Krzysztof **Rychlicki-Kicior**

Helion

Java EE 6

Programowanie **aplikacji WWW**

- Szybko i bez kłopotów poznaj Java Enterprise Edition
- Naucz się praktycznie tworzyć ciekawe aplikacje WWW
- Dołącz do elity programistów nowoczesnych rozwiązań webowych

Już dziś sięgnij po jedyne kompendium wiedzy na temat Java EE!

Java EE 6. Programowanie aplikacji WWW

Autor: Krzysztof Rychlicki-Kicior
ISBN: 978-83-246-2659-5
Format: 158×235, stron: 232

- Szybko i bez kłopotów poznaj Java Enterprise Edition
- Naucz się praktycznie tworzyć ciekawe aplikacje WWW
- Dołącz do elity programistów nowoczesnych rozwiązań webowych

Już dziś sięgnij po jedyne kompendium wiedzy na temat Java EE!

Java Enterprise Edition to standard tworzenia aplikacji biznesowych wykorzystujących język Java. Opracowany przez firmę Sun Microsystems, działa w oparciu o wielowarstwową architekturę komponentową, oferując programistom bardzo rozbudowane możliwości tworzenia oprogramowania funkcjonującego na niemal dowolnym sprzęcie, w każdym systemie operacyjnym, z wykorzystaniem licznych serwerów aplikacji. Duża popularność rozwiązań Java EE i coraz powszechniejszy dostęp do technologii WWW sprawiają, że programiści sprawnie posługujący się tego rodzaju narzędziami rzadko figurują na listach osób poszukujących pracy, a jeśli już jakimś cudem się na nich znajdą, bardzo szybko otrzymują atrakcyjne propozycje zatrudnienia. Nauka swobodnego poruszania się w tym środowisku może też być wspaniałą, poszerzającą horyzonty przygodą, a gdy poznasz platformę Java EE, będziesz dysponował potężnym narzędziem, ułatwiającym tworzenie nawet najbardziej skomplikowanych aplikacji internetowych w bardzo efektywny i szybki sposób.

Studenci, programiści i hobbyści pragnący poznać środowisko Java Enterprise Edition często napotykają problem ze znalezieniem solidnych źródeł wiedzy, które pozwoliłyby im szybko i łatwo wejść w świat tej coraz bardziej popularnej technologii. Lukę tę z powodzeniem wypełnia książka „Java EE 6. Programowanie aplikacji WWW".

Dzięki niej wszyscy zainteresowani tematem zyskają możliwość poznania Java EE od podstaw i zdobycia praktycznej wiedzy, na podstawie której będą mogli rozwijać swoje umiejętności programistyczne w przyszłości. Ten podręcznik pozwala na szybkie rozpoczęcie przygody z tworzeniem aplikacji webowych, skutecznie wprowadzając w zagadnienia wykorzystywanych przy tym platform i mechanizmów, lecz nie pomijając też informacji o charakterze ogólnym. Jeśli niewiele mówią Ci skróty JSP, JPA, JSF czy JPQL, a chciałbyś zmienić ten stan rzeczy, bez wątpienia powinieneś sięgnąć po tę książkę, podobnie jak wszystkie osoby zainteresowane bezproblemowym używaniem całego spektrum nowoczesnych narzędzi oferowanych przez środowisko Java EE.

- Tworzenie serwletów
- Zastosowanie szablonów JSP
- Integracja danych z aplikacjami za pomocą mechanizmu JPA
- Używanie interfejsów i komponentów
- Korzystanie z technologii JSF
- Uniwersalny i wygodny dostęp do danych, czyli język JPQL
- Praktyczne przykłady realizacji

Spraw, aby tworzenie aplikacji WWW z wykorzystaniem Java EE nie miało przed Tobą tajemnic

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Serwlet — na dobry początek

Aplikacja z poprzedniego rozdziału wprowadziła kilka istotnych elementów, których omawianiem zajmiemy się w przeciągu najbliższych trzech rozdziałów. Rozpoczniemy od podstawy podstaw, czyli elementu, który jest wykorzystywany pośrednio lub bezpośrednio we wszystkich aplikacjach webowych — mowa o serwlecie.

Serwlet, czyli klasa rozszerzająca możliwości serwera aplikacji, może być traktowany jako pojęcie niesłychanie ogólne. Praktycznie jedynym istotnym wymaganiem stawianym serwletom jest działanie w trybie żądanie — odpowiedź — serwlet powinien generować treść odpowiedzi w oparciu o to, co otrzyma w żądaniu. W poprzednim rozdziale spotkałeś się z jednym z typowych zastosowań serwletów — generowaniem kodu HTML. Nie jest to jednak w żadnym razie kres ich możliwości — nic nie stoi na przeszkodzie, aby za pomocą serwletów generować zarówno dane tekstowe (np. w formacie XML), jak i dane binarne (np. pliki wykonywalne, obrazy, etc.). Zanim jednak zabierzemy się za praktyczne przykłady (pierwszy z nich mogłeś przeanalizować w poprzednim rozdziale), konieczne jest krótkie wprowadzenie teoretyczne, w którym dowiesz się, jak serwlet współpracuje z serwerem aplikacji, a także jakie podstawowe opcje związane z serwletami można ustawić w pliku *web.xml*.

Życie serwletu

Gdy uruchamiasz zwykłą aplikację, graficzną lub konsolową, w swoim systemie operacyjnym, możesz w większości przypadków określić precyzyjnie, kiedy rozpoczyna się, a kiedy kończy jej działanie. W przypadku popularnych technologii dynamicznych stron internetowych (np. PHP) pliki są interpretowane na bieżąco (aczkolwiek istnieje możliwość ich pośredniej kompilacji). Jak można opisać cykl życia serwletu?

Zacznijmy od klasy w takiej postaci, jaką już znamy — nieskompilowanego kodu źródłowego. Zanim serwer zostanie uruchomiony, wszystkie pliki klas muszą zostać poddane kompilacji. Powstałe pliki (o rozszerzeniu *.class*) są kopiowane do odpowiednich

katalogów. Dopiero wtedy serwer może być uruchomiony, na nowo lub ponownie. Na szczęście w nowszych wersjach serwerów aplikacji (np. Apache Tomcat 6) istnieje możliwość automatycznego wykrywania i aktualizacji klas w trakcie działania serwera.

Gdy uruchamiasz serwer aplikacji, z punktu widzenia naszego serwletu nie dzieje się nic istotnego. Następuje wtedy, rzecz jasna, inicjalizacja samego serwera, a także niektórych ustawień całej aplikacji webowej. Sam serwlet pozostaje jednak nienaruszony. Cała zabawa zaczyna się, gdy dowolny użytkownik Twojej aplikacji po raz pierwszy spróbuje z niego skorzystać. Serwer wykonuje wtedy następujące czynności:

- załadowanie klasy serwletu,
- utworzenie instancji serwletu,
- wywołanie metody `init()`,
- wywołanie metody `service()`.

Gdy serwlet znajdzie się w trakcie wywołania metody `service()`, może on rozpocząć normalną obsługę żądań. Od tego momentu w przypadku otrzymania przezeń dowolnego żądania HTTP, nastąpi próba wywołania odpowiedniej metody serwletu, według schematu `nazwa/doNazwa()`, np. `GET/doGet()`, `POST/doPost()`, itd.

Sporo pracy, nieprawdaż? Na szczęście do obowiązków programisty należy obsługa wybranych metod ze słowem `do` w nazwie. Jeśli więc chcesz, aby serwlet obsługiwał tylko żądanie `GET`, zadeklaruj jedynie metodę `doGet()`.

W przypadku klasy serwletu utworzonej przez Netbeans, proces tworzenia serwletu został uproszczony jeszcze bardziej. Twórcy szablonu założyli (skądinąd słusznie), że znamienita większość programistów korzysta jedynie z metod HTTP `GET` i `POST`. Z tego względu w klasie serwletu są przesłaniane dwie metody — `doGet()` i `doPost()`, które odwołują się do jednej i tej samej metody — o nazwie `processRequest()`. Z jednej strony ułatwia to życie w większości sytuacji, z drugiej jednak mogą się zdarzyć sytuacje, w których inaczej chcemy zareagować w przypadku żądania `GET`, a inaczej w przypadku `POST`. W takiej sytuacji należy usunąć wygenerowany mechanizm i napisać własne metody obsługi `doGet()` i/lub `doPost()`.

Serwlet pod lupą

Przed chwilą poznałeś przepływ sterowania w serwlecie; najwyższa pora, abyś zapoznał się pokrótce z kluczowymi klasami powiązanymi z obsługą serwletów. Omówię jedynie najważniejsze elementy; warto je zapamiętać, ponieważ będą się one pojawiać także w dalszych przykładach, ilustrujących kolejne omawiane technologie.

Jak już wspomniałem, serwlety, którymi zajmujemy się w niniejszej książce, dziedziczą po klasie `HttpServlet`. Ze względu na fakt, że serwlet z założenia jest konstrukcją niezwykle uniwersalną, w hierarchii dziedziczenia pojawiają się dodatkowe elementy, które ową uniwersalność wprowadzają. Oto krótki opis elementów hierarchii dziedziczenia, począwszy od tych najbardziej ogólnych:

- Interfejs `Servlet` — określa najważniejsze metody, które muszą implementować wszystkie serwlety. Metody te są niezależne od stosowanych protokołów przesyłania danych, a dotyczą one głównie zarządzania cyklem życia serwletu (`init()`, `service()`, `destroy()`).
- Abstrakcyjna klasa `GenericServlet` — podstawowa implementacja interfejsów `Servlet` i `ServletConfig`, dająca dostęp do parametrów i ustawień serwletu. Klasa ta zawiera proste implementacje metod obu interfejsów, dzięki czemu stanowi podstawę dla klasy `HttpServlet` i wszystkich innych klas serwletów.
- Klasa `HttpServlet` — to właśnie po tej klasie będziesz dziedziczył, tworząc własne serwlety. Poza własnymi implementacjami metod ze wspomnianych wcześniej interfejsów, klasa `HttpServlet` udostępnia metody `do*`, czyli `doGet()`, `doPost()` etc. Dzięki temu we własnych serwletach musisz zdefiniować jedynie te metody, które Twój serwlet zamierza obsługiwać.

Protokół HTTP zawiera definicje ośmiu metod: `GET`, `POST`, `PUT`, `HEAD`, `OPTIONS`, `TRACE`, `DELETE`, `CONNECT`. Serwlety mogą obsługiwać wszystkie metody, na wyżej omówionej zasadzie. W praktyce zdecydowanie najczęściej stosuje się metody `GET` i `POST` i to na nich skupimy się w dalszej części tego rozdziału.

Żądanie — odpowiedź

Mimo niewątpliwie istotnej roli klasy `HttpServlet`, w trakcie pracy z serwletami częściej przyjdzie Ci zapewne korzystać z interfejsów `HttpServletRequest/HttpServlet`
`↪Response`. Reprezentują one odpowiednio obiekty żądania i odpowiedzi, przekazywane do metod `doGet()`, `doPost()` etc. Pełny nagłówek metody `doGet()` wygląda następująco:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, java.io.IOException
```

Twoim zadaniem jest takie zdefiniowanie metod `do*`, aby generowały one odpowiedź, zazwyczaj zależną od przesłanych parametrów. Wszystkie niezbędne metody znajdziesz w dwóch wyżej wspomnianych klasach. Zacznijmy od interfejsu `HttpServletRequest` — to na jego podstawie będziemy w kolejnych przykładach generować odpowiedzi przesyłane za pomocą interfejsu `HttpServletResponse`.

Niemal wszystkie metody `HttpServletRequest` są faktycznie przydatne, niemniej w tym miejscu omówimy metody najistotniejsze z punktu widzenia samych serwletów:

- `Object getParameter(String nazwa)` — pobiera parametr o danej nazwie przesłany w żądaniu.
- `Enumeration<String> getParameterNames()` — pobiera nazwy wszystkich parametrów znajdujących się w danym żądaniu.
- `String getRemoteUser()` — zwraca login uwierzytelnionego użytkownika lub `null`, w przypadku braku uwierzytelnienia.

- `Cookie[] getCookies()` — zwraca tablicę ciasteczek — specjalnych plików przechowywanych na komputerze użytkownika.
- `String getQueryString()` — zwraca łańcuch parametrów, przesłanych w adresie URL (za znakiem zapytania).
- `String getHeader(String nazwa)` — zwraca wartość nagłówka HTTP o podanej nazwie.
- `int getIntHeader(String nazwa)` — zwraca wartość nagłówka HTTP o podanej nazwie jako liczbę całkowitą.
- `long getDateHeader(String nazwa)` — zwraca wartość nagłówka HTTP o podanej nazwie jako liczbę milisekund, począwszy od początku epoki (1 stycznia 1970 roku). Wartość ta może być przekazana w konstruktorze klasy `Date`.
- `String getContextPath()` — zwraca ścieżkę kontekstu aplikacji.
- `String getServletPath()` — zwraca ścieżkę dostępu do serwletu.
- `String getPathInfo()` — zwraca dodatkowe informacje zawarte w ścieżce.

Trzy ostatnie metody są ze sobą związane, ponieważ zwracają one kolejne elementy adresu URL, wykorzystanego do wykonania żądania. Przeanalizujmy poniższy przykład, prezentujący wiadomość o określonym identyfikatorze:

```
http://localhost:8080/MojaAplikacja/serwlety/info/235/
```

Pomijamy oczywiście nazwę protokołu (*http*) i nazwę serwera z portem (*localhost:8080*). Zostaje nam więc ciąg:

```
/MojaAplikacja/serwlety/info/235/
Metoda getContextPath() zwraca fragment adresu określający naszą aplikację:
/MojaAplikacja
```

Ścieżka do kontekstu zawsze zaczyna się od ukośnika (ale nigdy na nim się nie kończy!), chyba że aplikacja zostanie umieszczona w katalogu głównym serwera — wtedy zwracana wartość to łańcuch pusty. Fragment ten jest wspólny dla wszystkich plików wchodzących w skład tej aplikacji. Kolejny fragment adresu określa ścieżkę do serwletu. W naszym przypadku jest to fragment:

```
/serwlety/info
```

Powyższy łańcuch znaków musi pasować do odpowiednich wzorców, zdefiniowanych w deskryptorze wdrożenia (pamiętasz znacznik `<url-pattern>` z poprzedniego rozdziału?). Zasady określania odpowiednich ścieżek do serwletów omówimy w następnym rozdziale; na razie wystarczy Ci informacja, że ten fragment adresu umożliwia jednoznaczne zidentyfikowanie serwletu.

Ostatni fragment ścieżki (`/235/`) zostanie zwrócony przez metodę `getPathInfo()`. Dokładnie rzecz biorąc, metoda `getPathInfo()` zwraca fragment adresu URL od ścieżki serwletu do początku łańcucha parametrów (czyli do znaku zapytania). Oznacza to, że nawet dołączenie parametrów, tak jak w poniższym przykładzie, nie zmieni wartości ścieżki.

```
http://localhost:8080/MojaAplikacja/serwlety/info/235?param=1
```

Przesyłanie odpowiedzi

Po przeanalizowaniu wszystkich możliwych atrybutów żądania musisz odesłać klientowi odpowiedź. Do tego celu służy obiekt interfejsu `HttpServletResponse`. W jego przypadku otrzymujemy nieco mniejszy zestaw metod, jednak nie oznacza to wcale mniejszych możliwości. Przede wszystkim musimy określić, jakie operacje chcemy wykonywać w związku z przesyłaniem odpowiedzi do klienta:

- przesłanie odpowiedzi w postaci danych tekstowych lub binarnych,
- utworzenie i przesłanie ciasteczek,
- dodanie do odpowiedzi dowolnych nagłówków,
- przekierowanie żądania lub przesłanie kodu błędu.

Transmisja danych

Chociaż technologie internetowe mają swoją specyfikę, nie zapominajmy, że żyjemy w świecie Javy. Z tego względu operacje zarówno odczytu, jak i zapisu wiążą się z wykorzystaniem strumieni i/lub obiektów klas `Reader/Writer`. Nie inaczej jest w tym przypadku: zanim prześlemy jakiekolwiek dane, musimy uzyskać odpowiednie obiekty zapisujące:

- `ServletOutputStream getOutputStream()` — zwraca strumień zapisu dla danych binarnych
- `PrintWriter getWriter()` — zwraca obiekt zapisujący dla danych tekstowych.

W przypadku danych binarnych możemy skorzystać z obiektu klasy `ServletOutput↪Stream`. Jest to zwykły strumień zapisu, rozszerzony o możliwość zapisywania dowolnych danych typów prymitywnych, a także łańcuchów znaków (za pomocą metod `print()` i `println()`). Z tej klasy należy korzystać w przypadku przesyłania plików — jeśli serwer musi w dynamiczny sposób wygenerować treść takiego pliku.

Znacznie częściej przyjdzie Ci jednak korzystać z danych tekstowych. W tym przypadku zazwyczaj będziesz korzystać z obiektu klasy `PrintWriter` i jego metody `println()`.

Nagłówki i ciasteczka

O ile w przypadku żądania mamy do czynienia z odczytem nagłówków i ciasteczek przesłanych przez klienta, o tyle w przypadku odpowiedzi występuje proces odwrotny. Aby dodać ciasteczko, wystarczy skorzystać z metody `addCookie()`:

```
void addCookie(Cookie c)
```

Więcej na temat ciasteczek w osobnym podrozdziale. W przypadku nagłówków sytuacja jest nieco bardziej skomplikowana — do dyspozycji mamy dwie metody (wraz z odpowiednikami dla liczb i dat):

```
void addHeader(String nazwa, String wartość)
void setHeader(String nazwa, String wartość)
```

Na czym polega różnica? Otóż metoda `addHeader()` doda podaną wartość do już istniejącej zawartości nagłówka, natomiast metoda `setHeader()` zastąpi wartość, jeśli takowa już istnieje. Tak samo działają bliźniacze metody `addIntHeader()`, `addDateHeader()`, `setIntHeader()` i `setDateHeader()`.

Kody odpowiedzi, błędy i przekierowania

Do obowiązków odpowiedzi HTTP należy także przekazywanie kodów odpowiedzi, jeśli chcemy zaznaczyć, że odpowiedź nie zostanie zakończona w zwykły sposób. Aby przekazać kod odpowiedzi, korzystamy z metody `setStatus()`:

```
void setStatus(int kod)
```

W ten sposób przekazujemy kody, które nie określają sytuacji problematycznych. W przypadku błędów (np. 404 — brak zasobu) zaleca się zdecydowanie wykorzystywanie metody `sendError()`:

```
void sendError(int kod)
void sendError(int kod, String komunikat)
```

Jak widać, istnieje możliwość przesłania dodatkowej informacji na temat samego błędu. Ostatnią funkcjonalność związaną z kodami odpowiedzi stanowi przekierowanie. Chociaż z technicznego punktu widzenia przekierowanie jest też rodzajem kodu odpowiedzi, do przekierowania wykorzystuje się oddzielną metodę:

```
void sendRedirect(String adres)
```

Korzystając z metod `sendError()` i `sendRedirect()`, należy pamiętać o subtelnych kwestiach związanych z fizycznym przesyłaniem danych do klienta. Przesyłanie komunikatów o błędach lub przekierowań wiąże się z dość brutalną ingerencją w proces przesyłania odpowiedzi. Proces ten jest natychmiast przerywany, a klient otrzymuje odpowiedź z wybranym kodem odpowiedzi. Co jednak stanie się, gdy zdążymy wysłać do klienta jakieś dane?

Odpowiedź jest prosta — nastąpi błąd. Po wysłaniu danych nie możesz ingerować w treść nagłówków, przez co nie możesz ustawić kodu odpowiedzi, a co za tym idzie także przekierowania. Czy oznacza to, że musisz uważać, gdzie wywołujesz metodę `println()` obiektu `PrintWriter`? Na szczęście nie do końca.

Domyślnym zachowaniem obiektu w przypadku odpowiedzi jest zapisywanie danych do bufora. Oznacza to, że dane zostaną wysłane po zakończeniu metody lub w przypadku wywołania metody `flush()` tego obiektu. Co za tym idzie, poniższa konstrukcja (wewnątrz metody `doGet()`) nie spowoduje wygenerowania błędu:

```
PrintWriter out = response.getWriter();
out.println("test");
response.sendRedirect("url/do/innego/serwletu");
```

Jeśli przed wywołaniem metody `sendRedirect()` wywołasz metodę `out.flush()`, wtedy błąd nastąpi. Zazwyczaj jednak takie wywołanie jest pomijane, dzięki czemu problem występuje stosunkowo rzadko.

Om nom nom, czyli ciasteczka w pełnej krasie

Twórcy aplikacji webowych, podobnie jak Ciasteczkowy Potwór, mają szczególny sentyment do ciasteczek (ang. *cookies*). Są to niewielkie pliki przechowywane na komputerach użytkowników aplikacji webowych, dzięki czemu jesteśmy w stanie zapamiętywać ich preferencje, loginy i hasła itd. Metody operujące na ciasteczkach poznaliśmy w poprzednim podrozdziale, ale teraz zaprezentujemy ich działanie w praktycznym przykładzie (listing 3.1):

Listing 3.1. *Przykład obsługi ciasteczek*

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
throws ServletException, IOException {
 response.setContentType("text/html;charset=UTF-8");
 PrintWriter out = response.getWriter();
 try {
  Cookie lastVisit = null;
  for (Cookie c : request.getCookies())
   if (c.getName().equals("obecnosc")) {
    lastVisit = c;
    break;
   }
   if (lastVisit != null)
    out.println("Twoja ostatnia wizyta na stronie miała miejsce w dniu " +
        lastVisit.getValue());
   else
    out.println("Do tej pory nie odwiedziłeś/aś naszej strony. Wstydź się!");
   lastVisit = new Cookie("obecnosc", new Date().toString());
   response.addCookie(lastVisit);
  } finally {
   out.close();
  }
}
```

Zadaniem powyższego serwletu jest przechowywanie informacji o dacie ostatniej wizyty na stronie i wyświetlanie jej. W przypadku braku ciasteczka z datą (co jest równoznaczne z pierwszymi odwiedzinami na tej stronie, przynajmniej od czasu wyczyszczenia ciasteczek w przeglądarce) wyświetlamy inną informację. Warto zwrócić uwagę na dwie kwestie. Po pierwsze, jeśli chcemy odczytać już istniejące ciasteczka — korzystamy z metody `getCookies()` znajdującej się w obiekcie `request`. Jeśli chcemy dodać ciasteczko — korzystamy z obiektu `response`. Nigdy odwrotnie! Sprawa druga, znacznie bardziej przykra — powyższy sposób dostępu do ciasteczek użytkownika (pętla `for..in`) stanowi jedyną metodę znajdywania ciasteczek o określonej nazwie. W przypadku tworzenia prawdziwych aplikacji trzeba zdefiniować osobną metodę do wyszukiwania ciasteczek.

Sesje — nie tylko dla studentów

Obsługa sesji jest kolejnym nieodłącznym elementem niemal wszystkich aplikacji webowych. W przypadku JEE interakcje z sesją możemy prowadzić na różne sposoby,

także za pomocą poznanych już klas. W tym rozdziale poznamy sposób na dostęp do sesji za pomocą obiektu klasy `HttpServletRequest`. Kluczową rolę odgrywa metoda `getSession()`, występująca w dwóch wariantach:

```
HttpSession getSession()
HttpSession getSession(boolean czyTworzyc)
```

Na wstępie zaznaczę, że pierwszy wariant tej metody jest równoważny drugiemu wywołanemu z parametrem `true`. Drugi wariant postępuje różnie w zależności od przekazanej wartości logicznej:

- Jeśli parametr ma wartość `true`, metoda zwraca obiekt sesji lub tworzy nowy, jeśli ten nie istnieje.
- Jeśli parametr ma wartość `false`, metoda zwraca obiekt sesji lub `null`, jeśli ten nie istnieje.

Jak widać, wartość `true` należy przekazać, jeśli chcesz po prostu uzyskać dostęp do sesji. Wartość `false` stosuje się, gdy chcesz sprawdzić, czy sesja istnieje. Można skorzystać z tego mechanizmu, aby sprawdzić, czy dane żądanie jest pierwszym żądaniem użytkownika w danej sesji. Mechanizm ten jest realizowany w poniższym przykładzie z listingu 3.2:

Listing 3.2. *Przykład wykorzystania sesji*

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            if (request.getSession(false)==null)
            {
                out.println("Witaj na stronie po raz pierwszy!");
                request.getSession();
            }
            else
                out.println("Witaj na stronie po raz kolejny!");
        } finally {
            out.close();
        }
    }
```

Jeśli po utworzeniu sesji chcesz sprawdzić, czy sesja została dopiero co utworzona, skorzystaj z metody `isNew()`:

- `boolean isNew()` — zwraca `true`, jeśli obiekt sesji został utworzony podczas tego żądania.

Korzystanie z obiektu sesji

Podstawowa funkcjonalność obiektu sesji sprowadza się do dwóch metod:

- `Object getAttribute(String` *`nazwa`*`)` — zwraca atrybut sesji o podanej nazwie.

- `void setAttribute(String nazwa, Object wartość)` — dodaje obiekt do sesji, przypisując mu podany klucz (nazwę). Jeśli jakiś obiekt o takiej samej nazwie już istniał, zostanie on zastąpiony.

Wiemy już, jak utworzyć sesję, wiemy też, jak z niej skorzystać. Pozostało nam omówienie, jakie są warunki zakończenia sesji. Może ono nastąpić w wyniku kilku różnych sytuacji:

- ręczne zakończenie sesji przez programistę,
- upłynięcie czasu życia sesji,
- zamknięcie okna przeglądarki przez użytkownika.

Ostatni przypadek, jest rzecz jasna, najprostszy — nie wymaga on naszej ingerencji. Ręczne zakończenie sesji wiąże się z wywołaniem następującej metody:

- `void invalidate()` — kończy sesję.

Najciekawiej sytuacja wygląda w przypadku określania terminu ważności sesji. Istnieją bowiem dwie możliwości określenia tej wartości — pierwsza z nich jest stosowana w pliku konfiguracyjnym *web.xml*:

```
<web-app>
  <session-config>
    <session-timeout>10</session-timeout>
  </session-config>
</web-app>
```

Podana wartość określa czas ważności sesji w minutach. Obowiązuje on dla wszystkich sesji, chyba że skorzystasz z możliwości określenia czasu życia sesji w kodzie:

- `void setMaxInactiveInterval(int czas)` — określa czas życia sesji w sekundach. Podanie wartości 0 i ujemnych powoduje, że sesja nigdy nie wygasa (do jej zakończenia jest więc konieczne wywołanie metody `invalidate()` lub zamknięcie okna przeglądarki przez użytkownika).

Konfiguracja w kodzie Javy — można tego uniknąć

Podczas tworzenia większości aplikacji programiści muszą zmierzyć się z problemem obsługi różnego rodzaju ustawień wpływających na działanie aplikacji. Problemem staje się lokalizacja tych ustawień. Z jednej strony nikt nie chce utrudniać sobie życia — w końcu nie ma nic prostszego, niż wczytać wartość umieszczoną w stałej/zmiennej. Z drugiej jednak strony zmiana takich ustawień wymagałaby rekompilacji całego projektu, w najlepszym przypadku — jednej biblioteki.

Z tego względu powszechnym standardem stało się umieszczanie różnego rodzaju ustawień w zewnętrznych źródłach danych — plikach binarnych, tekstowych, XML;

rzadziej w bazach danych. W przypadku aplikacji webowych JEE miejscem takim jest deskryptor wdrożenia — plik *web.xml*. Poza licznymi ustawieniami związanymi z funkcjonowaniem aplikacji jako takiej (część z nich już poznałeś), w pliku *web.xml* możesz także zdefiniować parametry dla poszczególnych serwletów, a także całej aplikacji webowej.

Parametry serwletów

Parametry serwletów możesz określać za pomocą znacznika `<init-param>` w następujący sposób:

```
<servlet>
        <servlet-name>ParameterServlet</servlet-name>
        <servlet-class>pl.helion.jeeweb.ParameterServlet</servlet-class>
        <init-param>
            <param-name>autor</param-name>
            <param-value>Krzysztof Rychlicki-Kicior</param-value>
        </init-param>
</servlet>
```

Po dwóch znanych już znacznikach (`servlet-name` i `servlet-class`) następuje dowolna liczba znaczników `init-param`. Każdy taki znacznik zawiera dwa kolejne, określające nazwę i wartość parametru. Parametry serwletów można też dodawać w środowisku Netbeans, podczas tworzenia serwletu (w ostatnim kroku kreatora).

Pierwszy parametr utworzony, najwyższa pora, aby odczytać go we wnętrzu serwletu. Do zarządzania parametrami serwletów służy interfejs `ServletConfig`, który jest implementowany przez znane nam klasy `GenericServlet` i `HttpServlet`. Dwie metody tego interfejsu, które interesują nas w tej chwili najbardziej, to:

- `String getInitParameter(String nazwa)` — zwraca wartość parametru o podanej nazwie.
- `String[] getInitParameterNames()` — zwraca wszystkie nazwy parametrów danego serwletu.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("Autorem serwletu jest " + this.getInitParameter("autor"));
        } finally {
            out.close();
        }
    }
```

Dzięki umieszczeniu konfiguracji w pliku XML odnieśliśmy wymierną korzyść. Zmiana wartości w pliku XML nie wymaga rekompilacji kodów źródłowych, a jedynie przeładowania aplikacji (w przypadku Tomcata istnieje także opcja automatycznego wykrywania zmian i przeładowywania aplikacji).

Interfejs `ServletConfig` poza dwoma poznanymi metodami udostępnia metodę `get ↪ServletName()`, zwracającą nazwę serwletu, a także metodę `getServletContext()`. Zwraca ona (a jakżeby inaczej) kontekst serwletów — jeden z najważniejszych obiektów w całym świecie aplikacji webowych JEE.

Kontekst serwletów

Kontekst serwletów to obiekt, który służy do komunikacji serwletów z kontenerem. Dzięki niemu możesz dynamicznie dodawać serwlety do aplikacji, uzyskiwać dostęp do zasobów znajdujących się w jej obrębie, zapisywać logi do serwerowego dziennika, a co najważniejsze z obecnego punktu widzenia — możesz korzystać z parametrów aplikacji webowej (kontekstu). Od parametrów serwletów różni je zasięg oddziaływania. Każdy parametr kontekstu jest widoczny we wszystkich serwletach i innych plikach. Parametry serwletu są określane w podobny sposób jak w przypadku serwletów:

```
<web-app>
    <context-param>
        <param-name>tytul </param-name>
        <param-value>Java EE 6. Tworzenie aplikacji webowych</param-value>
    </context-param>
    …
</web-app>
```

Również sposób wykorzystywania parametrów kontekstu przypomina ten znany z serwletów:

```
try {
  out.println("Wszystkie przykłady pochodzą z książki " +
      this.getServletContext().getInitParameter("tytul"));
} finally {
   out.close();
}
```

Jedyną różnicę stanowi odwołanie się do obiektu kontekstu. Reszta pozostaje bez zmian — nawet nazwa metody. Ciekawostkę stanowi metoda wprowadzona w specyfikacji Java Servlets 3.0. Otóż aż do momentu wprowadzenia tej specyfikacji parametry, zarówno serwletów, jak i kontekstu, były wartościami tylko do odczytu. Jedyną możliwością zmiany parametrów była edycja pliku *web.xml*. W wersji JavaServlet 3.0 API pojawiła się jednak innowacja — możliwość dynamicznego ustawiania parametrów kontekstu za pomocą metody `setInitParameter()`. Wynika to z wprowadzenia dużej elastyczności — klasa `ServletContext` w wersji 3.0 uzyskała wiele metod, takich jak `addServlet()`, czy `addFilter()`, które umożliwiają dynamiczne dodawanie różnych składników aplikacji, do tej pory deklarowanych jedynie w pliku *web.xml*. Nie należy jednak nadużywać tej metody.

Kontekst serwletów pojawi się ponownie już niebawem, tymczasem nadszedł czas, aby zmierzyć się z przeciwnikiem o wiele ważniejszym od parametrów — mowa o atrybutach.

Trzech muszkieterów?

Parametry, czy to serwletów, czy to aplikacji, mają swoje zastosowania i bywają niezwykle przydatne. Mimo to głównym środkiem komunikacji między serwletami, kontenerem, sesją, użytkownikiem i obiektem żądania — czyli z grubsza między wszystkimi elementami aplikacji — są atrybuty. Z technicznego punktu widzenia między parametrami i atrybutami występują dwie zasadnicze różnice:

- W przypadku parametrów zarówno klucz, jak i wartość są łańcuchami znaków, zaś w przypadku atrybutów — klucz jest łańcuchem, wartość może być obiektem.
- Parametry z założenia są tylko do odczytu (choć w świetle ostatniej wersji specyfikacji wygląda to inaczej…), natomiast atrybuty są przeznaczone zarówno do odczytu, jak i do zapisu.

Niezwykłe znaczenie ma także wprowadzenie zasięgu atrybutów. Atrybut dodany w zasięgu żądania (*request*) nie będzie widoczny w innych zasięgach. Tabela 3.1 przedstawia zestawienie parametrów i atrybutów w poszczególnych zakresach.

Tabela 3.1. *Możliwości zapisywania i odczytywania parametrów i atrybutów w poszczególnych zakresach*

Zakres	Parametry		Atrybuty	
	Zapis	Odczyt	Zapis	Odczyt
Żądanie	nie	tak	tak	tak
Serwlet	nie	tak	brak	brak
Sesja	brak	brak	tak	tak
Kontekst aplikacji	tak (od wersji 3.0)	tak	tak	tak

Na podstawie powyższej tabeli wydać wyraźnie, że parametry pełnią jedynie funkcję ustawień, opcji konfiguracyjnych, które ułatwiają zmianę w działaniu aplikacji bez konieczności ponownej rekompilacji kodu. Atrybuty natomiast mają zastosowanie o wiele szersze — służą do wymiany informacji pomiędzy poszczególnymi elementami aplikacji.

W dalszej części rozdziału skupimy się tylko na atrybutach. Ich ogromna przydatność ma bowiem pewne ograniczenia. Jedno z nich jest związane z najważniejszą chyba cechą odróżniającą aplikacje webowe od aplikacji typu standalone — konieczność jednoczesnej obsługi wielu użytkowników.

Atrybuty a mnogość żądań

Jedna aplikacja webowa może być używana nawet przez setki czy tysiące użytkowników jednocześnie. Każde żądanie (*HTTP request*) jest obsługiwane przez kontener w osobnym wątku. Istotną kwestią jest więc zapewnienie integralności operacji wyko-

nywanych przez każdego z nich — nie może być tak, że operacje jednego użytkownika wpłyną na efekt operacji innego.

W przypadku parametrów problem ten raczej nie występuje. Co prawda, w wersji 3.0 pojawiła się możliwość modyfikowania parametrów kontekstu aplikacji, jednak możliwość ta powinna być używana w bardzo sporadycznych sytuacjach, gdy obsługa wielu użytkowników nie powinna sprawiać problemów (np. z powodu wywoływania takiego kodu przez superadministratora witryny). Jeśli jednak zabezpieczenie jest konieczne, można zrealizować je w sposób analogiczny do tego, który zaprezentuję za chwilę.

Zdecydowanie bardziej skomplikowana sytuacja występuje w przypadku atrybutów. Wszystkie trzy przypadki omówię w kolejnych podrozdziałach.

Atrybuty żądania

W przypadku atrybutów żądania sytuacja jest stosunkowo prosta. Żądanie jest realizowane przez jednego użytkownika; w dodatku pojedyncze żądanie nie wiąże się w żaden sposób z innymi żądaniami (nawet tego samego użytkownika), dlatego problem jednoczesnego dostępu przez wielu użytkowników nie występuje. Pojawia się jednak inne pytanie — skoro obiekt żądania nie wchodzi w interakcje z innymi żądaniami, po co miałby korzystać z atrybutów?

Takie rozwiązanie wynika ze stosowanych w praktyce mechanizmów obsługi stron. Serwlety same w sobie rzadko generują treść — na ogół wykonują one różnorodne operacje (np. pobranie danych z bazy, realizacja logiki biznesowej — choć w większych aplikacjach i te zadania są delegowane), a następnie przekazują sterowanie do pliku JSP. W takiej sytuacji konieczne jest przekazanie informacji między serwletem a plikiem JSP. Voilà! — znaleźliśmy zastosowanie atrybutów żądania. Dokładne wyjaśnienie i przykłady poznasz w rozdziale poświęconym JSP.

Atrybuty sesji

W nieco gorszej sytuacji są atrybuty sesji. Wiemy już, że jedna sesja jest powiązana z konkretnym użytkownikiem. Teoretycznie nie powinno więc być problemów. Ale użytkownicy bywają okrutni — wyobraź sobie, co mogłoby się stać, gdyby użytkownik uruchomił Twoją aplikację webową w dwóch zakładkach i próbował jednocześnie ładować różne (lub te same) serwlety?

Odpowiedź jest prosta: mogłoby dojść do jednoczesnego dostępu do sesji. Odczyt danych nie stanowiłby problemu, ale atrybuty sesyjne mogą być przecież również zapisywane. Taka sytuacja to potencjalny problem. Jak więc mu zaradzić?

Powiem krótko: należy skorzystać ze standardowego mechanizmu Javy, chroniącego dane przed zapisem przez wiele wątków jednocześnie — synchronizacji. Teraz musimy określić, dostęp do czego dokładnie chcemy synchronizować.

Na początek odrzućmy obiekt żądania (klasy `HttpServletRequest`). Jest to obiekt związany tylko z jednym, konkretnym żądaniem, więc zablokowanie dostępu do niego nie wpłynęłoby na inne obiekty żądań — nadal wszystkie one mogłyby korzystać bez

skrępowania z sesji. Nie ma sensu również blokada obiektu serwletu — dostęp do sesji mają różne serwlety, więc zablokowanie jednego z nich nie powstrzyma innych od zapisu do sesji. Jedynym sensownym rozwiązaniem pozostaje zablokowanie obiektu sesji, do którego uzyskujemy dostęp za pomocą obiektu żądania. Poniższy kod, wstawiony we wszystkich serwletach, pozwoli na zliczenie wszystkich wywołań serwletów, które miały miejsce w aplikacji webowej dla danego użytkownika:

```
HttpSession sesja = request.getSession();
synchronized(sesja) {
  if (sesja.isNew())
    sesja.setAttribute("licznik", 1);
  else
  {
     int licznik = Integer.parseInt(sesja.getAttribute("licznik").toString());
     sesja.setAttribute("licznik", licznik + 1);
  }
}
```

W ten sposób, gdy jeden serwlet wejdzie w blok synchronizowany, uzyskujemy gwarancję, że żaden inny serwlet w tym momencie dostępu do sesji nie uzyska. Wszystkie inne serwlety będą musiały czekać, aż pierwszy serwlet zwolni blokadę.

Atrybuty kontekstu serwletów

Największe niebezpieczeństwo niesie za sobą korzystanie z atrybutów należących do kontekstu aplikacji. Każdy taki atrybut może być odczytany i zmodyfikowany w dowolnym niemal miejscu aplikacji. Z tego względu każda próba korzystania z atrybutów (zwłaszcza zapisu) powinna być synchronizowana.

Zasada działania jest taka sama, jak w przypadku sesji. W tym przypadku musimy jednak synchronizować obiekt kontekstu. Kod synchronizujący przedstawia się następująco:

```
ServletContext sc = this.getServletContext();
synchronized(sc)
{
  Object licznik = sc.getAttribute("licznik");
  if (licznik == null)
    sc.setAttribute("licznik", 1);
  else
  {
    licznik = sc.getAttribute("licznik");
    sc.setAttribute("licznik", Integer.parseInt(licznik.toString()) + 1);
  }
}
```

Powyższy kod realizuje funkcjonalność podobną do przykładu z sesją — tym razem zliczamy jednak wszystkie wywołania serwletów wykonane przez wszystkich użytkowników.

Z obiektami żądań, sesji i kontekstu, jak również z ich atrybutami, wiążą się ważne klasy — słuchaczy zdarzeń. Choć istnieje możliwość tworzenia całych aplikacji webowych bez świadomości istnienia tych klas, zdarzają się sytuacje, w których znajomość tego typu mechanizmów jest niezbędna.

Słuchowisko

Słuchacze zdarzeń to obiekty spotykane w Javie niezwykle często. Początkujący programiści Javy spotykają się z nimi np. podczas tworzenia prostych aplikacji graficznych. Słuchacz zdarzeń powiązany z przyciskiem pozwalał na wykonanie dowolnego kodu np. po jego kliknięciu. Pojęcie słuchacza zdarzeń nie ogranicza się oczywiście do tworzenia aplikacji z graficznym interfejsem — również aplikacje webowe dają słuchaczom zdarzeń spore pole do popisu.

W poniższych podrozdziałach przedstawię interfejsy słuchaczy zdarzeń przeznaczone do użycia w aplikacjach webowych. Nie jest to może najciekawszy fragment niniejszej książki, ale prędzej czy później znajdziesz się w sytuacji, w której będziesz musiał skorzystać z opisanych w następnych akapitach mechanizmów.

ServletContextListener

Jest to najrzadziej chyba wykorzystywany słuchacz zdarzeń. Zawiera dwie metody: `contextInitialized()` i `contextDestroyed()`, które są wywoływane w momencie utworzenia/usunięcia kontekstu aplikacji, czyli — w momencie startu i zakończenia aplikacji. Obydwie metody przyjmują parametr typu `ServletContextEvent` — umożliwia on pobranie kontekstu aplikacji za pomocą metody `getServletContext()`.

ServletContextAttributeListener

Drugim słuchaczem zdarzeń związanym z kontekstem aplikacji jest słuchacz obserwujący kolekcję atrybutów kontekstu. Reaguje on na dodawanie (metoda `attribute➥Added()`), usuwanie (`attributeRemoved()`) i zamianę (`attributeReplaced()`) atrybutów. Wszystkie trzy metody przyjmują jeden parametr typu `ServletContextAttribute➥Event` — umożliwia on pobranie nazwy modyfikowanego atrybutu (`getName()`), jego wartości (`getValue()`).

ServletRequestAttributeListener i ServletRequestListener

Obydwa interfejsy pełnią analogiczne funkcje, co ich „kontekstowi” koledzy — nawet nazwy metod są podobne (w przypadku pierwszego interfejsu — identyczne, w przypadku drugiego — `requestInitialized()` i `requestDestroyed()`). Jedyną realną zmianą jest wprowadzenie dodatkowej funkcjonalności do klas argumentów zdarzeń — `ServletRequestAttributeEvent` i `ServletRequestEvent`. Udostępniają one metodę `getServletRequest()`. Pozwala ona na skorzystanie z obiektu żądania, którego zdarzenia dotyczą.

HttpSessionAtributteListener i HttpSessionListener

Również słuchacze zdarzeń powiązani z sesjami zostały utworzone zgodnie z omówionymi powyżej zasadami. Słuchacz `HttpSessionListener` jest wykorzystywany przy tworzeniu (metoda `sessionCreated()`) i kończeniu sesji (`sessionDestroyed()`). Przekazywany argument — obiekt klasy `HttpSessionEvent` — udostępnia metodę `getSession()`, która daje dostęp do utworzonej (zakończonej) sesji. W przypadku interfejsu `HttpSessionAttributeListener` mamy do dyspozycji te same trzy metody, co w poprzednich przypadkach. Typ zdarzenia to `HttpSessionBindingEvent`. Jego możliwości sprowadzają się do pobrania obiektu sesji i nazwy/wartości dodawanego/usuwanego/zmienianego atrybutu.

HttpSessionBindingListener

Nareszcie coś ciekawego! Tytułowy interfejs odbiega nieco od schematu, z jakim mieliśmy do czynienia przez ostatnie trzy podrozdziały. Wszystkie trzy interfejsy z członem `AttributeListener` w nazwie odpowiadały za informowanie o zmianach zachodzących w kolekcji atrybutów. Dla odmiany interfejs `HttpSessionBindingListener` powinien być implementowany przez klasy, których obiekty będą umieszczane w sesji! Jeśli więc tworzysz własne klasy do przechowywania danych, które w momencie zapisu do sesji powinny być w jakiś sposób przetworzone, powinieneś skorzystać z tego interfejsu. Metody tego interfejsu to `valueBound()` i `valueUnbound()`, wywoływane odpowiednio w momencie dołączenia obiektu do sesji lub jego usunięcia. Klasa argumentu zdarzeń to znana z poprzedniego akapitu `HttpSessionBindingEvent`.

Sesja + wiele JVM = HttpSessionActivationListener

Java EE to technologia uniwersalna, która może być z powodzeniem stosowana w rozbudowanych aplikacjach webowych i biznesowych. Jednym z ważnych aspektów tworzenia takich aplikacji jest skalowalność — możliwość poprawnego i wydajnego działania aplikacji w dowolnych warunkach (niewielkiego, jak i bardzo dużego obciążenia), bez konieczności zmiany samej aplikacji. Bardzo często, w celu poprawy wydajności, ta sama aplikacja jest instalowana na wielu komputerach połączonych w sieć, a specjalny serwer, będący bramą do świata zewnętrznego (czyli do klientów), kieruje „ruchem” i przydziela poszczególne żądania do tych komputerów, które w danym momencie są najmniej obciążone. Taka technika nosi nazwę równoważenia obciążenia (ang. *load balancing*).

Nie będziemy zagłębiać się teraz w szczegóły tej techniki; w niniejszej książce będzie nas interesować co najwyżej wpływ tej techniki na kod aplikacji lub informacje zawarte w deskryptorze wdrożenia. W tym podrozdziale zajmiemy się słuchaczem zdarzeń, który jest związany z dwoma istotnymi zagadnieniami: obsługą sesji i równoważeniem obciążenia.

Gdy użytkownik tworzy obiekt sesji, jest on zapisywany (np. w postaci pliku) na komputerze, który obsługiwał w danej chwili żądanie. Następne żądanie użytkownika może jednak być przydzielone do zupełnie innego komputera wewnątrz sieci — dlatego konieczne jest wprowadzenie mechanizmu, który przeniesie obiekt sesji z pierwszego komputera na drugi.

Na szczęście mechanizm ten jest automatyczny. Jedynym (opcjonalnym) zadaniem programisty jest wykonanie operacji przed przeniesieniem sesji z komputera pierwotnego i po przeniesieniu sesji na komputer docelowy. Służą do tego celu metody:

- `void sessionWillPassivate(HttpSessionEvent hse)` — metoda wywoływana tuż przed przesłaniem sesji do innego komputera;
- `void sessionDidActivate(HttpSessionEvent hse)` — metoda wywoływana tuż po otrzymaniu sesji od pierwszego komputera.

Filtry

Do tej pory wszystkie przykłady omawialiśmy, korzystając z modelu obsługi żądania/odpowiedzi. Żądanie HTTP przesłane przez klienta powodowało utworzenie nowego wątku, wykorzystującego obiekt odpowiedniego serwletu. Na podstawie metody żądania HTTP następował wybór odpowiedniej metody klasy `HttpServlet` (`doGet()`, `doPost()` itd.). Model ten można jednak rozszerzyć o dodatkowe elementy — filtry.

Filtr umożliwia wykonywanie operacji w momencie nadejścia żądań do serwletów i wygenerowania przezeń odpowiedzi, przy czym nie ingeruje on w działanie samego serwletu. Można zatem np. zapisać w dzienniku datę każdego żądania, jego parametry, a także sprawdzić, jaka jest długość odpowiedzi wygenerowanej przez serwlet.

W praktyce serwlety stosuje się również do kontroli, a ewentualnie także modyfikacji obiektów żądania i odpowiedzi. Dzięki filtrom możesz skompresować całą treść odpowiedzi, zanim zostanie przesłana ostatecznie do klienta. Możesz też w uniwersalny sposób odrzucać żądania, które nie spełniają określonych warunków (np. wartość ustalonych parametrów). Największą zaletą filtrów jest możliwość podłączenia ich do dowolnej grupy serwletów — filtry są łączone z serwletami za pomocą znacznika `url-pattern`, działającego analogicznie jak w przypadku serwletów. Jak widać, dodanie jednego znacznika pozwala na włączenie szyfrowania, kompresji lub kontroli dostępu dla całej grupy serwletów.

Techniczny aspekt filtrów

Z technicznego punktu widzenia filtr jest klasą implementującą interfejs `javax.servlet.Filter`. Interfejs ów składa się z trzech metod:

- `void init(FilterConfig fc)` — metoda wywoływana przy utworzeniu filtru. Pozwala na uzyskanie obiektu ustawień filtru — obiektu interfejsu `FilterConfig`.

- `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)` — metoda wywoływana w momencie nadejścia żądania. Dokładny opis w dalszej części podrozdziału.
- `void destroy()` — metoda wywoływana przez serwer w momencie zakończenia działania filtru.

Największe znaczenie ma, rzecz jasna, metoda `doFilter()`. Typowy sposób jej wykorzystania przedstawiam poniżej:

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) {
  if (!req.isSecure())
   ((HttpServletResponse)res).sendError(HttpServletResponse.SC_RESPONSE);
  chain.doFilter(req, res);
  bool spakowany = false;
  if (req.getParameter("format") != null &&
      req.getParameter("format").equals("spakowany"))
   spakowany = true;
  if (spakowany) {
 // pobierz treść odpowiedzi, spakuj ją, a następnie
 // zapisz, korzystając z obiektu res.
  }
}
```

Treść powyższej metody można podzielić na trzy części. Na początku sprawdzamy, czy dane żądanie jest realizowane w trybie bezpiecznym, czyli z wykorzystaniem protokołu SSL. Jeśli nie — odsyłamy użytkownikowi kod błędu 403 — *forbidden* (brak uprawnień do zrealizowania żądania).

Najważniejszym fragmentem metody jest wywołanie `doFilter()` obiektu `chain`. Zbieżność nazw w tej sytuacji jest przypadkowa (klasy `FilterChain` i `Filter` nie są ze sobą w żaden formalny sposób związane). Zadaniem interfejsu `FilterChain` jest zapewnienie komunikacji między filtrem (lub grupą filtrów) a serwletem. W zależności od kolejności deklaracji filtrów w deskryptorze wdrożenia żądanie HTTP przechodzi przez kolejne filtry (dzięki wywołaniom metody `doFilter()` w każdym filtrze), aż w końcu dociera do serwletu. Po zakończeniu obsługi przez serwlet, sterowanie powraca do kolejnych filtrów (w naszym przypadku od deklaracji zmiennej `spakowany`, aż do opuszczenia ostatniego filtru).

Konfiguracja filtrów w pliku web.xml

Samo utworzenie klas filtrów to za mało. Musisz dodać także odpowiednie wpisy w pliku *web.xml*. Zasada działania filtrów przypomina tę znaną z serwletów, dlatego również informacje podawane w deskryptorze wdrożenia powinny wydać Ci się znajome:

```
<filter>
    <filter-name>Pierwszy filtr</filter-name>
    <filter-class>pl.helion.jeeweb.Filtr<filter-class>
</filter>
<filter-mapping>
    <filter-name>Pierwszy filtr</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Tak jak w przypadku serwletów, mamy do czynienia z dwoma głównymi sekcjami — powiązaniem klasy filtru z abstrakcyjną nazwą (znacznik `filter`) i połączeniem tej nazwy z adresem URL (jedno- lub wieloznacznym — znacznik `filter-mapping`). Reguły podawania wartości (nazw klas, nazw abstrakcyjnych i wzorców URL) są takie same, jak w przypadku serwletów.

Ciekawostką jest możliwość podania w znaczniku `filter-mapping` znacznika `dispatcher`. Pozwala on określić, czy filtr ma być stosowany także w przypadkach specjalnych. Domyślna wartość `REQUEST` określa, że filtr ma być stosowany tylko przy bezpośrednich żądaniach użytkownika. Pliki, które będą dynamicznie ładowane przez serwlety (za pomocą mechanizmu dołączania lub przekazania), nie będą przekształcane przez filtr. Jeśli chcesz zmienić ustawienia domyślne, musisz podać wszystkie pożądane sytuacje za pomocą znaczników `dispatcher`:

```
<filter-mapping>
    <filter-name>Pierwszy filtr</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

Ostatni z przypadków powoduje wywołanie filtru także przy przekierowaniach na strony błędów.

Java Enterprise Edition to standard tworzenia aplikacji biznesowych wykorzystujących język Java. Opracowany przez firmę Sun Microsystems, działa w oparciu o wielowarstwową architekturę komponentową, oferując programistom bardzo rozbudowane możliwości tworzenia oprogramowania funkcjonującego na niemal dowolnym sprzęcie, w każdym systemie operacyjnym, z wykorzystaniem licznych serwerów aplikacji. Duża popularność rozwiązań Java EE i coraz powszechniejszy dostęp do technologii WWW sprawiają, że programiści sprawnie posługujący się tego rodzaju narzędziami rzadko figurują na listach osób poszukujących pracy, a jeśli już jakimś cudem się na nich znajdą, bardzo szybko otrzymują atrakcyjne propozycje zatrudnienia. Nauka swobodnego poruszania się w tym środowisku może też być wspaniałą, poszerzającą horyzonty przygodą, a gdy poznasz platformę Java EE, będziesz dysponował potężnym narzędziem, ułatwiającym tworzenie nawet najbardziej skomplikowanych aplikacji internetowych w bardzo efektywny i szybki sposób.

Studenci, programiści i hobbyści pragnący poznać środowisko Java Enterprise Edition często napotykają problem ze znalezieniem solidnych źródeł wiedzy, które pozwoliłyby im szybko i łatwo wejść w świat tej coraz bardziej popularnej technologii. Lukę tę z powodzeniem wypełnia książka **„Java EE 6. Programowanie aplikacji WWW"**. Dzięki niej wszyscy zainteresowani tematem zyskają możliwość poznania Java EE od podstaw i zdobycia praktycznej wiedzy, na podstawie której będą mogli rozwijać swoje umiejętności programistyczne w przyszłości. Ten podręcznik pozwala na szybkie rozpoczęcie przygody z tworzeniem aplikacji webowych, skutecznie wprowadzając w zagadnienia wykorzystywanych przy tym platform i mechanizmów, lecz nie pomijając też informacji o charakterze ogólnym. Jeśli niewiele mówią Ci skróty JSP, JPA, JSF czy JPQL, a chciałbyś zmienić ten stan rzeczy, bez wątpienia powinieneś sięgnąć po tę książkę, podobnie jak wszystkie osoby zainteresowane bezproblemowym używaniem całego spektrum nowoczesnych narzędzi oferowanych przez środowisko Java EE.

- Tworzenie serwletów
- Zastosowanie szablonów JSP
- Integracja danych z aplikacjami za pomocą mechanizmu JPA
- Używanie interfejsów i komponentów
- Korzystanie z technologii JSF
- Uniwersalny i wygodny dostęp do danych, czyli język JPQL
- Praktyczne przykłady realizacji

SPRAW, ABY TWORZENIE APLIKACJI WWW Z WYKORZYSTANIEM JAVA EE NIE MIAŁO PRZED TOBĄ TAJEMNIC.

Cena: 37,00 zł

Nr katalogowy: 5615

Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
44-100 Gliwice, skr. poczt. 462
32 230 98 63
http://helion.pl
e-mail: helion@helion.pl

ISBN 978-83-246-2659-5
9 788324 626595