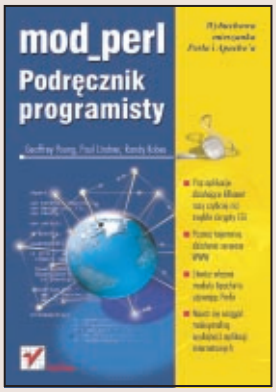
mod_perl. Podręcznik programisty



Autorzy: Geoffrey Young, Paul Lindner, Randy Kobes
Tłumaczenie: Przemysław Kowalczyk
ISBN: 83-7197-799-9
Tytuł oryginału: mod_perl Developer's Cookbook
Format: B5, stron: 564
Przykłady na ftp: 105 kB

Być może Perl jest najlepszym językiem służącym do pisania skryptów CGI, ale skrypty CGI nie są najlepszym sposobem tworzenia aplikacji internetowych. Potrzeba większej wydajności, lepszej integracji z serwerem WWW i pełniejszego wykorzystania jego możliwości doprowadziła do stworzenia modułu mod_perl. Pozwala on na pisanie modułów serwera Apache w Perlu i na pełny dostęp do funkcji API Apache'a z poziomu Perla.

mod_perl jest bardzo rozbudowany, dzięki czemu daje programiście ogromne możliwości. Książka „mod_perl. Kompendium programisty" będzie nieocenioną pomocą w poznawaniu jego potencjału. Nauczysz się z niej podstaw mod_perla, a gdy już je opanujesz, poznasz tajniki pisania dużych i skomplikowanych aplikacji.

W książce opisano między innymi:

- Instalację i konfigurację mod_perla
- Komunikację mod_perla z Apache
- Działania na adresach URL
- Obsługę plików w środowisku mod_perla
- Tworzenie własnych aplikacji w oparciu o mod_perla
- Osiągania maksymalnej wydajności aplikacji używających mod_perla
- Dodatkowe moduły współpracujące z mod_perlem

Po przeczytaniu tej książki uzyskasz nową perspektywę na programowanie aplikacji sieciowych w Perlu. Programiści Slashdot.org, Internet Movie Database i Wired wybrali mod_perl jako platformę do tworzenia aplikacji. Może i Ty powinieneś pójść w ich ślady?

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

Spis treści

Podziękowania ... 9

O Autorach ... 12

Przedmowa ... 13

Wprowadzenie ... 15

Część I Instalacja i konfiguracja ... 19

Rozdział 1. Instalacja modułu mod_perl ... 21

Wprowadzenie ... 21
1.1. Dystrybucja binarna dla Uniksa ... 22
1.2. Instalacja w systemie Windows ... 25
1.3. Instalacja w systemie Mac OS X ... 27
1.4. Kompilacja w systemie Unix ... 27
1.5. Kompilacja w systemie Windows ... 31
1.6. Kompilacja w systemie Mac OS X ... 35
1.7. Kompilacja modułu mod_perl jako biblioteki dzielonej ... 37
1.8. Testowanie instalacji ... 38
1.9. Zmiana katalogów instalacji serwera Apache ... 39
1.10. Dodawanie modułu mod_perl do działającego serwera Apache ... 40
1.11. Ponowne użycie opcji kompilacji ... 40
1.12. Odtwarzanie instalacji modułu mod_perl ... 41
1.13. Instalacja modułu mod_perl na wielu komputerach ... 42
1.14. Sprawdzanie istniejącego serwera ... 45
1.15. Instalacja modułów serwera Apache z archiwum CPAN ... 46
1.16. Śledzenie rozwoju modułu mod_perl ... 47
1.17. Więcej niż dostęp przez CVS ... 48
1.18. Kompilacja modułu mod_perl przy użyciu innej wersji Perla ... 50

Rozdział 2. Konfigurowanie modułu mod_perl 53
Wprowadzenie 53
2.1. Przenoszenie skryptów CGI 53
2.2. Moduł Apache::Registry 55
2.3. Skrypt startup.pl 58
2.4. Dzielenie przestrzeni nazw w środowisku Apache::Registry 61
2.5. Wczesne ładowanie skryptów w środowisku Apache::Registry 62
2.6. Ustawianie zmiennych środowiskowych CGI 63
2.7. Ustawianie innych zmiennych środowiskowych 64
2.8. Ustawianie opcji interpretera Perla 65
2.9. Bloki BEGIN i END w skrypcie startup.pl 66
2.10. Zarządzanie własnymi bibliotekami 67
2.11. Trwałe połączenia z bazą danych 69
2.12. Wcześniejsze nawiązywanie połączeń 70
2.13. Nietrwałe połączenia do bazy danych w środowisku Apache::DBI 72
2.14. Ustawianie zmiennych specyficznych dla modułu mod_perl 73
2.15. Ustawianie bardziej skomplikowanych zmiennych 74
2.16. Dynamiczna konfiguracja serwera Apache 75
2.17. Zachowywanie kolejności w sekcjach <Perl> 77
2.18. Używanie opcji w wierszu poleceń 78
2.19. Uruchamianie podwójnego serwera 79
2.20. Używanie modułu mod_proxy do przekazywania żądań do serwera Apache z modułem mod_perl 80
2.21. Używanie modułu mod_proxy_add_forward 80

Część II Interfejs API modułu mod_perl 83

Rozdział 3. Obiekt żądania 87
Wprowadzenie 87
3.1. Obiekt żądania 87
3.2. Komunikat żądania HTTP 89
3.3. Żądanie klienta 91
3.4. Dostęp do nagłówków żądania 92
3.5. Dostęp do pól formularzy HTML 95
3.6. Dane wysłane metodą POST 97
3.7. Obsługa cookies 98
3.8. Obsługa plików wysyłanych na serwer 100
3.9. Ustawianie nagłówków odpowiedzi serwera 103
3.10. Sterowanie pamięcią podręczną 105
3.11. Wysyłanie nagłówków odpowiedzi serwera 106
3.12. Ustawianie statusu odpowiedzi 108
3.13. Ustawianie nagłówków w przypadku błędu 111
3.14. Nagłówki o wielu wartościach 113
3.15. Żądania wewnętrzne 115
3.16. Ustawianie nagłówków żądania wewnętrznego 117
3.17. Rozpoznawanie żądań wewnętrznych 118
3.18. Metoda HTTP żądania 118
3.19. Dostęp do obiektu żądania z podprogramu XS 120

Rozdział 4. Komunikacja z serwerem Apache ... 127
Wprowadzenie ... 127
4.1. Obiekt Apache::Server ... 127
4.2. Symulowanie dyrektyw IfModule i IfDefine ... 130
4.3. Dostęp do dyrektyw ServerRoot i DocumentRoot ... 132
4.4. Zapis do dziennika błędów ... 134
4.5. Dostęp do dyrektywy ErrorLog ... 136
4.6. Wartość LogLevel ... 138
4.7. Obiekt Apache::Connection ... 140
4.8. Zdalne adresy IP i nazwy serwerów ... 141
4.9. Wykrywanie zerwania połączenia ... 143
4.10. Zamykanie procesu potomnego serwera Apache ... 145

Rozdział 5. Przetwarzanie adresów URI ... 149
Wprowadzenie ... 149
5.1. Żądany adres URI ... 150
5.2. Określanie dyrektywy <Location> dla adresu URI ... 152
5.3. Zmiana żądanego adresu URI ... 155
5.4. Konstruowanie nowego adresu URI ... 157
5.5. Kodowanie znaków specjalnych w adresie URI ... 159
5.6. Wymuszenie typu MIME za pomocą adresu URI ... 161
5.7. Pobieranie zawartości żądania wewnętrznego ... 162
5.8. Użycie klasy Apache::Util poza środowiskiem modułu mod_perl ... 166

Rozdział 6. Obsługa plików ... 169
Wprowadzenie ... 169
6.1. Tworzenie uchwytów plików ... 170
6.2. Tworzenie plików tymczasowych ... 172
6.3. Wysyłanie całego pliku ... 173
6.4. Wczytywanie zawartości plików do zmiennych ... 176
6.5. Pobieranie informacji o żądanym pliku ... 176
6.6. Nagłówki warunkowe ... 180
6.7. Żądania fragmentów plików ... 183
6.8. Nagłówki związane z datami ... 187
6.9. Opróżnianie buforów wyjściowych ... 188
6.10. Przekierowanie uchwytów plików wyjściowych ... 190

Rozdział 7. Tworzenie programów obsługi ... 193
Wprowadzenie ... 193
7.1. Tworzenie programu obsługi ... 194
7.2. Konfiguracja programów obsługi ... 197
7.3. Dodawanie niewielkich programów obsługi ... 199
7.4. Przygotowanie modułu do publikacji ... 201
7.5. Tworzenie archiwum programu TAR ... 202
7.6. Tworzenie binarnej dystrybucji PPM ... 204
7.7. Testowanie modułu ... 207
7.8. Własne dyrektywy konfiguracyjne ... 214
7.9. Rozszerzanie prototypów własnych dyrektyw ... 223
7.10. Łączenie własnych dyrektyw ... 225
7.11. Zastępowanie dyrektyw rdzeniowych ... 231
7.12. Dodawanie znaczników serwera ... 236
7.13. Publikowanie modułu w archiwum CPAN ... 237

Rozdział 8. Współpraca z programami obsługi ... 239
Wprowadzenie ... 239
8.1. Wykrywanie zmian programów obsługi ... 239
8.2. Dzielenie danych wewnątrz procesu potomnego ... 241
8.3. Tworzenie dzielonego bufora ... 244
8.4. Zachowywanie stanu ... 247
8.5. Wewnętrzne przekierowania ... 251
8.6. Tworzenie własnych stron o błędach ... 254
8.7. Przywracanie domyślnych stron o błędach ... 257
8.8. Łańcuchy programów obsługi ... 259
8.9. Łańcuchy programów obsługi w języku C ... 261
8.10. Dostęp do zmiennych środowiskowych ... 264
8.11. Dzielenie danych między fazami ... 265
8.12. Określanie aktualnej fazy żądania ... 268
8.13. Dane konfiguracyjne modułu Perla ... 269
8.14. Dane konfiguracyjne modułu języka C ... 270

Rozdział 9. Dostrajanie serwera Apache i modułu mod_perl ... 275
Wprowadzenie ... 275
9.1. Zbieranie podstawowych informacji o serwerze ... 277
9.2. Tworzenie raportu zużycia pamięci ... 281
9.3. Zużycie pamięci przez procesy serwera Apache ... 283
9.4. Bardziej szczegółowe informacje o zużyciu pamięci przez procesy serwera ... 284
9.5. Zużycie pamięci przez moduły Perla ... 286
9.6. Redukcja narzutu przy imporcie modułów ... 288
9.7. Zmniejszanie całkowitego zużycia pamięci ... 289
9.8. Zwiększanie obszaru pamięci dzielonej ... 291
9.9. Regulacja liczby procesów potomnych ... 293
9.10. Ograniczanie wzrostu zużycia pamięci przez procesy ... 294
9.11. Zamykanie niekontrolowanych procesów ... 296
9.12. Profilowanie programów obsługi ... 298
9.13. Znajdowanie wąskich gardeł wydajności ... 299
9.14. Dostrajanie wydajności serwera ... 301
9.15. Serwer Apache jako serwer proxy ... 305
9.16. Używanie programu uruchomieniowego Perla z modułem mod_perl ... 308
9.17. Wyszukiwanie błędów w skryptach Apache::Registry ... 310
9.18. Redukcja narzutu uruchomieniowego ... 311
9.19. Wyszukiwanie błędów przy naruszeniach segmentacji ... 313

Rozdział 10. Programowanie obiektowe przy użyciu modułu mod_perl ... 315
Wprowadzenie ... 315
10.1. Tworzenie klas i obiektów ... 316
10.2. Dziedziczenie metod ... 318
10.3. Tworzenie obiektowych programów obsługi ... 321
10.4. Używanie obiektowych programów obsługi ... 323
10.5. Dziedziczenie po klasie Apache ... 326
10.6. Dziedziczenie po klasie Apache przy użyciu modułów XS ... 328
10.7. Dziedziczenie po klasie Apache::Registry ... 330
10.8. Dziedziczenie po klasie Apache::Request ... 333

Część III Oprogramowywanie cyklu życiowego serwera Apache 339

Rozdział 11. PerlInitHandler 345
Wprowadzenie 345
11.1. Przetwarzanie każdego żądania 346
11.2. Przetwarzanie każdego żądania w danej dyrektywie zbiorczej 347
11.3. Mierzenie czasu żądania 348
11.4. Przerywanie cyklu obsługi żądania 350

Rozdział 12. PerlTransHandler 353
Wprowadzenie 353
12.1. Żądania pliku favicon.ico 354
12.2. Rozpoznawanie serwerów wirtualnych w żądaniach 355
12.3. Identyfikatory sesji w adresach URL 358
12.4. Współdzielenie dyrektywy DocumentRoot 360
12.5. Sterowanie wbudowanym serwerem proxy 362
12.6. Redukcja wywołań funkcji stat() 364

Rozdział 13. PerlAccessHandler, PerlAuthenHandler i PerlAuthzHandler 371
Wprowadzenie 371
13.1. Prosta kontrola dostępu 372
13.2. Ograniczanie dostępu „chciwym" klientom 375
13.3. Identyfikacja podstawowa 376
13.4. Ustawianie danych użytkownika 379
13.5. Warunkowa identyfikacja 381
13.6. Autoryzacja użytkownika 383
13.7. Tworzenie własnego mechanizmu autoryzacji 386
13.8. Identyfikacja przy użyciu funkcji skrótu 392

Rozdział 14. PerlTypeHandler i PerlFixupHandler 401
Wprowadzenie 401
14.1. Przywracanie domyślnego programu obsługi generowania zawartości 402
14.2. Wybór programu obsługi na podstawie rozszerzenia nazwy pliku 404
14.3. Zmiana typu MIME i programu obsługi 409
14.4. Zmiana domyślnych typów MIME 413
14.5. Własny mechanizm buforujący 414

Rozdział 15. PerlHandler 421
Wprowadzenie 421
15.1. Podstawowy PerlHandler 422
15.2. Zarządzanie wieloma programami obsługi typu PerlHandler 425
15.3. Wysyłanie poczty 427
15.4. Filtrowanie generowanej zawartości 431
15.5. Zapobieganie atakom skryptowym 435
15.6. Moduł Text::Template 439
15.7. Moduł HTML::Template 443
15.8. Moduł Apache::ASP 445
15.9. Pakiet Template Toolkit 450
15.10. Moduł HTML::Embperl 454
15.11. Moduł HTML::Mason 458
15.12. Generowanie dokumentów XML 461

15.13. Generowanie ogólnych dokumentów XML ... 464
15.14. Dokumenty XML i arkusze XSLT ... 467
15.15. Pakiet AxKit ... 470
15.16. Tworzenie serwera SOAP ... 472

Rozdział 16. PerlLogHandler i PerlCleanupHandler ... 481
Wprowadzenie ... 481
16.1. Dziennik w bazie danych ... 482
16.2. Dziennik w zwykłym pliku ... 485
16.3. Zmiana wiersza żądania ... 488
16.4. Zapisywanie niestandardowych informacji ... 489
16.5. Rejestrowanie warunkowe ... 490
16.6. Przechwytywanie błędów ... 491

Rozdział 17. PerlChildInitHandler, PerlChildExitHandler, PerlRestartHandler i PerlDispatchHandler ... 499
Wprowadzenie ... 499
17.1. Konfiguracja kodu poza obsługą żądania ... 501
17.2. Uruchamianie kodu podczas restartu serwera ... 503
17.3. Jednokrotne ładowanie konfiguracji ... 504
17.4. Przeładowywanie skryptów Registry w procesie nadrzędnym ... 506
17.5. Identyfikacja procesów potomnych ... 507
17.6. Wczesne łączenie ze źródłem danych ... 509
17.7. Śledzenie użycia modułów Perla ... 511
17.8. Zastępowanie programów obsługi ... 512

Dodatki ... 517

Dodatek A Dostępne punkty zaczepienia i opcje kompilacji modułu mod_perl ... 519
Punkty zaczepienia modułu mod_perl ... 519
Opcje kompilacji modułu mod_perl ... 523

Dodatek B Dostępne stałe ... 531
Wartości zwracane przez programy obsługi ... 531
Stałe określone przez protokół HTTP ... 531
Stałe używane przez programy obsługi dyrektyw ... 533
Stałe sterujące zapisem w dzienniku ... 536
Stałe serwera ... 536

Dodatek C Zasoby związane z modułem mod_perl ... 537
Zasoby sieciowe ... 537
Książki ... 540

Skorowidz ... 543

6.

Obsługa plików

Wprowadzenie

Podczas obsługi każdego żądania serwera Apache nasza aplikacji musi czytać i przetwarzać zawartość plików na dysku. W Perlu można to zrealizować wieloma sposobami. Aplikacje WWW jednak, a w szczególności aplikacje modułu mod_perl, mają specjalne wymagania, które najlepiej wypełnia nowy interfejs obsługi plików. Zadania w tym rozdziale przedstawiają typowe problemy i rozwiązania spotykane przy posługiwaniu się plikami.

Apache zawiera interfejs API obsługi plików zoptymalizowany pod kątem działania serwera WWW. Moduł mod_perl udostępnia elegancki, obiektowy interfejs do tych funkcji w klasie `Apache::File`. Korzystając z tej klasy, nasza aplikacja zyska na jakości.

- **Działa szybciej**. Klasa `Apache::File` używa skompilowanego kodu języka C, aby wykonać większość zadań.
- **Jest bardziej stabilna**. Pliki tymczasowe i zasoby tworzone dla żądania są automatycznie czyszczone.
- **Pełniej wykorzystuje możliwości protokołu HTTP**. Klasa `Apache` (a w konsekwencji także `Apache::File`) obsługuje zaawansowane możliwości protokołu HTTP/1.1, takie jak żądania fragmentów (*byte range*) plików czy nowe nagłówki.

Ten rozdział zawiera także recepty na typowe sytuacje.

- Konwersja dat modyfikacji plików (i dowolnych innych) na odpowiednie nagłówki HTTP.
- Opróżnienie bufora danych wyjściowych i wysłanie ich do klienta przed zakończeniem przetwarzania.
- Przekierowanie wyjścia istniejącego uchwytu pliku (jak `STDOUT` czy `STDERR`).

Omówienie klasy `Apache::File` stanowi koniec naszego wprowadzenia do klas rdzeniowych modułu mod_perl. Kolejne rozdziały pokażą, jak posługiwać się nimi w konkretnych aplikacjach.

6.1. Tworzenie uchwytów plików

Chcemy utworzyć nowy uchwyt pliku do czytania lub pisania.

Rozwiązanie

Użyjemy metod `new()` i `open()` klasy `Apache::File`, która stanowi obiektowy interfejs do uchwytów plików (*filehandle*).

Wydruk 6.1. *Przykładowy program obsługi*

```
use Apache::Constants qw(OK NOT_FOUND);
use Apache::File;

use strict;

sub handler {

  my $r = shift;

  # Otwieramy żądany plik w trybie tylko do odczytu.
  my $fh = Apache::File->new($r->filename);

  return NOT_FOUND unless $fh;

  # Jakieś przetwarzanie...
  $r->send_http_header;
  $r->send_fd($fh);

  # Nie trzeba zamykać uchwytu pliku, jest on zamykany automatycznie,
  # kiedy wychodzi z zasięgu widoczności.
  return OK;
}
```

Komentarz

Jest wiele sposobów obsługi wejścia-wyjścia plikowego w Perlu. Najczęściej używa się modułów *FileHandle.pm* i `IO::File` oraz funkcji `open()` i `sysopen()`. Klasa `Apache::File` stanowi jeszcze jedno rozwiązanie, dostarczając obiektowy interfejs do uchwytów plików, podobny do modułów *FileHandle.pm* i `IO::File`. Stylistycznie klasa `Apache::File` dobrze wkomponowuje się w moduł mod_perl, ponieważ większą część jego interfejsu API stanowią wywołania metod

różnych klas, więc „obiektowy" dostęp do plików rozjaśnia kod. Dodatkowo klasa `Apache::File` posiada zaletę w postaci większej wydajności, nie musimy się więc przejmować spowolnieniem operacji na plikach, jak w przypadku modułu `IO::File`.

Konstruktor `new()` zwraca nowy uchwyt pliku. Jeżeli parametrem jest nazwa pliku, jak w naszym przykładzie (wydruk 6.1), wywołuje metodę `Apache::File->open()` i zwraca otwarty uchwyt pliku. Domyślnie pliki są otwierane w trybie tylko do odczytu (znacznik `O_RDONLY`), ale możemy użyć tych samych parametrów w metodzie `Apache::File->open()` co w perlowej funkcji `open()`. Chodzi tu oczywiście o starszą wersję tej funkcji, nie tę z trzema parametrami, wprowadzoną w wersji 5.6 Perla.

```
my $fh = Apache::File->new;

# Otwieramy w trybie do dopisywania.
$fh->open('>>'.$r->server_root_relative('logs/access_log'))
  or return SERVER_ERROR;
```

Jedną z zalet używania konstruktora `new()` w stosunku do metody `open()` jest zwracanie wartości `undef` w przypadku błędu, co pozwala stosować prosty mechanizm obsługi sytuacji wyjątkowych.

Poniższa tabela przedstawia listę metod klasy `Apache::File`.

Tabela 6.1. *Metody klasy Apache::File*

Metoda	Opis
`new()`	Tworzy nowy uchwyt pliku, opcjonalnie otwierając wskazany plik.
`open()`	Otwiera wskazany plik.
`close()`	Zamyka uchwyt pliku.
`tmpfile()`	Tworzy plik tymczasowy i zwraca jego nazwę i uchwyt w kontekście listowym albo tylko uchwyt w kontekście skalarnym.

Chociaż klasa `Apache::File` umożliwia wygodną obsługę uchwytów plików, jak również dodatkowe korzyści, które opisujemy w kolejnych zadaniach, posiada niestety pewne ograniczenia. Między innymi nie implementuje wszystkich metod, których moglibyśmy wymagać od uchwytu pliku. To utrudnienie wychodzi na jaw w zadaniu 6.6, kiedy klasa `File::MMagic` wymaga wywołania metody `$fh->seek()`. Oczywiście uchwyt pliku, utworzony przez klasę `Apache::File`, jest normalnym, perlowym uchwytem pliku, więc zawsze możemy na nim wywołać perlową funkcję `seek()` w sposób „nieobiektowy".

Innym utrudnieniem jest nakład czasu w trakcie wykonywania spowodowany przez interfejs obiektowy. Jeżeli zdecydujemy się pozostać przy perlowej funkcji `open()`, możemy skorzystać z automatycznego tworzenia anonimowych referencji (*autovivification*), wprowadzonego w wersji 5.6 Perla, co pozwala opuścić wywołanie metody `Symbol::gensym()`. Jeżeli jednak używamy starszej wersji Perla i chcemy użyć funkcji `open()`, moduł mod_perl udostępnia metodę `Apache->gensym()`, byśmy nie musieli dołączać modułu `Symbol` do naszego programu obsługi.

```
my $fh = Apache->gensym;
open($fh, "layline.html");
```

6.2. Tworzenie plików tymczasowych

Chcemy utworzyć plik tymczasowy, który istnieje tylko podczas przetwarzania żądania.

Rozwiązanie

Użyjemy metody `tmpfile()` z klasy `Apache::File`.

Wydruk 6.2. *Moduł Rules.pm*

```
package Cookbook::Rules;

use Apache::Constants qw(OK);
use PDF::Create;

use strict;

sub handler {

  my $r = shift;

  my ($filename, $fh) = Apache::File->tmpfile;

  my $pdf = PDF::Create->new(filename => $filename);

  my $page = $pdf->new_page;

  $page->string($pdf->font, 20, 1, 770, "mod_perl jest fajny!");

  $pdf->close;

  # Ustawiamy wskaźnik pliku na początku.
  seek $fh, 0, 0;

  $r->set_content_length(-s $filename);

  $r->send_http_header('application/pdf');

  # Wysłamy plik.
  $r->send_fd($fh);

  # Plik $filename jest usuwany po zakończeniu żądania.
  return OK;
}
1;
```

Komentarz

Czasem potrzebujemy pliku tymczasowego, na przykład kiedy tworzymy duży dokument i nie chcemy przechowywać go w pamięci przed wysłaniem do klienta. W takiej sytuacji (i w podobnych) metoda `tmpfile()` stanowi wygodny sposób tworzenia plików tymczasowych, które są usuwane po zakończeniu przetwarzania żądania.

Metoda `tmpfile()` może zostać wywołana na dwa sposoby: w kontekście listowym zwraca nazwę nowego pliku i otwarty uchwyt, a w kontekście skalarnym — tylko uchwyt. W obu przypadkach plik otwierany jest przy użyciu znaczników `O_RDWR|O_CREAT|O_EXCL`, czyli w trybie do odczytu i zapisu.

Pliki tymczasowe utworzone w ten sposób różnią od plików tworzonych przez metodę `IO::File->new_tmpfile()` pod dwoma względami: mamy dostęp do nazwy pliku, a sam plik nie jest usuwany, kiedy jego uchwyt wychodzi z zasięgu widoczności. Są to wymarzone cechy dla programistów modułu mod_perl, pozwalające łatwo użyć tego samego pliku tymczasowego w różnych fazach przetwarzania żądania. Możemy na przykład skorzystać z metody `pnotes()`, opisanej w zadaniu 8.11, do przekazania nazwy lub uchwytu do pliku tymczasowego w łańcuchu programów obsługi.

```
# Zapamiętujemy nazwę pliku tymczasowego "na później".
$r->pnotes(TEMPFILE => $filename);
```

Warto też zauważyć, że wywołanie funkcji `seek()` nie jest niezbędne w naszym przykładzie (wydruk 6.2), gdyż moduł `PDF::Create` nie korzysta bezpośrednio z uchwytu pliku utworzonego za pomocą metody `tmpfile()`. W ogólnym przypadku jednak jeżeli chcemy pisać do wygenerowanego pliku tymczasowego (albo jakiegokolwiek innego), a później wypisać jego zawartość przy użyciu tego samego uchwytu pliku, musimy użyć perlowej funkcji `seek()`, aby ustawić wskaźnik pliku z powrotem na jego początku, jak w poniższym przykładzie (wydruk 6.3).

Wydruk 6.3. *Użycie funkcji seek()*

```
my $fh = Apache::File->tmpfile;

print $fh 'Mów mi Ishmael.';

# Przestawiamy wskaźnik pliku na początek.
seek $fh, 0, 0;

# Wysyłamy plik do klienta.
$r->send_fd($fh);
```

6.3. Wysyłanie całego pliku

Chcemy wysłać cały plik do klienta.

Rozwiązanie

Użyjemy metody `Apache->send_fd()`.

Wydruk 6.4. *Przykładowy skrypt*

```
my $fh = Apache::File->new('enchilada.html');

if (my $length = $r->param('length')) {
  # Wysyłamy część pliku...
  $r->send_fd($fh, $length);
}
else {
  # ...albo cały.
  $r->send_fd($fh)
}
```

Komentarz

Wszystkie dotychczasowe przykłady w tym rozdziale używały metody `$r->send_fd()`, aby przesłać plik bezpośrednio do klienta. Zazwyczaj można się spotkać z użyciem funkcji `print()` do wypisania zawartości pliku, jak na przykład:

```
print while <$fh>;
```

Ponieważ jednak wysyłanie pliku do klienta jest często potrzebne w aplikacjach WWW, metoda `send_fd()` pozwala na wykonanie tej czynności łatwo i efektywnie. Jej parametrem jest otwarty uchwyt pliku; używa ona interfejsu API języka C serwera Apache, aby wysłać zawartość pliku do przeglądarki klienta możliwie wydajnie. Zwraca długość w bajtach przesłanych danych na wypadek, gdybyśmy chcieli znać różnicę między liczbą wszystkich wysłanych bajtów a pochodzących z pliku. Drugim opcjonalnym parametrem może być liczba bajtów do wysłania, używa się go rzadko, ale w pewnych warunkach jest użyteczny, co można zobaczyć w zadaniu 6.7.

Rozważmy na przykład sytuację, w której chcielibyśmy uruchomić usługę, umożliwiającą (zaufanemu) klientowi zażądanie zawartości pliku konfiguracyjnego serwera. Moglibyśmy zrealizować to za pomocą następującego programu obsługi (wydruk 6.5).

Wydruk 6.5. *Moduł ViewConf.pm*

```
package Cookbook::ViewConf;

use Apache::Constants qw(OK SERVER_ERROR NOT_FOUND);
use Apache::File;

use strict;

sub handler {

  my $r = shift;
```

```
  # Pobieramy nazwę żądanego pliku.
  my $file = $r->filename;

  # Upewniamy się, że istnieje.
  unless (-f $r->finfo) {
    $r->log_error("Plik $file nie istnieje.");
    return NOT_FOUND;
  }

  # Otwieramy uchwyt pliku.
  my $fh = Apache::File->new($file);

  unless ($fh) {
    $r->log_error("Nie można otworzyć pliku $file: $!");
    return SERVER_ERROR;
  }

  $r->send_http_header('text/plain');

  # Pobieramy rozmiar pliku i wysyłamy go.
  my $size = -s _;
  my $sent = $r->send_fd($fh);

  $r->print(<<"END");

--------------------------------------------------
Rozmiar pliku: $size
Wysłano bajtów: $sent
--------------------------------------------------

END

  return OK;
}
1;
```

Aby uruchomić moduł `Cookbook::ViewConf`, należy użyć następujących dyrektyw w pliku *httpd.conf*:

```
PerlModule Cookbook::ViewConf

Alias /conf /usr/local/apache/conf
<Location /conf>
  SetHandler perl-script
  PerlHandler Cookbook::ViewConf
  # Dostęp tylko z zaufanych źródeł.
  Order Deny,Allow
  Deny from All
  Allow from localhost
</Location>
```

W ten sposób klient może otrzymać tekstową wersję pliku konfiguracyjnego serwera, przykładowo *httpd.conf*, żądając adresu URI *http://localhost/conf/httpd.conf*. Pod zawartością pliku raportowany jest jeszcze rozmiar pliku na dysku serwera i liczba wysłanych bajtów. Te dwie liczby mogą się różnić, na przykład w systemie Win32, z powodu użycia różnych sekwencji nowego wiersza.

6.4. Wczytywanie zawartości plików do zmiennych

Chcemy przechowywać zawartość całego pliku w zmiennej, aby móc na niej operować.

Rozwiązanie

Użyjemy perlowego idiomu `local $/`, aby „wessać" (*slurp*) cały plik, ale należy przy tym zachować ostrożność!

```
my $fh = Apache::File->new($filename);

my $file = do {local $/; <$fh>}
```

Komentarz

Lokalizacja specjalnej zmiennej `$/` jest to perlowy idiom, służący do wczytywania całej zawartości pliku do zmiennej tekstowej. Aby być wydajnym programistą Perla, należy znać ten idiom i podobne. W przypadku modułu mod_perl trzeba jednak głębiej zastanowić się nad jego znaczeniem.

Jak już pisaliśmy w rozdziale 2., moduł mod_perl jest tak użyteczny między innymi dlatego, że interpreter Perla jest wbudowany w serwer Apache. Ma to wiele zalet, jak na przykład zmniejszenie nakładu czasowego za każdym uruchomieniem skryptu środowiska `Apache::Registry`. Jedną z największych wad takiego rozwiązania jest jednak fakt, że pamięć, której używa interpreter Perla, nie jest zwracana do systemu operacyjnego, dopóki nie zakończy się działanie odpowiedniego procesu potomnego *httpd*. Oznacza to, że jeżeli jakiś beztroski program obsługi postanowi wczytać 10-megabajtowy plik do zmiennej, to pamięć, której zmuszony będzie użyć interpreter Perla, nie zostanie zwolniona dopóki nie zostanie zakończony proces potomny serwera Apache.

Generalnie należy więc unikać operowania na zawartości całych plików jako zmiennych i starać się zrealizować pożądaną funkcjonalność inaczej. Zdarzają się jednak sytuacje, kiedy nie istnieje inna możliwość, jak w przypadku pobierania całych plików z bazy danych (jak w zadaniu 3.11) albo gdy używamy klasy `Apache::Filter` (która zapisuje wygenerowaną zawartość w zmiennej). Jeżeli koniecznie potrzebujemy takiej funkcjonalności w naszej aplikacji, powinniśmy upewnić się, że stosujemy odpowiedni mechanizm utrzymywania rozmiarów procesów potomnych w ryzach, na przykład `Apache::SizeLimit`.

6.5. Pobieranie informacji o żądanym pliku

Chcemy użyć funkcji `stat()` na żądanym pliku albo przeprowadzić testy.

Rozwiązanie

Użyjemy metody `$r->finfo()`, aby przeprowadzić testy i zastąpić wywołania funkcji `stat()` bezpośrednio funkcją `$r->filename()`.

Wydruk 6.6. *Moduł XBitHack.pm*

```
package Cookbook::XBitHack;

use Apache::Constants qw(OK DECLINED OPT_INCLUDES);
use Apache::File;

use Fcntl qw(S_IXUSR S_IXGRP);

use strict;

sub handler {
  # Implementujemy dyrektywę "XBitHack full" w programie PerlFixupHandler.

  my $r = shift;

  return DECLINED unless
    (-f $r->finfo                    &&    # plik istnieje
     $r->content_type eq 'text/html' &&    # i jest dokumentem HTML
     $r->allow_options & OPT_INCLUDES);    # i ustawiono opcję +Includes

  # Sprawdzamy, czy plik jest wykonywalny dla właściciela i jego grupy.
  my $mode = (stat _)[2];

  # Wykonywalność dla właściciela musimy sprawdzić osobno.
  return DECLINED unless ($mode & S_IXUSR);

  # Ustawiamy nagłówek Last-Modified, jeżeli plik jest wykonywalny dla grupy.
  $r->set_last_modified((stat _)[9]) if ($mode & S_IXGRP);

  # Upewniamy się, że moduł mod_include zajmie się żądaniem.
  $r->handler('server-parsed');

  return OK;
}
1;
```

Komentarz

Jak już wiemy, metoda `$r->filename()` zwraca nazwę fizycznego pliku dla żądania, której można użyć w rozmaitych operacjach testowych na pliku czy w funkcji `stat()`. Jednak metoda `$r->finfo()` stanowi efektywniejszy sposób uzyskiwania tej samej informacji i oszczędza czas przy wielu wywołaniach funkcji `stat()`, która zużywa dużo zasobów systemu. Użycie jej dowodzi przy okazji, że posiedliśmy biegłość w posługiwaniu się zaawansowanymi elementami modułu mod_perl.

Kiedy serwer Apache zmapuje żądany adres URI na plik fizyczny, wywołuje funkcję `stat()` dla własnych potrzeb i gromadzi informację w polu `finfo` rekordu żądania. Kiedy wywoływana jest metoda `$r->finfo()`, moduł mod_perl wydobywa tę informację z rekordu żądania, wewnętrznie wypełnia specjalny perlowy uchwyt pliku _ i zwraca go. Ponieważ uchwyt _ używany jest do buforowania informacji dla przyszłych wywołań funkcji `stat()`, programiści modułu mod_perl mogą uniknąć straty czasu, jaka zazwyczaj towarzyszy sprawdzaniu, czy plik istnieje, pobieraniu czasu ostatniej modyfikacji i tym podobnym.

Program obsługi `Cookbook::XBitHack` (wydruk 6.6) stanowi implementację dyrektywy `XBitHack` z modułu mod_include. Standardowo dyrektywa ta pozwala administratorowi serwera Apache wskazać, które pliki są przetwarzane przez mechanizm SSI (*Server Side Include engine* — serwerowy mechanizm włączania plików) w oparciu o uprawnienia dostępu do pliku i dyrektywę `Options`. Aby zaimplementować całą funkcjonalność dyrektywy `XBitHack full` przy minimum wysiłku, czynimy użytek z „dróg na skróty", które zapewnia moduł mod_perl i tym podobnych sztuczek.

Pierwszym z wywołań funkcji opartych o `stat()` jest operator testu pliku `-f`, który sprawdza, czy plik istnieje i jest zwykłym plikiem. Ponieważ parametrem tego operatora jest `$r->finfo()`, „obchodzimy" w ten sposób wywołanie systemowej funkcji `stat()`, używając informacji przygotowanej przez serwer Apache. Pozostałe wywołania `stat()` używają uchwytu _, świeżo zainicjalizowanego przez `$r->finfo()`, dzięki czemu korzystamy z wewnętrznego bufora interpretera Perla i oszczędzamy na wywołaniach metody `$r->finfo()`.

Ponieważ oryginalna dyrektywa `XBitHack` rozróżnia uprawnienia dla właściciela i grupy, operator testu `-x` nie przyda się nam, jeżeli chcemy zachować z nią zgodność. Porównujemy więc uprawnienia do pliku, zwrócone przez funkcję `stat()`, z odpowiednimi stałymi, zaimportowanymi z pakietu `Fcntl`, aby wyizolować uprawnienie do wykonywania pliku przez właściciela i grupę.

W naszym programie obsługi pozostaje już tylko sprawdzić wartość dyrektywy `Options`, ustawić nagłówek `Last-Modified` i upewnić się, że moduł mod_include zajmie się fazą generowania zawartości. Aby sprawdzić ustawienie dyrektywy `Options`, używamy operatora koniunkcji bitowej `&` na wartości zwróconej przez metodę `$r->allows_options()` i jeszcze jednej stałej z modułu `Apache::Constants`. Metody tej używa się bardzo rzadko w programach modułu mod_perl, ale przydaje się w sytuacjach, takich jak ta, kiedy chcemy wymusić ustawienia pliku *.htaccess*. Użycie metody `set_last_modified()` jest dokładniej opisane w następnym zadaniu, a użycie `$r->handler()` — w podrozdziale 14.1.

Moduł `Cookbook::XBitHack`, użyty jako program obsługi typu `PerlFixupHandler`, ma identyczną funkcjonalność, jak moduł mod_include, z jednym istotnym wyjątkiem. W systemach Win32 nie istnieje rozróżnienie między uprawnieniami do pliku dla właściciela i grupy, więc moduł mod_include stosuje specjalną obsługę dla tej platformy (i kilku innych). Sprawdza wtedy po prostu tylko, czy użytkownik może wykonywać dany plik i zawsze ustawia nagłówek `Last-Modified`. Chociaż wydaje się to rozsądnym rozwiązaniem, nie rozwiązuje jeszcze problemu użytkowników Windows, gdyż system ten uważa za pliki wykonywalne tylko te, które mają odpowiednie rozszerzenie, jak *.exe* czy *.bat*. Tak więc, chyba że używamy SSI do przetwarzania dokumentów o nazwie typu *index.exe*, dyrektywa `XBitHack` staje się bezużyteczna na platformie Win32 mimo „najlepszych intencji" modułu mod_include.

Poniżej (wydruk 6.7) prezentujemy alternatywę dla implementacji dyrektywy XBitHack z modułu mod_include w wersji dostosowanej do specyfiki systemu Win32.

Wydruk 6.7. *Moduł WinBitHack.pm*

```
package Cookbook::WinBitHack;

use Apache::Constants qw(OK DECLINED OPT_INCLUDES);
use Apache::File;

use Win32::File qw(READONLY ARCHIVE);

use strict;

sub handler {
  # Implementujemy dyrektywę "XBitHack full" w programie PerlFixupHandler,
  # wersja dla systemu Win32.

  my $r = shift;

  return DECLINED unless (
     -f $r->finfo                    &&    # plik istnieje
     $r->content_type eq 'text/html' &&    # i jest dokumentem HTML
     $r->allow_options & OPT_INCLUDES);    # i ustawiono opcję +Includes

  # Pobieramy atrybuty pliku.
  my $attr;
  Win32::File::GetAttributes($r->filename, $attr);

  # Zwracamy stałą DECLINED, jeżeli plik ma ustawiony atrybut ARCHIVE.
  return DECLINED if $attr & ARCHIVE;

  # Ustawiamy nagłówek Last-Modified, jeżeli nie jest ustawiony atrybut READONLY.
  $r->set_last_modified((stat _)[9]) unless $attr & READONLY;

  # Upewniamy się, że moduł mod_include zajmie się żądaniem.
  $r->handler('server-parsed');

  return OK;
}
1;
```

Zamiast użyć uprawnień do pliku, moduł Cookbook::WinBitHack sprawdza atrybuty ARCHIVE (gotowy do archiwizacji) i READONLY (tylko do odczytu) przy użyciu pakietu Win32::File, dostępnego w dystrybucji *libwin32* w archiwum CPAN. Jeżeli atrybut ARCHIVE *nie jest* ustawiony, nasz program obsługi przekazuje go modułowi mod_include do przetworzenia. Ponieważ atrybut ARCHIVE trzeba usunąć z utworzonego pliku celowo, schemat działania dyrektywy XBitHack pozostaje bez zmian. Nasza nowa implementacja ustawia także nagłówek Last-Modified, ale tylko, gdy nie jest ustawiony atrybut READONLY — jeżeli plik nie może być zmodyfikowany, nie kłopoczemy się ustawianiem nagłówka.

W zależności od wersji systemu operacyjnego może być wiele sposobów przełączania atrybutów pliku, ale najbardziej uniwersalne jest użycie programu ATTRIB z wiersza poleceń:

```
C:\Apache\htdocs> ATTRIB -A underway.html
```

6.6. Nagłówki warunkowe

Chcemy właściwie posługiwać się nagłówkami warunkowymi, na przykład wysyłać w odpowiedzi nagłówek `Last-Modified`*, czy sprawdzać nagłówek żądania* `If-Modified-Since`*.*

Rozwiązanie

Użyjemy metod dodanych do klasy `Apache` przez klasę `Apache::File`, jak `set_last_modified()` czy `meets_condition()` (spełnia warunek).

Wydruk 6.8. *Moduł SendSmart.pm*

```
package Cookbook::SendSmart;

use Apache::Constants qw( OK NOT_FOUND );
use Apache::File;

use File::MMagic;
use IO::File;

use strict;

sub handler {
  # Wysyłamy plik statyczny z odpowiednimi nagłówkami tylko, kiedy
  # klient posiada starą wersję.

  my $r = shift;

  # Nie Apache::File->new() ponieważ File::MMagic potrzebuje $fh->seek().
  my $fh = IO::File->new($r->filename);

  return NOT_FOUND unless $fh;

  # "Magicznie" ustawiamy typ MIME.
  $r->content_type(File::MMagic->new->checktype_filehandle($fh));

  # Ustawiamy nagłówek Last-Modified na podstawie czasu modyfikacji pliku...
  $r->set_last_modified((stat $r->finfo)[9]);

  # ...oraz nagłówki Etag i Content-Length.
  $r->set_etag;
  $r->set_content_length;

  # Jeżeli wszystkie warunki z nagłówków If-* są spełnione, wysyłamy nagłówki.
  # W przeciwnym razie zwracamy status do serwera Apache.
  if ((my $status = $r->meets_conditions) == OK) {
    $r->send_http_header;
  }
  else {
    return $status;
  }
```

```
    # Ustawiamy wskaźnik pliku na początek i wysyłamy zawartość.
    seek $fh, 0, 0;
    $r->send_fd($fh);

    return OK ;
}
1;
```

Komentarz

W zadaniu 3.10 pokazaliśmy, jak, używając metody `no_cache()`, ograniczyć „nadgorliwość" przeglądarki klienta w buforowaniu oglądanych dokumentów, teraz zobaczymy, jak „przekonać" przeglądarkę do użycia lokalnej kopii dokumentu, kiedy tylko to możliwe. W tym celu będziemy sprawdzać i ustawiać zestaw odpowiednich nagłówków.

Częścią specyfikacji protokołu HTTP/1.1 jest pojęcie *warunkowego żądania GET* (*conditional GET request*), czyli żądania przy użyciu metody `GET` opartego na dodatkowej informacji, zawartej w nagłówkach żądania i odpowiedzi. Nowoczesne przeglądarki zapamiętują odpowiedzi serwera, jak również dodatkowe informacje o żądaniu, w pamięci podręcznej. Informacje te są wysyłane z następnymi żądaniami w celu ograniczenia przesyłu danych.

Obsługa żądań, które *mogą* wygenerować odpowiedź, jest skomplikowana: samo przeczytanie opisu nagłówków z rodziny `If-*` w dokumencie RFC 2616 może przyprawić o ból głowy. Na szczęście interfejs API serwera Apache dostarcza kilku metod, które zajmują się analizą i ustawianiem nagłówków warunkowych. Kod tych metod, jak również wyjaśnienia, pomocne w „rozszyfrowaniu" specyfikacji HTTP/1.1, znajdują się w pliku *http_protocol.c* w dystrybucji kodu źródłowego serwera Apache. Jak zwykle, możemy dostać się do tych metod dzięki modułowi mod_perl, w tym przypadku za pośrednictwem klasy `Apache::File`.

Tabela 6.2 przedstawia metody dostępne przez obiekt żądania. Inaczej niż do pozostałych metody klasy `Apache`, dostęp do tych jest możliwy dopiero po użyciu instrukcji `use Apache::File`.

Dla dokumentów statycznych odpowiednimi nagłówkami warunkowymi żądania i odpowiedzi zajmuje się domyślny program obsługi serwera Apache. Przetwarzanie ich przez aplikacje generujące dynamiczną zawartość wymaga trochę więcej wysiłku niż po prostu wywoływanie wyżej wymienionych metod. Należy najpierw zadecydować, co ma wpływ na zawartość, którą generujemy: dane źródłowe, ich zmiany czy też inne czynniki, które mogą być subtelne, ale ważne.

Moduł `Cookbook::SendSmart` (wydruk 6.8) pokazuje, jak w prostym programie obsługi zawartości użyć metod obsługujących nagłówki warunkowe. Po pobraniu danych z żądanego zasobu statycznego przy użyciu metody `$r->filename()` ustawiamy odpowiednie nagłówki odpowiedzi i obiektu. Wywołanie metody `meets_conditions()` powoduje użycie interfejsu API serwera Apache, aby zdecydować, czy „świeża" zawartość powinna zostać wygenerowana na podstawie nagłówków `Etag`, `If-Match`, `If-Unmodified-Since`, `If-None-Match`, `If-Modified-Since` i `Range`

Tabela 6.2. *Metody dodane do klasy Apache przez klasę Apache::File*

Metoda	Opis
`discard_request_body()`	Usuwa ciało komunikatu z nadchodzącego żądania.
`each_byterange()`	Zwraca pozycje początkowe i długości każdego fragmentu wyspecyfikowanego w żądaniu.
`meets_conditions()`	Sprawdza, czy spełnione są warunki z nagłówków `If-*`. Jeżeli zwróci `OK`, zawartość powinna zostać wysłana do klienta.
`mtime()`	Umożliwia dostęp do czasu ostatniej modyfikacji żądanego zasobu, przechowywanego w rekordzie żądania serwera Apache.
`set_byterange()`	Zwraca wartość „prawda", jeżeli żądanie dotyczy fragmentów pliku.
`set_content_length()`	Ustawia nagłówek `Content-Length` na wskazaną wartość albo na długość żądanego pliku (jeżeli jest dostępna).
`set_etag()`	Generuje i ustawia nagłówek `Etag`.
`set_last_modified()`	Ustawia nagłówek `Last-Modified` na czas ostatniej modyfikacji żądanego pliku, opcjonalnie wywołując metodę `update_mtime()`z podaną wartością.
`update_time()`	Ustawia czas ostatniej modyfikacji żądanego pliku w rekordzie żądania tylko, kiedy nowo ustawiany czas jest późniejszy od dotychczasowego.

żądania. Metoda `meets_conditions()` zwraca wartość `OK`, jeżeli z analizy nagłówków i innych informacji, które dostarczyliśmy na temat zasobu, jak na przykład czas ostatniej modyfikacji, wynika, że należy wysłać klientowi aktualną wersję zawartości. Jeżeli zwrócona wartość jest różna od `OK`, powinna zostać przekazana do serwera Apache, aby mógł podjąć odpowiednią reakcję, na przykład wysłać odpowiedź `304 Not Modified` (zasób niezmieniony).

Można by pomyśleć, że ustawianie nagłówków odpowiedzi przed wywołaniem metody `meets_conditions()` to strata czasu. Jednak metoda ta używa nagłówka `Last-Modified` w swoich porównaniach, a ponadto niektóre nagłówki można zwracać także z odpowiedzią `304 Not Modified`, na przykład `Etag`, `Last-Modified`, `Keep-Alive` i inne.

Chociaż większość metod w tabeli 6.2 może być używana przy wysyłaniu zarówno dynamicznej, jak i statycznej zawartości, metody `set_etag()` powinno się używać tylko przy wysyłaniu niezmienionych, statycznych plików, ponieważ obliczenie nagłówka `Etag` jest bardzo kosztowne, gdyż musi być zagwarantowana jego unikalność dla danego zasobu w danym stanie; nie jest dopuszczalne, aby jakiekolwiek dwie wersje zasobu mogły mieć ten sam nagłówek `Etag`.

Warto również omówić osobno metodę `update_mtime()`. Wpływa ona bezpośrednio na czas ostatniej modyfikacji, który zostanie wysłany w nagłówku `Last-Modified` odpowiedzi, jeżeli użyjemy metody `set_last_modified()`. Metodę `update_mtime()` możemy wywoływać dowolną ilość razy — nagłówek `Last-Modified` będzie miał w rezultacie wartość *najpóźniejszą* z tych, które będziemy próbowali ustawić, co ułatwia wyrażanie skomplikowanych warunków logicznych dotyczących dat w naszym kodzie. Dobrą ilustrację tej cechy stanowią zadania 6.7 i 8.2.

Poniższy wynik działania metody `$r->as_string()` pokazuje komunikację między klientem a serwerem dla wcześniejszego przykładu (wydruk 6.8). Pierwszy zestaw nagłówków reprezentuje żądanie zasobu, którego przeglądarka klienta jeszcze nie posiada w pamięci podręcznej, a drugi — powtórne żądanie tego samego zasobu.

```
GET /Smart/index.html HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en,pdf
Connection: Keep-Alive
Host: www.example.com
User-Agent: Mozilla/4.73  (Windows NT 5.0; U)

HTTP/1.0 200 OK
Last-Modified: Thu, 17 May 2001 18:17:46 GMT
ETag: "6b82a-18a-3b0415ca"
Content-Lenght: 394
Connection: close
Content-Type: text/html

GET /Smart/index.html HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en,pdf
Connection: Keep-Alive
Host: www.example.com
If-Modified-Since: Thu, 17 May 2001 18:17:46 GMT; lenght=394
User-Agent: Mozilla/4.73  (Windows NT 5.0; U)

HTTP/1.0 304 Not Modified
Last-Modified: Thu, 17 May 2001 18:17:46 GMT
ETag: "6b82a-18a-3b0415ca"
Content-Lenght: 394
Connection: close
```

6.7. Żądania fragmentów plików

Chcemy obsługiwać żądania fragmentów plików, wymaganych na przykład przez moduły rozszerzające przeglądarki do obsługi dokumentów w formacie PDF.

Rozwiązanie

Użyjemy metod `Apache->set_byterange()` i `Apache->each_byterange()`, dodanych przez klasę `Apache::File`.

Wydruk 6.9. *Moduł SendAnyDoc.pm*

```
package Cookbook::SendAnyDoc;

use Apache::Constants qw(OK NOT_FOUND);
use Apache::File;
```

```
use DBI;
use DBD::Oracle;
use MIME::Types qw(by_suffix);
use Time::Piece;

use strict;

sub handler {

  my $r = shift;

  my $user  = $r->dir_config('DBUSER');
  my $pass  = $r->dir_config('DBPASS');
  my $dbase = $r->dir_config('DBASE');

  # Tworzymy obiekt Time::Piece na później.
  my $time = localtime;

  my $dbh = DBI->connect($dbase, $user, $pass,
   {RaiseError => 1, AutoCommit => 1, PrintError => 1}) || die $DBI::errstr;

  # Znajdujemy nazwę tablicy i pliku w dodatkowej informacji w ścieżce.
  # Przykładowy adres URI: http//localhost/SendAnyDoc/docs/file.pdf
  my ($table, $filename) = $r->path_info =~ m!/(.*)/(.*)!;

  # Tworzymy zapytanie, które zwraca zawartość pliku i czas ostatniej modyfikacji
  # w sekundach od początku epoki, ale względem aktualnej strefy czasowej
  # (inaczej niż perlowa funkcja time()).
  my $sql= qq(
     select document,
       (last_modified - to_date('01011970','DDMMYYYY')) * 86400
     from $table
     where name = ?
  );

  # Specjalne ustawienia dla pól BLOB.
  $dbh->{LongReadLen} = 10 * 1024 * 1024;  # 10M

  my $sth = $dbh->prepare($sql);

  $sth->execute($filename);

  my ($file, $last_modified) = $sth->fetchrow_array;

  $sth->finish;

  return NOT_FOUND unless $file;

  # Informujemy przeglądarkę, że akceptujemy żądania fragmentów pliku.
  $r->headers_out->set('Accept-Ranges' => 'bytes');

  # Ustawiamy typ MIME na podstawie rozszerzenia nazwy pliku.
  $r->content_type(by_suffix($filename)->[0]);

  # Niech Apache zdecyduje, który czas jest najpóźniejszy:
  #   1. czas pobrany z bazy danych
  #   2. czas modyfikacji pliku źródłowego modułu.
```

```
  # W przypadku czasu z bazy danych upewniamy się, ze jest w GMT.
  (my $package = __PACKAGE__) =~ s!::!/!g;
  $r->update_mtime($last_modified - $time->tzoffset);
  $r->update_mtime((stat $INC{"$package.pm"})[9]);
  $r->set_last_modified;

  # Sprawdzamy, czy jest to żądanie fragmentów _po_ ustawieniem Content-Length
  # ale _przed_ wysłaniem nagłówków, gdyż funkcja ap_set_byterange modyfikuje je.
  # Pamiętajmy, że ustawienie Content-Length jest _konieczne_.
  $r->set_content_length(length($file));

  my $range_request = $r->set_byterange;

  # Tak albo nie.
  if ((my $status = $r->meets_conditions) == OK) {
    $r->send_http_header;
  }
  else {
    return $status;
  }

  # Nie wysyłamy zawartości, jeżeli o nią nie proszono.
  return OK if $r->header_only;

  # Teraz zajmiemy się fragmentami pliku, w przypadku np. dokumentu PDF.
  if ($range_request) {
    while( my($offset, $length) = $r->each_byterange) {
      print substr($file, $offset, $length);
    }
  }
  else {
    print $file;
  }

  return OK ;
}
1;
```

Komentarz

Chociaż powinniśmy pozwolić serwerowi Apache obsługiwać wszystkie możliwe dokumenty statyczne, kiedy plik jest przechowywany w bazie danych, możemy nie mieć innego wyjścia, jak tylko obsłużyć żądanie „własnoręcznie". W takim przypadku dodatkowy wysiłek, włożony w zapewnienie właściwej obsługi różnych nagłówków protokołu HTTP/1.1, pozwoli znacznie obniżyć obciążenie naszego łącza.

Nasz przykładowy program obsługi (wydruk 6.9) stanowi bardzo zmodyfikowaną wersję programu `Cookbook::SendWordDoc` z zadania 3.11. W nowej wersji dodaliśmy kilka usprawnień na podstawie dotychczasowych rozwiązań, w tym użycie dodatkowej informacji w adresie URI, aby określić nazwę tabeli w bazie danych i pliku, który chcemy z niej uzyskać. Dodaliśmy też możliwość inteligentnej obsługi żądań warunkowych w oparciu o czas modyfikacji zarówno

pliku w bazie danych, jak i naszego modułu — oba są używane do określenia, czy zawartość jest „świeża". I, aby spełnić obietnicę zawartą w tytule podrozdziału, dodaliśmy możliwość przesyłania fragmentów pliku.

Możliwość żądania jedynie wskazanych fragmentów pliku została dodana przez wprowadzenie odpowiedniego zestawu nagłówków w specyfikacji protokołu HTTP/1.1. Pełna implementacja tych nagłówków przez klienta i serwer redukuje transfery dużych plików, kiedy użytkownik mógłby być zainteresowany jedynie niektórymi fragmentami. Chociaż taka koncepcja stanowi fascynujące rozwiązanie problemu „zapchania" przepustowości sieci, a serwer Apache implementuje ją w pełni we wbudowanym, domyślnym programie obsługi zawartości statycznej, przeglądarki klientów rzadko ją wykorzystują, z wyjątkiem żądań plików PDF[1].

Mechanizm pobierania fragmentów pliku, używany aktualnie przez moduły rozszerzające przeglądarkę o obsługę dokumentów PDF, może się wydać nieco dziwny na pierwszy rzut oka, gdyż wywoływanych jest kilka żądań, z których pierwsze jest przerywane, aby wywołać następne, dotyczące fragmentów. Chociaż może się to wydawać niezgodne z intuicją w przypadku projektu mającego redukować przesyłanie danych, jednak po zidentyfikowaniu zasobu jako dokumentu PDF, pochodzącego z serwera obsługującego żądania fragmentów, przeglądarka natychmiast przerywa aktualne żądanie, aby wywołać jedno lub kilka następnych z odpowiednimi nagłówkami dotyczącymi fragmentów pliku.

Jak opisaliśmy w zadaniu 4.9, zerwanie połączenia jest natychmiast rozpoznawane przez serwer Apache, który z kolei zamienia wszystkie operacje pisania na operacje puste w celu zaoszczędzenia cykli procesora. Żądania fragmentów pliku mają zmniejszać obciążenie sieci, niekoniecznie zaś obciążenie naszego serwera.

Jeżeli to wszystko brzmi skomplikowanie, to dlatego, że jest skomplikowane. Przykładowy dialog żądanie-odpowiedź dla programu z wydruku 6.9 mógłby wyglądać następująco:

```
GET /SendAnyDoc/docs/file.pdf HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png. */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en,pdf
Connection: Keep-Alive
Host: www.example.com
User-Agent: Mozilla/4.73  (Windows NT 5.0; U)

HTTP/1.0 200 OK
Content-Lenght: 1397596
Accept-Ranges: bytes
Last-Modified: Fri, 29 Jun 2001 16:08:59 GMT
Connection: close
Content-Type: application/pdf

GET /SendAnyDoc/docs/file.pdf HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png. */*
```

[1] Możliwość tę wykorzystują również coraz popularniejsze programy, zwane menedżerami pobierania (*download manager*). W przypadku zerwania połączenia podczas pobierania dużego pliku, program próbuje, przy ponownej inicjacji połączenia, rozpocząć pobieranie od miejsca, w którym zostało przerwane — *przyp. tłum*.

```
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en,pdf
Connection: Keep-Alive
Host: www.example.com
Range: bytes=1396572-1397595,1389404-1396571, [reszta usunięta...]
Request-Range: bytes=1396572-1397595,1389404-1396571, [reszta usunięta...]
User-Agent: Mozilla/4.73  (Windows NT 5.0; U)

HTTP/1.0 206 Partial Content
Content-Lenght: 1392080
Accept-Ranges: bytes
Last-Modified: Fri, 29 Jun 2001 16:08:59 GMT
Connection: close
Content-Type: multipart/x-byteranges; boundary=3b3cbeae439
```

Metody `set_byterange()` i `each_byterange()` stanowią klucz do dynamicznego wysyłania dokumentów PDF. Obie zostają dodane do klasy `Apache` po zaimportowaniu (przy użyciu instrukcji `use`) klasy `Apache::File`. Pierwsza analizuje nagłówki żądania, a następnie ustawia odpowiednio nagłówki `Content-Type` i `Content-Range` w razie potrzeby. Ponieważ wartość nagłówka `Content-Range` zależy od wielkości zawartości, *musimy* wywołać metodę `set_content_length()` *przed* wywołaniem metody `set_byterange()`, która z kolei musi zostać wywołana *przed* metodą `send_http_header()` — kolejność jest tu bardzo istotna. Kiedy zostanie już stwierdzone, że żądano fragmentów pliku, wywołanie metody `each_byterange()` zwraca listę par, zawierających początek i długość każdego fragmentu żądanego przez klienta. Możemy użyć tej listy, aby pobrać odpowiednie fragmenty pliku i przesłać je klientowi.

W naszym przykładzie pobieraliśmy plik z bazy danych, ale jeżeli obsługujemy dynamicznie żądania fragmentów plików statycznych na dysku, lepszym rozwiązaniem będzie użyć dwuargumentowej wersji metody `send_fd()`, omówionej w zadaniu 6.3, niż wczytywać cały plik do pamięci i dzielić na fragmenty za pomocą funkcji `substr()`. Na przykład w pętli przetwarzającej listę zwróconą przez metodę `each_byterange()` w naszym programie obsługi moglibyśmy wprowadzić następujące zmiany, aby przetwarzać pliki statyczne:

```
if ($range_request) {
  while( my($offset, $length) = $r->each_byterange) {
    seek $fh, $offset, 0;
    $r->send_fd($fh, $length);
  }
}
else {
  $r->send_fd($fh);
}
```

6.8. Nagłówki związane z datami

Chcemy bezpośrednio odczytywać i zmieniać wartość nagłówków związanych z datami.

Rozwiązanie

Użyjemy funkcji `ht_time()` i `parsedate()`, dostarczonych przez klasę `Apache::Util`.

```
$r->log->info("Nagłówek Last-Modified ustawiony na: ",
              Apache::Util::ht_time($r->mtime));
```

Komentarz

Ponieważ omawiamy teraz nagłówki związane z datami, jest to dobre miejsce, aby szerzej omówić dwie metody z klasy `Apache::Util`, o których wspomnieliśmy tylko w rozdziale 5. Funkcje `ht_time()` i `parsedate()` zapewniają wygodny sposób konwersji dat między formatem protokołu HTTP a liczbą sekund od początku epoki, którą zwraca wiele funkcji perlowych. Funkcja `parsedate()` przelicza datę z formatu HTTP na odpowiednią liczbę sekund od początku epoki. Funkcja `ht_time()` zamienia sekundy na datę w formacie HTTP, która zawsze jest wyrażana w czasie GMT.

Chociaż te metody są wygodne, nie zawsze mogą wykonać za nas całą pracę. Na przykład jeżeli obliczamy liczbę sekund w innej strefie czasowej niż GMT (jak na wydruku 6.9), będziemy musieli wykonać konwersję samodzielnie, używając modułu, takiego jak `Time::Piece`.

6.9. Opróżnianie buforów wyjściowych

Chcemy opróżnić wewnętrzne bufory wyjściowe serwera Apache.

Rozwiązanie

Użyjemy metody `Apache->rflush()`.

```
while (<$fh>) {
  # Wypisujemy każdy wiersz pliku i wysyłamy go od razu do klienta.
  # Bardzo zły pomysł w większości wypadków.
  print;
  $r->rflush;
}
```

Komentarz

Serwer Apache w normalnych warunkach buforuje dane wypisywane przez program obsługi, wysyłając je do klienta dopiero, kiedy bufor jest pełny albo program obsługi zakończy działanie. Mogą się jednak zdarzyć sytuacje, kiedy trzeba wysłać dane do klienta natychmiast, na przykład

gdy nasz program jest w trakcie jakiegoś relatywnie długiego procesu i chcielibyśmy, aby w tym czasie pojawiło się coś w przeglądarce klienta. W takim przypadku możemy użyć metody `$r->rflush()`, aby opróżnić bufor. Należy jednak używać jej z umiarem, gdyż obniża znacznie wydajność serwera.

Jako przykład a także przypomnienie kilku metod, które poznaliśmy dotychczas, rozważmy następujący program obsługi, który wybiera losowy obraz z katalogu *ServerRoot/icons* i wysyła go klientowi.

Wydruk 6.10. *Moduł SendIcon.pm*

```
package Cookbook::SendIcon;

use Apache::Constants qw(OK SERVER_ERROR);
use Apache::File;

use DirHandle;

use strict;

sub handler {

  my $r = shift;

  # Pobieramy położenie podkatalogu icons/ względem ServerRoot.
  my $icons = $r->server_root_relative('icons');
  my $dh = DirHandle->new($icons);
  unless ($dh) {
    $r->log_error("Nie można otworzyć katalogu $icons: $!");
    return SERVER_ERROR;
  }

  # Pobieramy zawartość katalogu i dodajemy do listy @icons wszystkie znalezione
  # pliku typu GIF.
  my @icons;

  foreach my $icon ($dh->read) {
    my $sub = $r->lookup_uri("/icons/$icon");
    next unless $sub->content_type eq 'image/gif';
    push @icons, $sub->filename;
  }

  # Wybieramy losowy obraz.
  my $image = $icons[rand @icons];

  # Otwieramy wybrany plik obrazu i wysyłamy go klientowi.
  my $fh = Apache::File->new($image);
  unless ($fh) {
    $r->log_error("Nie można otworzyć pliku $image: $!");
    return SERVER_ERROR;
  }

  binmode $fh;  # wymagane dla systemów Win32

  $r->send_http_header('image/gif');
```

```
  $r->send_fd($fh);

  # Opróżniamy bufor, aby obraz został wysłany natychmiast.
  $r->rflush;

  # Symulujemy jakiś długotrwały proces...
  sleep(5);

  return OK;
}
1;
```

Bez wywołania metody `$r->rflush()` klient nie ujrzałby obrazu, zanim nie skończyłby się długotrwały proces (w naszym przykładzie symulowany wywołaniem funkcji `sleep()`). Jednakże, podkreślamy po raz kolejny, metoda `$r->rflush()` może znacznie obniżyć wydajność serwera, więc powinna być używana tylko w razie potrzeby.

6.10. Przekierowanie uchwytów plików wyjściowych

Chcemy zmienić domyślne przypisanie strumieni wyjściowych `STDOUT` *i* `STDERR`.

Rozwiązanie

Użyjemy interfejsu `TIEHANDLE`, pochodzącego z klasy `Apache` lub innej, aby zmienić zachowanie uchwytów plików.

Wydruk 6.11. *Przykładowy program obsługi*

```
use Apache::Constants qw(OK SERVER_ERROR);
use Net::FTP;

use strict;

sub handler {

  my $r = shift;

  return SERVER_ERROR unless chdir "/tmp";

  $r->send_http_header('text/plain');

  # Przekierowujemy strumień STDERR do przeglądarki klienta.
  # W ten sposób ukaże mu się cała komunikacja z serwerem FTP.
  tie *STDERR, 'Apache';
```

```
  my $ftp = Net::FTP->new("ftp.cpan.org", Debug => 5);
  $ftp->login("anonymous");
  $ftp->binary;
  $ftp->cwd("pub/CPAN/modules/by-module/Apache/");
  $ftp->get("mod_perl-1.26.tar.gz");
  $ftp->quit;

  # Strumień STDERR ponownie skierowany do dziennika błędów.
  untie *STDERR;

  return OK;
}
```

Komentarz

Mechanizm `tie()` to „zdradzieckie" narzędzie, które może zostać użyte do zmiany zachowania wielu perlowych typów danych. W zadaniu 2.17 widzieliśmy, jak można dowiązać (*tie*) mapę do klasy `Tie::DxHash`, aby zachować kolejność wstawiania elementów i umożliwić powtarzanie się kluczy — normalnie mapa nie dopuszcza takiej funkcjonalności. Tak samo łatwo możemy za pomocą instrukcji `tie()` zmusić uchwyty plików `STDOUT` i `STDERR` do wykonania naszych „diabelskich sztuczek".

Moduł mod_perl podłącza strumienie `STDOUT` i `STDIN` do przeglądarki klienta, używając interfejsu `TIEHANDLE` z klasy `Apache`, natomiast `STDERR` jest kierowany do pliku wskazanego przez wartość `error_log` w rekordzie serwera. Chociaż nie ma dużego sensu przekierowywanie strumienia `STDIN`, to samo w przypadku strumieni wyjściowych może przynieść nieoczekiwane korzyści w rękach cudotwórców, magików, guru i tym podobnych. Możemy poczuć smaczek tej „magii", kiedy używamy klasy `Apache::Filter` do łączenia wyniku działania kilku perlowych programów obsługi — w serwerze Apache 1.3 filtrowanie danych wyjściowych jest po prostu niemożliwe przy użyciu istniejącego interfejsu API języka C. Programy obsługi modułu mod_perl mogą ominąć to ograniczenie, używając przekierowanych uchwytów plików i paru innych, bardzo pomysłowych rozwiązań, aby osiągnąć imponujące rezultaty. W zadaniu 15.4 mamy przykład, jak używać klasy `Apache::Filter`.

Jeżeli interesuje nas przekierowanie strumienia `STDOUT` zamiast `STDERR`, możemy go dowiązać do klasy, takiej jak `IO::String`, `IO::Scalar` (z dystrybucji `IO-stringy`) lub naszego własnego interfejsu `TIEHANDLE`. Należy tylko pamiętać, aby „zwrócić" strumień `STDOUT` do serwera Apache, kiedy skończymy. Oto bardzo prosty przykład.

Wydruk 6.12. *Przekierowanie strumienia STDOUT*

```
use Apache::Constants qw(OK);
use IO::Scalar;

use strict;

sub handler {

  my $r = shift;
```

```
    my $string;

    $r->send_http_header('text/plain');

    print "Jeszcze niedowiązany...\n";

    tie *STDOUT, 'IQ::Scalar', \$string;

    print "Piszemy do zmiennej \$string";

    tie *STDOUT, 'Apache';

    print "Ponownie dowiązany: $string\n";

    return OK;
}
```

Aby nasz program zachowywał się grzecznie i przewidywalnie, nie powinniśmy zakładać, że strumień `STDOUT` jest dowiązany do klasy `Apache`. Właściwe rozwiązanie polega na zapamiętaniu klasy, do której strumień był dowiązany i odtworzenie go, kiedy skończymy już nasze „czary".

```
my $old = ref tied(*STDOUT);

# abra-tie-kadabra...

tie *STDOUT, $old;
```