiej separacji, w tym także pakiet obsługi szablonów o nazwie Smarty.

Czym są szablony i dlaczego warto je stosować?

W aplikacjach PHP najczęściej stosowanym sposobem separacji prezentacji od logiki aplikacji jest zastosowanie szablonów. Ogólnie rzecz biorąc, szablon jest dokumentem HTML zawierającym specjalne znaczniki i (lub) struktury sterujące. W rzeczywistości w początkowym okresie swojego rozwoju PHP został zaprojektowany jako prosty język makr działający podobnie do mechanizmów obsługi szablonów.

Separacja najczęściej używanych elementów

Kiedy język PHP zyskał nieco na popularności, ze względu na swoją prostotę i łatwość bardzo szybko zaczął być stosowany przez programistów aplikacji internetowych. Ta właśnie dzięki tej łatwości stosowania PHP jest jednym z najlepszych języków do szybkiego tworzenia aplikacji oraz ich prototypów. Niestety, te same cechy, dzięki którym PHP jest doskonałym językiem do tworzenia prototypów rozwiązań, sprawiają jednocześnie, iż łatwo można w nim tworzyć kod bardzo trudny do zarządzania. Programiści zdają sobie sprawę z tego, iż wraz z powiększaniem się witryn rośnie także znaczenie zwiększenia ich modularyzacji. Najczęściej stosowanym rozwiązaniem tego zagadnienia jest wydzielanie wspólnych elementów witryny, których następnie można używać w jej dowolnych miejscach (przy użyciu instrukcji `include`). Na przykład niemal wszystkie witryny WWW można podzielić na trzy elementy: nagłówek, stopkę oraz treść. Listing 7.1 pokazuje, w jaki sposób można podzielić standardową stronę WWW na wymienione wcześniej trzy segmenty.

Listing 7.1. *Kod HTML typowej strony WWW*

```
<?php
    /* segments.php */

    function display_head($title="Typowa strona WWW") {
    ?>
    <HTML>
    <HEAD><TITLE><?=$title?></TITLE></HEAD>
    <BODY>
    <TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
    <TR>
    <TD>
      <TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
      <TR><TD><A HREF="products.php">Produkty</A></TD></TR>
      <TR><TD><A HREF="contact.php">Kontakt</A></TD></TR>
      <TR><TD><A HREF="about.php">O nas</A></TD></TR>
      </TABLE>
    </TD>
    <TD>
    <?php } // koniec funkcji display_head()

    function display_foot() {
    ?>
    </TD>
    </TR>
    </TABLE>
    </BODY>
    </HTML>
    <?php } // koniec funkcji display_foot()
?>

<?php
    /* index.php */

    include('segments.php');
    display_head();
?>

Welcome to my web site.

<?php display_foot(); ?>
```

Nawet pobieżne przejrzenie listingu 7.1 uświadamia, iż takie rozwiązanie wykazuje znaczące zalety w porównaniu z klasycznym sposobem konstrukcji stron WWW. Wydzielenie wspólnych elementów — w tym przypadku nagłówka i stopki strony — i umieszczenie ich w niezależnych funkcjach może w ogromnym stopniu uprościć zarządzanie całą witryną. Dzięki takiemu rozwiązaniu wykonanie trywialnej operacji, takiej jak dodanie nowego hiperłącza do menu, wymaga wprowadzenia modyfikacji tylko w jednym pliku, niemniej jednak zmiany te będą widoczne w całej witrynie. Niemal we wszystkich mniejszych witrynach, nad którymi pracuje jedna lub dwie osoby (i wszystkie znają się na PHP), rozwiązanie takie doskonale zdaje egzamin.

Niestety na żadnej witrynie tworzonej przez dwa zespoły, z których jeden zajmuje się układem i prezentacją stron, a drugi skryptami PHP, takie rozwiązanie nie okaże się szczególnie przydatne. Choć pozwala ono na zmniejszenie ilości powtarzającego się kodu stosowanego w witrynie, to jednak i tak wymaga umieszczania kodu PHP bezpośrednio w dokumentach HTML.

Prosty przykład systemu szablonów

W sytuacjach, które faktycznie wymagają oddzielenia prezentacji od logiki aplikacji, konieczne jest wykorzystanie prawdziwego systemu obsługi szablonów. Choć w dalszej części rozdziału zaprezentuję profesjonalny system do obsługi szablonów o nazwie Smarty, to jednak informacje te nie na wiele by się przydałby, gdyby Czytelnik nie znał podstawowych idei działania rozwiązań tego typu. Aby przedstawić te zasady, stworzę własny, bardzo prosty system obsługi szablonów o nazwie QuickTemplate. Analizując ten przykład, Czytelnik nie tylko będzie mógł poznać podstawowe idee związane z systemami obsługi szablonów oraz sposoby ich działania, lecz także nauczyć się nieco o poprawnych metodach tworzenia złożonych skryptów PHP.

Jednak zanim przedstawię i przeanalizuję sam skrypt QuickTemplate (w zasadzie jest to klasa), chciałbym opisać cele, do których dążymy. Aby całkowicie oddzielić kod HTML od kodu PHP, musimy w jakiś sposób zaznaczyć wybrane miejsca dokumentu, w których zostanie umieszczona zawartość wygenerowana przez kod PHP. W przypadku naszej przykładowej klasy `QuickTemplate` znaczniki szablonu mają postać łańcuchów znaków składających się z wielkich liter (A – Z) umieszczonych pomiędzy znakami procentu (`%`). Na przykład stosunkowo łatwo można by zdefiniować szablon (listing 7.2) podobny do dokumentu przedstawionego na listingu 7.1.

Listing 7.2. *Plik szablonu QuickTemplate*

```
<HTML>
<HEAD><TITLE>%TITLE%</TITLE></HEAD>
<BODY>
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
<TR>
<TD>%LEFTNAV%</TD>
<TD>%CONTENT%</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Jak widać, w kodzie HTML przedstawionym na listingu 7.2 nie ma żadnego kodu PHP. Bez najmniejszych problemów można go tworzyć i modyfikować przy wykorzystaniu edytorów HTML działających w trybie WYSIWYG (ang. *what you see is what you get* — dostajesz to, co widzisz), a mimo to zapewnia on ten sam, wysoki poziom kontroli nad dynamiczną zawartością strony, którym dysponowaliśmy w rozwiązaniu przedstawionym na listingu 7.1. W tym przypadku wydzieliłem tabelę nawigacyjną i umieściłem ją w osobnym pliku, przedstawionym na listingu 7.3.

Listing 7.3. *Kod HTML definiujący połączenia nawigacyjne*

```
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
      <TR><TD><A HREF="products.php">Produkty</A></TD></TR>
      <TR><TD><A HREF="contact.php">Kontakty</A></TD></TR>
      <TR><TD><A HREF="about.php">O nas</A></TD></TR>
</TABLE>
```

W praktyce kody przedstawione na obu ostatnich listingach powinny zostać zapisane w osobnych plikach (zakładam, że te fragmenty kodu zostały zapisane odpowiednio w plikach *index.html* oraz *links.html*; w dalszej części rozdziału Czytelnik zrozumie, dlaczego poczyniłem takie założenie). Teraz, kiedy już zdefiniowaliśmy nasze szablony, możemy ich użyć i przetworzyć je przy użyciu klasy `QuickTemplate`.

Podobnie jak wszystkie systemy obsługi szablonów, które przedstawię w niniejszej książce, także i klasa `QuickTemplate` działa w oparciu tablice o złożonej strukturze. Poniższy przykład przedstawia zawartość typowej tablicy, używanej przez klasę `QuickTemplate` do obsługi szablonów z listingów 7.2 oraz 7.3.

Listing 7.4. *Typowa tablica używana przez klasę QuickTemplate*

```
<?php

$temp_data = array('main' => array('file' =>
                                   'index.thtml'),
                   'leftnav' => array('file' =>
                                   'link.html'),
                   'content' => array('content' =>
                                   'To jest zawartość: %DYNAMIC%'),
                   'title' => array('content' =>
                                   'Typowa witryna używająca szablonów'),
                   'dynamic' => array('content' =>
                                   'To jest dalsza zawartość witryny')
                   );
?>
```

Jak widać, ta wielowymiarowa tablica asocjacyjna zawiera na głównym poziomie grupę elementów, których klucze odpowiadają znacznikom użytym w szablonie przedstawionym na listingu 7.2 (jedynym wyjątkiem jest tu element o kluczu `main`). Z kluczami tymi są skojarzone tablice zawierające jeden element. Klucz tego elementu (`file` lub `content`) jest skojarzony z wartością (nazwą pliku lub łańcuchem znaków) reprezentującą dane, którymi należy zastąpić znacznik umieszczony w szablonie. Na przykład w szablonie z listingu 7.2 zdefiniowałem znacznik o nazwie `%CONTENT%`. Zostanie on zastąpiony wartością pobraną z tablicy `$temp_data`, z elementu o kluczu `content`. Wartością tego elementu jest tablica zawierająca jeden element o kluczu `content`, dlatego też znacznik umieszczony w szablonie zostanie zastąpiony łańcuchem znaków `'To jest zawartość: %DYNAMIC%'`. Niemniej jednak przed zastąpieniem znacznika `%CONTENT%` także zastępujący go łańcuch znaków zostanie przetworzony. Poniżej zamieściłem opis wykonywanych czynności:

1. Znacznik `%CONTENT%` jest zastępowany wartością skojarzoną z kluczem `content`.
2. Znacznik `%DYNAMIC%` umieszczony w wartości skojarzonej z kluczem `content` jest zastępowany wartością skojarzoną z kluczem `dynamic`.

W rezultacie po zakończeniu całego procesu każde wystąpienie znacznika `%CONTENT%` zostanie zastąpione łańcuchem znaków: `"To jest zawartość: jest dalsza zawartość witryny"`.

Ten sam proces jest powtarzany dla każdego znacznika umieszczonego w szablonie, aż do chwili, gdy nie będzie w nim już żadnych znaczników, które należy zastąpić. Jeśli z jakichkolwiek powodów w szablonie znajdą się znaczniki, które nie będą istnieć w tablicy używanej przez klasę `QuickTemplate` (listing 7.4), to zostaną one zastąpione komentarzami HTML zawierającymi stosowne komunikaty.

Teraz, po przedstawieniu podstawowych zasad działania systemu obsługi szablonów QuickTemplate, mogę przedstawić jego kod. W zależności od tego, w jakim stopniu Czytelnik zrozumiał podane wcześniej wyjaśnienia dotyczące sposobu przetwarzania znaczników szablonu, może on uznać, że kod całego rozwiązania nie jest lub jest dla niego zbyt złożony i trudny. Niemniej jednak wszystkich zachęcam do dalszej lektury — w praktyce okazuje się, że cała klasa składa się jedynie z 40 wierszy kodu!

Cały kod klasy został przedstawiony na listingu 7.5.

Listing 7.5. *Klasa QuickTemplate*

```
<?php
    class QuickTemplate {

        private $t_def;

        public function parse_template($subset = 'main') {

            $noparse = false;
            $content = "";
            $temp_file = $this->t_def[$subset]['file'];

            if(isset($temp_file)) {

                if(strlen($temp_file) > 6) {
                    $ext = substr($temp_file, strlen($temp_file)-6);
                }

                if(strcasecmp($ext, ".thtml") != 0) {
                    $noparse = true;
                }

                $fr = fopen( $temp_file, "r" );

                if(!$fr) {
                    $content = "<!-- Błąd podczas wczytywania pliku '$temp_file' //-->";
                } else {
                    $content = fread($fr, filesize($temp_file));
                }

                @fclose($fr);

            } else {

                if(isset($this->t_def[$subset]['content'])) {
                    $content = $this->t_def[$subset]['content'];
                } else {
                    $content = "<!-- Zawartość znacznika '$subset' nie została
                    zdefiniowana //-->";
                }

            }

            if(!$noparse) {

                $content=preg_replace("/\%([A-Z]*)\%/e",
```

```
                "QuickTemplate::parse_template(strtolower('$1'))",
                $content);
            }

            return $content;

        }

        function __construct($temp='') {

            if(is_array($temp)) $this->t_def = $temp;

        }
    }
?>
```

Jak widać, oprócz trywialnego konstruktora klasa przedstawiona na listingu 7.5 definiuje tylko jedną funkcję: `parse_template()`. Zacznę zatem od przedstawienia sposobu jej działania.

Działanie naszego mechanizmu przetwarzania szablonów opiera się na zastosowaniu rekurencji (podejrzewam, że w taki sam sposób działa większość mechanizmów obsługi szablonów). To właśnie dzięki rekurencji tak łatwo i szybko można zastępować znaczniki umieszczone wewnątrz wartości zastępujących inne znaczniki.



Czy Czytelnik jest pewny, że wie, co to jest rekurencja? Otóż najprościej rzecz ujmując, funkcja *rekurencyjna* to funkcja, która wywołuje samą siebie. Proces ten został zilustrowany na poniższym przykładzie, prezentującym funkcję wyznaczającą największy wspólny podzielnik dwóch liczb:

```
<?php
    function gcd($a, $b) {
      return ($b > 0) ? gcd($b, $a % $b) : $a;
    }
```

To tylko jeden (bardzo dobry) z wielu przykładów pokazujących, jak bardzo użyteczne i potężne mogą być funkcje rekurencyjne.

Jak pokazuje definicja funkcji `parse_template()`, w jej wywołaniu można podać jeden, opcjonalny parametr `$subset`, którego domyślną wartością jest `'main'`. Parametr ten nigdy jednak nie powinien być używany przez samego programistę korzystającego z klasy `QuickTemplate`. Służy on do określenia aktualnie przetwarzanego klucza tablicy. Ponieważ mechanizm przetwarzający szablony musi rozpocząć działanie od jakiegoś miejsca, to właśnie klucz `'main'` został wybrany przeze mnie jako punkt początkowy przetwarzania wszystkich szablonów.

Pierwszą czynnością wykonywaną na samym początku przetwarzania szablonu jest inicjalizacja trzech zmiennych: `$content`, `$noparse` oraz `$temp_file`. Pierwsza z nich — `$content` — będzie przechowywać kod HTML wygenerowany przez mechanizm przetwarzający dla aktualnie przetwarzanego fragmentu szablonu. Zmienna `$noparse` zawiera wartość logiczną, określającą, czy wartość aktualnie przetwarzanego znacznika szablonu powinna być dodatkowo przetwarzana. Dzięki temu możemy używać zarówno plików HTML zawierających znaczniki (które należy dalej przetwarzać), jak również plików

zawierających wyłącznie kod HTML (które nie wymagają dalszego przetwarzania). Choć nasz mechanizm obsługi szablonów byłby znacznie łatwiejszy, gdybyśmy nie zaprzątali sobie głowy taką możliwością, to jednak zastosowaliśmy ją ze względu na wydajność, aby nie dopuszczać do niepotrzebnego przetwarzania plików. Ostatnia ze zmiennych — `$temp_file` — jest kluczem `file` aktualnie przetwarzanego podzbioru. Jeśli klucz ten będzie dostępny, to jego wartość powinna określać nazwę pliku, który należy przetworzyć. Jeśli klucz `file` nie zostanie zdefiniowany, to funkcja spróbuje odszukać klucz `content`, a jeśli i to się nie uda — zgłosi błąd. Właśnie w tym celu w kolejnym wierszu kodu sprawdzamy, czy zmienna `$temp_file` została zdefiniowana (używając w tym celu funkcji `isset()`). Jeśli zmienna została zdefiniowana, to zawartość wskazanego pliku jest odczytywana i zapisywana w zmiennej `$content`. (W czasie wykonywania tych operacji wykorzystywane są funkcje PHP do obsługi systemu plików.) Z kolei jeśli zmienna `$temp_file` nie została zdefiniowana, funkcja sprawdza, czy jest dostępny klucz `content` zawierający łańcuch znaków, który ma zastąpić znacznik szablonu. Jeśli nie uda się odnaleźć tego klucza, funkcja generuje błąd.

Do tej pory opisałem wyłącznie operacje związane z inicjalizacją i obsługą błędów. Dopiero teraz przedstawię kluczowe operacje wykonywane przez funkcję `parse_template()`. Co ciekawe, ta „kluczowa operacja" sprowadza się do wywołania jednej funkcji: `preg_replace()` wykorzystującej modyfikator `/e`. Funkcja `preg_match()` została szczegółowo opisana w rozdziale 3. „Wyrażenia regularne", przypomnę jednak, iż jej działanie polega na dopasowaniu wyrażenia regularnego do łańcucha znaków i zastąpieniu dopasowanego fragmentu innym łańcuchem. W naszym przypadku zadaniem funkcji `preg_replace()` jest odszukanie wszystkich fragmentów łańcucha składających się z dowolnej ilości dużych liter umieszczonych pomiędzy znakami procentu (`%`) i rekurencyjne wywołanie funkcji `parse_template()`. Wartość zwrócona przez tę funkcję zastępuje następnie fragment łańcucha znaków dopasowany do wyrażenia regularnego.

Wyniki zwracane przez tę funkcję stanowią kluczowy element zapewniający działanie naszego mechanizmu obsługi szablonów. Zastosowanie funkcji `preg_match()` zapewnia, że każdy znacznik szablonu pasujący do wyrażenia regularnego zostanie odszukany i zastąpiony. Wynikowy kod uzyskany w efekcie tego zastąpienia jest zapisywany w zmiennej `$content`, która jest następnie zwracana bądź to do początkowego skryptu, który utworzył obiekt `QuickTemplate`, bądź też do rekurencyjnie wywołanej funkcji `parse_template()`.

To już cały kod klasy `QuickTemplate`! Jest on dosyć prosty, a jednocześnie działa całkowicie zgodnie z naszymi oczekiwaniami. W ramach podsumowania chciałbym przedstawić praktyczne zastosowanie klasy `QuickTemplate` (listing 7.6).

Listing 7.6. *Zastosowanie klasy QuickTemplate*

```
<?php

    include('quicktemplate.php');// Definicja klasy

    $temp_data = array('main' => array('file' =>
                                       'index.thtml'),
                       'leftnav' => array('file' =>
                                       'link.html'),
                       'content' => array('content' =>
                                       'To jest zawartość: %DYNAMIC%'),
```

```
                        'title' => array('content' =>
                                        'Typowa witryna używająca szablonów'),
                        'dynamic' => array('content' =>
                                         'To jest dalsza zawartość witryny')
                        );

    $engine = new QuickTemplate($temp_data);
    echo $engine->parse_template();
?>
```

Wykonanie powyższego skryptu, wykorzystującego szablony przedstawione na listingach 7.2 oraz 7.3, spowoduje wygenerowanie strony o następującym kodzie HTML:

```
<HTML>
<HEAD><TITLE>Typowa witryna używająca szablonów</TITLE></HEAD>
<BODY>
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
<TR>
<TD><TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
      <TR><TD><A HREF="products.php">Produkty</A></TD></TR>
      <TR><TD><A HREF="contact.php">Kontakty</A></TD></TR>
      <TR><TD><A HREF="about.php">O nas</A></TD></TR>
</TABLE>
</TD>
<TD>To jest zawartość: To jest dalsza zawartość witryny</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Mam nadzieję, że Czytelnik jest już w stanie docenić nakład pracy konieczny do stworzenia nawet prostego mechanizmu obsługi szablonów. Co gorsza, przedstawiona tu klasa `QuickTemplate` nie obsługuje wielu naprawdę przydatnych możliwości, na przykład struktur sterujących. Jednak w odróżnieniu od rozwiązania polegającego na dzieleniu kodu i zapisywaniu go w niezależnych plikach klasa `QuickTemplate` pozwala na całkowite oddzielenie kodu HTML związanego z prezentacją strony od obsługującego go kodu logiki aplikacji.

Oczywiście taka separacja kodu prezentacji i kodu logiki aplikacji ma swoją cenę. Czytelnik zapewne zdaje już sobie sprawę z tego, iż stosowanie własnego (lub dowolnego innego) mechanizmu obsługi szablonów sprawia, iż tworzone strony WWW stają się znacznie mniej intuicyjne. Dlatego też przed rozpoczęciem lektury kolejnej części rozdziału koniecznie należy zrozumieć, co robi klasa `QuickTemplate` (choć niekoniecznie trzeba w pełni rozumieć, *jak* to robi). Jeśli Czytelnik ma problemy ze zrozumieniem zasad stosowania klasy `QuickTemplate`, to zapewne nie jest jeszcze gotów do poznania systemów obsługi szablonów, takich jak Smarty, gdyż są one znacznie bardziej złożone i zaawansowane. Dlatego też, jeśli Czytelnik nie rozumie pewnych zagadnień przedstawionych we wcześniejszej części rozdziału, to przed rozpoczęciem dalszej lektury polecałbym ponowne przeanalizowanie informacji na temat `QuickTemplate`.

Mechanizm obsługi szablonów Smarty

Smarty jest niezwykle złożonym i dysponującym ogromnymi możliwościami mechanizmem obsługi szablonów, z którego mogą korzystać programiści PHP. Prawdopodobnie jest to najlepsze z rozwiązań stanowiących bazę do tworzenia aplikacji, z którymi się do tej pory spotkałem. Smarty w optymalny sposób równoważy oddzielenie prezentacji od logiki aplikacji bez niepotrzebnego ograniczania użyteczności. Choć Smarty działa w oparciu o swój własny język skryptowy używany podczas tworzenia szablonów, to jednak korzystanie z tych możliwości nie jest niezbędne.

Kiedy po raz pierwszy zetknąłem się z pakietem Smarty, pierwszy złożony przykład, który zobaczyłem, bardzo mnie zniechęcił do tego rozwiązania. Zastosowane w nim szablony stanowiły zupełnie niezależny program, zawierały własne struktury sterujące oraz wywołania własnych funkcji pakietu. Cały czas powracałem w myślach do podstawowego celu stosowania szablonów — wyeliminowania sytuacji, w których projektanci musieliby tworzyć kod PHP lub posługiwać się nim. Początkowo wydawało mi się, iż choć Smarty faktycznie umożliwia oddzielenie kodu PHP od kodu HTML, to jednak chcąc używać tego pakietu, projektanci muszą poznać zupełnie nowy język „skryptowy" używany w jego szablonach. W efekcie byłem bardzo rozczarowany i usunąłem pakiet ze swojego komputera.

Jakiś czas później prowadziłem badania związane z artykułem poświęconym separacji logiki aplikacji i logiki biznesowej, który właśnie pisałem, i ponownie natknąłem się na pakiet Smarty. Aby wyczerpująco przedstawić opisywane rozwiązania, zdecydowałem się także wspomnieć w artykule o tym pakiecie, co oczywiście zmusiło mnie do nieco dokładniejszego poznania jego możliwości i sposobów zastosowania. Kiedy dokładniej przejrzałem dokumentację pakietu i samodzielnie utworzyłem kilka szablonów, zacząłem zmieniać zdanie na jego temat. Choć faktycznie udostępniał on pewne niezwykle złożone możliwości i mechanizmy, to jednak pozwalał także na stosowanie zwyczajnego zastępowania zmiennych, które przedstawiłem w pierwszej części rozdziału, poświęconej klasie `QuickTemplate`. Smarty obsługiwał także podstawowy zbiór struktur sterujących, takich jak instrukcje warunkowe oraz pętle, które umożliwiały pełną separację prezentacji od logiki aplikacji. Wziąwszy pod uwagę wszystkie te możliwości, można by się spodziewać, że mechanizm Smarty działa wolno; jednak najbardziej zaskoczyło mnie to, iż w rzeczywistości był on *szybki* — szybszy od wszystkich innych mechanizmów obsługi szablonów stworzonych w PHP, z którymi się wcześniej zatknąłem! Wszystkie te czynniki sprawiły, że diametralnie zmieniłem zdanie o mechanizmie Smarty i obecnie jestem jego wielkim zwolennikiem.

Co sprawia, że Smarty jest tak wyjątkowym narzędziem? Otóż wykorzystuje on rozwiązanie, które (z tego co wiem) jest całkowicie unikalne wśród wszystkich innych mechanizmów obsługi szablonów przeznaczonych do użycia w aplikacjach PHP — kompiluje szablony do postaci normalnego kodu PHP. Tak więc za pierwszym razem, gdy szablon jest wczytywany, Smarty kompiluje go do postaci zwykłego skryptu PHP, zapisuje, a następnie wykonuje ten skrypt. Dzięki temu szablony działają niemal równie szybko jak zwyczajne skrypty PHP, a jednocześnie są niezwykle skalowalne. Co więcej, mechanizm Smarty został zaprojektowany w taki sposób, iż udostępniane przez niego struktury sterujące są zamieniane bezpośrednio na kod PHP, dzięki czemu zyskują jego elastyczność i możliwości, a jednocześnie ukrywają wszystkie związane z nim złożoności.

Instalacja pakietu Smarty

Aby móc używać pakietu Smarty, należy go najpierw poprawnie zainstalować. W pierwszej kolejności trzeba pobrać jego najnowszą wersję z witryny *http://smarty.php.net/*.

Po pobraniu pliku zawierającego pakiet Smarty i rozpakowaniu go na dysku zostaje utworzonych kilka katalogów i plików. Tylko kilka spośród wszystkich rozpakowanych plików stanowi część samego mechanizmu Smarty; są to trzy pliki zawierające klasy (*Smarty.class.php*, *Smarty_Compile.class.php* oraz *Config_File.class.php*) oraz katalog *plugins* wraz z zawartością. Wszystkie trzy pliki należy umieścić w jednym z katalogów, w których PHP automatycznie poszukuje dołączanych plików. Jeśli nie mamy możliwości umieszczenia tych plików w żadnym z automatycznie przeglądanych katalogów (gdyż na przykład nie mamy prawa dostępu do pliku *php.ini* lub *.htaccess*), to możemy postąpić na dwa sposoby:

Sposób 1: Możemy umieścić te pliki w dowolnie wybranym katalogu, a następnie w czasie działania skryptu odpowiednio zmodyfikować dyrektywę konfiguracyjną `include_path`, posługując się przy tym funkcjami `ini_set()` oraz `ini_get()`:

```
<?php ini_set("include_path",
             ini_get("include_path"). ";/ścieżka/do/Smarty/" ?>
```

Sposób 2: Możemy skopiować te pliki do dowolnego katalogu, a następnie zdefiniować stałą `SMARTY_DIR`, zapisując w niej ścieżkę do niego:

```
<?php define('SMARTY_DIR', '/scieżka/do/Smarty/'); ?>
```

Kolejnym etapem instalacji jest utworzenie trzech (a może nawet czterech) katalogów, które będą wykorzystywane przez pakiet. Tworząc te katalogi, koniecznie należy zwrócić uwagę na możliwe zagrożenia bezpieczeństwa i odpowiednio postępować. Poniżej przedstawiłem listę katalogów, które należy utworzyć, by móc korzystać z mechanizmu Smarty.



Podane poniżej nazwy katalogów można zmienić. Jeśli zdecydujemy się na zastosowanie innych nazw, musimy podać je podczas konfigurowania składowych głównej klasy mechanizmu Smarty (zagadnienie to opisałem w dalszej części rozdziału).

templates Ten katalog powinien się znajdować poza drzewem katalogów witryny WWW; jest on wykorzystywany do przechowywania szablonów używanych przez mechanizm Smarty.

templates_c Ten katalog musi się znajdować w drzewie katalogów witryny WWW, a służy do przechowywania skompilowanych szablonów (skryptów PHP), które są wykonywane w celu wyświetlenia żądanej strony WWW. Zarówno PHP, jak i serwer WWW musi mieć prawo zapisu plików w tym katalogu.

configs Ten katalog powinien się znajdować poza drzewem katalogów witryny WWW; służy on do przechowywania plików konfiguracyjnych używanych przez szablony obsługiwane przez mechanizm Smarty (zagadnienie to opiszę w dalszej części rozdziału).

cache Także ten katalog powinien znajdować się poza drzewem katalogów witryny WWW; pełni on funkcję pamięci podręcznej służącej do przechowywania szablonów (zagadnienie to opiszę w dalszej części rozdziału). Zarówno PHP, jak i serwer WWW musi mieć prawo zapisu plików w tym katalogu.

PHP musi dysponować prawami dostępu do tych katalogów (przy czym katalogi *configs* oraz *templates* mogą być przeznaczone tylko do odczytu, natomiast pozostałe muszą także pozwalać na zapis plików). Z myślą o osobach, które mogą nie znać terminologii, wyjaśniam, iż określenie „poza drzewem katalogów witryny WWW" oznacza, iż zawartość danego katalogu nie jest dostępna dla serwera WWW, zatem użytkownik nie może wyświetlić w przeglądarce żadnego z przechowywanych w nim plików.

Po skopiowaniu niezbędnych plików i utworzeniu wszystkich katalogów kolejnym krokiem jest skonfigurowanie mechanizmu Smarty. W tym celu należy otworzyć plik *Smarty.class.php* w dowolnym edytorze i zmodyfikować wartości składowych (umieszczonych na samym początku kodu klasy). Choć w samym kodzie klasy zostały podane krótkie opisy każdej ze składowych, niemniej jednak poniżej podałem wyjaśnienia dotyczące najważniejszych składowych konfiguracyjnych mechanizmu Smarty:

`$template_dir` Określa ścieżkę, w której Smarty będzie poszukiwać używanych szablonów; składowa ta powinna zawierać ścieżkę do utworzonego wcześniej katalogu. (Jej domyślną wartością jest `templates`).

`$compile_dir` Określa ścieżkę, w której Smarty będzie zapisywać skompilowane wersje szablonów. (Jej domyślną wartością jest `templates_c`).

`$plugins_dir` Określa ścieżkę (lub ścieżki), w której Smarty będzie poszukiwać pluginów używanych przez mechanizm przetwarzający; składowa ta zawiera tablicę łańcuchów znaków, z których każdy jest ścieżką do katalogu. (Domyślną zawartością tej składowej jest `array('plugins')`).

`$compile_check` Jej wartość określa, czy Smarty będzie sprawdzać, czy szablon wymaga powtórnej kompilacji. Jeśli jej wartość jest różna od `true`, to Smarty nigdy nie będzie aktualizować (ponownie kompilować) zmodyfikowanych szablonów. (Domyślnie składowa ta ma wartość `true`).

Po wprowadzeniu niezbędnych zmian w konfiguracji mechanizmu Smarty można go przetestować, by upewnić się, czy wszystko działa poprawnie. W tym celu należy utworzyć dwa pliki przedstawione na listingach 7.7 oraz 7.8 i nadać im odpowiednio nazwy: *test_template.tpl* oraz *test_smarty.php*.

Listing 7.7. *Testowy szablon*

```
Poniżej powinien zostać wyświetlony napis 'PHP 5. Księga eksperta':<BR>
{$testvar}<BR><BR>
Poniżej powinna zostać wyświetlona tabela zawierająca kolejne liczby od 1 do 10:
<TABLE CELLPADDING=3 BORDER=1>
<TR>
{section name=testsection loop=$testdata}
<TD>{$testdata[testsection]}</TD>
{/section}
</TR>
</TABLE>
<BR>
Wyrażenie '{$testvar}' składa się z {$testvar|count_characters} znaków.
```

Listing 7.8. *Skrypt testujący działanie mechanizmu Smarty*

```
<?php

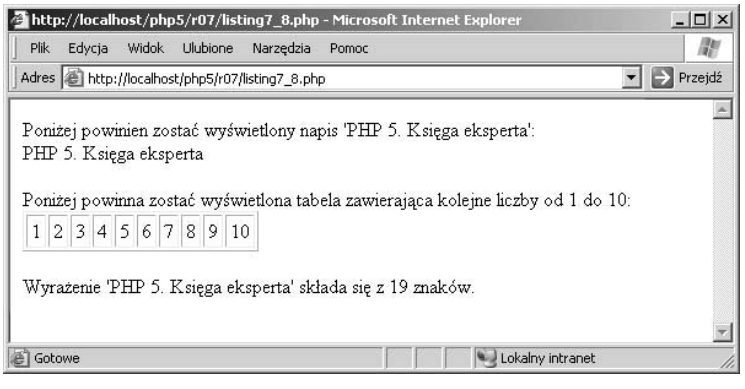
    require("Smarty.class.php");

    $smarty = new Smarty;

    $smarty->assign("testvar", 'PHP 5. Księga eksperta');
    $smarty->assign("testdata", range(1,10));
    $smarty->display("test_template.tpl");

?>
```

Aby przetestować instalację pakietu Smarty, plik *test_template.tpl* należy umieścić w katalogu *templates* (jeśli tak został nazwany), a plik *test_smarty.php* w dowolnym katalogu należącym do drzewa witryny WWW. Później pozostaje już tylko uruchomienie przeglądarki i podjęcie próby wyświetlenia w niej pliku *test_smarty.php*. W rezultacie strona wyświetlona w przeglądarce powinna wyglądać tak samo jak na rysunku 7.1.



Rysunek 7.1. *Okno przeglądarki prezentujące wyniki wykonania skryptu test_smarty.php*

Jeśli podczas działania skryptu pojawiły się błędy lub z jakichkolwiek innych przyczyn skrypt nie został wykonany poprawnie, to w pierwszej kolejności należy sprawdzić, czy zostały utworzone niezbędne katalogi oraz czy poprawnie określono prawa dostępu do nich. Oprócz tego należy powtórnie sprawdzić składowe konfiguracyjne umieszczone w pliku *Smarty.class.php* i upewnić się, czy ich wartości odpowiadają utworzonym katalogom. Jeśli wszystkie te operacje nie dadzą żadnego efektu, to należy poszukać dodatkowych informacji w dokumentacji pakietu Smarty. Jeśli jednak test zakończył się pomyślnie, to gratuluję — mechanizm Smarty został zainstalowany na serwerze, a Czytelnik może przejść do lektury dalszej części rozdziału.

Podstawy działania mechanizmu Smarty: zmienne i modyfikatory

Teraz, kiedy Czytelnik zainstalował już pakiet Smarty na swoim serwerze, możemy zacząć poznawać sposoby jego zastosowania! W tym celu opiszę, w jaki sposób można zrealizować proste zastąpienie zmiennej, podobne do tego, które realizowała klasa `QuickTemplate`.

W przypadku mechanizmu Smarty zmienne są domyślnie zapisywane w postaci {$nazwa_zmiennej}, o czym mogliśmy się przekonać analizując przykładowy szablon przedstawiony na listingu 7.7. Należy zapamiętać, iż jest to domyślny sposób zapisu zmiennych w szablonach obsługiwanych przez mechanizm Smarty. Ograniczniki, domyślnie są to nawiasy klamrowe {}, można jednak zmieniać przy użyciu zmiennych konfiguracyjnych $left_delimiter oraz $right_delimiter. W każdym razie, aby łańcuch znaków umieszczony pomiędzy tymi ogranicznikami został uznany za zmienną, musi się zaczynać od znaku $. Stosowane w pakiecie Smarty reguły określające, jakie znaki mogą być używane w nazwach zmiennych, odpowiadają regułom stosowanym w języku PHP. Podobnie jak PHP, także i mechanizm Smarty zwraca uwagę na wielkość liter w nazwach zmiennych. Poniżej przedstawiłem przykład bardzo prostego szablonu:

```
{* Bardzo prosty szablon *}
Witam, dziękuję {$name} za kupienie książki PHP 5. Księga eksperta.
```



Podobnie jak kod skryptów PHP, także i szablony obsługiwane przez mechanizm Smarty mogą zawierać komentarze. Komentarze rozpoczynają się od łańcucha znaków {* (przy czym nawias klamrowy można zastąpić dowolnie wybranym ogranicznikiem), a kończą łańcuchem *}. Podobnie jak komentarze umieszczane w skryptach, także i komentarze umieszczane w szablonach są ignorowane i nigdy nie będą wyświetlane w przeglądarce.

W powyższym szablonie zdefiniowaliśmy jedną zmienną {$name}. Aby szablon ten można było wykorzystać, należy go zapisać w katalogu *templates*. Zazwyczaj szablony zapisuje się w plikach z rozszerzeniem *tpl* określającym przeznaczenie pliku, niemniej jednak nie jest to konieczne. W naszym przykładzie zakładam, iż szablon został zapisany w pliku o nazwie *simple.tpl*.

Po utworzeniu i zapisaniu szablonu w pliku można napisać skrypt PHP, który będzie z niego korzystać. Pierwszym etapem zastosowania mechanizmu Smarty w skryptach PHP jest stworzenie obiektu klasy Smarty. W tym celu na początku skryptu należy umieścić następujący fragment kodu:

```
<?php require('Smarty.class.php');
$smarty = new Smarty(); ?>
```

Od tego momentu z mechanizmu Smarty będziemy korzystali za pośrednictwem obiektu klasy Smarty zapisanego w zmiennej $smarty.

Aby można było przetwarzać szablony mechanizmu Smarty, oprócz dołączenia niezbędnego pliku i utworzenia obiektu klasy Smarty konieczne jest wykonanie kilku dodatkowych czynności. Pierwszą z nich jest określenie wartości wszystkich zmiennych używanych w szablonie. Do tego celu służy funkcja assing() klasy Smarty. Funkcja ta umożliwia podanie jednego lub dwóch parametrów, przy czym sposób jej wywołania określa czynności, jakie zostaną wykonane.

Pierwszy sposób wywołania funkcji assing() polega na przekazaniu dwóch parametrów, z których pierwszy jest łańcuchem znaków, a drugi wartością. W tym przypadku Smarty przypisze zmiennej określonej przy użyciu pierwszego parametru wartość przekazaną jako drugi parametr. W naszym przypadku, aby przypisać wartość zmiennej {$name}, możemy użyć poniższego wiersza kodu:

```
<?php $smarty->assign('name', 'John Coggeshall'); ?>
```

Drugi sposób wywoływania funkcji `assing()` jest przydatny, gdy należy określić wartości wielu zmiennych. W tym przypadku w wywołaniu funkcji przekazywana jest jedynie tablica asocjacyjna. Klucze elementów tej tablicy reprezentują zmienne, natomiast wartości tych elementów to wartości zmiennych. Na przykład gdyby w szablonie były używane zmienne `{$foo}` oraz `{$bar}`, to ich wartości można by określić przy użyciu poniższego wywołania funkcji `assing()`:

```
<?php $smarty->assing(array('foo' => 10, 'bar' => 'Witaj Świecie!')); ?>
```

W powyższym przykładzie zmiennej `{$foo}` przypisaliśmy wartość całkowitą `10`, a zmiennej `{$bar}` łańcuch znaków `'Witaj Świecie!'`.

W powyższym przykładzie użyliśmy tablicy, aby określić wartości zmiennych `{$foo}` oraz `{$bar}`. Ale tablice mogą być także wartościami zmiennych. Zapisanie tablicy w zmiennej nie różni się niczym od standardowego określenia wartości zmiennej. Na przykład w poniższym przykładzie zmiennej `{$myarray}` przypisywana jest tablica zawierająca liczby całkowite od `5` do `10`:

```
<?php $smarty->assing('myarray', array(5,6,7,8,9,10)); ?>
```

Choć tworzenie tablicy w szablonach odbywa się podobnie jak w skryptach PHP, to jednak odwoływanie się do ich elementów ma inną postać niż korzystanie z wartości skalarnych. W przypadku tablic, których klucze są liczbami całkowitymi, takich jak tablica przedstawiona na powyższym przykładzie, sposób odwoływania się do jej elementów w szablonie jest taki sam jak w skryptach PHP — do nazwy zmiennej należy dodać odpowiedni indeks zapisany w nawiasach kwadratowych. A zatem wyrażenie `{$myarray[2]}` umieszczone w szablonie odwołuje się do tej samej wartości co wyrażenie `$myarray[2]` umieszczone w skrypcie PHP (zakładając oczywiście, że obie zmienne zawierają identyczne tablice). Jednak w przypadku posługiwania się tablicami asocjacyjnymi tablice stosowane w szablonach Smarty działają w odmienny sposób — zamiast zapisywać klucz w nawiasach kwadratowych, należy go poprzedzić kropką i zapisać bezpośrednio za nazwą zmiennej. Sposób korzystania z tablic asocjacyjnych w szablonach Smarty został przedstawiony na listingu 7.9.

Listing 7.9. *Odwoływanie się do tablic indeksowanych liczbami całkowitymi*

```
<HTML><HEAD><TITLE>{$title}</TITLE></HEAD>
<BODY>
Trzecim elementem tablicy $myarray jest: {$myarray[2]}<BR>
Element o kluczu 'mykey' w tablicy
$anotherarray ma wartość: {$anotherarray.mykey}<BR>
</BODY>
</HTML>
```

Przy założeniu, że wartość klucza `mykey` jest tablicą asocjacyjną, odwołanie do tej wartości w kodzie PHP miałoby postać: `$anotherarray['mykey']`. Jednak w szablonach odwołanie to ma postać `{$anotherarray.mykey}`.

Teraz, kiedy już wiemy, jak określać wartości i jak odwoływać się do zmiennych w szablonach Smarty, pokażę, czym one się różnią od zmiennych stosowanych w skryptach PHP. Konkretnie rzecz biorąc, chciałbym wprowadzić pojęcie modyfikatorów zmiennych.

Modyfikatory zmiennych, zgodnie z tym co sugeruje ich nazwa, służą do odpowiedniej zmiany zawartości wskazanej zmiennej. Modyfikatory są umieszczane bezpośrednio w szablonach, przy czym aby je zastosować, należy zapisać nazwę wybranego modyfikatora po nazwie zmiennej, oddzielając obie nazwy znakiem potoku (|). Domyślnie w mechanizmie Smarty zdefiniowanych jest 19 modyfikatorów; istnieje też możliwość dodawania nowych modyfikatorów, tworzonych w formie pluginów. Na przykład jeden ze standardowych modyfikatorów nosi nazwę `upper`, a jego działanie polega na zamianie wszystkich małych liter w łańcuchu znaków na duże. Spróbujmy zatem zastosować ten modyfikator — za jego pomocą wyświetlimy zawartość zmiennej `{$name}` z jednego z poprzednich przykładów dużymi literami:

```
Witam, dziękuję {$name|upper} za kupienie książki PHP 5. Księga eksperta.
```

W większości przypadków będziemy chcieli dostosować działanie modyfikatorów do własnych potrzeb. W tym celu konieczne będzie przekazanie do modyfikatora odpowiednich parametrów. Parametry przekazuje się, zapisując po nazwie modyfikatora dwukropek i wartość przekazywanego parametru, przy czym w ten sam sposób można przekazać większą ilość parametrów. Należy jednak pamiętać, że nie wszystkie modyfikatory pozwalają na przekazywanie parametrów (przykładem modyfikatora, który na to nie pozwala, jest `upper`). Sposób przekazywania parametrów przedstawiłem na przykładzie modyfikatora `wordwrap` (listing 7.10), który ogranicza ilość znaków wyświetlanych w jednym wierszu.

Listing 7.10. *Zastosowanie modyfikatora wordwrap*

```
<HTML><HEAD><TITLE>{$title}</TITLE></HEAD>
<BODY>
W poniższym wierszu zostanie wyświetlonych tylko 30 znaków:<BR><BR>

{$excerpt|wordwrap:30:"<br>\n"}
</BODY>
</HTML>
```

Jak widać, w powyższym przykładzie oprócz samego modyfikatora `wordwrap` zostały podane także dwa parametry. Pierwszy z nich jest liczbą określającą ilość znaków, które należy wyświetlać w wierszu (w naszym przykładzie wyświetlanych będzie 30 znaków). Z kolei drugi parametr to łańcuch znaków, który będzie wstawiany do zawartości zmiennej co określoną wcześniej ilość znaków. Ponieważ zawartość zmiennej ma być wyświetlana w przeglądarce WWW, zatem prócz standardowego znaku nowego wiersza w momencie przenoszenia pozostałej części tekstu do nowego wiersza należy też dodawać odpowiedni znacznik HTML. To, na jakich danych faktycznie operuje modyfikator `wordwrap` użyty w powyższym przykładzie, nie ma najmniejszego znaczenia, gdyż wszelkie operacje są wykonywane przy użyciu modyfikatorów stosowanych wyłącznie w szablonach.

Gdy konieczne jest zastosowanie kilku modyfikatorów operujących na jednej zmiennej, kolejny modyfikator można zapisać tuż po parametrach poprzedniego, oddzielając go od nich znakiem potoku (|). Sposób zastosowania dodatkowego modyfikatora został przedstawiony na listingu 7.11, który określa domyślne wartości zmiennych `{$excerpt}` oraz `{$title}`, jeśli nie zostały one podane wcześniej.

Listing 7.11. *Zastosowanie kilku modyfikatorów operujących na jednej zmiennej*

```
<HTML><HEAD><TITLE>{$title|default:"Nie podano"}</TITLE></HEAD>
<BODY>
W poniższym wierszu zostanie wyświetlonych tylko 30 znaków:<BR><BR>
{$excerpt|wordwrap:30:"<br>\n"|default:"Brak danych!"}
</BODY>
</HTML>
```

Pełną listę wszystkich dostępnych modyfikatorów można znaleźć w dokumentacji pakietu Smarty na witrynie *http://smarty.php.net/*.

Teraz, kiedy znamy już zarówno zmienne, jak i modyfikatory zmiennych, nadszedł czas, by przyjrzeć się specjalnej zmiennej o nazwie `{$smarty}`.



Nie należy mylić zmiennej `{$smarty}` używanej w szablonach ze zmienną `$smarty`, której używaliśmy w skryptach PHP do zapisania obiektu klasy `Smarty`.

Tworząc szablony Smarty, można korzystać z kilku zmiennych predefiniowanych. Wszystkie te zmienne są dostępne jako klucze zmiennej `{$smarty}`. Dzięki tym zmiennym przy tworzeniu szablonów można uzyskać dostęp do danych związanych z żądaniem HTTP (na przykład danych przesłanych metodami `GET` lub `POST`), wewnętrznych danych pakietu Smarty, danych serwera oraz środowiskowych. Poniżej podałem listę informacji, do których można uzyskać dostęp za pośrednictwem zmiennej `{$smarty}`:

`{$smarty.get}`	Tablica zamiennych przesłanych metodą `GET` (odpowiada superglobalnej tablicy `$_GET`).
`{$smarty.post}`	Tablica zmiennych przesłanych metodą `POST` (odpowiada superglobalnej tablicy `POST`).
`{$smarty.cookie}`	Tablica danych przesłanych w cookies (odpowiada superglobalnej tablicy `$_COOKIE`).
`{$smarty.server}`	Tablica danych związanych z serwerem (odpowiada superglobalnej tablicy `$_SERVER`).
`{$smarty.env}`	Tablica danych środowiskowych (odpowiada superglobalnej tablicy `$_ENV`).
`{$smarty.session}`	Tablica zarejestrowanych danych sesyjnych (odpowiada superglobalnej tablicy `$_SESSION`).
`{$smarty.request}`	Tablica zawierająca wszystkie dane pochodzące z tablic superglobalnych: `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER` oraz `$_ENV` (odpowiada tablicy superglobalnej `$_REQUEST`).
`{$smarty.now}`	Zawiera aktualny czas (wyrażony jako ilość sekund, jakie minęły od 1 stycznia 1970 roku). W celu wyświetlenia bieżącego czasu, daty itp. należy użyć tej zmiennej wraz z modyfikatorem `date_format`.
`{$smarty.template}`	Nazwa aktualnie przetwarzanego szablonu.



Powyższa lista nie jest kompletna. Nie zawiera ona kilku zmiennych, gdyż zagadnienia, z jakimi się one wiążą, nie zostały jeszcze opisane. Zmienne te przedstawię w dalszej części rozdziału przy okazji prezentacji zagadnień, z którymi się one wiążą.

Pliki i funkcje konfiguracyjne

Po przedstawieniu zmiennych oraz modyfikatorów zmiennych dostępnych w pakiecie Smarty chciałbym się skoncentrować na innych jego możliwościach. W pierwszej kolejności przedstawię funkcje oraz ich zastosowanie w tworzeniu praktycznych skryptów wykorzystujących szablony.

Domyślnie pakiet Smarty udostępnia około 12 funkcji, których można używać w tworzonych szablonach. Dzięki nim w szablonach można korzystać także ze struktur sterujących oraz logicznych (takich jak instrukcje `if`), jak również z innych przydatnych możliwości. Funkcje dostępne w pakiecie Smarty przypominają nieco modyfikatory zmiennych, gdyż tak jak i one posiadają predefiniowane parametry. Jednak w odróżnieniu od modyfikatorów funkcje często operują na całych blokach kodu HTML (przykładem może być generowanie poszczególnych wierszy tworzących tabelę).

W pakiecie Smarty funkcje zapisuje się wewnątrz tych samych ograniczników, w których są zapisywane nazwy zmiennych; natomiast parametry funkcji są od siebie oddzielane znakami odstępu (a nie przecinkami). Co więcej, jeśli funkcja ma operować na bloku kodu, to jego koniec należy zdefiniować przy użyciu nazwy funkcji poprzedzonej znakiem ukośnika (/, bez żadnych parametrów). Na początku przyjrzyjmy się najprostszej z możliwych funkcji, jakie można utworzyć i stosować w szablonach Smarty. Funkcje tego typu tworzy się po to, aby uprościć życie projektantów stron, gdyż upraszczają one tworzenie szablonów stron HTML. Zacznę od przedstawienia funkcji `{literal}`.

Funkcja `{literal}` jest wykorzystywana do oznaczania bloku kodu HTML, który należy pominąć podczas przetwarzania, lecz powinien zostać wyświetlony. Możliwość generowania takich bloków kodu jest niezwykle ważna w przypadku korzystania ze skryptów wykonywanych w przeglądarce (na przykład pisanych w języku JavaScript), których kod może przysporzyć problemów podczas analizy i przetwarzania szablonów. Funkcja `{literal}` nie wymaga podawania żadnych parametrów — po prostu możemy w niej umieścić blok kodu, który zostanie całkowicie zignorowany podczas przetwarzania szablonów. Przykład zastosowania funkcji `{literal}` przedstawiłem na listingu 7.12.

Listing 7.12. *Zastosowanie funkcji {literal}*

```
{literal}
    <script language="JavaScript">
        if(foo) {
            window.status = 'To prawda!';
        }
    </script>
{/literal}
```



Podobne rozwiązanie stosuje się, by poprawnie wyświetlić znaki, które są używane w szablonach jako ograniczniki. Do tego celu Smarty udostępnia dwie funkcje: `{ldelim}` oraz `{rdelim}`. Powodują one wyświetlenie odpowiednio: łańcucha lewego i prawego ogranicznika.

Wszyscy, którzy mają pewne doświadczenie w tworzeniu stron HTML, wiedzą, iż umieszczanie odstępów pomiędzy znacznikami może powodować, że różne przeglądarki mogą wyświetlać tę samą stronę w nieco odmienny sposób. W większości przypadków różnice

te są tak małe, iż można je pominąć. Niemniej jednak mogą się także zdarzyć przypadki, w których różnice te będą całkiem znaczące. Niestety, usunięcie wszystkich znaków odstępu z dokumentów HTML sprawia, iż stają się one bardzo nieczytelne. Aby rozwiązać ten problem, Smarty udostępnia funkcję {strip}, przedstawioną na listingu 7.13. Podczas przetwarzania szablonu funkcja ta usuwa z umieszczonego wewnątrz niej kodu HTML wszystkie znaki odstępu, dzięki czemu będzie on wyświetlany poprawnie.

Listing 7.13. *Zastosowanie funkcji {strip}*

```
{strip}
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
     <TR>
          <TD>Witam</TD>
     </TR>
</TABLE>
{/strip}
```

Kod HTML wygenerowany przez ten szablon będzie mieć następującą postać:

```
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0><TR><TD>Witam</TD></TR></TABLE>
```

Ostatnia z prostych funkcji pakietu Smarty, którą chciałbym przedstawić, jest związana z dołączaniem do szablonów kodu PHP. Smarty pozwala na umieszczanie w szablonach kodu PHP, jeśli zaistnieje taka konieczność, choć rozwiązanie takie nie jest polecane. Kod PHP można umieszczać w szablonach bądź to przy użyciu funkcji {php} stosowanej tak samo jak przedstawione wcześniej funkcje {literal} i {strip}, bądź to przy użyciu funkcji {include_php} pozwalającej na dołączenie do szablonu wskazanego pliku PHP. Funkcja {php} zastępuje standardowe ograniczniki skryptów PHP — <?php ?> — i nie wymaga podawania żadnych parametrów. Z kolei wywołanie funkcji {include_php} ma poniższą postać:

```
{include_php file=<nazwa_pliku> [once=<tylko_raz>] [assing=<zmienna>]}
```

Przy czym parametr <nazwa_pliku> określa nazwę pliku, który należy dołączyć do szablonu, <tylko_raz> jest parametrem logicznym określającym, czy wskazany plik powinien być dołączony tylko jeden raz (na zasadzie podobnej do działania instrukcji include_once PHP), a ostatni parametr — <zmienna> — określa nazwę zmiennej, w której zostaną zapisane wyniki wygenerowane przez dołączany plik (zamiast bezpośredniego przekazania ich do kodu generowanego przez szablon).

Po przedstawieniu najprostszych funkcji pakietu Smarty czas poznać funkcje nieco bardziej skomplikowane. Większość z funkcji, które opiszę, będzie związana z możliwościami określania logiki działania szablonów. Oczywiście wszędzie, gdzie pojawia się zagadnienie logiki działania, nie sposób nie wspomnieć o operacjach warunkowych — dlatego też w pierwszej kolejności opiszę funkcję {if}. Poniżej przedstawiłem jej składnię:

```
{if <warunek>}
     ...
{elseif <warunek>}
     ...
[{elseif <warunek>}
     ...
[{else}]
     ...
{/if}
```

Podczas posługiwania się funkcjami zmienne umieszczane wewnątrz ich ograniczników nie muszą być zapisywane w dodatkowych ogranicznikach. Na przykład gdy używamy funkcji {if}, w parametrze <warunek> możemy używać dowolnych zmiennych w sposób przedstawiony na listingu 7.14.



Zmiennych można używać w funkcjach pakietu Smarty nie tylko podczas tworzenia wyrażeń warunkowych! Mogą one także określać wartości parametrów funkcji.

Listing 7.14. *Zastosowanie funkcji {if}*

```
<HTML>
<HEAD><TITLE>Przykład zastosowania funkcji {if}.</TITLE></HEAD>
<BODY>
{if $secured == true}
     Cześć {$name}, witamy w obszarze chronionym!
{else}
     Nie masz praw niezbędnych do oglądania tej strony.
{/if}
</BODY>
</HTML>
```

W powyższym przykładzie w wyrażeniu warunkowym została użyta zmienna {$secured}. Wyrażenia warunkowe używane w szablonach Smarty mogą być bardzo proste lub dowolnie skomplikowane, a zasady ich działania są takie same jak zasady działania wyrażeń warunkowych stosowanych w skryptach PHP.

Kolejnym przykładem bardzo przydatnej funkcji pakietu Smarty jest funkcja {include}. Pozwala ona na dołączenie w miejscu wywołania szablonu zapisanego w innym, wskazanym pliku. Jej działanie odpowiada zatem zastosowaniu instrukcji include w skryptach PHP. Poniżej przedstawiłem składnię wywołania tej funkcji.

```
{include file=<nazwa_pliku> [assing=<zmienna_wyn>] [<zmienna>=<wartosc> ...]}
```

Gdzie parametr <nazwa_pliku> określa nazwę pliku szablonu, który chcemy dołączyć, a parametr <zmienna_wyn> określa nazwę zmiennej, w której zostaną zapisane wyniki wygenerowane przez dołączony plik (zamiast bezpośredniego przekazania ich do wyników generowanych przez aktualnie przetwarzany szablon). W wywołaniu funkcji {include} można także opcjonalnie podać dowolną ilość par nazwa-wartość. Na podstawie tych par wewnątrz dołączanego szablonu zostaną utworzone zmienne. Przykładowe wywołanie funkcji {include} przedstawione na listingu 7.15 spowoduje dołączenie pliku *header.tpl* i zastąpienie umieszczonej wewnątrz niego zmiennej wartością przekazaną w wywołaniu.

Listing 7.15. *Zastosowanie funkcji {include}*

```
<!-- To główny plik szablnu -->
{include file=header.tpl title="Oto tytuł strony!"}

<!-- Zawartość pliku header.tpl -->
<HTML><HEAD><TITLE>{$title}</TITLE></HEAD><BODY>
```



Jeśli pamięć podręczna mechanizmu Smarty jest włączona, to pliki dołączane przy użyciu funkcji {include} będą w niej zapisywane. Aby dołączyć wskazany plik do szablonu bez umieszczania go w pamięci podręcznej, należy wykorzystać funkcję {insert}. Funkcja ta działa niemal identycznie jak funkcja {include}, a jedyna różnica polega na tym, iż dołączany plik nie zostanie zapisany w pamięci podręcznej.

Jeśli dojedziemy do wniosku, że chcemy zapisać w zmiennej kod, który normalnie byłby wygenerowany jako wynik działania funkcji {include}, to można do tego celu wykorzystać parametr assing. Pozwala on na podanie nazwy zmiennej, w której zostaną zapisane wyniki wygenerowane przez dołączany szablon (na przykład parametr assing=foo spowoduje, że zawartość pliku zostanie zapisana w zmiennej $foo).

Podczas tworzenia szablonów bardzo wiele czasu może nam zaoszczędzić możliwość wielokrotnej realizacji tej samej czynności (w szczególności dotyczy to takich operacji jak generowanie tabel HTML). Mechanizm Smarty udostępnia dwa sposoby wielokrotnego wykonywania tych samych czynności — są to dwie funkcje: {section} oraz {foreach}. Obie z nich operują na zmiennych tablicowych, przy czym pierwsza służy do przeglądania tablic indeksowanych liczbami całkowitymi, natomiast druga operuje na tablicach asocjacyjnych. Zacznijmy od funkcji operującej na tablicach indeksowanych liczbami, czyli od funkcji {section}. Oto postać jej wywołania:

```
{section name=<licznik>
         loop=<zmienna>
         [start=<wart_pocz>]
         [step=<krok>]
         [max=<wart_maks>]
         [show=<pokazywac>]}
    ... zawartośc pętli ...
[{sectionelse}]
    ... zawartość wyświetlana gdy wszystkie elementy tablicy zostaną pobrane ...
{/section}
```

Gdzie parametr <licznik> to nazwa (a nie prawdziwa zmienna) umożliwiająca odwoływanie się do indeksu bieżącego elementu tablicy, a parametr <zmienna> określa, na jakiej tablicy ma operować pętla. Pierwszy z parametrów opcjonalnych — <wart_pocz> — określa początkową wartość indeksu (jest to oczywiście liczba całkowita). Parametr <krok> określa, o ile będzie się zwiększał licznik pętli podczas każdej iteracji. Parametr <wart_max> definiuje maksymalną ilość iteracji pętli. Natomiast ostatni z parametrów opcjonalnych — <pokazywac> — określa, czy dana sekcja będzie aktywna (czyli czy jej wyniki będą wyświetlane). Wewnątrz sekcji można stosować dowolne zmienne oraz funkcje pakietu Smarty, takie jak {if}, a nawet inne funkcje {section}.



Jeśli zdefiniujemy segment {sectionelse}, to zostanie on przetworzony, nawet jeśli parametrowi show funkcji {section} będzie przypisana wartość true.

Wewnątrz funkcji {section} dostępnych jest kilka zmiennych dostarczających przeróżnych informacji na temat jej bieżącego stanu. Ponieważ w większości przypadków funkcja ta będzie używana do wyświetlania odpowiednio sformatowanej zawartości tablicy, zatem chciałbym pokazać, w jaki sposób można jej używać do wyświetlania zawartości konkretnego elementu tablicy. Jak wiadomo, w celu wyświetlenia wartości wybranego elementu tablicy indeksowanej liczbami za nazwą zmiennej tablicowej należy zapisać parę nawiasów kwadratowych zawierających liczbę określającą indeks elementu. Jednak w przypadku korzystania z sekcji, zamiast określać konkretną wartość indeksu, w nawiasach kwadratowych należy użyć nazwy podanej w parametrze name. Stosowny przykład został przedstawiony na listingu 7.16. Zawiera on szablon, który wyświetla zawartość tablicy {$myarray} w formie tabeli HTML.

Listing 7.16. *Zastosowanie funkcji {section}*

```
<TABLE CELLPADDING=0 CELLSPACING=0 BORDER=1>
<TR>
{section name=countvar loop=$myarray}
<TD>{$myarray[countvar]}</TD>
{/section}
</TR>
</TABLE>
```

Segment `{sectionelse}` instrukcji `{section}` pozwala na wyświetlenie stosownego komunikatu w przypadku, gdy podana tablica nie istnieje, jest pusta lub gdy nie ma w niej elementów o indeksach określonych przez parametry funkcji. Przykład zastosowania segmentu `{sectionelse}` został przedstawiony na listingu 7.17, przy czym zakładam w nim, iż tablica `$myarray` jest pusta.

Listing 7.17. *Zastosowanie funkcji {section} wraz z {sectionelse}*

```
{section name=countvar loop=$myarray max=2}
Aktualna wartość to: {$myarray[countvar]}<BR>
{sectionelse}
Nie ma już wartości, które możnaby wyświetlić!<BR>
{/section}
```

Zgodnie z tym, o czym pisałem wcześniej, w przypadku korzystania z sekcji mechanizm Smarty udostępnia wiele dodatkowych informacji. Są one przechowywane w specjalnej zmiennej o nazwie `$smarty`, o której wspominałem już we wcześniejszej części rozdziału. Aby uzyskać dostęp do tych informacji, należy użyć wyrażenia o następującej postaci:

```
{$smarty.section.<nazwa_sekcji>.<nazwa_zmiennej>}
```

Gdzie `<nazwa_sekcji>` jest wartością przypisaną parametrowi `name` funkcji `{section}`, a `<nazwa_zmiennej>` jest jedną z wartości przedstawionych w tabeli 7.1.

Tabela 7.1. *Specjalne zmienne dostępne w funkcji {section}*

`index`	Liczba całkowita określająca bieżącą wartość indeksu (jest ona zależna od parametrów `start`, `step` oraz `max`).
`index_prev`	Liczba całkowita określająca poprzednią wartość indeksu (jeśli poprzedni indeks nie jest dostępny, zmienna ta przyjmuje wartość –1).
`index_next`	Liczba całkowita określająca następną wartość indeksu.
`iteration`	Liczba określająca, ile razy pętla już została wykonana. Wartość tej zmiennej nie zależy od żadnych parametrów funkcji `{section}`.
`first`	Wartość logiczna określająca, czy pętla jest wykonywana po raz pierwszy.
`last`	Wartość logiczna określająca, czy pętla jest wykonywana po raz ostatni.
`loop`	Ta zmienna określa ostatnią wartość indeksu użytą wewnątrz pętli (można z niej korzystać nawet poza funkcją `{section}` po jej wykonaniu).
`show`	Wartość informująca, czy pętla została wykonana.
`total`	Wartość określająca, ile razy pętla zostanie wykonana (można z niej korzystać poza funkcją `{section}` po jej wykonaniu).

Na przykład aby określić, ile razy zostanie wykonana funkcja {section} o nazwie countvar, można wykorzystać poniższy fragment kodu:

```
Poniżej zostanie wyświetlona lista {$smarty.section.countvar.total} elementów..<BR>
```

Ostatnim przykładem zastosowania funkcji {section} będzie szablon przedstawiony na listingu 7.18. Jego zadaniem jest wygenerowanie tabeli prezentującej listę imion naszych znajomych (stworzoną przez odpowiedni skrypt PHP). Jeśli skrypt nie zwróci listy imion, szablon wygeneruje stosowny komunikat.

Listing 7.18. *Zastosowanie parametru show funkcji {section}*

```
{section name=myfriends loop=$friends show=$show_friends}
{if $smarty.section.myfriends.first}
<TABLE CELLPADDING=0 CELLSPACING=3 BORDER=1>
{/if}
<TR><TD>{$friends[myfriends]}</TD></TR>
{if $smarty.section.myfriends.last}
</TABLE>
{/if}
{sectionelse}
Przykro mi... nie mam znajomych!
{/section}
```

Z myślą o sytuacjach, w których chcemy operować na tablicach asocjacyjnych, a nie na tablicach indeksowanych liczbami całkowitymi, pakiet Smarty udostępnia funkcję {foreach}. Działa ona bardzo podobnie do instrukcji foreach języka PHP i ma następującą składnię:

```
{foreach from=<zmienna_tablicowa>
         item=<biezacy_elem>
         [key=<klucz>]
         [name=<nazwa_petli>]}
   ...
[{foreachelse}]
   ...
{/foreach}
```

Gdzie <zmienna_tablicowa> to tablica, na której funkcja ma operować, <biezacy_elem> oraz <klucz> to nazwy zmiennych, w których będzie zapisywany odpowiednio: bieżący element tablicy oraz jego klucz, a <nazwa_petli> to nazwa tej funkcji {foreach}. Podobnie jak było w przypadku funkcji {section}, także i funkcja {foreach} udostępnia opcjonalny segment {foreachelse}, który zostanie wykonany, jeśli wskazana tablica będzie pusta. Sposób zastosowania tej funkcji został przedstawiony na listingu 7.19, prezentującym szablon, który wyświetla pary klucz-wartość zapisane w tablicy {$myarray}.

Listing 7.19. *Zastosowanie funkcji {foreach}*

```
<TABLE CELLPADDING=0 CELLSPACING=3 BORDER=1>
{foreach from=$myarray item=curr_item key=curr_key}
<TR>
<TD>{$curr_key}</TD><TD>{$curr_item}</TD>
</TR>
{/foreach}
</TABLE>
```

Podobnie jak funkcja {section}, także i funkcja {foreach} udostępnia kilka zmiennych, z których można korzystać wewnątrz niej przy wykorzystaniu specjalnej zmiennej {$smarty.foreach}. Zmienne te mają takie same nazwy jak zmienne przedstawione w tabeli 7.1, jednak w przypadku korzystania z funkcji {foreach} nie wszystkie z nich są dostępne. Dostępne są jedynie zmienne iteration, first, last, show oraz total.

Ostatnią wewnętrzną funkcją pakietu Smarty, którą chciałem przedstawić, jest funkcja {capture}. Zapewnia ona dokładnie te same możliwości funkcjonalne co parametr assign funkcji {include}, lecz nie wymaga umieszczania kodu w osobnym pliku. Cały kod szablonu umieszczony wewnątrz tej funkcji nie zostanie przekazany bezpośrednio do przeglądarki, lecz zapisany w zmiennej. Podobnie jak wszystkie inne funkcje pakietu Smarty, także i ta może być umieszczana wewnątrz innych funkcji. Poniżej przedstawiłem składnię wywołania funkcji {capture}:

```
{capture name=<nazwa_zmiennej>}
   ...
{/capture}
```

Gdzie parametr <nazwa_zmiennej> określa nazwę, jaką należy skojarzyć z generowanym i zapamiętywanym fragmentem kodu. W celu uzyskania dostępu do zapamiętanego kodu należy się odwołać do specjalnej zmiennej {$smarty.capture.<nazwa_zmiennej>} w sposób przedstawiony na listingu 7.20.

Listing 7.20. *Zastosowanie funkcji {capture}*

```
{capture name=mytable}
{include file="gen_table.tpl"}
{/capture}
Oto zawartość mojej tabeli:
{$smarty.capture.mytable}
```

Zanim przejdę do kolejnych zagadnień, chciałbym zakończyć prezentację funkcji dostępnych w pakiecie Smarty informacją, iż bynajmniej nie wyczerpałem tego zagadnienia. Opisałem bowiem jedynie wbudowane funkcje pakietu — jednak w rzeczywistości dostępnych funkcji jest znacznie więcej. Można bowiem stosować opcjonalne funkcje (jak również modyfikatory zmiennych) definiowane w formie pluginów. Najbardziej popularne z tych pluginów zostały dołączone do standardowej wersji pakietu Smarty, a informacje dotyczące sposobu ich stosowania można znaleźć w jego dokumentacji. Pełną listę oficjalnych pluginów pakietu Smarty można znaleźć na jego witrynie (*http://smarty.php.net/*), tam też można je pobrać lub zdobyć wszelkie informacje na ich temat.

Teraz przyjrzyjmy się, w jaki sposób pakiet Smarty umożliwia posługiwanie się danymi, które na ogół uznawane są za stałe. Dane tego typu określają zazwyczaj kolor tła witryny lub inne informacje, które nie zmieniają się podczas obsługi kolejnych żądań. Choć jest całkiem prawdopodobne, że znaczna część informacji przechowywanych w plikach konfiguracyjnych będzie wykorzystywana w szablonach, to jednak warto pamiętać, iż takich stałych wartości konfiguracyjnych można używać także w logice aplikacji.

Pliki konfiguracyjne używane w pakiecie Smarty mają bardzo podobną strukturę do pliku *php.ini*. Przykład pliku konfiguracyjnego, z którego można korzystać w pakiecie Smarty, przedstawiłem na listingu 7.21.

Listing 7.21. *Plik konfiguracyjny, z którego można korzystać w pakiecie Smarty*

```
# plik myconfiguration.ini

# Wartości określające kolory
[Colors]
background=#FFFFFF
link=#FF0000
vlink=#FF00FF
alink=#00FF00

# Statyczne teksty używane na witrynie
[StaticText]
base_url=http://www.phphaven.com/

[.DatabaseSettings]
host=localhost
username=user
password=mypassword
```

Jak widać na listingu 7.21, plik konfiguracyjny zawiera kilka zmiennych, które mogą być używane podczas tworzenia witryn wykorzystujących pakiet Smarty. W pikach konfiguracyjnych pakietu Smarty, podobnie jak w pliku *php.ini*, nazwy sekcji (zapisywane w nawiasach kwadratowych —`[]`) oraz nazwy zmiennych konfiguracyjnych muszą być zgodne z zasadami określającymi nazwy zmiennych PHP. Należy także zauważyć, iż komentarze tworzy się, umieszczając na początku wiersza znak # lub `;`.

Jedną z różnic pomiędzy plikiem konfiguracyjnym PHP oraz plikami konfiguracyjnymi pakietu Smarty jest sekcja `[.DatabaseSettings]`. Jak już wspominałem, nazwy sekcji muszą być zgodne z zasadami nazewnictwa zmiennych PHP, dlatego też wydaje się, iż ta nazwa sekcji jest nieprawidłowa (gdyż zaczyna się od znaku kropki). I choć można by sądzić, iż jest to niezamierzony błąd w tekście książki, okazuje się, że ta nazwa jest całkowicie poprawna. Kropka umieszczona na początku nazwy sekcji lub zmiennej konfiguracyjnej nakazuje, by dana sekcja bądź zmienna nie była „widoczna" w szablonach. Choć taka „ukryta" zmienna nie będzie dostępna w szablonach, to jednak będzie można z niej korzystać w skryptach PHP. Dane konfiguracyjne tego typu doskonale nadają się do przechowywania ważnych informacji (takich jak nazwy użytkowników i hasła), gdyż nie będą z nich mogli korzystać projektanci tworzący szablony.

Kolejnym zagadnieniem związanym ze stosowaniem danych konfiguracyjnych będzie przedstawienie sposobu, w jaki można z nich korzystać w szablonach. Aby uzyskać dostęp do zmiennych konfiguracyjnych, w pierwszej kolejności należy wczytać plik konfiguracyjny, w którym są one zapisane. Do tego celu służy funkcja `{config_load}`.

Poniżej przedstawiłem sposób jej wywołania oraz parametry:

```
{config_load file=<nazwa_pliku> section=<sekcja> scope=<zakres>}
```

Przy czym parametr `<nazwa_pliku>` określa nazwę pliku konfiguracyjnego, który należy wczytać, parametr `<sekcja>` sekcję umieszczoną w tym pliku, której zawartość należy wczytać, a parametr `<zakres>` definiuje, jakie szablony będą miały dostęp do wczytanych wartości konfiguracyjnych. Wartości konfiguracyjne, którymi można się posługiwać w mechanizmie Smarty, mogą należeć do jednego z trzech zakresów: `local` (który

oznacza, że wartości konfiguracyjne będą dostępne wyłącznie w bieżącym pliku szablonu), `parent` (który oznacza, że wartości będą dostępne zarówno w bieżącym pliku szablonu, jak i w szablonie, z którego został on wywołany) oraz `global` (w tym przypadku wartości będą dostępne we wszystkich szablonach).

Po wczytaniu pliku konfiguracyjnego do umieszczonych w nim zmiennych można się odwoływać zapisując ich nazwy pomiędzy ogranicznikami o postaci {# oraz #}. Listing 7.22 pokazuje, w jaki sposób można korzystać ze zmiennych konfiguracyjnych zdefiniowanych w pliku przedstawionym na listingu 7.21.



Jeśli stosuje się niestandardowe ograniczniki (inne niż nawiasy klamrowe { oraz }), to także w przypadku korzystania ze zmiennych konfiguracyjnych należy zastąpić standardowe ograniczniki {# oraz #} własnymi. I tak, jeśli stosowane ograniczniki mają postać <!-- oraz -->, to nazwy zmiennych konfiguracyjnych należy umieszczać pomiędzy łańcuchami: <!--# oraz #-->.

Listing 7.22. *Stosowanie plików konfiguracyjnych w szablonach Smarty*

```
{config_load file="myconfiguration.ini" section="Colors" scope="local"}
<HTML>
<HEAD>
<BODY LINK={#link#} ALINK={#alink#} VLINK={#vlink#} BGCOLOR={#background#}>
Witamy w <A HREF="{#base_url#}">PHPHaven.com!</A>
</BODY>
</HTML>
```

Jak już wspominałem, zmienne konfiguracyjne oraz sekcje, których nazwy rozpoczynają się od kropki, nie będą dostępne w przypadkach wczytywania plików konfiguracyjnych przy użyciu funkcji `{config_load}`. Aby uzyskać dostęp do tych zmiennych i sekcji, konieczne będzie wczytanie pliku konfiguracyjnego przy użyciu specjalnej klasy `Config_File` zdefiniowanej w pliku *Config_File.class.php*.

Klasa `Config_File` jest używana przez Smarty do wczytywania zmiennych z plików konfiguracyjnych i można jej używać zupełnie niezależnie od wykorzystania samego mechanizmu Smarty. Klasa ta posiada kilka składowych, których wartości można modyfikować, aby dostosować działanie klasy do naszych potrzeb.

`$overwrite`	(Wartość logiczna) Jeśli składowa ta przyjmie wartość `true`, to zmienne konfiguracyjne o tych samych nazwach będą się wzajemnie przesłaniać.
`$booleanize`	(Wartość logiczna) Jeśli składowa ta przyjmie wartość `true`, to wartości `true` i `yes` oraz ich odpowiedniki będą automatycznie zamieniane na wartości logiczne PHP — `true` oraz `false`.
`$read_hidden`	(Wartość logiczna) Jeśli składowa ta przyjmie wartość `false`, to ukryte zmienne konfiguracyjne oraz sekcje plików konfiguracyjnych nie będą dostępne.
`$fix_newlines`	(Wartość logiczna) Jeśli składowa ta przyjmie wartość `true`, to klasa będzie automatycznie konwertować pliki zapisane w formatach innych niż uniksowy (w których nowy wiersz jest oznaczany znakami specjalnymi `\r` lub `\r\n`) na format uniksowy (w którym nowe wiersze są oznaczane wyłącznie znakiem specjalnym `\r`).

Uwaga: Działanie składowej `$fix_newlines` nie zmienia zawartości pliku konfiguracyjnego, a jedynie postać odczytanych z niego informacji.

Ogólnie rzecz biorąc, te pliki konfiguracyjne można zapewne pozostawić w niezmienionej postaci i nie będzie się to wiązać z żadnymi konsekwencjami. Pierwszą operacją wykonywaną podczas korzystania z klasy `Config_File` jest określenie miejsca, w którym są przechowywane pliki konfiguracyjne, na których klasa ta ma operować. Ścieżkę tę można podać w wywołaniu konstruktora podczas tworzenia nowego obiektu klasy `Config_File` bądź też przy użyciu metody `set_path()`. Zarówno konstruktor klasy, jak i metoda `set_path()` wymagają przekazania jednego parametru, którym jest właśnie ścieżka dostępu do katalogu przechowującego pliki konfiguracyjne. Po określeniu ścieżki można już pobierać dane konfiguracyjne. Do tego celu jest wykorzystywana metoda `get()`. Poniżej przedstawiłem składnię tej metody:

```
$obiekt->get($nazwapliku[, $sekcja[, $zmienna]])
```

Gdzie parametr `$nazwapliku` określa nazwę pliku konfiguracyjnego, który należy wczytać (przechowywanego w katalogu wskazanym w wywołaniu konstruktora lub metody `set_path()`), parametr `$sekcja` określa nazwę wybranej sekcji zdefiniowanej we wskazanym pliku konfiguracyjnym, a ostatni parametr — `$zmienna` — określa nazwę konkretnej zmiennej, którą chcemy wczytać. Jeśli w wywołaniu metody zostanie pominięty parametr `$zmienna`, to zostaną wczytane wszystkie dane konfiguracyjne podane we wskazanej sekcji. Jeśli jednak nie zostanie podany żaden z opcjonalnych parametrów, to zostaną zwrócone wszystkie zmienne konfiguracyjne nienależące do żadnej z sekcji.



Odwołując się do ukrytych zmiennych i sekcji konfiguracyjnych, w ich nazwach nie należy podawać początkowej kropki! Klasa `Config_File` automatycznie usuwa bowiem te kropki z nazw zarówno zmiennych, jak i sekcji.

Innym sposobem na odczytanie wartości z pliku konfiguracyjnego jest zastosowanie kolejnej metody klasy `Config_File` — `get_key()`. Metoda ta wymaga podania jednego parametru — nazwy zmiennej konfiguracyjnej, której wartość chcemy odczytać; nazwa ta jest podawana w następującym formacie:

```
nazwa pliku/nazwa sekcji/nazwa zmiennej
```

A zatem aby odczytać wartość zmiennej konfiguracyjnej o nazwie `myvalue` zapisanej w pliku *test.ini* w sekcji `mysection`, w wywołaniu metody `get_key()` należałoby użyć następującej nazwy:

```
test.ini/mysection/myvalue
```

Podobnie jak w przedstawionej wcześniej metodzie `get()`, także w nazwach stosowanych w metodzie `get_key()` można pomijać dwa prawe fragmenty (nazwę sekcji oraz nazwę zmiennej); określenie nazwy pliku konfiguracyjnego jest konieczne.

Aby pokazać praktyczny sposób wykorzystania klasy `Config_File`, powróćmy do przykładowego pliku konfiguracyjnego z listingu 7.21 i spróbujmy wczytać podane w nim informacje dotyczące połączenia z bazą danych (listing 7.23).

Listing 7.23. *Zastosowanie klasy Config_File*

```
<?php
    require("Config_File.class.php");

    $config = new Config_File("");
    $dbsettings = $config->get("phphaven_config.ini", "DatabaseSettings");
    echo <<< OUTPUT
```

```
Nazwa użytkownika $dbsettings[user]<BR>
Hasło użytkownika $dbsettings[password]<BR>
OUTPUT;
?>
```

Poniżej podałem kilka innych, przydatnych metod klasy `Config_File`:

`get_file_names()`	Zwraca tablicę zawierającą nazwy plików konfiguracyjnych wczytanych przez ten obiekt `Config_File`.
`get_section_names($plik)`	Zwraca tablicę zawierającą nazwy sekcji dostępnych we wskazanym pliku konfiguracyjnym (określonym przez parametr `$plik`).
`get_var_names($plik[, $sekcja])`	Zwraca tablicę zawierającą nazwy wszystkich wartości zdefiniowanych we wskazanym pliku konfiguracyjnym (określonym przy użyciu parametru `$plik`), które nie należą do żadnej z sekcji, bądź też, jeśli zostanie podany parametr `$sekcja`, nazwy zmiennych należących do wskazanej sekcji.
`clear($plik)`	Usuwa z pamięci wszystkie zmienne konfiguracyjne zdefiniowane we wskazanym pliku.

Podsumowanie

Po przeczytaniu rozdziału Czytelnik zapewne zdaje już sobie sprawę z faktu, iż korzystanie z szablonów nie tylko upraszcza zarządzanie i utrzymanie kodu, lecz także przyczynia się do stosowania dobrych praktyk programistycznych. W tym rozdziale przedstawiłem całe spektrum rozwiązań pozwalających na stosowanie szablonów, zaczynając od funkcji `include()`, a kończąc na pakiecie Smarty, który zawiera swój własny język programowania. Wyłącznie od Czytelnika zależy, które z tych rozwiązań zastosuje w swoich skryptach. Jednak w przypadku wykorzystania szablonów niezwykle prawdziwym okazuje się stare powiedzenie, by nie wytaczać armat na wróble — ponieważ nieprzemyślane lub błędne wykorzystanie szablonów może przysporzyć większych problemów niż całkowita rezygnacja z ich stosowania.