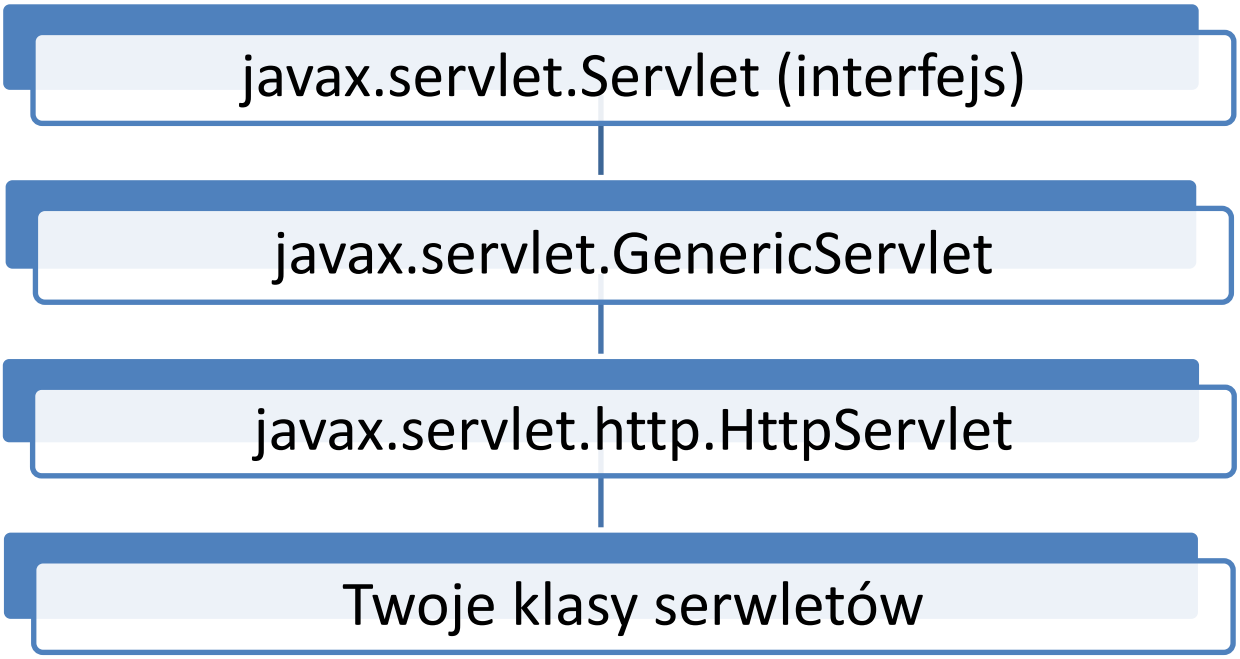
Moduł 2 – Serwlety – podstaw aplikacji webowych

W pierwszej aplikacji webowej, utworzonej przez nas w module 1., skorzystaliśmy ze strony JSP, aby wyświetlić informację użytkownikowi. Mechanizm stron JSP jest opisany w module 3., niemniej jak się w swoim czasie okaże, ten i wiele mechanizmów mają silny związek z podstawowym i najważniejszym składnikiem aplikacji webowych Java EE, który choć ostatnimi czasy nie jest aż tak intensywnie wykorzystywany w codziennej pracy z aplikacjami webowymi, nadal pełni rolę absolutnie kluczową i wiedza na jego temat jest niezwykle istotna.

Jak nietrudno się domyślić, mam na myśli zagadnienie tytułowe, czyli serwlet. Serwlet, zgodnie z definicją w standardzie Java EE, jest niewielką aplikacją uruchamianą w kontenerze webowym do obsługi żądań klienckich i generowania odpowiedzi. Definicja ta, choć pod pewnymi względami precyzyjna, z pewnością nie wyczerpuje tematu. Przede wszystkim, serwlet jest klasą dziedziczącą po klasie `javax.servlet.GenericServlet` (dokładna hierarchia dziedziczenia jest przedstawiona na rysunku 1.). Oczywiście sam fakt dziedziczenia to nie wszystko – tuż za nim idą zobowiązania związane z koniecznością implementacji pewnych metod.



Rysunek 3.1. Hierarchia dziedziczenia serwletów.

Jak wynika z rysunku 3.1., na czele hierarchii dziedziczenia znajduje się interfejs `Servlet`. Nie należy jednak w żadnym przypadku implementować bezpośrednio tego właśnie interfejsu. Dużo większe znaczenie ma klasa `GenericServlet`, która stanowi bazę, podstawę dla twórców serwletów. Dlaczego więc na rysunku 3.1. znajduje się jeszcze jeden

element? Otóż w założeniu serwlet nie musi być stosowany jedynie w aplikacjach webowych działających przy użyciu protokołu HTTP – nic nie stoi na przeszkodzie, aby tworzyć serwlety komunikujące się przy użyciu innych protokołów, o ile tylko zachowany jest model żądanie/odpowiedź. Tak się jednak składa, że ogromna większość serwletów, którą będziesz tworzyć w swoich aplikacjach, będzie działać w związku z protokołem HTTP, dlatego Twoje serwlety będą dziedziczyć bezpośrednio po klasie `HttpServlet`, znacznie lepiej przygotowanej do obsługi wspomnianego protokołu. Warto zauważyć, że podobne rozgraniczenie występuje także w przypadku innych ważnych typów w świecie serwletu – istnieją ogólne typy żądania/odpowiedzi (`ServletRequest`/`ServletResponse`), jak również te wyspecjalizowane dla protokołu HTTP (`HttpServletRequest`/`HttpServletResponse`).

Struktura aplikacji webowej

Zanim przejdziemy do tworzenia i konfiguracji serwletów, musimy omówić kilka kwestii bez znajomości których tworzenie serwletów jest procesem niezwykle trudnym i żmudnym. Pierwszą z tych kwestii jest umiejętność swobodnego poruszania się po aplikacji webowej, co niewątpliwie wymaga zaznajomienia się z jej strukturą.

Niektóre elementy aplikacji webowej są niezbędne do jej działania, podczas gdy obecność innych wynika z korzystania z określonego środowiska programistycznego, w którym obowiązują określone konwencje dotyczące struktury projektu. Z racji wykorzystania w tym szkoleniu środowiska Eclipse, w tym momencie skupimy się na strukturze projektu właśnie w tym środowisku, niemniej różnice w przypadku stosowania innych IDE (Integrated Development Environment – zintegrowane środowisko programistyczne) nie są na ogół duże.

Najpierw skoncentrujemy się na obowiązkowych składnikach projektu – tych, bez których funkcjonowanie aplikacji webowej nie jest możliwe. W projekcie aplikacji webowej kluczowe znaczenie ma podkatalog `WebContent` projektu, ponieważ to właśnie w nim znajdują się pliki, które tworzą aplikację webową, gotową do późniejszego wdrożenia. Warto pamiętać, że nazwa tego katalogu – `WebContent` – to tylko konwencja stosowana w środowisku Eclipse, można ją nawet zmienić na etapie tworzenia projektu. Obecność katalogu służy do oddzielenia esencji aplikacji webowej od pozostałych składników. Oto elementy tego katalogu:

- `/WebContent` – wszystkie pliki i katalogi **poza** podkatalogami `WEB-INF` i `META-INF` są publicznie dostępne, tj. użytkownik może uzyskać do nich dostęp, wprowadzając odpowiedni adres URL. W związku z tym, to w tym katalogu często są umieszczane pliki widoków (wspomniane strony JSP, zasoby statyczne takie jak CSS/JavaScript, itd.). Sytuacja może ulec zmianie, jeśli użytkownik wprowadzi określone ustawienia w pliku konfiguracyjnym.
- `/WebContent/WEB-INF` – podkatalog zawierający niemal wszystkie pliki, które nie mogą/nie powinny być bezpośrednio dostępne dla użytkowników.

- `/WebContent/WEB-INF/web.xml` – deskryptor aplikacji webowej. Jeden z najważniejszych elementów – jest to centralny plik konfiguracyjny dla całej aplikacji webowej. W Javie EE 6 jego znaczenie nieco maleje, ponieważ sporo elementów konfiguracji można przenieść do kodu Javy dzięki wprowadzeniu adnotacji.
- `/WebContent/WEB-INF/lib` – w tym podkatalogu należy umieszczać wszelkie biblioteki (pliki JAR), które muszą być użyte w aplikacji.
- `/WebContent/WEB-INF/classes` – ten podkatalog zawiera skompilowane pliki źródłowe utworzone w języku Java.
- `/WebContent/META-INF/` – obecność tego podkatalogu wynika z faktu, że aplikacja webowa gotowa do wdrożenia ma postać pojedynczego pliku o rozszerzeniu .WAR (Web ARchive), będącego w istocie specyficznym przypadkiem pliku .JAR (Java ARchive). W plikach JAR z kolei właśnie w tym podkatalogu znajduje się manifest archiwum – plik `MANIFEST.MF`. Czasami w tym podkatalogu są umieszczane dodatkowe informacje, takie jak plik jednostki utrwalania stosowany w technologii JPA.

Na tym kończy się opis zasadniczej części projektu aplikacji webowej Java EE, jednak środowisko Eclipse wprowadza także kilka dodatkowych elementów, bez których trudno wyobrazić sobie pełnoprawny projekt takiej aplikacji:

- `Java Resources` – zbiór zasobów języka Java, wśród których znajdują się pliki źródłowe klas i innych typów, a także można przejrzeć zawartość zaimportowanych bibliotek.
- `JAX-WS Web Services` – zawiera wszelkie informacje związane z usługami sieciowymi, m.in. informacje o końcówkach usług, czy też deklaracje własnych usług.
- `JPA Content` – jednostki utrwalania wykorzystywane w aplikacji – tylko, jeśli korzystamy z technologii JPA (omawiana szerzej w module 5.).
- `Deployment Descriptor` – bezpośredni dostęp do deskryptora wdrożenia, przydatny, jeśli nie lubisz dokonywać zmian w kodzie XML.

Na zakończenie jeszcze raz przypomnę, że wymienione bezpośrednio powyżej elementy nie wchodzą w skład podstawowego, obowiązkowego schematu projektu aplikacji webowej Java EE, choć są one częściej (jak pliki źródłowe Java) lub rzadziej (jak JAX-WS Web Services) wykorzystywane.

Plik konfiguracyjny web.xml

Plik konfiguracyjny web.xml będzie nam towarzyszył przez niemalże całe szkolenie, jednak przygodę z nim zaczniemy już teraz – od informacji i znaczników niezbędnych do wykorzystania serwletów w aplikacji webowej.

Na początku musimy zdać sobie sprawę w jaki sposób użytkownik, wpisujący w pasek adresu przeglądarki określony adres, może doprowadzić do wywołania serwletu, które w konsekwencji zwróci nam pożądaną odpowiedź. Otóż wszystkie informacje, które są potrzebne do wykonania tego procesu, znajdują się w pliku konfiguracyjnym `web.xml`.

Na początku konieczne jest powiązanie klasy serwletu (których tworzeniem zajmiemy się już niebawem) z pewną abstrakcyjną nazwą. Dodatkowo możemy określić różne inne własności takiego serwletu, takie jak uruchamianie serwletu przy starcie aplikacji czy też parametry inicjalizacji, o których powiemy niebawem. Aby osiągnąć taki efekt należy zadeklarować znacznik `servlet`:

```
<servlet>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <servlet-class>devcastzone.javaee.serwlety.SerwletListaUzytkownikow</servlet-
class>
</servlet>
```

To jednak wciąż za mało, aby użytkownik po wpisaniu adresu mógł trafić do naszego serwletu, nawet jeśli jest on znany pod bardziej przyjazną nazwą. Skoro brakuje nam powiązania pomiędzy adresem, a nazwą serwletu, musimy je po prostu stworzyć. Posłuży nam do tego celu znacznik `servlet-mapping`:

```
<servlet-mapping>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <url-pattern>/uzytkownicy/</url-pattern>
    <url-pattern>/innaLista/</url-pattern>
</servlet-mapping>
```

W znaczniku `servlet-mapping` jeden serwlet musi otrzymać co najmniej jeden wzorzec URL. Jeśli żądanie użytkownika będzie pasować do podanego wzorca, zostanie ono przekazane do serwletu o podanej nazwie, co w konsekwencji doprowadzi do naszej klasy serwletu. *Voila*!

Zanim przejdziemy dalej, pomówimy jeszcze chwilę o wzorcach URL. Wzorce te mogą bowiem przyjmować postać bardziej skomplikowaną, niż ta przedstawiona powyżej. Powyższe wzorce stanowią przykład wzorców dokładnych – fragment adresu URL znajdujący się za fragmentem bazowym aplikacji. Adresem bazowym może być np. nazwa domenowa z ukośnikiem, jeśli serwer zostanie odpowiednio skonfigurowany – np. *http://www.serwer.com/* – ale na ogół jest to nazwa domenowa wraz z katalogiem aplikacji – np. *http://www.serwer.com/HelloWorld/*. W tym drugim przypadku użytkownik trafi do serwletu `Lista Uzytkownikow`, jeśli wprowadzi adres *http://www.serwer.com/HelloWorld/uzytkownicy/* **lub** *http://www.serwer.com/HelloWorld/innaLista/*. Jeśli jednak chcielibyśmy, aby serwlet otrzymał żądania zawierające adresy o pewnej części wspólnej, wystarczy skorzystać ze znaku `*`:

```
<servlet-mapping>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <url-pattern>/uzytkownicy/*</url-pattern>
    <url-pattern>/innaLista/</url-pattern>
</servlet-mapping>
```

W ten sposób zostaną obsłużone zarówno adres `/uzytkownicy/`, jak i `/uzytkownicy/pierwszy`. Trzeci wariant formatu wzorca pozwala na przypisanie serwletu do żądań do plików o określonym rozszerzeniu. Nie podajemy wtedy fragmentów wzorca, a jedynie rozszerzenia wraz ze znakiem wieloznacznym:

```
<servlet-mapping>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <url-pattern>*.html</url-pattern>
    <url-pattern>/innaLista/</url-pattern>
  </servlet-mapping>
```

W ten sposób do serwletu trafi np. żądanie `/uzytkownicy/index.html`, ale już nie `/uzytkownicy/`. Warto zauważyć, że plik `index.html` nie musi wcale istnieć – w aplikacjach webowych Java EE adres URL to tylko łańcuch znaków i może on być przetwarzany przy użyciu powyższych wzorców. O tym, co zostanie zwrócone do klienta decyduje serwlet, który otrzyma żądanie.

Zanim na jakiś czas rozstaniemy się z plikiem `web.xml`, nauczymy się jeszcze dodawać do niego parametry inicjalizacyjne. Parametr inicjalizacyjny serwletu pozwala na określenie pewnej wartości w kodzie XML, która następnie może być odczytana w serwlecie. Można zapytać, czy nie prościej po prostu zadeklarować stałą w kodzie serwletu. Prościej – być może, ale jakakolwiek zmiana stałej wymaga rekompilacji plików źródłowych, co nie dotyczy oczywiście zmian w pliku XML – w takiej sytuacji wystarczy jedynie zrestartowanie serwera. Parametry inicjalizacyjne deklarujemy w ramach znacznika `servlet`:

```
<servlet>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <servlet-class>devcastzone.javaee.serwlety.SerwletListaUzytkownikow</servlet-
class>
    <init-param>
      <param-name>ZrodloDanych</param-name>
      <param-value>Pamiec</param-value>
    </init-param>
</servlet>
```

Tworzenie serwletów

Nadszedł czas, aby wreszcie utworzyć nasz pierwszy serwlet. Mimo że Eclipse udostępnia własny kreator serwletów, który znacznie upraszcza proces ich tworzenia, nasz pierwszy serwlet stworzymy od podstaw. Utwórz więc klasę `SerwletListaUzytkownikow` w pakiecie `devcastzone.javaee.serwlety`, pamiętając o dziedziczeniu po klasie `javax.servlet.http.HttpServlet`. Następnie uzupełnij kod zgodnie z poniższym listingiem:

```
package devcastzone.javaee.serwlety;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.*;

public class SerwletListaUzytkownikow extends HttpServlet {
	@Override
	protected void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
		res.getWriter().println("Lista użytkowników:");
	}
}
```

Jak widać, pusta deklaracja klasy `SerwletListaUzytkownikow` rozszerzyła się o metodę `doGet()`. Metoda ta, pochodząca z klasy `HttpServlet`, jest wywoływana w momencie, gdy serwlet otrzymuje żądanie przy użyciu metody GET protokołu HTTP. Jeśli ktokolwiek spróbuje wywołać nasz serwlet przy użyciu metody POST, to niestety taka próba się nie powiedzie. Można to jednak nadrobić, deklarując metodę o zbliżonej nazwie – `doPost()`:

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
	doGet(req, res);
}
```

Jest to częsta praktyka – tylko jedna z metod ze słowem do w nazwie zawiera faktyczną treść, podczas gdy reszta niejako przekierowuje żądania do tej samej metody. W ten sposób bez względu na użytą metodę, użytkownik otrzymuje ten sam rezultat.

Skupmy się jednak teraz na samej konstrukcji metody. Każda z metod obsługi otrzymuje dwa obiekty – żądanie i odpowiedź. Dzięki żądaniu może ona uzyskać informacje niezbędne do prawidłowego działania, podczas gdy odpowiedź służy do wygenerowania treści, którą otrzyma użytkownik.

Aby wygenerować najprostszą, de facto statyczną odpowiedź, nie potrzebujemy w ogóle obiektu żądania. Przy użyciu obiektu odpowiedzi możemy uzyskać obiekt strumienia tekstowego – typu `PrintWriter`. W tym momencie wystarczy skorzystać z jednego z licznych wariantów metod `print/println()`, aby dodać do strumienia tekst. Jak nietrudno się domyślić, ten tekst powędruje do użytkownika. Oczywiście nie musi to być zwykły tekst – można także, a nawet trzeba, wysyłać kod HTML, dzięki czemu przeglądarka będzie mogła wyświetlić użytkownikowi elegancko sformatowaną informację. W naszym przykładzie pojawia się jednak mały szkopuł – przeglądarka niemal na pewno wyświetli tekst, w którym zamiast polskich znaków pojawią się "krzaczki". Wynika to z faktu, że nie

ustawiliśmy kodowania odpowiedzi. W tym celu musimy dodać jeszcze jedną instrukcję do treści metody `doGet()` – oczywiście przed wywołaniem metod `println()`:

```
res.setContentType("text/plain; charset=utf-8");
```

Co więcej, jeśli zamiast metody `getWriter()` skorzystasz z metody `getOutputStream()`, uzyskasz dostęp do strumienia binarnego, co stanowi wstęp do przesłania do klienta plików binarnych! Oto pełny przykład przesłania do klienta danych binarnych w taki sposób, aby użytkownik mógł plik zapisać na dysku:

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException
{
     byte[] b = ... // pobranie pliku z dysku
     response.setContentType("image/png");
     response.addHeader("Content-Disposition", "attachment;
     filename=obrazek.png");
     response.setContentLength(b.length);
     OutputStream os = response.getOutputStream();
     os.write(b);
     os.flush();
}
```

Jak widać, obiekt odpowiedzi udostępnia także inne możliwości, np. ustawienie typu MIME odpowiedzi, dodanie nagłówka, jak również określenie długości żadania. Najważniejsze jest jednak zapisanie do strumienia wyjściowego danych binarnych pochodzących z pliku.

Do tej pory dużo mówiliśmy o odpowiedzi, ale tak naprawdę (poza możliwością dodania do odpowiedzi ciasteczek przy użyciu metody `addCookie()`), to już niemal koniec ich możliwości. Sama interakcja z obiektem odpowiedzi nie jest interesująca – za to o wiele większe znaczenie ma proces tworzenia treści, która w tej odpowiedzi się znajdzie.

Do tworzenia treści wykorzystuje się zaś obiekt żądania. Można powiedzieć, że obiekty żądania i odpowiedzi pod wieloma względami są do siebie podobne. Obiekt żądania pozwala na odczytywanie ciasteczek (`getCookies()`), podczas gdy obiekt odpowiedzi pozwala na ich dodanie (`addCookie()`). Analogicznie wygląda sytuacja z nagłówkami (`getHeader()/addHeader()`). W przypadku żądania kluczowe znaczenie ma jednak metoda `getParameter()`, która pozwala na pobranie wartości parametru przesłanego w ramach żądania, niezależnie od tego, czy jest to parametr przesłany przy użyciu łańcucha zapytań (metoda GET), czy też w sposób niewidoczny dla użytkownika (POST).

Jeśli więc poniższy kod:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
      res.setContentType("text/plain; charset=utf-8");
      res.getWriter().println("Lista użytkowników:");
      res.getWriter().println("Wybrany użytkownik: " +
req.getParameter("uzytkownik"));
}
```

wywołamy przy użyciu adresu:

http://localhost:8080/HelloWorld/uzytkownicy/index.html?uzytkownik=jan

to wtedy zostanie wyświetlona wartość parametru uzytkownik. Jeśli parametr ten nie zostanie określony, wtedy zostanie wyświetlona domyślna wartość zwracana przez metodę `getParameter()`, czyli `null`.

Obsługa sesji

Mimo tego, że omówiliśmy raptem kilka podstawowych metod obiektów odpowiedzi i żądania, tak naprawdę zyskaliśmy całkiem potężne narzędzia, dzięki którym teoretycznie jest możliwe osiągnięcie niemal każdego pożądanego efektu – możemy bowiem przyjmować dane od użytkownika i generować dla niego informacje, i to na różne sposoby. Czegóż chcieć więcej? Przede wszystkim – jednego z najbardziej irytujących aspektów protokołu HTTP – jego bezstanowości.

Bezstanowość protokołu HTTP sprowadza się do tego, że serwer WWW nie jest w (nomen omen) stanie zidentyfikować klienta na przestrzeni wielu żądań. Jeśli więc klient (przeglądarka) wykona pierwsze żądanie w celu uzyskania treści strony HTML (tudzież kodu HTML wygenerowanego przez serwlet), a następnie spróbuje pozyskać obrazek, wykorzystywany w tymże kodzie, to oba te żądania będą stanowić dla serwera zupełnie odrębną sprawę. Jest to niezwykle kłopotliwe, bo w praktyce wszelkie systemy wykorzystujące różne aspekty zarządzania tożsamością/użytkownikami musiałyby za każdym razem pytać użytkownika o dane logowania. Nie trzeba chyba mówić jak znacząco obniżyłoby to wrażenia z wykorzystywania aplikacji, nie mówiąc już o innych mechanizmach, takich jak koszyk użytkownika w sklepie. W związku z tym opracowano mechanizm sesji, który naprawia ten problem.

Idea sesji polega na tym, że obsługując dowolne żądanie HTTP serwlet może zdecydować się na rozpoczęcie sesji z użytkownikiem, od którego dane żądanie pochodzi. W związku z tym, do odpowiedzi na to żądanie zostanie dołączona informacja – specjalny identyfikator. Przy okazji kolejnych żądań realizowanych przez tego samego klienta ów identyfikator będzie odsyłany, a na jego podstawie serwer będzie w stanie odczytać dane przechowane dla tego właśnie identyfikatora.

Problemem pozostaje sposób przekazywania identyfikatora. Najczęściej służy do tego celu ciasteczko, ale alternatywną metodę (stosowaną zwłaszcza dawniej) stanowiło dołączanie identyfikatora do adresów URL żądań, dzięki czemu zawierały one ciąg losowych i

nieistotnych, jak mogło się wydawać, znaków. My jednak nie musimy martwić się metodą, ani samym identyfikatorem. Oczywiście sesja swym zasięgiem obejmuje nie tylko jeden serwlet. Aby rozpocząć działanie sesji, należy wywołać metodę `getSession()` obiektu żądania:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
	HttpSession sesja = req.getSession();
	Object o = sesja.getAttribute("licznik");
	int licznik = 0;
	if (o != null)
		licznik = Integer.parseInt(o.toString());
	sesja.setAttribute("licznik", ++licznik);
	res.setContentType("text/plain; charset=utf-8");
	res.getWriter().println("Licznik odwiedzin: " + licznik);
}
```

Cała zabawa z sesją sprowadza się do użycia metod `get/setAttribute()`, pozwalających na traktowanie sesji jak zwykłej, prostej mapy. Nie jest to jednak jedyna funkcjonalność sesji – istotne są również mechanizmy z kończeniem sesji. Istnieją bowiem trzy sposoby zakończenia sesji:

- zamknięcie przeglądarki przez użytkownika (w tym przypadku nie trzeba nic robić),
- wyczyszczenie sesji z poziomu kodu,
- upłynięcie określonego limitu czasu

Aby wyczyścić sesję ręcznie, należy po prostu wywołać metodę `invalidate()`. Nasz wpływ na trzecią metodę kończenia sesji polega na możliwości zmiany owego limitu. Do manipulacji służy para metod `get/setMaxInactiveInterval()`, która pozwala na określenie limitu w sekundach (podanie negatywnej wartości spowoduje wyłączenie limitu czasu).

Atrybuty i ich zasięgi

Na przykładzie sesji poznaliśmy nieświadomie jeden z najważniejszych aspektów aplikacji webowych Java EE – możliwość wykorzystywania atrybutów. Atrybuty to zwykłe obiekty – od wartości prymitywnych opakowanych w odpowiednie *wrappery*, aż po bardziej skomplikowane instancje rozmaitych klas. Każdy obiekt atrybutu ma swój klucz, dzięki któremu można odszukać go w mapie atrybutów. Istnieją trzy odrębne mapy atrybutów, zależne od **zasięgu**, w którym one występują. I tak, mamy do czynienia z następującymi zasięgami:

- żądania (*request*)
- sesji (*session*)
- kontekstu aplikacji (*application context*, *servlet context*)

Wszystkie trzy mapy są oczywiście zupełnie odrębne, dlatego też każda z nich może zawierać różne atrybuty o takich samych kluczach (nie należy jednak doprowadzać zbyt często do

takich sytuacji – więcej na ten temat w module 3.). W tym momencie skupmy się jednak na samych zasięgach, ponieważ korzystanie z map atrybutów nie jest zbyt ekscytujące – sprowadza się do wywoływania metod `getAttribute()` i `setAttribute()`.

Umieszczenie atrybutu w zasięgu żądania sprawi, że dany atrybut będzie dostępny w przeciągu całego żądania. W tej chwili może wydawać się to bezsensowne – w końcu obsługa żądania przy naszym obecnym stanie wiedzy zaczyna się i kończy w jednej z metod do*(). Jak się jednak dowiemy niebawem, będziemy w stanie m.in. przekazywać żądania pomiędzy serwletami, a taka sytuacja sprawia, że atrybuty żądania nabierają całkiem dużego sensu.

O sensie istnienia atrybutów sesji już dyskutowaliśmy – ich obecność jest całkiem zasadna. Wiemy również jak należy rozpatrywać czas życia sesji – jego rozpoczęcie powoduje wywołanie metody `getSession()`, a przyczyny jej zakończenia mogą być różnorakie.

Zdecydowanie najbardziej tajemniczym z całej trójki jest zasięg kontekstu aplikacji. Kontekst aplikacji jest bowiem specjalnym obiektem, tworzonym przy starcie aplikacji, a usuwanym przy jej zakończeniu, dostępnym w obrębie **całej** aplikacji webowej. Obiekt ten udostępnia rozmaite informacje na temat samej aplikacji, a także jej ustawień, a ponadto (od Javy EE 6) pozwala na dynamiczne dodawanie do niej serwletów, filtrów i innych mechanizmów (więcej na ten temat w module 6.). Poza tym, pozwala on na ustawianie i pobieranie atrybutów, dzięki czemu możliwe jest udostępnianie obiektów w obrębie całej aplikacji. Choć z jednej strony takie rozwiązanie wydaje się być panaceum na pewne problemy, to rodzi ono ze sobą pewne istotne problemy, które nie mają miejsca w przypadku dwóch pozostałych zasięgów.

Problemem tym jest dostęp współbieżny. O ile sam odczyt atrybutów o zasięgu kontekstu aplikacji nie stanowi problemu, o tyle próba współbieżnego zapisu czy też zmiany już istniejącego atrybutu rodzi pytanie – kto wygra, kto będzie szybszy? Z tego względu, wszelkie próby wykonywania operacji zmiany wartości atrybutów o zasięgu kontekstu powinny być synchronizowane. Jednocześnie, należy zwrócić uwagę, że zbyt często występująca synchronizacja może spowolnić działanie aplikacji, dlatego też atrybuty kontekstu aplikacji nie powinny być zbyt często modyfikowane. Jeśli w Twojej aplikacji dokonuje się zbyt wielu modyfikacji atrybutów, z pewnością warto rozważyć przyczyny, dla których jest stosowane takie zachowanie. Warto również zwrócić uwagę, że teoretycznie dostęp do atrybutów sesji również nie jest bezpieczny wątkowo, ponieważ użytkownik może uruchomić dwa żądania w tym samym czasie. Jest to jednak sytuacja o wiele mniej prawdopodobna, dlatego na ogół nie trzeba podejmować aż takich środków ochronnych, chyba, że w szczególnych sytuacjach.

Na zakończenie tej sekcji warto jeszcze wspomnieć o metodach dostępu do obiektu kontekstu aplikacji. Możemy to uczynić na trzy sposoby:

```
ServletContext sc = this.getServletContext();
sc = req.getServletContext();
sc = this.getServletConfig().getServletContext();
```

Jak widać, kontekst serwletów (aplikacji) jest dostępny po prostu z poziomu obiektu serwletu, ale również z poziomu obiektu żądania i obiektu konfiguracji serwletów –

`ServletConfig`. Trzecia metoda stanowi ciekawostkę, ponieważ obiekt konfiguracji serwletu stosuje się niezwykle rzadko – te same informacje są dostępne po prostu bezpośrednio z poziomu serwletu.

Jak już wspomniałem, przeprowadzanie operacji na atrybutach po uzyskaniu dostępu do odpowiednich obiektów jest dziecinnie proste:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
            ServletContext kontekst = this.getServletContext();
            HttpSession sesja = req.getSession();

            req.setAttribute("atrybutŻądania", "1");
            sesja.setAttribute("atrybutSesji", "2");
            synchronized (kontekst) {
                  kontekst.setAttribute("atrybutKontekstu", "3");
            }
            res.setContentType("text/plain; charset=utf-8");
            res.getWriter().println("Żądanie: " +
req.getAttribute("atrybutŻądania"));
            res.getWriter().println("Sesja: " +
sesja.getAttribute("atrybutSesji"));
            res.getWriter().println("Kontekst: " +
kontekst.getAttribute("atrybutKontekstu"));
}
```

Cykl życia aplikacji a nasłuchiwanie zdarzeń

Na przestrzeni tego modułu nauczyliśmy się do tej pory korzystać z obiektów żądania, sesji i kontekstu aplikacji. Samo ich stosowanie (wraz z obiektem odpowiedzi) w ramach metod `doGet/doPost`/itd., to nie wszystko, co możemy zdziałać w aplikacji webowej. Dzięki bogatej palecie interfejsów pozwalających na nasłuchiwanie rozmaitych zdarzeń związanych z aplikacją webową, jesteśmy w stanie reagować na praktycznie każde możliwe do wyobrażenia zdarzenie z nią związane.

Ze względu na dużą liczbę interfejsów (i jeszcze większą liczbę metod w nich zawartych) do ich omawiania podejdziemy w sposób usystematyzowany. Wszelkie typy związane z parametrami zdarzeń omówimy zbiorczo na końcu tej sekcji. Zaczniemy od interfejsów, które dotyczą bezpośrednio naszych trzech głównych obiektów – żądania, sesji i kontekstu aplikacji. Wszystkie trzy interfejsy będą dotyczyć zdarzeń utworzenia i usunięcia wymienionych obiektów:

1. `ServletRequestListener` – dotyczy zdarzeń cyklu życia obiektu żądania.
    a. `requestInitialized(ServletRequestListener evt)` – metoda wywoływana tuż przed dodaniem obiektu żądania do aplikacji.
    b. `requestDestroyed(ServletRequestListener evt)` – metoda wywoływana tuż przed pozbyciem się obiektu żądania z aplikacji.
2. `HttpSessionListener` – dotyczy zdarzeń cyklu życia obiektu sesji.

   a. `sessionCreated(HttpSessionEvent evt)` – metoda wywoływana tuż po utworzeniu obiektu sesji.
   b. `sessionDestroyed(HttpSessionEvent evt)` – metoda wywoływana tuż przed usunięciem (wygaśnięciem) sesji.
3. `ServletContextListener` – dotyczy zdarzeń cyklu życia aplikacji.
   a. `contextInitialized(ServletContextEvent evt)` – metoda wywoływana przy starcie aplikacji webowej (przed inicjalizacją jakichkolwiek serwletów i filtrów).
   b. `contextDestroyed(ServletContextEvent evt)` – metoda wywoływana tuż przed zakończeniem aplikacji webowej (po zakończeniu działania jakichkolwiek serwletów i filtrów).

Druga grupa interfejsów do nasłuchiwania zdarzeń pozwala na reagowanie na dodanie, usunięcie lub zmianę wartości atrybutu:

1. `ServletRequestAttributeListener` – dotyczy atrybutów o zasięgu żądania.
   a. `attributeAdded(ServletRequestAttributeEvent evt)` – metoda wywoływana tuż po dodaniu atrybutu do zasięgu żądania.
   b. `attributeRemoved(ServletRequestAttributeEvent evt)` – metoda wywoływana tuż po usunięciu atrybutu z zasięgu żądania.
   c. `attributeReplaced(ServletRequestAttributeEvent evt)` – metoda wywoływana tuż po zastąpieniu jednego atrybutu innym w zasięgu żądania.
2. `HttpSessionAttributeListener` – dotyczy atrybutów o zasięgu sesji.
   a. `attributeAdded(HttpSessionBindingEvent evt)` – metoda wywoływana tuż po dodaniu atrybutu do zasięgu sesji.
   b. `attributeRemoved(HttpSessionBindingEvent evt)` – metoda wywoływana tuż po usunięciu atrybutu z zasięgu sesji.
   c. `attributeReplaced(HttpSessionBindingEvent evt)` – metoda wywoływana tuż po zastąpieniu jednego atrybutu innym w zasięgu sesji.
3. `ServletContextAttributeListener` – dotyczy atrybutów o zasięgu kontekstu aplikacji.
   a. `attributeAdded(ServletContextAttributeEvent evt)` – metoda wywoływana tuż po dodaniu atrybutu do zasięgu kontekstu aplikacji.
   b. `attributeRemoved(ServletContextAttributeEvent evt)` – metoda wywoływana tuż po usunięciu atrybutu z zasięgu kontekstu aplikacji.
   c. `attributeReplaced(ServletContextAttributeEvent evt)` – metoda wywoływana tuż po zastąpieniu jednego atrybutu innym w zasięgu kontekstu aplikacji.

Wreszcie, możemy przejść do dwóch pozostałych interfejsów do nasłuchiwania, które z pewnością są znacznie ciekawsze od tych już przedstawionych:

1. `HttpSessionBindingListener` – pozwala na informowanie obiektów umieszczanych jako atrybuty w zasięgu sesji:
    a. `valueBound(HttpSessionBindingEvent evt)` – metoda wywoływana w momencie umieszczania obiektu klasy, w której ta metoda została zadeklarowana, w zasięgu sesji.
    b. `valueUnbound(HttpSessionBindingEvent evt)` – metoda wywoływana w momencie usunięcia obiektu klasy, w której ta metoda została zadeklarowana, z zasięgu sesji.
2. `HttpSessionActivationListener` – pozwala na informowanie obiektów umieszczanych jako atrybuty w zasięgu sesji o przeniesieniu sesji do innej maszyny wirtualnej Java:
    a. `sessionDidActivate(HttpSessionEvent evt)` – informuje obiekt, że sesja została przywrócona na nowej maszynie wirtualnej.
    b. `sessionWillPassivate(HttpSessionEvent evt)` – informuje obiekt, że sesja zostanie wygaszona przed przeniesieniem do nowej maszyny wirtualnej.

Oczywiście utworzenie klasy to nie wszystko – konieczne jest dodanie niezbędnego fragmentu kodu do pliku `web.xml` (tudzież zastosowanie adnotacji):

```
<listener>
  <listener-class>devcastzone.javaee.DziennikAtrybutowSesji</listener-class>
</listener>
```

Na koniec warto wspomnieć o możliwościach, jakie udostępniają nam parametry metod zdarzeń. Typy owych parametrów występują mniej licznie, niż same interfejsy, ale i tak warto znać różnice pomiędzy nimi:

- `ServletRequestListener` – udostępnia informacje o danym obiekcie żądania i kontekście serwletów (aplikacji).
- `ServletRequestAttributeEvent` – udostępnia informacje o atrybucie w zasięgu żądania.
- `ServletContextEvent` – udostępnia informacje wyłącznie o kontekście aplikacji.
- `ServletContextAttributeEvent` – udostępnia informacje o atrybucie w zasięgu kontekstu aplikacji.
- `HttpSessionEvent` – udostępnia informacje o obiekcie sesji.
- `HttpSessionBindingEvent` – udostępnia informacje o atrybucie w zasięgu sesji.

Bezpieczeństwo aplikacji webowych

Bezpieczeństwo aplikacji webowych to bez wątpienia jedna z najważniejszych kwestii, którą trzeba zająć się po utworzeniu niezbędnych funkcjonalności. Na szczęście, w aplikacjach webowych Java EE implementowanie rozmaitych zasad i reguł związanych z bezpieczeństwem nie stanowi dużego problemu. Niemal wszystkie informacje potrzebne do

zdefiniowania prawidłowych zasad bezpieczeństwa można podać w pliku `web.xml`. Dzięki temu, prace nad bezpieczeństwem aplikacji można niemal zupełnie odseparować od prac programistycznych nad tworzeniem serwletów, co pozytywnie wpływa na prowadzenie projektu w rozbudowanych zespołach programistycznych.

Aby zdać sobie sprawę z możliwości, jakie dają nam mechanizmy wbudowane w Javę EE, należy przypomnieć sobie kilka pojęć związanych z bezpieczeństwem:

- użytkownikiem będziemy nazywać klienta aplikacji, który jest w stanie dostarczyć poświadczenia (ang. *credentials*) potwierdzające jego tożsamość
- role pozwalają na określanie poziomów dostępu do funkcjonalności aplikacji – jednym słowem odpowiadają one za autoryzowanie użytkowników. Przykładami ról mogą być menedżer, administrator, moderator, kierownik.
- wzorce URL, zwłaszcza te wieloznaczne, są wykorzystywane do określania możliwości poszczególnych ról. Na przykład, jest możliwe, aby dostęp do puli adresów pasujących do wzorca `/admin/*` był dopuszczony tylko dla użytkowników o roli administrator.

W tym momencie możemy przejść już do części praktycznej, czyli zapoznania się z przykładowymi zasadami bezpieczeństwa, na przykładzie naszego serwletu `Lista Uzytkownikow`. Kluczowym znacznikiem dla definiowania zasad bezpieczeństwa jest `security-constraint`:

```
<servlet>
    <servlet-name>Lista Uzytkownikow</servlet-name>
    <servlet-
class>devcastzone.javaee.serwlety.SerwletListaUzytkownikow</servlet-class>
    <init-param>
      <param-name>ZrodloDanych</param-name>
      <param-value>Pamiec</param-value>
    </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Lista Uzytkownikow</servlet-name>
  <url-pattern>/uzytkownicy/lista</url-pattern>
</servlet-mapping>
<security-constraint>
      <web-resource-collection>
            <web-resource-name>Zasoby użytkowników</web-resource-name>
            <url-pattern>/uzytkownicy/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>admin</role-name>
      </auth-constraint>
      <user-data-constraint>
            <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
</security-constraint>
<login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>default</realm-name>
</login-config>
```

Powyższy kod jest dość długi – dla porządku umieściliśmy od razu całość, wraz z opisem serwletu, jednak skupmy się głównie na znaczniku `security-constraint`. Znacznik ten zawiera przede wszystkim znacznik `web-resource-collection`. W ten sposób możemy zdefiniować wzorzec URL i metody, które zostaną zabezpieczone dla tego wzorca. Oznacza to, że w powyższym przykładzie wykonanie żądania PUT do adresów URL zawierających część `/uzytkownicy/` powiedzie się. Nie musi to nas jednak martwić, bo w naszym serwlecie obsługujemy wszak jedynie metody GET i POST.

Obok wzorca URL określamy rolę, która otrzymuje pozwolenie na dostęp. W razie potrzeby możemy deklarować więcej ról, bo przecież dostęp do listy użytkowników może być potrzebny także kierownikom.

Ostatni znacznik umieszczony w ramach wytycznej bezpieczeństwa to wytyczna dotycząca zasad transferu danych. W tym przypadku nie wymuszamy w żaden sposób bezpiecznego transferu – odpowiada za wartość `NONE`. Ponadto można także skorzystać z wartości `CONFIDENTIAL`, wymuszającej na kontenerze webowym utworzenie bezpiecznego połączenia, uniemożliwiającego podsłuchanie transmisji (niegdyś wykorzystywano również wartość `INTEGRAL`, zapobiegającą możliwości modyfikacji danych, jednak w praktyce serwery najczęściej traktują obie wartości tak samo, stosując szyfrowanie SSL/TLS).

Sam znacznik `security-constraint` wprowadza całkiem sporo informacji, jednak z praktycznego punktu widzenia brakuje nam jednej istotnej informacji – skąd aplikacja ma wziąć dane o użytkownikach i rolach? A poza tym w jaki sposób ma ona uwierzytelnić użytkownika? Na drugie pytanie odpowiada znacznik `login-config` – pozwala on na wybór jednej z czterech metod logowania:

- `BASIC`: tradycyjne logowanie przy użyciu protokołu HTTP (przeglądarka wyświetla okno z prośbą o wprowadzenie loginu i hasła). Dane uwierzytelniające są przesyłane po przetworzeniu na format `Base64`.
- `DIGEST`: najmniej popularna metoda, rzadko implementowana w serwerach aplikacji. Stanowi ona bezpieczniejszą odmianę metody BASIC, ponieważ dane są szyfrowane.
- `CLIENT-CERT`: uwierzytelnianie z użyciem certyfikatów. Metoda najbezpieczniejsza (wykorzystuje SSL do szyfrowania danych), jednak równocześnie najtrudniejsza do powszechnego stosowania – w praktyce spotykana w aplikacjach z niewielką (stałą bądź ograniczoną) liczbą zaufanych klientów.
- `FORM`: pozwala na pobranie danych logowania przy użyciu formularza, dzięki czemu jest możliwe zintegrowanie zwykłych widoków HTML z całym systemem bezpieczeństwa. Umożliwia podanie dwóch dodatkowych znaczników: `form-login-page`, zawierającego adres strony logowania, i `form-error-page`, zawierającego adres strony o błędzie logowania.

W tym momencie wiemy już jak można przeprowadzić logowanie, jednak cały czas nie wiemy skąd aplikacja bierze wszystkie niezbędne dane – loginy i hasła użytkowników, a także ich powiązania z rolami. Odpowiedź na to pytanie zależy już od serwera aplikacji. W przypadku serwera Tomcat najczęściej spotykane są dwie możliwości:

- skorzystanie z pliku `tomcat-users.xml`, zawierającego wszystkie niezbędne informacje (wariant domyślny)
- dodanie do pliku konfiguracyjnego znacznika, definiującego źródło informacji o użytkownikach.

Wariant domyślny jest przydatny, jeśli chcemy jedynie przetestować podstawowe reguły bezpieczeństwa, tudzież nasze grono użytkowników aplikacji jest i będzie ograniczone. W każdym innym przypadku warto zastosować drugi wariant. Oto przykładowy kod z pliku `conf/server.xml`, umieszczony bezpośrednio w znaczniku `Service`:

```
<Realm className="org.apache.catalina.realm.JDBCRealm" driverName="com.mysql.jdbc.Driver"
digest="MD5" connectionURL="jdbc:mysql://localhost/testowa?user=root" userTable="uzytkownik"
userNameCol="login" userCredCol="haslo" userRoleTable="rola" roleNameCol="rola" />
```

Poza niezbędnym opisem tabel i kolumn w bazie danych mamy możliwość definicji sposobu szyfrowania hasła, a także określenie sterownika JDBC, który wykorzystujemy w aplikacji.

Filtry

Ostatnim zagadnieniem, które poruszymy w tym module, są filtry. Filtry pozwalają na wykonanie wstępnego i/lub końcowego przetwarzania żądań, zanim trafią one do wyznaczonych serwletów, tudzież już po zakończeniu ich przetwarzania właśnie przez serwlety. Filtr w dużej mierze przypomina serwlet – jego konfiguracja wygląda niemal identycznie – filtry, za pośrednictwem abstrakcyjnych nazw, są wiązane z wzorcami URL:

```
<filter>
  <filter-name>Filtr reklamowy</filter-name>
  <filter-class>devcastzone.javaee.filtry.FiltrSpamowy</filter-class>
</filter>
<filter-mapping>
  <filter-name>Filtr reklamowy</filter-name>
  <url-pattern>/uzytkownicy/*</url-pattern>
</filter-mapping>
```

W ten sposób, każde żądanie wysłane pod adres z grupy `/uzytkownicy/`, zostanie przetworzone przy użyciu filtru `FiltrSpamowy`. Jego konstrukcja jest prostsza, niż mogłoby się wydawać:

```
public class FiltrSpamowy implements Filter {
      public void destroy() { }
      public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
            response.setContentType("text/plain; charset=UTF-8");
            response.getWriter().println("REKLAMA: TANIO KUPIĘ/SPRZEDAM");
            chain.doFilter(request, response);
            response.getWriter().println("PONOWNIE: REKLAMA: TANIO
KUPIĘ/SPRZEDAM");
      }
      public void init(FilterConfig fConfig) throws ServletException { }
}
```

Kluczowym elementem klasy filtru jest metoda `doFilter()` (kolejne prawdopodobieństwo do klasy serwletu, nieprawdaż?). Do strumienia odpowiedzi wypisujemy, w dość ordynarny sposób, treść reklamy. Co więcej, robimy to zarówno przed przetworzeniem żądania, jak i po – cały proces przetwarzania odbywa się bowiem w momencie wywołania metody `doFilter()` obiektu łańcucha wywołań. Łańcuch ten odpowiada za wywołanie kolejnych filtrów, a na koniec – za przekazanie sterowania do serwletu. Oczywiście trzeba mieć świadomość, że takie zachowanie – zwłaszcza ustawianie nagłówków – może kolidować z zachowaniem serwletu (rozpoczęcie wypisywania treści odpowiedzi uniemożliwia zmianę nagłówków). Z drugiej strony, nic nie stoi na przeszkodzie, aby w ogóle nie dopuścić do wywołania serwletu – wystarczy uzależnić wywołanie metody `doFilter()` od spełnienia jakiegoś warunku. W praktyce, filtry są wykorzystywane głównie do wszelkiego rodzaju uniwersalnego przetwarzania żądań, czy też odpowiedzi, np.:

- szyfrowanie/deszyfrowanie,
- kompresja,
- rejestrowanie informacji w dzienniku,

- wyrafinowane uwierzytelnianie/autoryzacja, jeśli standardowe mechanizmy bezpieczeństwa omówione w poprzedniej sekcji okazują się niewystarczające.