Java Servlet – programowanie.
Wydanie 2

Autorzy: Jason Hunter, William Crawford
Tłumaczenie: Jacek Smycz, Adam Grochowina, Tomasz Miszkiel
ISBN: 83-7197-527-9
Tytuł oryginału: Java Servlet Programming 2nd Edition
Format: B5, stron: 666
Przykłady na ftp: 193 kB

W ciągu kilku ostatnich lat serwlety Javy zdobyły uznanie społeczności twórców oprogramowania działającego po stronie serwera. Obecnie, po wprowadzeniu wersji 2.3 Servlet API, serwlety wynoszą Javę na nowy poziom tworzenia oprogramowania dla sieci WWW.

Serwlety zapewniają szybkie, potężne i przenośne środowiska do tworzenia dynamicznej zawartości stron WWW. Są one wykonywane na serwerze, co pozwala im działać efektywniej w porównaniu z innymi rozwiązaniami. Serwlety posiadają pełny dostęp do różnych API Javy, a także klas niezależnych komponentów, są również integralną częścią Java 2 Enterprise Edition (J2EE). Ich najważniejszą zaletą jest możliwość przenoszenia pomiędzy systemami operacyjnymi i serwerami – serwlety można „utworzyć raz, używać wszędzie". Użytkownicy rozpoczynający dopiero pracę z serwletami znajdą w książce opis wykorzystania serwletów do tworzenia potężnych, interaktywnych aplikacji WWW. Tematy tej książki to między innymi dynamiczne strony HTML, dokumenty XML, WAP, multimedialna zawartość stron, zintegrowane śledzenie sesji oraz wydajna łączność z bazami danych za pomocą JDBC. Osobom znającym już serwlety książka ta oferuje uaktualnione informacje na takie tematy jak archiwa aplikacji WWW (WAR), integracja J2EE, zarządzane przez serwer systemy bezpieczeństwa, zoptymalizowana współpraca serwerów, a także JavaServer Pages (JSP) oraz wiele innych.

Drugie wydanie bestsellerowej książki „Java Servlet programming" jest doskonałym wprowadzeniem do świata servletów. Książka opisuje metody wykorzystania serweltów do stworzenia profesjonalnych, interaktywnych aplikacji sieciowych.

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

Spis treści

Wstęp *9*

Rozdział 1. Wprowadzenie *19*

Historia aplikacji WWW 20
Obsługa serwletów 24
Potęga serwletów 28

Rozdział 2. Podstawy serwletów HTTP *31*

Podstawy HTTP 32
Interfejs API (Servlet API) 34
Tworzenie strony 36
Aplikacje WWW 42

Rozdział 3. Czas istnienia (cykl życia) serwletu *49*

Alternatywa serwletu 49
Odnawianie (powtórne ładowanie) serwletu 55
Metody „Init" i „Destroy" 56
Model jednowątkowy (Single Thread Model) 63
Przetwarzanie w tle 65
Ładowanie i uruchamianie 67
Buforowanie podręczne po stronie klienta 68
Buforowanie podręczne po stronie serwera 70

Rozdział 4. Pobieranie informacji *81*

Serwlet 82
Serwer 85
Klient 93

Rozdział 5. Wysyłanie informacji HTML 131
Struktura odpowiedzi 132
Przesyłanie standardowej odpowiedzi 132
Używanie trwałych połączeń 134
Buforowanie odpowiedzi 135
Kody statusu 138
Nagłówki HTTP 140
Rozwiązywanie problemów 147
Sześć sposobów uzyskiwania korzyści z serwletów 158

Rozdział 6. Wysyłanie zawartości multimedialnej 163
WAP i WML 163
Obrazki 171
Zawartość skompresowana 187
Serwer cykliczny 190

Rozdział 7. Śledzenie sesji 195
Uwierzytelnianie użytkownika 196
Ukryte pola danych formularza 197
Przepisywanie URL-u 200
Trwałe cookies 202
API — śledzenie sesji 206

Rozdział 8. Bezpieczeństwo 223
Uwierzytelnienie poprzez HTTP 224
Uwierzytelnienie na podstawie formularza 230
Uwierzytelnienie niestandardowe 233
Certyfikaty cyfrowe 239
Protokół bezpiecznej transmisji danych (SSL) 241

Rozdział 9. Łączność z bazami danych 249
Relacyjne bazy danych 251
JDBC API 253
Ponowne użycie obiektów bazy danych 265
Transakcje 267
Serwlet księgi gości 275
Zaawansowane techniki JDBC 280
Co dalej? 283

Rozdział 10. Komunikacja aplet-serwlet.....285
Opcje komunikacji..... 285
Serwer daytime..... 291
Serwer chat..... 321

Rozdział 11. Współpraca serwletów 339
Dzielenie informacji..... 339
Dzielenie kontroli..... 343

Rozdział 12. Serwlety korporacyjne i J2EE 351
Ładowanie rozproszone..... 352
Integracja z J2EE..... 355

Rozdział 13. Internacjonalizacja.....361
Języki zachodnioeuropejskie..... 362
Hołdowanie lokalnym zwyczajom..... 365
Języki spoza Europy Zachodniej..... 367
Więcej języków..... 371
Dynamiczna negocjacja języka..... 373
Formularze HTML..... 382

Rozdział 14. Szkielet Tea 389
Język Tea..... 390
Początki..... 391
Informacja o żądaniu..... 393
Administracja Tea..... 396
Zastosowania Tea..... 400
Aplikacja „Narzędzia”..... 405
Ostatnie słowo..... 415

Rozdział 15. WebMacro 417
Szkielet WebMacro..... 418
Instalacja WebMacro..... 421
Dyrektywy WebMacro..... 426
Szablony WebMacro..... 429
Aplikacja „Narzędzia”..... 434
Filtry..... 439

Rozdział 16. Element Construction Set 441
Elementy strony jako obiekty 441
Wyświetlanie zbioru wyników 443

Rozdział 17. XMLC 453
Prosta kompilacja języka XML 454
Klasa manipulacyjna 459
Aplikacja „Narzędzia” 463

Rozdział 18. JavaServer Pages 471
Wykorzystywanie JavaServer Pages 472
Zasady działania 473
Wyrażenia i deklaracje 476
Dyrektywy 477
JSP i JavaBeans 482
Dołączenia i przekazania 487
Aplikacja „Narzędzia” 489
Biblioteki własnych znaczników 493

Rozdział 19. Informacje dodatkowe 499
Analiza parametrów 499
Wysyłanie poczty elektronicznej 504
Stosowanie wyrażeń regularnych 507
Uruchamianie programów 511
Stosowanie metod rodzimych 514
Występowanie jako klient RMI 515
Usuwanie błędów 517
Poprawa wydajności 524

Rozdział 20. Zmiany w Servlet API 2.3 527
Zmiany w Servlet API 2.3 527
Konkluzja 541

Dodatek A Krótki opis Servlet API 543

Dodatek B Krótki opis HTTP Servlet API 571

Dodatek C Krótki opis DTD deskryptora aplikacji WWW *597*

Dodatek D Kody statusu HTTP *627*

Dodatek E Encje znakowe *635*

Dodatek F Kodowania *643*

Skorowidz *647*

3

Czas istnienia (cykl życia) serwletu

W tym rozdziale:

- Alternatywa serwletu,
- Odnawianie (powtórne ładowanie) serwletu,
- Inicjalizacja i usuwanie,
- Model jednowątkowy (Single Thread Model),
- Przetwarzanie drugoplanowe,
- Ładowanie i uruchamianie,
- Buforowanie podręczne po stronie klienta,
- Buforowanie podręczne po stronie serwera.

Czas istnienia (cykl życia) serwletu jest jednym z bardziej interesujących aspektów dotyczących serwletów. Czas istnienia jest hybrydą czasów używanych w środkach programowania CGI oraz środkach programowania niskiego poziomu WAI/NSAPI i ISAPI, tak jak zostało to omówione w rozdziale 1. („Wprowadzenie").

Alternatywa serwletu

Czas istnienia (cykl życia) serwletów pozwala ich kontenerom na odniesienie się do wydajności, do problemów związanych z CGI oraz do problemów dotyczących bezpieczeństwa niskopoziomowych środków programowania API dla serwerów. Kontenery serwletów uruchamiają zwykle wszystkie serwlety razem, w jednej maszynie wirtualnej Javy (JVM). Dzięki umiejscowieniu serwletów w tej samej JVM mogą one skutecznie wymieniać dane między sobą, jednak język Java nie daje możliwości wglądu jednemu serwletowi do „prywatnych" danych znajdujących się w drugim. Serwlety mogą istnieć w JVM pomiędzy zleceniami — jako egzemplarze obiektów.

Dzięki temu zajęte jest mniej pamięci niż w przypadku pełnych procesów, a serwlety są nadal w stanie utrzymać odniesienia do zewnętrznych zasobów. Cykl życia serwletów jest wysoce elastyczny. Jedyną rzeczą niezmienną i konieczną w tym cyklu jest to, iż kontener serwletu musi przestrzegać następnych zasad:

1. Stworzyć oraz zainicjalizować serwlet.
2. Obsłużyć wywołania usługi od klientów.
3. Usunąć serwlet i zwolnić przydzieloną mu pamięć.

W przypadku serwletów jest rzeczą całkowicie naturalną, iż są one ładowane, tworzone i przechowywane w swojej własnej maszynie wirtualnej Javy — tylko po to, aby być usuniętymi, nie obsłużywszy żadnych zleceń od klientów lub po obsłużeniu tylko jednego takiego zlecenia. Jednak kontenery serwletów zachowujące się w taki sposób, nie utrzymają się długo na rynku. W tym rozdziale zostaną omówione najpopularniejsze oraz najczulsze realizacje czasów istnienia serwletów HTTP.

Pojedyncza maszyna wirtualna Javy

Większość kontenerów wykonuje wszystkie serwletyw jednej JVM w celu maksymalizacji zdolności serwletów do wymiany informacji (wyjątkiem są tutaj kontenery wyższej klasy, które realizują rozproszone wykonywanie serwletu na wielu serwerach, tak jak zostało to omówione w rozdziale 12., „Serwlety korporacyjne i J2EE").

Wykonania wyżej wspomnianej pojedynczej maszyny wirtualnej Javy mogą być odmienne na różnych serwerach:

- Na serwerze napisanym w Javie, np. „Apache Tomcat", sam serwer może być wywoływany w JVM wraz ze swoimi serwletami.
- Na pojedynczo przetwarzającym, wielowątkowym serwerze WWW, zapisanym w innym języku, wirtualna maszyna Javy może zostać zawarta w procesie serwera. JVM, jako część procesu serwera, zwiększa wydajność, ponieważ serwlet staje się w pewnym sensie kolejnym rozszerzeniem serwera API niskiego poziomu. Serwer taki może wywołać serwlet z „lekkim" połączeniem kontekstu, może również dostarczyć informacje o zleceniach poprzez bezpośrednie wywołania metod.
- Wieloprocesowy serwer WWW (który uruchamia kilka procesów, aby obsłużyć zlecenia) właściwie nie może zawrzeć JVM bezpośrednio w swoim procesie, ponieważ takiego nie posiada. Ten typ serwerów zwykle uruchamia zewnętrzną JVM, której procesy może współdzielić. Taki sposób oznacza, iż każde wejście do serwletu będzie się wiązać ze skomplikowanym połączeniem kontekstu przypominającym *FastCGI*. Jednakże wszystkie serwlety będą nadal dzieliły ten sam zewnętrzny proces.

Na szczęście, z perspektywy serwletów (a tym samym z naszej — jako ich twórców), implementacja serwerów nie ma większego znaczenia, ponieważ zachowują się one zawsze w te sam sposób.

Trwałość egzemplarza

Tak jak to zostało już opisane, serwlety istnieją pomiędzy zleceniami jako egzemplarze obiektów (instancje). Inaczej mówiąc, w czasie ładowania kodu serwletu, serwer tworzy pojedynczy egzemplarz, który obsługuje wszystkie zlecenia przeznaczone dla niego.. Poprawia to wydajność z trzech powodów:

- Zajmowana powierzchnia pamięci jest mała.
- Eliminowane są obciążenia tworzenia obiektu (w przeciwnym wypadku konieczne byłoby utworzenie nowego obiektu serwletu). Serwlet może być już ładowany w maszynie wirtualnej, kiedy zlecenie dopiero wchodzi, pozwalając mu na rozpoczęcie wywoływania natychmiast.
- Umożliwione jest trwanie — serwlet może mieć wszystko, czego potrzebuje podczas obsługi zlecenia, np. już połączenie z bazą danych może zostać ustanowione raz i używane wielokrotnie, a z takiego połączenia może korzystać wiele serwletów. Kolejnym przykładem jest serwlet koszyka zakupów, który ładuje do pamięci listę cen wraz z informacją o ostatnio połączonych klientach. Inne serwlety w sytuacji, kiedy otrzymują to samo zlecenie, pobierają całe strony przetrzymywane w pamięci podręcznej, celem zaoszczędzenia czasu.

Serwlety nie tylko trwają pomiędzy zleceniami, lecz także wykonują wszystkie wątki stworzone przez siebie. Taka sytuacja nie jest może zbyt korzystna w przypadku serwletu „run-of-the-mill", jednak daje interesujące możliwości. Rozważmy sytuację, w której podrzędny wątek przeprowadza pewne kalkulacje, podczas gdy inne wyświetlają ostatnie rezultaty. Podobnie jest w przypadku apletu animacyjnego, w którym jeden wątek zamienia obraz, a inny nanosi kolory.

Liczniki

W celu przedstawienia cyklu życia (czasu istnienia serwletu) posłużymy się prostym przykładem. Przykład 3.1 pokazuje serwlet, który zlicza i wyświetla liczbę połączeń z nim realizowanych. Dla uproszczenia wynik przedstawiany jest jako zwykły tekst (kod do wszystkich przykładów dostępny jest w Internecie; zobacz „Wstęp").

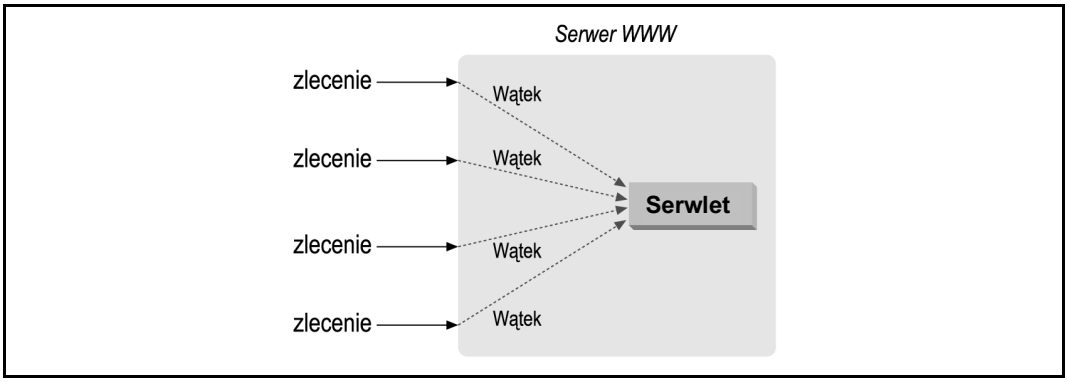
Przykład 3.1. Przykładowy prosty licznik

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleCounter extends HttpServlet {

  int count = 0;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType("text/plain; charset=ISO-8859-2");
    PrintWriter out = res.getWriter();
    count++;
    out.println("Od pierwszego uruchomienia z serwletem połączono się
                ➥" + count + " razy.");
  }
}
```

Kod jest prosty: zwiększa oraz wyświetla wartość zmiennej nazwanej `count`, ale dobrze ukazuje „potęgę" trwałości. Kiedy serwer ładuje ten serwlet, tworzy pojedynczy egzemplarz celem obsłużenia wszystkich zleceń nałożonych na ten serwlet, dlatego właśnie kod bywa taki prosty. Kolejne wywołania tego serwletu będą odwoływać się do tego samego obiektu.

Liczniki zsynchronizowane

Z punktu widzenia projektantów serwletów każdy klient to kolejny wątek, który wywołuje serwlet poprzez metody typu: `service()`, `doGet()`, `doPost()`, jak to pokazuje rysunek 3.1[1].



Rysunek 3.1. Wiele wątków — jeden egzemplarz serwletu

Jeżeli nasze serwlety odczytują tylko zlecenia, piszą w odpowiedziach i zapisują informacje w lokalnych zmiennych, czyli w zmiennych określonych w metodzie, nie musimy obawiać się interakcji pomiędzy wątkami. Jeżeli informacje zostają zapisane w zmiennych nielokalnych, czyli w zmiennych określonych w klasie, lecz poza szczególną metodą, musimy być świadomi, iż każdy z wątków klientów może operować tymi zmiennymi serwletu. Bez odpowiednich środków ostrożności sytuacja taka może spowodować zniszczenie danych oraz sprzeczności. I tak np. jeżeli serwlet `SimpleCounter` założy fałszywie, że zwiększanie licznika oraz wyprowadzenie wartości są przeprowadzane niepodzielnie (bezpośrednio jeden po drugim, nieprzerwanie), to gdy dwa zlecenia zostaną złożone do `SimpleCounter` prawie w tym samym czasie, każdy z nich może wskazać tę samą wartość dla `count`. Jak? Na przykład jeden wątek zwiększa wartość dla `count` i zaraz po tym, zanim jeszcze pierwszy watek wypisze wynik `count`, drugi wątek również zwiększa wartość. W takim przypadku każdy z wątków wskaże tę samą wartość, po efektywnym zwiększeniu jej o 2[2].

---

[1] To, iż jeden egzemplarz serwletu może obsłużyć wiele zleceń w tym samym czasie, może wydawać się dziwne. Dzieje się tak prawdopodobnie dlatego, że kiedy obrazujemy program uruchamiający, zwykle obserwujemy, jak egzemplarze obiektów wykonują zadanie, wywołując nawzajem swoje metody. Mimo że przedstawiony model działa w prostych przypadkach, nie przedstawia rzeczywistości dokładnie. Prawdziwa sytuacja wygląda tak, że wszystkie zadania wykonują wątki. Egzemplarze obiektów nie są niczym więcej jak tylko strukturami danych, którymi operują wątki. Dlatego możliwa jest sytuacja, w której dwa działające wątki używają w tym samym czasie tego samego obiektu.

[2] Ciekawostka: jeżeli wartość `count` byłaby zamiast 32-bitowego `int`, 64-bitowym `long`, teoretycznie możliwa byłaby sytuacja, że inkrementacja będzie dokonana tylko w połowie, tzn. do czasu gdy przerwie mu inny wątek. Dzieje się tak dlatego, że w Javie używana jest 32-bitowy stos.

Porządek wykonania wygląda mniej więcej w ten sposób:

```
count++           // Wątek 1

count++           // Wątek 2

out.println       // Wątek 1

out.println       // Wątek 2
```

W tym przypadku ryzyko sprzeczności nie stanowi poważnego zagrożenia, ale wiele innych serwletów zagrożonych jest poważniejszymi błędami. W celu zapobiegania tego typu błędom oraz sprzecznościom, które im towarzyszą, można dodać jeden lub więcej synchronizowanych bloków do kodu. To gwarancja, że wszystko, co znajduje się w bloku synchronizowanym lub w metodzie synchronizowanej, nie będzie wywoływane przez inny wątek. Zanim jakikolwiek z wątków rozpocznie wywoływanie kodu synchronizowanego, musi otrzymać monitor (zamek) na określony egzemplarz obiektu. Jeżeli inny wątek ma już monitor, np. dlatego że wywołuje ten sam blok synchronizowany lub inny z tym samym monitorem, wtedy pierwszy wątek musi zaczekać. Działa to na zasadzie „łazienki na stacji benzynowej" zamykanej na klucz (zawieszany zwykle na dużej, drewnianej desce). W naszym przypadku kluczem będzie monitor. Wszystko to dzieje się dzięki mechanizmom wbudowanym w język, tak więc obsługa jest łatwa. Synchronizacja powinna być jednak używana tylko w ostateczności. W przypadku niektórych platform sprzętowych otrzymanie monitora za każdym razem, kiedy wchodzimy do kodu synchronizowanego wymaga wiele operacji, a co ważniejsze — w czasie, kiedy jeden wątek wywołuje kod synchronizowany, pozostałe mogą być blokowane aż do zwolnienia monitora.

Dla `SimpleCounter` istnieją cztery sposoby rozwiązywania potencjalnych problemów. Po pierwsze można dodać slowo kluczowe `synchronized` do `doGet()`:

```
public synchronized void doGet(HttpServletRequest req,
                               HttpServletResponse res)
```

Taka sytuacja gwarantuje zgodność przez synchronizację całej metody. Nie jest to jednak najlepszy sposób, ponieważ oznacza, iż serwlet może w tym samym czasie obsłużyć tylko jedno zlecenie `GET`.

Drugim sposobem jest zsynchronizowanie tylko dwóch wierszy, które chcemy wywołać niepodzielnie:

```
PrintWriter out = res.getWriter();
synchronized(this) {
  count++;
  out.println ("Z serwletem połączono się " + count + " razy.");
}
```

Powyższa technika działa lepiej, ponieważ ogranicza czas, który serwlet spędza w swoim zsynchronizowanym bloku, osiągając ten sam cel zgodności w wyniku liczenia. Prawdą jest, iż technika ta nie różni się znacząco od pierwszej.

Trzecim sposobem ominięcia potencjalnych problemów jest utworzenie synchronizowanego bloku (który będzie robił wszystko, co musi być wykonane szeregowo), reszta będzie znajdować się poza blokiem synchronizowanym. W przypadku naszego serwletu liczącego możemy zwiększyć wartość zmiennej (`count`) w bloku synchronizowanym, zapisać zwiększoną wartość do lokalnej zmiennej (zmiennej określonej wewnątrz metody), a następnie wyświetlić wartość lokalnej zmiennej poza blokiem synchronizowanym:

```
PrintWriter out = res.getWriter();
int local_count;
synchronized(this) {
  local_count = ++count;
}
     out.println("Z serwletem połączono się " + local_count + " razy.");
```

Powyższa zmienna zawęża blok synchronizowany do najmniejszych możliwych rozmiarów, zachowując przy tym zgodność liczenia.

Celem zastosowania czwartej, ostatniej z metod musimy zadecydować, czy chcemy ponieść konsekwencje zignorowania wyników synchronizacji? Czasem bywa i tak, że konsekwencje te są do przyjęcia, np. zignorowanie synchronizacji może oznaczać, że klienci otrzymają wynik trochę niedokładny. Trzeba przyznać, iż to rzeczywiście nie jest wielki problem. Jeżeli jednak oczekiwanoby od serwletu liczb dokładnych, wtedy sprawa wyglądałaby trochę gorzej.

Mimo iż nie jest to opcja możliwa do zastosowania na omawianym przykładzie, to na innych serwletach można dokonać zamiany dotychczasowych zmiennych na zmienne lokalne. Zmienne lokalne są niedostępne dla innych wątków i tym samym nie muszą być dokładnie strzeżone przed zniszczeniem. Jednocześnie zmienne lokalne nie istnieją pomiędzy zleceniami, tak więc nie możemy ich użyć do utrzymywania stałego stanu naszego licznika.

Liczniki całościowe

Model „jeden egzemplarz na jeden serwlet” wymaga jedynie ogólnego omówienia. Prawda jest taka, że każda nazwa zarejestrowana dla serwletu (lecz nie każde URL-owe dopasowanie do wzorca) jest związana z jednym egzemplarzem serwletu. Nazwa używana przy wchodzeniu do serwletu określa, który egzemplarz obsłuży zlecenie. Taka sytuacja wydaje się być właściwa, ponieważ klient powinien kojarzyć odmienne nazywanie serwletów z ich niezależnym działaniem. Osobne egzemplarze są ponadto wymogiem dla serwletów zgodnych z parametrami inicjalizacji, co zostało omówione poniżej.

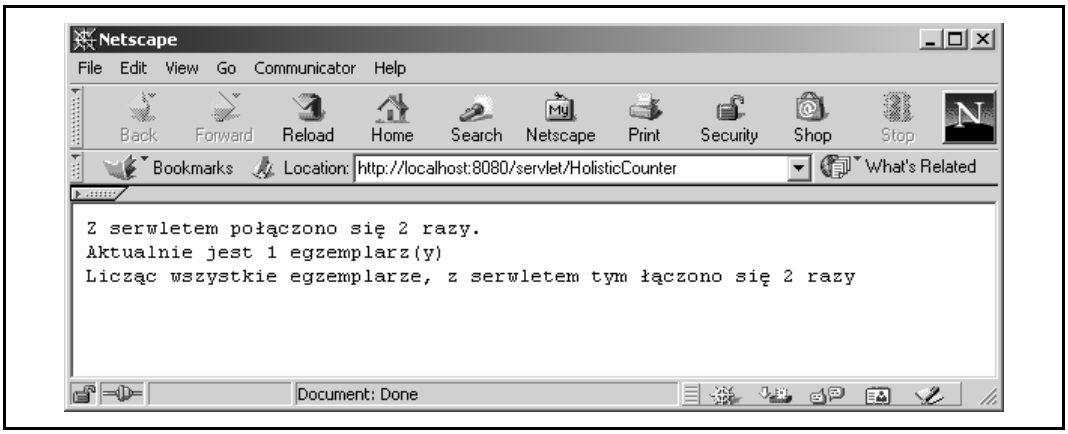
Nasz przykładowy `SimpleCounter` posługuje się zmienną przy zliczaniu liczby połączeń z nim wykonanych. Jeżeli zaistniałaby potrzeba liczenia dla wszystkich egzemplarzy (a tym samym wszystkich zarejestrowanych nazw), możliwe jest użycie klasy lub zmiennej statycznej. Zmienne takie są wspólne dla wszystkich egzemplarzy klasy. Przykład 3.2 ukazuje liczbę uruchomień serwletu, liczbę egzemplarzy utworzonych przez serwer (na jedną nazwę) oraz całkowitą liczbę połączeń z tymi egzemplarzami.

Przykład 3.2. Licznik całościowy

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HolisticCounter extends HttpServlet {

  static int  classCount = 0;    // współdzielony przez wszystkie egzemplarze
  int  count = 0;                // osobny dla każdego serwletu
  static Hashtable instances = new Hashtable();   // również współdzielony

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
```

```
    res.setContentType ("text/plain; charset=ISO-8859-2");
    PrintWriter out = res.getWriter();

     count++;
     out.println ("Z serwletem połączono się " + count + " razy.");

    // pamięta count poprzez wstawienie odwołania do niego w tablicy
    // asocjacyjnej instances.  Powtarzające się wpisy są ignorowane.
    // Metoda size() zwraca liczbę egzemplarzy
     instances.put(this, this);
     out.println("Aktualnie jest " + instances.size() + " egzemplarz(y)");

     classCount++;

     out.println ("Licząc wszystkie egzemplarze, z serwletem tym " +
                  ➥"łączono się " + classCount + " razy");

    }
}
```

Przedstawiony licznik całościowy (`HolisticCounter`), śledzi liczbę połączeń własnych za pomocą zmiennej `count`, liczbę połączeń wspólnych — za pomocą zmiennej `classCount` oraz liczbę egzemplarzy — za pomocą tablicy asocjacyjnej `instances` (kolejny wspólny element, który musi być zmienną klasy). Przykład ten został pokazany na rysunku 3.2.



Rysunek 3.2. Widok licznika całościowego

Odnawianie (powtórne ładowanie) serwletu

Jeśli ktoś próbował użyć omówionych liczników we własnym zakresie, być może zauważył, iż z każdą kolejną rekompilacją liczenie zaczyna się automatycznie od 1. Nie jest to defektem, tylko właściwością. Większość serwerów odnawia (powtórnie ładuje) serwlety, po tym jak zmieniają się ich pliki klasy (pod domyślnym katalogiem serwletów *WEB-INF/classes*). Jest to procedura wykonywana na bieżąco, która znacznie przyśpiesza cykl testu i rozbudowy oraz pozwala na przedłużenie czasu sprawnego działania serwera.

Odnawianie serwletu może wydawać się proste, jednak wymaga dużego nakładu pracy. Obiekty klasy `ClassLoader` zaprojektowane są do jednokrotnego załadowania klasy.

Aby ominąć to ograniczenie i wielokrotnie ładować serwlety, serwery używają własnych programów ładujących, które ładują serwlety ze specjalnych katalogów, takich jak *WEB-INF/classes*.

Kiedy serwer wysyła zlecenie do serwletu, najpierw sprawdza, czy plik klasy serwletu zmienił się na dysku. Jeżeli okaże się, że tak, wówczas serwer nie będzie już używał programu ładującego starej wersji pliku, tylko utworzy nowy egzemplarz własnego programu ładującego klasy — celem załadowania nowej wersji. Niektóre serwery poprawiają wydajność poprzez sprawdzanie znaczników modyfikacji czasu tylko co jakiś czas lub na wyraźne żądanie administratora.

W starszych wersjach Interfejsów API (sprzed 2.2) inne serwlety ładowane były przez odmienne programy ładujące — co powodowało czasem zgłoszenie `ClassCastException` jako wyjątku, kiedy serwlety wymieniały informacje (ponieważ klasa załadowana przez jeden program ładujący nie jest tym samym, co klasa ładowana przez inny, nawet jeżeli dane dotyczące klasy są identyczne).

Interfejs API 2.2 jest gwarancja, że problemy z `ClassCastException` nie pojawią się dla serwletów w tym samym kontekście. Tak więc obecnie większość implementacji serwerów ładuje każdy kontekst aplikacji WWW w jednym programie ładującym klasy oraz używa nowego programu ładującego do załadowania całego kontekstu, jeżeli jakikolwiek serwlet w kontekście ulegnie zmianie.

Skoro więc wszystkim serwletom oraz klasom obsługującym w kontekście zawsze odpowiada ten sam program ładujący, nie należy się obawiać żadnych, nieoczekiwanych wyjątków `ClassCastException` podczas uruchamiania. Powtórne ładowanie całego kontekstu powoduje mały spadek wydajności, który jednak występuje tylko podczas tworzenia.

Powtórne ładowanie (odnawianie) klasy nie jest przeprowadzane tylko wtedy, kiedy zmianie ulega klasaosbługująca. Celem większej efektywności określenia, czy jest konieczne odnawianie kontekstu, serwery sprawdzają tylko znaczniki czasu serwletów klasy. Klasy obsługujące w *WEB-INF/classes* mogą być także powtórnie załadowane, gdy kontekst jest odnowiony, lecz jeżeli klasa obsługująca jest jedyną klasą do zmiany, serwer prawdopodobnie nie zauważy tego.

Odnowienie serwletu nie jest także wykonywane dla wszystkich klas (serwletu lub innych) dostępnych w ścieżce klasy serwera. Klasy takie ładowane są przez rdzenny (pierwotny) program ładujący, a nie własny, konieczny do powtórnego załadowania. Są również ładowane jednorazowo i przechowywane w pamięci nawet wtedy, gdy ich pliki ulegają zmianie. Jeżeli chodzi o klasy globalne (takie jak klasy narzędziowe `com.oreilly.servlet`) to najlepiej jest umieścić je gdzieś na ścieżce klasy, gdzie unikną odnowienia. Przyśpiesza to proces powtórnego ładowania oraz pozwala serwletom w innych kontekstach wspólnie używać tych obiektów bez `ClassCastException`.

Metody „Init" i „Destroy"

Serwlety, podobnie jak aplety mogą określać metody `init()` i `destroy()`. Serwer wywołuje metodę `init()` zaraz po utworzeniu obiektu, jednak zanim jeszcze serwlet obsłuży jakiekolwiek

zlecenie. Serwer wywołuje metodę `destroy()` po wyłączeniu serwletu i po zakończeniu wszystkich zleceń lub po przekroczeniu ich limitu czasowego[3].

W zależności od rodzaju serwera oraz konfiguracji aplikacji WWW metoda `init()` może zostać wywołana w poniższych momentach:

- podczas uruchamiania serwletu,
- podczas pierwszego zlecenia obsługi, przed wywołaniem metody `service()`,
- na żądania administratora serwera.

W każdym przypadku metoda `init()` zostanie wywołana i zakończona zanim serwlet obsłuży swoje pierwsze zlecenie.

Metoda `init()` jest zwykle wykorzystywana do inicjalizacji serwletu — tworzenia lub ładowania obiektów używanych przez serwlet w procesie obsługi zleceń. W czasie wykorzystywania metody `init()` serwlet może chcieć odczytać swoje parametry inicjalizacji (`init`), które są dostarczane samemu serwletowi i nie są w jakikolwiek sposób związane z jednym zleceniem. Mogą one określać takie wartości początkowe jak wartość początkowa licznika lub wartości domyślne, takie jak np. szablon, który powinien zostać użyty w przypadku nie określenia tego w zleceniu. Parametry początkowe serwletu można znaleźć w deskryptorze wdrożenia (`web.xml`). Niektóre serwery mają graficzne interfejsy, mogące zmodyfikować ten plik (przykład 3.3).

Przykład 3.3. Ustalanie wartości parametrów w deskryptorze rozmieszczenia

```
<?xml version="1.0" encoding="ISO-8859-2"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            counter
        </servlet-name>
        <servlet-class>
            InitCounter
        </servlet-class>
        <init-param>
            <param-name>
                initial
            </param-name>
            <param-value>
                1000
            </param-value>
            <description>
                To jest wartość początkowa dla counter <!-- opcjonalnie -->
            </description>
```

[3] Specyfikacje projektów, mającego ukazać się na rynku Interfejsu API 2.3 (Servlet API 2.3), zakładają, że dodane zostaną metody cyklu życia (czasu istnienia), które umożliwią serwletom oczekiwanie na sygnały, kiedy kontekst lub sesja są tworzone lub zakańczane oraz podczas wiązania i rozwiązywania atrybutu z kontekstem lub sesją.

```
        </init-param>
      </servlet>
</web-app>
```

Wielokrotne elementy `<init-param>` mogą zostać umieszczone w znaczniku `<servlet>`. Znacznik `<description>` jest opcjonalny, pierwotnie miał być przeznaczony do graficznych programów narzędziowych. Pełną definicję typu dokumentu do pliku *web.xml* można znaleźć w Dodatku F „Kodowania".

Podczas stosowania metody `destroy()`, serwlet powinien zwolnić wszystkie zasoby, które wcześniej pozyskał, takie które nie będą automatycznie usnięte. Metoda `destroy()` daje również serwletowi możliwość zapisania składowanych informacji nie zapisanych dotychczas lub innych trwałych informacji, które powinny zostać odczytane podczas kolejnego wywołania metody `init()`.

Licznik z metodą Init

Parametry początkowe mają wiele zastosowań. Jednak przede wszystkim określają początkowe lub domyślne wartości zmiennych serwletu lub przekazują serwletowi informację, jak w określony sposób dostosować jego zachowanie. W przykładzie 3.4 nasz `SimpleCounter` został rozszerzony celem odczytania parametru początkowego (zwanego `initial`), który przechowuje wartość początkową dla naszego licznika. Poprzez ustawianie początkowego stanu licznika na wysokie wartości można sprawić, że nasza strona będzie popularniejsza niż w rzeczywistości.

Przykład 3.4. Licznik odczytujący parametry początkowe

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitCounter extends HttpServlet {

  int count;


  public void init() throws ServletException {
    String initial = getInitParameter("initial");
    try {
      count = Integer.parseInt(initial);
    }

    catch (NumberFormatException e) {
     count = 0;
    }
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {

    res.setContentType ("text/plain; charset=ISO-8859-2");
    PrintWriter out = res.getWriter();
    count++;
    out.println ("Od załadowania serwletu "
                ➥+ "(być może z wartością początkową pobraną z parametru)");
    out.println ("z serwletem tym łączono się");
    out.println (count + " razy.");
  }
}
```

Co się stało z super.init(config)?

W Interfejsie API 2.0 serwlet, implementujący metodę `init()`, musiał implementować również jej formularz, który przejmował parametr `ServletConfig` oraz wywołać `super.init(config)`:

```
public void init(ServletConfig config) throws ServletException {
 super.init(config);
 //  poniżej właściwy kodu metody}
```

Parametr `ServletConfig` dostarczał informacji serwletowi o konfiguracji, a wywołanie `super.init(config)` przekazywało obiekt konfiguracyjny do nadklasy `GenericServlet`, gdzie był zapisywany do użytku serlwetu. Klasa `GenericServlet` specjalnie używała przekazanego parametru `config` celem implementacji samego interfejsu `ServletConfig` (przekazując wszystkie wywołania do delegowanej konfiguracji i pozwalając serwletowi na wywołanie metod z `ServletConfig` — dla wygody).

Powyższa operacja była bardzo zawiła, lecz w Interfejsie API 2.1, została jednak uproszczona do tego stopnia, że obecnie wystarczy, aby serwlet implementował wersję bezargumentową `init()`, a obsługa `ServletConfig` i `GenericServlet` będzie zrealizowana na dalszym planie. Klasa `GenericServlet` współpracuje z bezargumentową metodą `init`, z kodem przypominającym poniższy:

```
public class GenericServlet implements Servlet, ServletConfig {

 ServletConfig _config = null;

 public void init(ServletConfig config) throws ServletException {
   _config = config;
   log("init wywołano");
   init();
 }


 public void init() throws ServletException { }

 public String getInitParameter(String name) {
   return _config.getInitParameter(name);
 }

// itd. ...
}
```

Zwróćmy uwagę, iż serwer wywołuje w czasie inicjalizacji metodę serwletu `init(ServletConfig config)`. Zmiana w wersji 2.1 dotyczyła tego, iż obecnie `GenericServlet` przekazuje to wywołanie do bezargumentowej metody `init()`, którą można zignorować, nie martwiąc się o `config`. Jeżeli chodzi o zgodność z poprzednimi wersjami należy nadal nadpisać `init(ServletConfig config)` i wywoływać `super.init(config)`. W przeciwnym wypadku być może nie będziemy mogli wywoływać metody bezargumentowej `init()`.

Niektórzy z programistów uważają, iż dobrze jest wywołać najpierw `super.destroy()` podczas implementacji `destroy()`. Powoduje to, że implementacja metody `destroy()` z `GenericServlet` informuje rejestr zdarzeń, że serwlet jest niszczony.

Metoda `init()` wykorzystuje metodę `getInitParameter()` w celu uzyskania wartości parametru zwanego `initial`. Metoda ta pobiera nazwę parametru jako `String` i zwraca wartość również jako `String`. Nie ma możliwości uzyskania wartości innego typu. Dlatego serwlet ten przekształca wartość `String` w wartość `int` lub w razie problemów domyślnie ustawia wartość na `0`. Należy pamiętać, że jeżeli chcemy wypróbować ten przykład, może się okazać konieczne powtórne uruchomienie serwera celem wprowadzenia zmian w *web.xml* oraz odniesienie się do serwletu poprzez użycie zarejestrowanej nazwy.

Licznik z metodami Init i Destroy

Dotychczas przykłady liczników demonstrowały, jak stan serwletu utrzymuje się pomiędzy połączeniami. To jednak rozwiązuje problem tylko częściowo. Za każdym razem, kiedy serwer jest wyłączany lub serwlet odnawiany, liczenie zaczyna się od nowa. Rzeczą naprawdę potrzebną jest trwanie licznika niezależnie od ładowań — licznik, który nie zaczyna ciągle od początku.

To zadanie mogą wykonać metody `init()` i `destroy()`. Przykład 3.5 poszerza wcześniej omówiony `InitCounter` poprzez dodanie serwletowi możliwość zachowania swojego stanu podczas wykonywania `destroy()` oraz ponownego odczytania w `init()`. Dla uproszczenia przyjmijmy, że serwlet ten nie jest zarejestrowany i dostępny tylko pod adresem *http://server:port/servlet/InitDestroyCounter*. Gdyby ten serwlet był zarejestrowany pod różnymi nazwami, musiałby zachowywać oddzielny stan dla każdej z nazw.

Przykład 3.5. Trwały licznik

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitDestroyCounter extends HttpServlet {

  int count;

  public void init() throws ServletException {
    //Spróbuj odczytać wartość początkową dla count z zapisanego
    //i trwałego stanu
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;
    try {
      fileReader = new FileReader("InitDestroyCounter.initial");
      bufferedReader = new BufferedReader(fileReader);
      String initial = bufferedReader.readLine();
      count  = Integer.parseInt(initial);
      return;
    }
    catch (FileNotFoundException ignored) { } // brak stanu zapisanego
    catch (IOException ignored) { }           // problem podczas czytania
    catch (NumberFormatException ignored) { } // niepoprawnie zapisany stan
    finally {
      // Nie zapomnij zamknąć pliku
      try  {
        if (bufferedReader != null) {
          bufferedReader.close();
        }
      }
      catch (IOException ignored) {}
    }
```

```
    // W razie niepowodzenia, sprawdź parametr inicjalizujący
    String initial = getInitParameter("initial");
    try {
      count = Integer.parseInt(initial);
      return;
    }
    catch (NumberFormatException ignored) {}  // null lub liczba nie całkowita

    // Domyślne dla początkowego stanu licznika "0"
    count = 0;
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType ("text/plain; charset=ISO-8859-2");
    PrintWriter  out  = res.getWriter();
    count++;
    out.println ("Z serwletem połączono się już " + count + " razy.");
  }

  public void destroy() {
    super.destroy(); //całkowicie opcjonalne
    saveState();
  }

  public void saveState() {
    // Spróbuj zapisać bieżącą wartość
    FileWriter fileWriter = null;
    PrintWriter printWriter = null;
    try {
      fileWriter = new FileWriter("InitDestroyCounter.initial");
      printWriter = new PrintWriter(fileWriter);
      printWriter.println(count);
      return;
    }
    catch (IOException e) { // problem podczas pisania
      // Zgłoś wyjątek. Patrz rozdział 5.
    }
    finally {
      // Nie zapomnij zamknąć plik
      if (printWriter != null) {
        printWriter.close();
      }
    }
  }
}
```

Za każdym razem, gdy serwlet jest usuwany, jego stan jest zachowywany w pliku o nazwie *InitDestroyCounter.initial*. Jeżeli nie ma dostarczonej ścieżki dostępu, plik jest zapisywany w bieżącym katalogu procesu serwera, zwykle jest to katalog startowy. Sposoby alternatywnej lokalizacji opisano w rozdziale 4., „Pobieranie informacji". Plik ten zawiera liczbę całkowitą, zapisaną jako ciąg znaków reprezentujący ostatnie liczenie.

Przy każdym ładowaniu serwera pojawia się próba odczytu z pliku zachowanego stanu licznika. Jeżeli z jakiegoś powodu próba odczytu nie powiedzie się (jak ma to miejsce podczas pierwszego uruchomienia serwletu, ponieważ plik jeszcze wtedy nie istnieje), serwlet sprawdza, czy parametr inicjalizujący jest określony.W razie niepowodzenia zaczyna od zera. Podczas stosowania metody `init()` zalecana jest najwyższa ostrożność.

Serwlety mogą zachowywać swój stan na wiele różnych sposobów. Niektóre z nich mogą posłużyć się formatem użytkowym pliku, tak jak zostało to pokazane w niniejszym rozdziale. Inne serwery zapisują swój stan jak zserializowane obiekty Javy lub umieszczają go w bazie danych. Niektóre wykorzystują nawet technikę *journaling*, powszechnie stosowaną przy bazach danych oraz przy kopiach zapasowych taśm, gdzie pełny stan serwletu jest zapisywany rzadko, podczas gdy plik dziennika wprowadza do pamięci przyrostowe aktualizacje w trakcie zmian. To, której metody użyje serwlet, zależy od sytuacji. Powinniśmy być zawsze świadomi tego, iż zapisywany stan nie podlega żadnym zmianom w tle.

Teraz może nasuwać się pytanie, co się stanie, jeżeli serwer ulegnie awarii? Metoda `destroy()` nie zostanie wywołana[4]. Nie jest to jednak problem dla metod `destroy()`, które muszą tylko zwolniać zasoby; przeładowany serwer równie dobrze się do tego nadaje (czasem nawet lepiej). Sytuacja taka jest problemem dla serwletu, który musi zapisywać swój stan we własnej metodzie `destroy()`. Ratunkiem dla tych serwletów jest częstsze zapisywanie swojego stanu. Serwlet może „wybrać" zapisanie swojego stanu po obsłudze każdego ze zleceń, jak powinien to zrobić serwer szachowy (chess server), że nawet gdy serwer jest ponownie uruchamiany, gra może zostać wznowiona z ostatnim układem na szachownicy. Inne serwlety mogą potrzebować zapisać stan tylko po zmianie jakiejś ważnej wartości — lista zakupów (shopping cart) serwlet musi zapisać swój stan tylko wtedy, gdy klient doda lub usunie pozycję z listy. I w końcu niektóre serwlety mogą tracić część swoich ostatnich zmian stanu. Takie serwlety mogą zapisywać stan po pewnej określonej liczbie zleceń, np. w naszym `InitDestoyCounter` wystarczającym powinno być zapisywanie stanu co dziesięć połączeń. Celem zaimplementowania tego można dodać prosty wiersz na końcu `doGet()`:

```
if (count % 10 == 0) saveState();
```

Można zapytać, czy jest to istotna zmiana? Wydaje się, że tak, biorąc pod uwagę zagadnienia związane z synchronizacją. Stworzyliśmy możliwość utraty danych (jeśli `saveState()` zostanie uruchomiony przez dwa wątki w tym samym czasie) oraz ryzyko, że `saveState()` nie będzie w ogóle wywołane, jeżeli liczenie zostanie zwiększone przez kilka wątków z rzędu przed sprawdzeniem. Załóżmy, że taka możliwość nie istniała, kiedy `saveState()` było wywoływane tylko z metody `destroy()`, bo metoda ta jest wywoływana tylko raz na jeden egzemplarz serwletu. Jednak teraz, kiedy `saveState()` jest wywoływana w metodzie `doGet()` musimy ponownie się nad tym zastanowić. Jeżeli zdarzyłoby się kiedyś, że serwlet ten byłby odwiedzany tak często, iż powstałoby więcej niż 10 jednocześnie wykonujących się wątków, jest prawdopodobne, że dwa serwlety (10 osobnych zleceń) będą w `saveState()` w tym samym czasie. Może to spowodować zniszczenie pliku z danymi lub doprowadzić do jednoczesnego zwiększenia count przez dwa watki, zanim któryś „zorientuje się", iż minął czas wywoływania `saveState()`. Rozwiązanie jest proste. Przemieśćmy kontrolę liczenia do bloku zsynchronizowanego, tam gdzie count jest zwiększane:

```
int local_count;
synchronized(this) {
  local_count = ++count;
```

[4] Jeżeli mamy szczęście i nasz serwer nie będzie miał awarii podczas stosowania metody `destroy()`. W przeciwnym razie możemy zostać z częściowo zapisanym plikiem stanu — pozostałościami zapisanymi na górze naszego poprzedniego stanu. Celem osiągnięcia całkowitego bezpieczeństwa, serwlet powinien zapisać swój stan w pliku tymczasowym, kopiując go następnie na góre oficjalnego pliku stanu w jednym poleceniu.

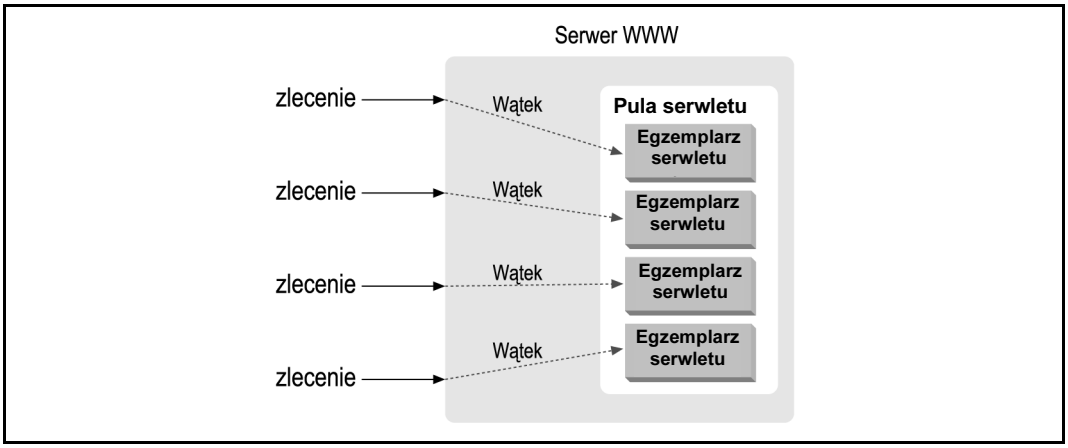
```
    if (count % 10 == 0)  saveState();
  }
  out.println("Z serwletem  połączono się " + count + " razy.");
```

Wniosek z powyższych rozważań jest jeden: bądźmy przezorni i chrońmy kod serwletu przed problemami związanymi z wielowątkowym dostępem.

Model jednowątkowy (Single Thread Model)

Mimo iż typową sytuacją jest jeden egzemplarz serwletu na jedną zarejestrowaną nazwę serwletu, to możliwa jest również pula egzemplarzy utworzonych dla każdej z nazw serwletu, której każdy egzemplarz obsługuje zlecenia. Serwlety sygnalizują taka chęć poprzez zaimplementowanie interfejsu `javax.servlet.SingleThreadModel`. Jest to prosty interfejs *tag*, który nie określa żadnych metod ani zmiennych, służy tylko do oznaczenia serwletu jako „wyrażającego chęć" zmiany cyklu życia.

Serwer, który ładuje serwlet typu `SingleThreadModel` musi gwarantować, zgodnie z dokumentacją Interfejsu API, że żadne dwa wątki nie będą wywoływały jednocześnie jego metody „service". W celu spełnienia powyższego warunku każdy wątek używa wolnego egzemplarza serwletu z puli, tak jak na rysunku 3.3, dzięki temu każdy serwlet, implementując `SingleThreadModel`, może zostać uznany za bezpieczny co do wątku oraz nie wymagający synchronizacji dostępu do jego zmiennych. Niektóre serwery dopuszczają konfigurację wielu egzemplarzy na pulę, inne — nie. Z kolei niektóre używają puli tylko z jednym egzemplarzem, powodując zachowanie identyczne z metodą zsynchronizowaną `service()`.



Rysunek 3.3. Model jednowątkowy

Czas istnienia modelu jednowątkowego (`SingleThreadModel`) nie jest używany do liczników lub innych aplikacji, które wymagają obsługi centralnego stanu. Czas istnienia może mieć pewne zastosowanie, jednak tylko w unikaniu synchronizacji, ciągle obsługując zlecenie sprawnie.

Dla przykładu, serwlety, które łączą się z bazami danych muszą czasem wykonać kilka poleceń wewnętrznych bazy, niepodzielnie jako część pojedynczej transakcji. Każda transakcja wykonywana na bazie danych wymaga wydzielonego obiektu połączenia z bazą, dlatego serwlet musi

zagwarantować, że żadne dwa wątki nie będą próbowały „wchodzić" na to samo połączenie w tym samym czasie. Można tego dokonać poprzez użycie synchronizacji i „pozwolenie" serwletowi na obsługę tylko jednego zlecenia w jednym czasie. Dzięki implementowaniu `SingleThreadModel` oraz posiadaniu tylko jedno obiektu połączenia na serwlet, może on w prosty sposób obsługiwać konkurencyjne (jednoczesne) zlecenia, ponieważ każdy egzemplarz będzie miał swoje połączenie. Zarys kodu pokazano w przykładzie 3.6.

Przykład 3.6. Obsługa połączeń z baza danych z użyciem modelu jednowątkowego

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SingleThreadConnection extends HttpServlet
                                    implements SingleThreadModel {

  Connection con = null; // połączenie z bazą danych,
                         // jedno na każdy egzemplarz z puli

  public void init() throws ServletException {
    // Rozpocznij połączenie dla tego egzemplarza
    try {
      con = estabilishConnection();
      con.setAutoCommit(false);
    }
    catch (SQLException e) {
      throw new ServletException(e.getMessage());
    }
  }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType ("text/plain");
    PrintWriter  out  = res.getWriter();

    try {
      // Użyj połączenia utworzonego specjalnie dla tego egzemplarza
      Statement stmt = con.createStatement();

      // Aktualizuj bazę danych jakimikolwiek sposobami

      // Zatwierdź transakcję
      con.commit();
    }
    catch (SQLException e) {
      try { con.rollback(); } catch (SQLException ignored) { }
    }
  }

  public void destroy() {
    if (con != null) {
      try { con.close(); } catch (SQLException ignored) { }
    }
  }

  private Connection estabilishConnection() throws SQLException {
    // Nie zaimplementowane. Patrz rozdział 9.
  }
}
```

W rzeczywistości `SingleThreadModel` nie jest najlepszym rozwiązaniem dla tego typu aplikacji. O wiele lepszym byłoby dla serwletu użycie wydzielonego obiektu `ConnectionPool` przechowywanego jako zmienna klasy, za pomocą którego mógłby zarządzać połączeniami. Połączenie usunięte z ewidencji może być przechowywane jako lokalna zmienna, zapewniająca wydzielony dostęp. Zewnętrzna pula zapewnia serwletowi więcej kontroli nad zarządzaniem połączeniami. Pula może również zweryfikować poprawność każdego połączenia i zostać skonfigurowana w taki sposób, że będzie zawsze tworzyła pewną minimalną liczbę połączeń, lecz nigdy większą niż określona liczba maksymalna. Podczas używania modelu jednowątkowego (`SingleThreadModel`), serwer mógłby utworzyć znacznie więcej egzemplarzy (a tym samym połączeń) niż baza danych może obsłużyć.

Zatem należy unikać stosowania metody `SingleThreadModel`. Większość innych serwletów mogłaby być lepiej implementowana z użyciem synchronizacji oraz puli zasobów zewnętrznych. Prawdą jest, iż interfejs daje pewien stopień kontroli programistom nie znającym programowania wielowątkowego; choć, gdy `SingleThreadModel` czyni sam serwlet bezpiecznym co do wątku, to interfejs nie czyni tego z systemem. Interfejs nie zapobiega problemom związanym z synchronizacją, które wynikają z jednoczesnego dostępu serwletów do wspólnych zasobów, takich jak np. zmienne statyczne czy obiekty poza zasięgiem serwletu. Problemy związane z wątkami będą się pojawiały zawsze podczas pracy w systemie wielowątkowym — z lub bez `SingleThreadModel`.

Przetwarzanie w tle

Serwlety potrafią więcej niż tylko utrzymywać się pomiędzy kolejnymi odwywołaniami do nich. Każdy wątek uruchomiony przez serwlet może kontynuować wykonywanie nawet po wysłaniu odpowiedzi. Możliwość ta najlepiej sprawdza się przy dłuższych zadaniach, których wyniki przyrostowe są udostępniane wielu klientom. Wątki drugoplanowe, uruchomione w `init()`, wykonują pracę w sposób ciągły, podczas gdy wątki obsługujące zlecenia wyświetlają stan bieżący za pomocą `doGet()`. Jest to technika podobna do używanej w apletach animacyjnych, gdzie jeden wątek dokonuje zmian na rysunku, a drugi nanosi kolory.

Przykład 3.7 ukazuje serwlet wyszukujący liczb pierwszych, większych od kwadryliona. Tak wielka liczba wybierana jest celowo, żeby uczynić liczenie powolnym i aby zademonstrować efekty buforujące (które będą potrzebne przy omawianiu dalszej części materiału). Algorytm używany przez serwlet jest bardzo prosty. Serwlet selekcjonuje wszystkie liczby nieparzyste i następnie próbuje podzielić je przez liczby nieparzyste całkowite z przedziału od 3 do ich pierwiastka kwadratowego. Jeżeli dana liczba nie jest podzielna bez reszty przez żadną tych liczb, wtedy jest uznawana za liczbę pierwszą[5].

Przykład 3.7. W poszukiwaniu liczb pierwszych

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

[5] Można zapytać, dlaczego sprawdzane są tylko czynniki mniejsze od pierwiastka kwadratowego? Ponieważ, jeżeli liczba miałaby dzielniki wśród liczb większych od tego pierwiastka, to musiałaby je także mieć wśród liczb od niego mniejszych.

```
public class PrimeSearcher extends HttpServlet implements Runnable {

  long lastprime = 0;                   // ostatnia znaleziona liczba pierwsza
  Date lastprimeModified = new Date();  // kiedy została znaleziona
  Thread searcher;                      // drugoplanowy wątek szukający

  public void init() throws ServletException {
    searcher = new Thread(this);
    searcher.setPriority(Thread.MIN_PRIORITY); // bądź dobrym obywatelem
    searcher.start();
  }

  public void run() {
    //              QTTTBBBMMMTTTOOO
    long candidate = 1000000000000001L;        // kwadrylion jeden

    // Rozpocznij szukanie liczb pierwszych
    while (true) {                             // szukaj cały czas
      if (isPrime(candidate)) {
        lastprime = candidate;                 // nowa liczba pierwsza
        lastprimeModified = new Date();        // nowy "czas" liczby pierwszej
      }
      candidate += 2;                          // liczby parzyste nie są
                                               // liczbami pierwszymi

      // Pomiędzy potencjalnymi liczbami pierwszymi rób przerwę 0.2 sekundy
      // Kolejny sposób aby być dobrym obywatelem w zasobach systemu
      try {
        searcher.sleep(200);
      }
      catch (InterruptedException ignored) {}
    }
  }

  private static boolean isPrime(long candidate) {
    // Spróbuj podzielić tę liczbę przez wszystkie liczby
    // nieparzyste z przedziału od 3 do jej pierwiastka kwadratowego

    long sqrt  = (long)Math.sqrt (candidate);
    for (long i = 3; i <= sqrt; i += 2) {
      if (candidate % i == 0) return false;  // znajdź czynnik
    }

    // jeśli jest pierwsza
    return true;
  }
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType ("text/plain; charset=ISO-8859-2");
    PrintWriter  out  = res.getWriter();
    if (lastprime == 0) {
      out.println ("Nadal szukam liczb pierwszych...");
    }
    else{
      out.println("Ostatnia liczba pierwsza została znaleziona " + lastprime);
      out.println(" w " + lastprimeModified);
    }
  }

  public void destroy() {
    searcher.stop();
  }
}
```

Wątek wyszukujący rozpoczyna wyszukiwanie w metodzie `init()`. Ostatnia liczba przez niego znaleziona zostaje zapisana w `lastprime`, a czas (w którym to się stało) w `lastprimeModified`. Za każdym razem, kiedy klient łączy się z serwletem, metoda `doGet()` informuje go, jaka największa liczba pierwsza została znaleziona do tej pory oraz w jakim czasie to się stało. Wątek niezależnie przetwarza połączenia klientów i nawet jeśli żaden z nich nie jest połączony, wątek nadal kontynuuje poszukiwania liczb pierwszych. Jeżeli kilku klientów połączy się w tym samym czasie z serwletem, dla wszystkich zostanie wyświetlony bieżący stan.

Zwróćmy uwagę, iż metoda `destroy()` zatrzymuje wątek wyszukujący. Jest to bardzo istotne, ponieważ gdy serwlet nie zatrzyma swoich drugoplanowych wątków, będą one działały tak długo, jak długo będzie istniała maszyna wirtualna. Nawet jeżeli serwlet zostanie powtórnie załadowany (odnowiony; jawnie bądź z powodu zmiany pliku klasy), jego wątki nie przestaną działać. Zamiast tego jest prawdopodobne, że nowy serwlet utworzy kopie dodatkowewątków drugoplanowych.

Ładowanie i uruchamianie

Aby sprawić, że `PrimeSearcher` zacznie szukać liczb pierwszych tak szybko jak to możliwe, możemy skonfigurować aplikację WWW serwletu w taki sposób, że będzie ładowała serwlet przy starcie serweru. Dokonuje się tego poprzez dodanie znacznika `<load-on-startup>` do wpisu `<servlet>` deskryptora wdrożenia, tak jak w przykładzie 3.8.

Przykład 3.8. Ładowanie serwletu przy rozruchu

```
<?xml version = "1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            ps
        </servlet-name>
        <servlet-class>
            PrimeSearcher
        </servlet-class>
        <load-on-startup/>
    </servlet>
</web-app>
```

Powyższe znaczniki powodują, że serwer utworzy egzemplarz klasy `PrimeSearcher` pod rejestrowaną nazwą `ps` oraz że zainicjalizuje serwlet podczas sekwencji uruchamiania serwera. Z serwletem można połączyć się wtedy na URL-u */servlet/ps*. Zwróćmy uwagę, iż egzemplarz serwletu obsługujący URL */servlet/PrimeSearcher* nie jest ładowany przy rozruchu.

W przykładzie 3.8 znacznik `<load-on-startup>` jest pusty. Znacznik może również zawierać dodatnią liczbę całkowitą, oznaczającą kolejność, w której serwlet powinien być załadowany w odniesieniu do innych serwletów kontekstu. Te z niższymi liczbami ładowane są przed mającymi większe liczby. Serwlety z wartościami ujemnymi lub niecałkowitymi mogą być ładowane w każdym momencie sekwencji uruchamiania, dokładna kolejność zależy od serwera, np. *web.xml*, ukazany w przykładzie 3.9, gwarantuje, że `first` będzie załadowany przed `second`, podczas gdy `anytime` może zostać załadowany w każdym momencie rozruchu serwera.

Przykład 3.9. Możliwości serwletu

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>
            first
        </servlet-name>
        <servlet-class>
            First
        </servlet-class>
        <load-on-startup >10</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>
            second
        </servlet-name>
        <servlet-class>
            Second
        </servlet-class>
        <load-on-startup >20</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>
            anytime
        </servlet-name>
        <servlet-class>
            Anytime
        </servlet-class>
        <load-on-startup/>
    </servlet>
</web-app>
```

Buforowanie podręczne po stronie klienta

Do tej pory mówiono, że serwlety obsługują zlecenia `GET` za pomocą metody `doGet()`. I jest to niemal prawda. Cała prawda jest taka, że nie do każdego zlecenia konieczne jest wywoływanie metody `doGet()`. Dla przykładu, przeglądarka WWW, która stale łączy się z `PrimeSearcher`, będzie musiała wywołać `doGet()` tylko po tym, jak wątek wyszukujący znajdzie nową liczbę pierwszą. Do tego czasu wszystkie wywołania `doGet()` generują po prostu stronę, którą użytkownik już oglądał, stronę prawdopodobnie przechowywaną w pamięci podręcznej przeglądarki. To, co jest teraz najbardziej potrzebne, to sposób w jaki serwlet mógłby informować o zmianach na wyjściu . I tutaj właśnie pomocne będzie opisanie metody `getLastModified()`.

Większość serwerów WWW zawiera, jako część swojej odpowiedzi, nagłówek `Last-Modified`, wtedy gdy odsyła dokument. Przykład wartości nagłówka `Last-Modified` mógłby wyglądać w następujący sposób:

```
Sun, 14 Oct 2001 19:15:06 GMT
```

Powyższy nagłówek informuje klienta o tym, kiedy ostatnio została zmieniona strona. Informacja sama w sobie nie jest specjalnie wartościowa, jednak zyskuje na wartości w momencie, gdy przeglądarka powtórnie ładuje stronę.

Większość przeglądarek WWW podczas odnawiania strony zawiera w swoich zleceniach nagłówek `If-Modified-Since`, którego struktura jest identyczna z nagłówkiem `Last-Modified`:

```
Sun, 14 Oct 2001 19:15:06 GMT
```

Nagłówek ten informuje serwer o czasie(`Last-Modified`) ostatnio pobranej przez przeglądarkę strony. Serwer może odczytać ten nagłówek oraz stwierdzić, czy plik był zmieniany od określonego czasu. Jeżeli okaże się, że plik został zmieniony, serwer musi przesłać nową treść. Jeżeli okażę się, że nie uległ zmianie, wtedy serwer może wysłać przeglądarce krótką odpowiedź oraz poinformować, że wystarczy powtórne wyświetlenie wersji dokumentu trzymanej w pamięci podręcznej przeglądarki (ta odpowiedź to: kod stanu 304 `Not Modified`).

Technika ta działa najlepiej w odniesieniu do stron statycznych. Serwer może użyć systemu plików w celu sprawdzenia, kiedy określony plik został zmodyfikowany po raz ostatni. Ale dla treści tworzonej dynamicznie, takiej jaka jest odsyłana przez serwlety, serwer potrzebuje dodatkowej pomocy. Wszystko, co może zrobić, to odtwarzać je bezpiecznie, zakładając jednocześnie, iż zawartość ulega zmianie z każdym połączeniem, eliminując w ten sposób konieczność użycia nagłówków `Last-Modified` oraz `If-Modified-Since`.

Serwlet może służyć dodatkową pomocą poprzez implementację metody `getLastModified()` w celu przesłania danych dotyczących czasu, w którym po raz ostatni zmienił swój wydruk wyjściowy. Serwery wywołują tę metodę dwa razy; pierwszy — kiedy odsyłają odpowiedzi (w celu wstawienia nagłówka odpowiedzi `Last-Modified`), a drugi — podczas obsługi zleceń `GET`, które zawierają nagłówek `If-Modified-Since`. Dzięki temu serwer może odpowiedzieć w sposób inteligentny. Jeżeli czas zwracany przez `getLastModified()` jest taki sam bądź wcześniejszy od czasu nadesłanego w nagłówku `If-Modified-Since`, serwer przesyła kod stanu `Not Modified`. W przeciwnym razie serwer wywołuje `metodę doGet()` oraz odsyła wynik wykonania serwletu[6].

Niektórym serwletom może sprawiać trudność ustalenie czasu ostatniej modyfikacji. Dlatego w takich sytuacjach najlepszym wyjściem jest użycie zachowania domyślnego o bezpieczeństwie (play it safe). Jednak większość serwletów doskonale daje sobie z tym radę. Rozważmy przypadek serwletu elektroniczny biuletyn informacyjny (bulletin board). Serwlet taki może zarejestrować i odesłać informację, kiedy po raz ostatni została zmieniona treść biuletynu. Nawet gdy ten sam serwlet obsługuje kilka biuletynów, nadal może przesyłać różne czasy modyfikacji — odpowiednie dla parametrów podanych w zleceniu. Oto metoda `getLastModified()` do naszego przykładu `PrimeSearcher`, która przesyła czas znalezienia ostatniej liczby pierwszej:

```
public long getLastModified(HttpServletRequest req) {
  return lastprimeModified.getTime() /1000 * 1000;
}
```

[6] Serwlet może wstawić swój nagłówek `Last-Modified` bezpośrednio w `doGet()` z użyciem technik omówionych w rozdziale 5., „Wysyłanie informacji HTML". Jednak do czasu, kiedy nagłówek zostanie wstawiony w `doGet()`, jest już zbyt późno, by zdecydować o tym, czy wywoływać `doGet()`, czy nie.

Zwróćmy uwagę, iż metoda ta przesyła wartość długą, która przedstawia czas jako liczbę milisekund, która upłynęła od 1 stycznia 1970 roku GMT. Takiej samej reprezentacji używa wewnętrznie Java w celu przechowywania wartości czasowych. W ten sposób serwlet używa metody `getTime()` w celu otrzymania `lastprimeModified` jako `long`.

Serwlet zanim odeśle tę wartość czasową, zaokrągla ją do najbliższej sekundy, dzieląc ją przez tysiąc, a następnie mnożąc przez tyle samo. Wszystkie czasy odesłane przez `getLastModified()` powinny być zaokrąglone w ten sposób. Jest to spowodowane tym, że nagłówki `Last-Modified` oraz `If-Modified-Since` są zaokrąglane do najbliższej sekundy. Jeżeli `getLastModified()` zwraca ten sam czas, lecz z wyższą rozdzielczością, to może on błędnie wydawać się kilka milisekund późniejszy od podanego przez `If-Modified-Since`. Załóżmy, że `PrimeSearcher` znalazł liczbę pierwszą, dokładnie 869 127 442 359 milisekund od wyżej wspomnianej daty. Fakt ten przekazywany jest przeglądarce, lecz tylko do najbliższej sekundy:

```
Thu, 17-Jul- 01   09:17:22 GMT
```

Teraz załóżmy, że użytkownik powtórnie ładuje stronę, a przeglądarka poprzez nagłówek `If--Modified-Since` podaje serwerowi czas, który uważa za ostatnią modyfikację:

```
Thu, 17-Jul-01   09:17:22 GMT
```

Niektóre serwery przyjmują ten czas, przeliczają go na dokładnie 869 127 442 000 milisekundy, uznają, iż jest on 359 milisekund wcześniejszy od odesłanego `przez getLastModified()`, a następnie fałszywie zakładają, że treść serwletu uległa zmianie. Dlatego właśnie, żeby zachować bezpieczeństwo (to play it safe), `getLastModified()` powinna zawsze zaokrąglać do najbliższego tysiąca milisekund. Obiekt `HttpServletRequest` jest przekazywany do `getLastModified()` w razie gdyby serwlet potrzebował oprzeć swoje rezultaty na informacjach specyficznych dla określonego zlecenia. Standardowy serwlet elektronicznego biuletynu informacyjnego może to wykorzystać, np. do określenia, który biuletyn jest przedmiotem zlecenia.

Buforowanie podręczne po stronie serwera

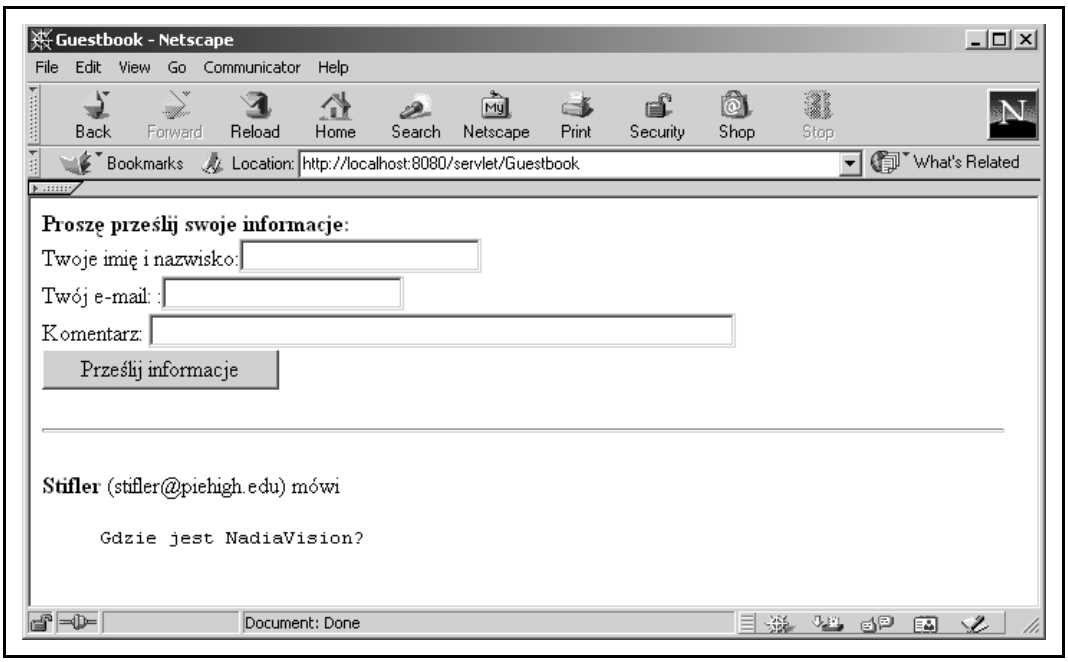
Metoda `getLastModified()` przy odrobinie pomysłowości może być pomocna w zarządzaniu pamięcią podręczną wyniku wykonania serwletu. Serwlety stosujące taki zabieg, mogą swoje wyniki wykonania przechwycać i umieszczać w pamięci podręcznej po stronie serwera, a następnie odsyłać do klientów, jak to ma miejsce w przypadku metody `getLastModified()`. Taka procedura może znacznie przyspieszyć tworzenie strony serwletu, szczególnie w odniesieniu do tych, którym dużo czasu zajmuje przetwarzanie, zmieniającego się rzadko, np. serwlety wyświetlające dane z bazy danych.

Celem zaimplementowania buforowania po stronie serwera, serwlet musi:

- rozszerzyć `com.oreilly.servlet.CacheHttpServlet` zamiast `HttpServlet`,
- zaimplementować metodę `getLastModified(HttpServletRequest)`.

Przykład 3.10 pokazuje serwlet korzystający z `CacheHttpServlet`. Jest to księga gości serwletu, która wyświetla złożone przez użytkowników komentarze. Serwlet przechowuje je w pamięci jako obiekty klasy `Vector` zawartych w `GuestbookEntry`.W rozdziale 9. („Łączność

z bazami danych") poznamy wersję tego serwletu działającą poza bazą danych. W celu symulacji czytania z wolnej bazy, pętla wyświetlająca ma półsekundowe opóźnienie na wejście. Im dłuższa lista wejść, tym wolniejsza wizualizacja strony. Z powodu rozszerzania przez serwlet `CacheHttpServlet`, wizualizacja musi mieć miejsce tylko podczas pierwszego zlecenia GET, po dodaniu nowego komentarza. Wszystkie późniejsze zlecenia GET wysyłają odpowiedź z pamięci podręcznej. Przykładowy rezultat wykonania został pokazany na rysunku 3.4.



Rysunek 3.4. Księga gości

Przykład 3.10. Księga gości używająca CacheHttpServlet

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

import com.oreilly.servlet.CacheHttpServlet;

public class Guestbook extends CacheHttpServlet {

  private Vector entries = new Vector();  // Lista wpisów użytkownika
  private long lastModified = 0;          // Czas, w którym został
                                          // dodany ostatni wpis

  // Wyświetl aktualne wpisy, następnie poproś o nowy wpis
  public void doGet (HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {
    res.setContentType ("text/html; charset=ISO-8859-2");
    PrintWriter out  = res.getWriter();

    printHeader(out);
    printForm(out);
```

```
    printMessages(out);
    printFooter(out);
  }

  // Dodaj nowy wpis, następnie odeślij z powrotem do doGet()
  public  void doPost (HttpServletRequest req, HttpServletResponse res)
                                  throws ServletException, IOException {
    handleForm(req, res);
    doGet(req, res);
  }

  private void printHeader(PrintWriter out) throws ServletException {
    out.println ("<HTML><HEAD><TITLE>Guestbook</TITLE></HEAD>");
    out.println ("<BODY>");
  }

  private void printForm(PrintWriter out) throws ServletException {
    out.println ("<FORM METHOD=POST>");   // przesyła do siebie
    out.println ("<B>Proszę prześlij swoje informacje:</B><BR>");
    out.println ("Twoje imię i nazwisko:<INPUT TYPE=TEXT NAME=name><BR>");
    out.println ("Twój e-mail: :<INPUT TYPE=TEXT NAME=email><BR>");
    out.println ("Komentarz: <INPUT TYPE=TEXT SIZE=50 NAME=comment><BR>");
    out.println ("<INPUT TYPE=SUBMIT VALUE=\"Prześlij informacje\"><BR>");
    out.println ("</FORM>");
    out.println ("<HR>");
  }

  private void printMessages(PrintWriter out) throws ServletException {
    String name, email, comment;

    Enumeration e = entries.elements();
    while (e.hasMoreElements()) {
      GuestbookEntry entry = (GuestbookEntry)e.nextElement();
      name = entry.name;
      if (name == null) name = "Nieznany użytkownik";
      email = entry.email;
      if(email == null) email = "Nieznany e-mail";
      comment = entry.comment;
      if (comment == null) comment = "Bez komentarza";
      out.println ("<DL>");
      out.println ("<DT><B>" + name + "</B> (" + email +") mówi");
      out.println ("<DD><PRE>" + comment + "</PRE>");
      out.println ("</DL>");

      // Wstrzymaj wykonywanie na pół sekundy w celu symulacji
      // powolnego źródła danych
      try {Thread.sleep(500);} catch  (InterruptedException ignored) { }
    }
  }

  private void printFooter(PrintWriter out) throws ServletException {
    out.println ("</BODY>");
  }

  private void handleForm (HttpServletRequest req, HttpServletResponse res) {
    GuestbookEntry entry =  new GuestbookEntry();

    entry.name = req.getParameter("name");
    entry.email = req.getParameter("email");
    entry.comment = req.getParameter("comment");
    entries.addElement(entry);
```

```
      // Zwróć uwagę, że mamy nowy czas ostatniej modyfikacji
      lastModified = System.currentTimeMillis();
    }

    public long getLastModified (HttpServletRequest req) {
      return lastModified;
    }
  }

  class GuestbookEntry {
    public String name;
    public String email;
    public String comment;
  }
```

Źródło `CacheHttpServlet` zostało pokazane w przykładzie 3.11. Kod, w tym miejscu książki, może wydawać się bardzo skomplikowany. Lepiej po lekturze rozdziału 5. spróbować odczytać kod tej klasy. Przed obsługą zlecenia klasa `CacheHttpServlet` sprawdza wartość `getLastModified()`. Jeżeli przetworzony kod w pamięci podręcznej jest co najmniej tak aktualny, jak czas ostatniej modyfikacji serwletu, wówczas kod ten wysyłany jest bez wywoływania metody `doGet()` serwletu.

Jeżeli klasa ta wykryje, że łańcuch zapytań, posiada dodatkową ścieżkę lub ścieżka serwletu została zmieniona, to w celu zachowania bezpieczeństwa, pamięć podręczna zostanie unieważniana i utworzona od nowa. Nie jest to jednak pamięć oparta na różnych nagłówkach zleceń lub cookies; w przypadku serwletów, które różną się od tych opartych na takich wartościach (tzn. serwerów śledzących sesję), klasa ta raczej nie powinna być używana albo metoda `getLastModified()` powinna odnosić się do nagłówków oraz cookies. Korzystanie z pamięci podręcznej nie jest stosowane przy zleceniach `POST`.

`CacheHttpServletResponse` oraz `CacheServletOutputStream` są klasami pomocniczymi dla klasy i nie powinny być używane bezpośrednio. Klasa została utworzona na podstawie Interfejsu API 2.2 (Servlet API 2.2), dlatego używanie jej z poprzednimi wersjami Interfejsu API przebiega poprawnie, choć używanie jej z przyszłymi wersjami prawdopodobnie nie będzie już przebiegało tak dobrze, ponieważ interfejs `HttpServletResponse`, który `CacheHttpServletResponse` musi zaimplementować, może ulec zmianie i w związku z tym niektóre metody interfejsowe pozostaną niezaimplementowane. Jeżeli natknęlibyśmy się na taki problem, to aktualna wersja tej klasy dostępna jest na stronie *http://www.servlets.com*.

Interesujący jest sposób w jaki `CacheHttpServlet` przechwytuje zlecenie w celu jego wczesnego przetworzenia. Otóż implementuje on metodę `service(HttpServletRequest, HttpServletResponse)`, którą serwer wywołuje w celu przekazania serwletowi kontroli nad obsługą zleceń. Standardowa implementacja `HttpServlet` tej metody przesyła zlecenie do `doGet()`, `doPost()` oraz innych metod zależnych od rodzaju zlecenia HTTP . `CacheHttpServlet` ignoruje taką implementację, zyskując tym samym pierwszeństwo kontroli nad obsługą zleceń. Kiedy klasa kończy przetwarzanie, może przekazać kontrolę z powrotem do `HttpServlet` za pomocą wywołania `super.service()`.

Przykład 3.11. Klasa CacheHttpServlet

```
package com.oreilly.servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public abstract class CacheHttpServlet extends HttpServlet {

  CacheHttpServletResponse cacheResponse;
  long cacheLastMod = -1;
  String cacheQueryString = null;
  String cachePathInfo = null;
  String cacheServletPath = null;
  Object lock = new Object();

  protected void service (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
  // Tylko dla zleceń GET korzystaj z buforowania
  String method = req.getMethod();
  if(!method.equals("GET")) {
    super.service(req, res);
    return ;
  }

  // Sprawdź czas ostatniej modyfikacji serwletu
  long servletLastMod = getLastModified(req);

  // Ostatnio zmodyfikowany -1 oznacza, że nie powinniśmy w ogóle
  // używać pamięci podręcznej
  if(servletLastMod == -1) {
    super.service(req, res);
    return ;
  }

  // jeżeli klient przysłał nagłówek If-Modified-Since, w lub po czasie
  // ostatniej modyfikacji serwletu prześlij krótki kod statusu
  // "Nie zmodyfikowany". Zaokrągl do najbliższej sekundy
  // jako że nagłówki klienta są w sekundach
  if ((servletLastMod / 1000 * 1000) <=
          req.getDateHeader("If-Modified-Since")) {
    res.setStatus(res.SC_NOT_MODIFIED);
    return ;
  }

  // Wykorzystaj istniejącą pamięć podręczną
  // jeżeli jest ona aktualna i poprawna
  CacheHttpServletResponse localResponseCopy = null;
  synchronized(lock) {
    if (servletLastMod <= cacheLastMod &&
            cacheResponse.isValid() &&
            equal (cacheQueryString, req.getQueryString()) &&
            equal (cachePathInfo, req.getPathInfo()) &&
            equal (cacheServletPath, req.getServletPath())) {
      localResponseCopy = cacheResponse;
    }
  }
```

```
    if (localResponseCopy != null) {
      localResponseCopy.writeTo(res);
      return ;
    }

    // W przeciwnym razie utwórz nowy bufor, aby zachować odpowiedź
    localResponseCopy = new CacheHttpServletResponse(res);
    super.service(req, localResponseCopy);
    synchronized (lock) {
      cacheResponse = localResponseCopy;
      cacheLastMod = servletLastMod;
      cacheQueryString = req.getQueryString();
      cachePathInfo = req.getPathInfo();
      cacheServletPath = req.getServletPath();
    }
  }

    private boolean equal(String s1, String  s2) {
      if (s1 == null && s2 == null) {
        return true;
      }
      else if (s1 == null || s2 == null) {
        return false;
      }
      else {
        return s1.equals(s2);
      }
    }
  }

  class CacheHttpServletResponse implements HttpServletResponse {
    // Przechowuj kluczowe zmienne odpowiedzi w celu
    // wstawienia ich później
    private int status;
    private Hashtable headers;
    private int contentLength;
    private String contentType;
    private Locale locale;
    private Vector cookies;
    private boolean didError;
    private boolean didRedirect;
    private boolean gotStream;
    private boolean gotWriter;

    private HttpServletResponse delegate;
    private CacheServletOutputStream out;
    private PrintWriter writer;

    CacheHttpServletResponse(HttpServletResponse res) {
      delegate = res;
      try {
        out = new CacheServletOutputStream(res.getOutputStream());
      }
      catch (IOException e) {
        System.out.println (
          "Wystąpił wyjątek IOException tworząc odpowiedź z pamięci podręcznej:
           ➥" + e.getMessage());
      }
      internalReset();
    }
```

```
private void internalReset() {
  status = 200;
  headers = new Hashtable();
  contentLength = -1;
  contentType = null;
  locale = null;
  cookies = new Vector();
  didError = false;
  didRedirect = false;
  gotStream = false;
  gotWriter = false;
  out.getBuffer().reset();
}

public boolean isValid() {
  // Nie przechowujemy w pamięci podręcznej stron z błędami
  // lub przekierowaniami
  return didError != true && didRedirect !=true;
}

private void internalSetHeader(String name, Object value) {
  Vector v = new Vector();
  v.addElement(value);
  headers.put(name, v);
}

private void internalAddHeader(String name, Object value) {
  Vector v = (Vector) headers.get(name);
    if (v == null) {
      v = new Vector();
  }
  v.addElement(value);
  headers.put(name, v);
}

public void writeTo(HttpServletResponse res) {
  // Zapisz kod statusu
  res.setStatus(status);
  // Zapisz nagłówki upraszczające
  if (contentType != null) res.setContentType(contentType);
  if (locale != null) res.setLocale(locale);
  // Zapisz cookies
  Enumeration enum = cookies.elements();
  while (enum.hasMoreElements()) {
    Cookie c =(Cookie) enum.nextElement();
    res.addCookie(c);
  }
  // Zapisz nagłówki standardowe
  enum = headers.keys();
  while(enum.hasMoreElements()) {
    String name = (String) enum.nextElement();
    Vector values = (Vector) headers.get(name); // może mieć
                                                // wielokrotne wartości
    Enumeration enum2 = values.elements();
    while (enum2.hasMoreElements()) {
      Object value = enum2.nextElement();
      if (value instanceof String) {
        res.setHeader(name, (String)value);
      }
      if (value instanceof Integer) {
        res.setIntHeader(name, ((Integer)value).intValue());
      }
```

```
      if (value instanceof Long) {
        res.setDateHeader(name, ((Long)value).longValue());
      }
    }
  }
  // Zapisz długość zawartości
  res.setContentLength(out.getBuffer().size());
  // Zapisz treść
  try {
    out.getBuffer().writeTo(res.getOutputStream());
  }
  catch (IOException e) {
    System.out.println(
      ➥"Wystąpił wyjątek IOException na pisemną odpowiedź tekstową
      ➥z pamięci podręcznej: " + e.getMessage());
  }
}

public ServletOutputStream getOutputStream() throws IOException {
  if (gotWriter) {
    throw new IllegalStateException(
      "Niemożliwy do otrzymania strumień wyjściowy po uzyskaniu
       ➥wykonawcy zapisu");
  }
  gotStream = true;
  return out;
}

public PrintWriter getWriter() throws UnsupportedEncodingException {
  if (gotStream) {
    throw new IllegalStateException(
      "Niemożliwy do otrzymania wykonawca zapisu po uzyskaniu
       ➥strumienia wyjściowego");
  }
  gotWriter = true;
  if (writer == null) {
    OutputStreamWriter w =
      new OutputStreamWriter(out, getCharacterEncoding());
    writer = new PrintWriter(w, true); // konieczne jest automatyczne
                                       // opróżnienie pamięci podręcznej
  }
  return writer;
}

public void setContentLength(int len) {
  delegate.setContentLength(len);
  // Nie ma potrzeby zapisywania długości, możemy obliczyć to później
}

public void setContentType(String type) {
  delegate.setContentType(type);
  contentType = type;
}

public String getCharacterEncoding () {
  return delegate.getCharacterEncoding ();
}

public void setBufferSize(int size) throws IllegalStateException {
  delegate.setBufferSize(size);
}
```

```
public int getBufferSize() {
  return delegate.getBufferSize();
}

public void reset() throws IllegalStateException {
  delegate.reset();
  internalReset();
}

public void resetBuffer() throws IllegalStateException {
  delegate.resetBuffer();
}

public boolean isCommitted() {
  return delegate.isCommitted();
}

public void flushBuffer() throws IOException {
  delegate.flushBuffer();
}

public void setLocale(Locale loc) {
  delegate.setLocale(loc);
  locale = loc;
}

public Locale getLocale(){
  return delegate.getLocale();
}

public void addCookie(Cookie cookie) {
  delegate.addCookie(cookie);
  cookies.addElement(cookie);
}

public boolean containsHeader(String name) {
  return delegate.containsHeader(name);
}

/**@deprecated*/
public void setStatus(int sc, String sm) {
  delegate.setStatus(sc, sm);
  status = sc;
}

public void setStatus(int sc) {
  delegate.setStatus(sc);
  status = sc;
}

public void setHeader(String name, String value) {
  delegate.setHeader(name, value);
  internalSetHeader(name, value);
}

public void setIntHeader(String name, int value) {
  delegate.setIntHeader(name, value);
  internalSetHeader(name, new Integer(value));
}
```

```
  public void setDateHeader(String name, long date) {
    delegate.setDateHeader(name, date);
    internalSetHeader(name, new Long(date));
  }

  public void sendError(int sc, String msg) throws IOException {
    delegate.sendError(sc, msg);
    didError = true;
  }

  public void sendError (int sc) throws IOException {
    delegate.sendError (sc);
    didError = true;
  }

  public void sendRedirect(String location) throws IOException {
    delegate.sendRedirect(location);
    didRedirect = true;
  }

  public String encodeURL(String url) {
    return delegate.encodeURL(url);
  }

  public String encodeRedirectURL(String url) {
    return delegate.encodeRedirectURL(url);
  }

  public void addHeader(String name, String value) {
    internalAddHeader(name, value);
  }

  public void addIntHeader(String name, int value) {
    internalAddHeader(name, new Integer(value));
  }

  public void addDateHeader(String name, long value) {
    internalAddHeader(name, new Long(value));
  }

  /**@deprected*/
  public String encodeUrl(String url) {
    return this.encodeURL(url);
  }

  /**@deprected*/
  public String encodeRedirectUrl(String url) {
    return this.encodeRedirectURL(url);
  }
}

class CacheServletOutputStream extends ServletOutputStream {

  ServletOutputStream delegate;
  ByteArrayOutputStream cache;

  CacheServletOutputStream(ServletOutputStream out) {
    delegate = out;
    cache = new ByteArrayOutputStream(4096);
  }
```

```
  public ByteArrayOutputStream getBuffer() {
    return cache;
  }

  public void write(int b) throws IOException {
    delegate.write(b);
    cache.write(b);
  }

  public void write(byte b[]) throws IOException {
    delegate.write(b);
    cache.write(b);
  }

  public void write(byte buf[], int offset, int len) throws IOException {
    delegate.write(buf, offset, len);
    cache.write(buf, offset, len);
  }
}
```