

Serwety i JSP

wg.

SERVLETS & JSP

B.Basham, K.Sierra, B.Bates

O'Reilly

Wyłącznie jako materiał porównawczy w ramach zajęć.



Rozwiązania ćwiczeń

Zadanie	Serwer WWW	Kontener	Serwlet
Tworzenie obiektów żądania i odpowiedzi		Bezpośrednio przed uruchomieniem wątku.	
Wywoływanie metody <code>service()</code>		Metoda <code>service()</code> wywołuje metodę <code>doGet()</code> lub <code>doPost()</code> .	
Uruchamianie nowych wątków dla obsługi przychodzących żądań		Uruchamia wątek serwletu.	
Konwersja obiektu odpowiedzi na postać odpowiedzi protokołu HTTP	Otrzymuje odpowiedź od kontenera.		
Znajomość protokołu HTTP	Wykorzystuje protokół HTTP do komunikacji z przeglądarką klienta.		
Dodawanie kodu HTML do obiektu odpowiedzi			Przeznaczona dla klienta dynamiczna treść strony.
Utrzymywanie referencji do obiektów odpowiedzi		Kontener przekazuje referencję do serwletu.	Wykorzystuje referencję do przekazywania odpowiedzi.
Odnajdywanie adresów URL w deskrytorze rozmieszczenia		W ten sposób odnajduje właściwy serwlet dla otrzymanego żądania.	
Usuwanie obiektów żądania i odpowiedzi		Po zakończeniu pracy przez serwlet.	
Koordynowanie tworzenia dynamicznej zawartości stron	Wie, jak przekazywać odpowiednie dane do kontenera.	Wie, który serwlet należy wywołać.	
Zarządzanie cyklami życia		Wywołuje metodę <code>service()</code> (i, jak się przekonamy, nie tylko).	
Posiadanie nazwy odpowiadającej elementowi <code><servlet-class></code> z deskryptora rozmieszczenia			<code>public class</code> Cokolwiek

Rozwiązania ćwiczeń, kontynuacja...

Serwlet

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

```
public class Ch2Dice extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request,
```

```
                        HttpServletResponse response)
```

```
        throws IOException {
```

```
        PrintWriter out = response.getWriter();
```

```
        String d1 = Integer.toString((int)((Math.random()*6)+1));  
        String d2 = Integer.toString((int)((Math.random()*6)+1));  
        out.println("<html> <body>" +  
                    "<h1 align=center>Serwlet rzucający kostkami do  
gry: Ch2Dice</h1>" +  
                    "<p>Wyrzucono liczby" + d1 + "i" + d2 +  
                    "</body> </html>");  
    }  
}
```

Deskryptor wdrożenia

```
<web-app ...>
```

```
    <servlet>
```

```
        <servlet-name>
```

```
        C2dice </servlet-name>
```

```
        <servlet-class>
```

```
        Ch2Dice
```

```
        </servlet-class>
```

```
    </servlet>
```

```
    <servlet-mapping>
```

```
        <servlet-name>
```

```
        C2dice
```

```
        </servlet-name>
```

```
        <url-pattern>
```

```
        /Dice
```

```
        </url-pattern>
```

```
    </servlet-mapping>
```

```
</web-app>
```

„Działający” deskryptor rozmieszczenia (DD)

Na razie nie musisz się martwić o rzeczywiste znaczenia poszczególnych fragmentów deskryptorów rozmieszczenia (w *dalszych* rozdziałach zdobędziesz odpowiednią wiedzę i zostaniesz z tej wiedzy rozliczony). W tym miejscu chcemy Ci jedynie przedstawić deskryptor rozmieszczenia *web.xml*, który faktycznie *działa*. W pozostałych przykładach prezentowanych w tym rozdziale pomijaliśmy znaczne fragmenty otwierającego znacznika `<web-app>` (teraz już widzisz, dlaczego jego stosowanie nie miało większego sensu).

Sposób, w jaki często przedstawiamy deskryptory rozmieszczenia w tej książce:

```
<web-app ...> ← Użyty tutaj otwierający znacznik
                  <web-app> jest niekompletny.

  <servlet>
    <servlet-name>Ch3 Piwo</servlet-name>
    <servlet-class>com.example.web.WyborPiwa</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch3 Piwo</servlet-name>
    <url-pattern>/WybierzPiwo.do</url-pattern>
  </servlet-mapping>

</web-app>
```

Nigdy NIE musisz zapamiętywać żadnego z wykorzystywanych w tym miejscu znaczników otwierających. Jeśli używasz kontenera, który jest zgodny ze specyfikacją servletów w wersji 2.4 (tak jest np. w przypadku serwera Tomcat 5), wystarczy te znaczniki kopiować i wklejać do deskryptorów.

RZECZYWISTA postać deskryptora rozmieszczenia:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Ch3 Piwo</servlet-name>
    <servlet-class>com.example.web.WyborPiwa</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Ch3 Piwo</servlet-name>
    <url-pattern>/WybierzPiwo.do</url-pattern>
  </servlet-mapping>

</web-app>
```


Jaka w tym wszystkim jest rola platformy J2EE?

Java 2 Enterprise Edition (w skrócie J2EE) jest pewnego rodzaju superspecyfikacją, ponieważ obejmuje wiele innych specyfikacji, włącznie ze specyfikacją serwletów w wersji 2.4 oraz specyfikacją stron JSP w wersji 2.0 (obie dotyczą kontenera WWW). Warto jednak pamiętać, że platforma J2EE 1.4 obejmuje także specyfikację komponentów Enterprise JavaBeans (w skrócie EJB) w wersji 2.1, która dotyczy z kolei kontenera EJB. Innymi słowy, kontener WWW ma na celu obsługę komponentów *WWW* (serwletów i stron JSP), natomiast zadaniem kontenera EJB jest obsługa komponentów *biznesowych*.

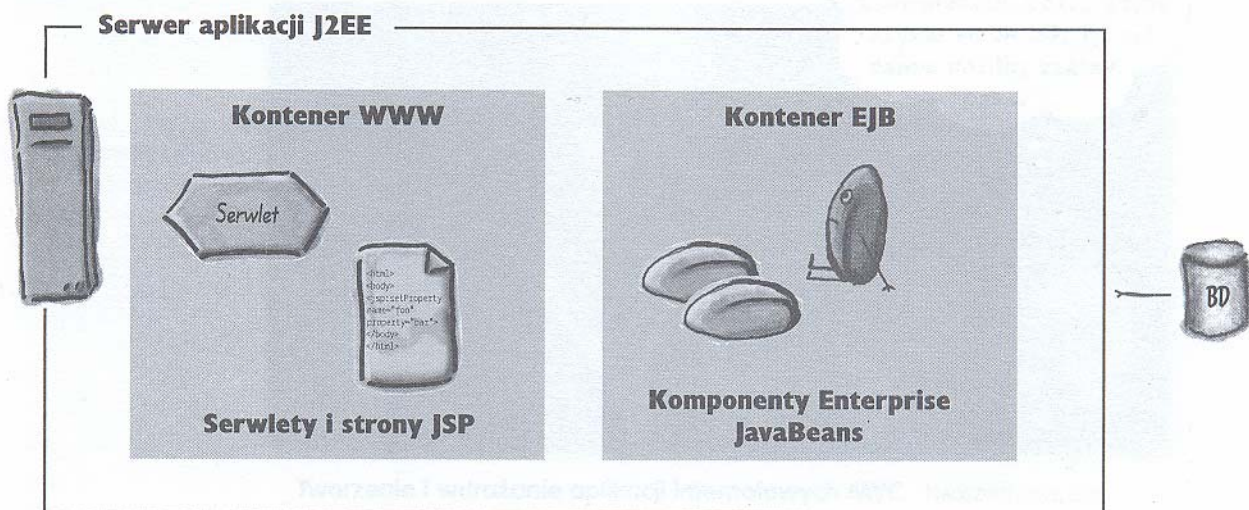
Serwer aplikacji w pełni zgodny ze specyfikacją J2EE musi zawierać zarówno kontener WWW, jak i kontener EJB (a także inne komponenty, takie jak implementacja standardów JNDI i JMS). Warto pamiętać, że np. Tomcat jest jedynie kontenerem WWW! Jego zgodność ze specyfikacją J2EE dotyczy wyłącznie elementów składających się na kontener WWW.

Tomcat jest kontenerem WWW, a nie kompletnym serwerem aplikacji J2EE, ponieważ nie oferuje funkcjonalności kontenera EJB.

Serwer aplikacji J2EE zawiera zarówno kontener WWW, JAK I kontener EJB.

Tomcat jest co prawda kontenerem WWW, ale NIE jest kompletnym serwerem aplikacji J2EE.

Specyfikacja J2EE 1.4 obejmuje specyfikację serwletów w wersji 2.4, specyfikację stron JSP w wersji 2.0 oraz specyfikację technologii EJB w wersji 2.1.



P: Czy fakt, iż Tomcat jest samodzielnym kontenerem WWW, oznacza, że istnieją samodzielne (autonomiczne) kontenery EJB?

O: W dawnych czasach (powiedzmy, że w roku 2000) istniały kompletne serwery aplikacji J2EE, samodzielne kontenery WWW oraz samodzielne kontenery EJB. Obecnie jednak niemal wszystkie kontenery EJB stanowią elementy składowe pełnych serwerów J2EE, choć istnieje jeszcze kilka samodzielnych kontenerów WWW (np. Tomcat i Resin).

Samodzielne kontenery WWW są zwykle konfigurowane w taki sposób, aby możliwa była ich współpraca z serwerami WWW

wykorzystującymi protokół HTTP (np. z serwerem Apache), chociaż np. Tomcat *sam* oferuje możliwość funkcjonowania w roli prostego serwera HTTP. Warto jednak pamiętać, że funkcjonalność Tomcata w tym zakresie daleka jest od możliwości Apache'a, zatem większość aplikacji internetowych, w których nie zastosowano komponentów EJB, wykorzystuje odpowiednio skonfigurowane Apache'a i Tomcata — gdzie Apache pełni rolę *serwera WWW*, natomiast Tomcat występuje w roli *kontenera*.

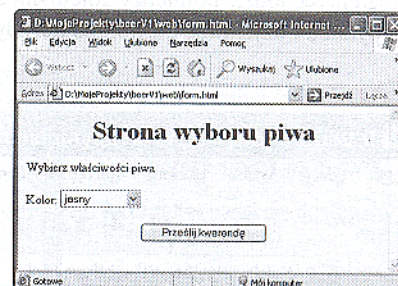
Do najbardziej popularnych serwerów J2EE należą: WebLogic firmy BEA, JBoss AS z otwartym dostępem do kodu źródłowego oraz WebSphere firmy IBM.

Zbudujemy prawdziwą (małą) aplikację internetową

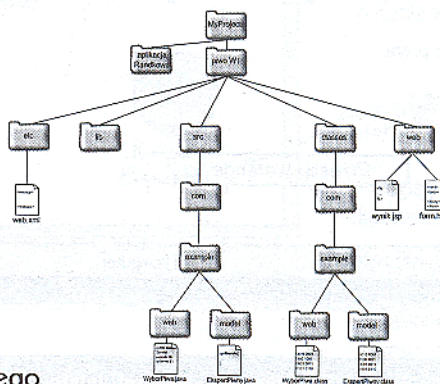
Przeanalizowaliśmy już rolę kontenera, wspominaliśmy o deskryptorach rozmieszczenia oraz rzuciliśmy okiem na architekturę MVC. Cały dotychczasowy materiał miał jednak charakter rozważań teoretycznych — trudno sobie jednak wyobrazić, abyś usiadł i przez cały dzień tylko *czytał* o aplikacjach internetowych, nadszedł czas, abyś wreszcie coś *zrobił*.

Cztery kroki, które będziemy musieli wykonać:

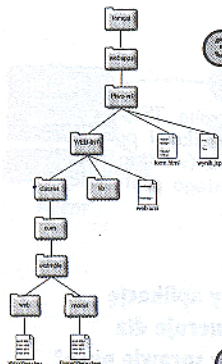
- ❶ Dokonamy przeglądu widoków (perspektyw) użytkownika (czyli tego, co jest wyświetlane w oknie przeglądarki) oraz architektury wysokiego poziomu.



- ② Stworzymy środowisko wytwarzania aplikacji, które będziemy wykorzystywali dla tego projektu (i którego będziesz mógł używać faktycznie w pozostałych przykładach omawianych w książce).



- 3 Stworzymy dla tego projektu środowisko wdrożenia (z którego także będziesz mógł korzystać podczas sprawdzania innych przykładów z książki).



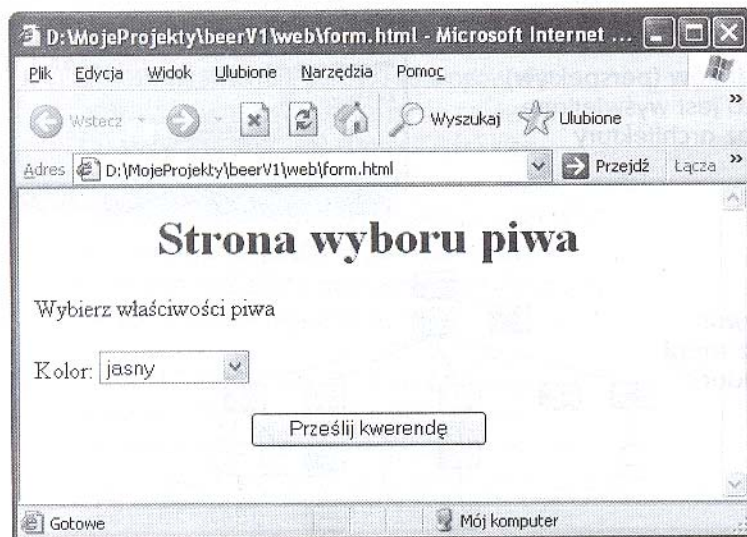
- ④ Przeprowadzimy iteracyjny proces wytwarzania i testowania zróżnicowanych komponentów składających się na naszą aplikację internetową (to fakt, ten punkt należy potraktować bardziej jak strategię niż tylko pojedynczy krok).

Uwaga: Zawsze zalecamy stosowanie strategii iteracyjnego wytwarzania i testowania aplikacji, choć nie wszędzie w tej książce będziemy analizowali *wszystkie* kroki tego procesu.

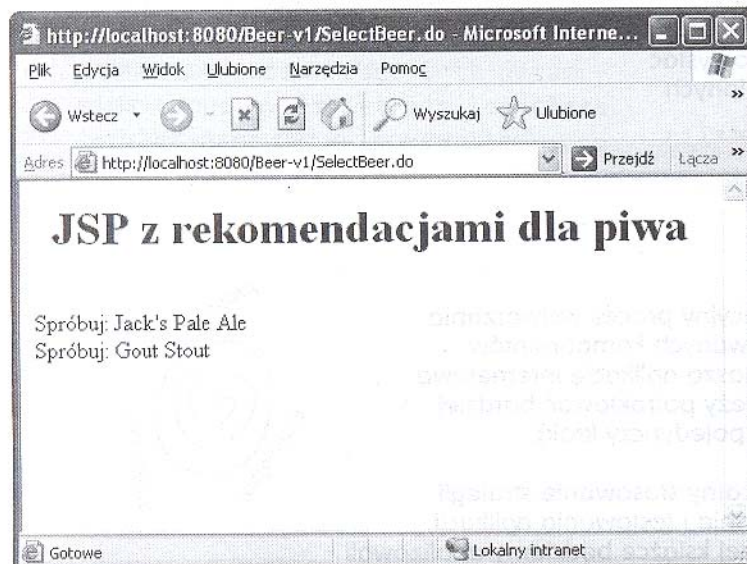


Widok użytkownika aplikacji internetowej — Doradca piwny

Naszą aplikację internetową nazwiemy *Doradca piwny*. Użytkownicy będą mogli przechodzić pomiędzy kolejnymi stronami naszej aplikacji, odpowiadać na pytania i uzyskiwać doskonałe porady na temat piwa.



Ta strona zostanie napisana w języku HTML i będzie generowała żądanie POST protokołu HTTP z określonym przez użytkownika kolorem piwa.



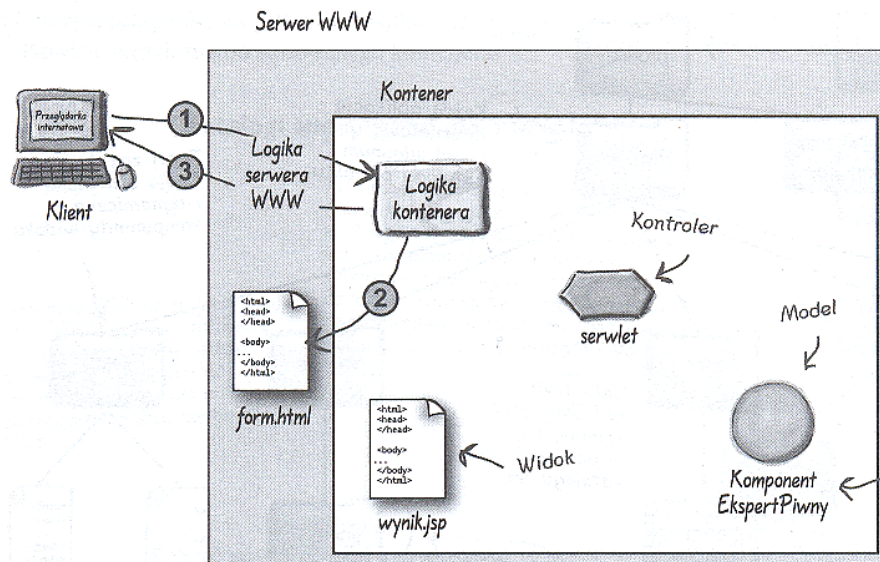
Ta strona będzie stroną JSP z udzieloną użytkownikowi poradą odnośnie marki piwa (według określonych przez niego preferencji w zakresie koloru).

P: Dlaczego piszemy aplikację internetową, która generuje dla użytkownika porady w sprawie piwa?

O: Po przeprowadzeniu wyczerpujących badań rynku, stwierdziliśmy, że 90% czytelników naszych książek ceni sobie smak piwa. W przypadku pozostałych 10% słowo „piwo” można po prostu zastąpić słowem „kawa”.

Oto nasza architektura

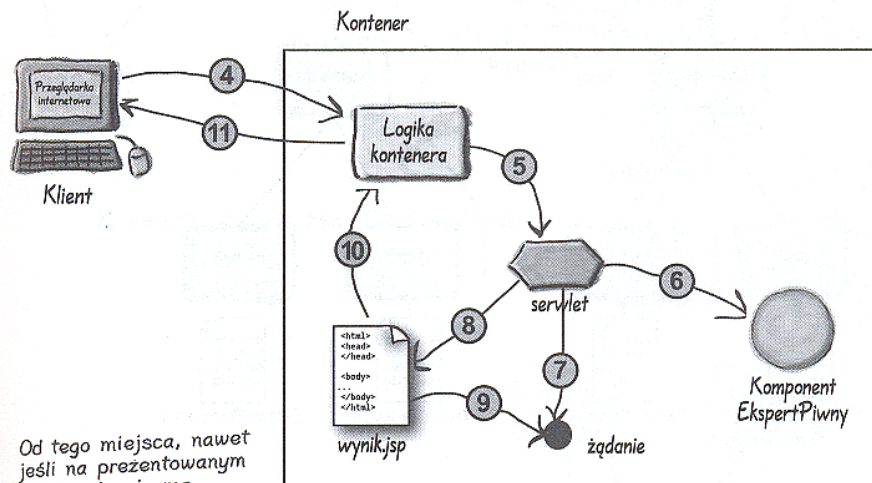
Chociaż prezentowana aplikacja jest bardzo skromna, zbudujemy ją zgodnie z prostą architekturą MVC. Dzięki temu, kiedy nasza aplikacja stanie się NAJPOPULARNIEJSZĄ witryną w internecie, będziemy mogli bez trudu rozszerzyć jej funkcjonalność.



1. Klient tworzy żądanie dotyczące strony `form.html`.
2. Kontener uzyskuje dostęp do strony `form.html`.
3. Kontener zwraca stronę do przeglądarki internetowej, gdzie użytkownik odpowiada na pytania zawarte w formularzu.

Zwykły, tradycyjny obiekt Javy

4. Przeglądarka przesyła do kontenera dane żądania.
5. Kontener odnajduje właściwy serwlet na podstawie określonego przez klienta adresu URL i przekazuje do tego serwletu otrzymane żądanie.

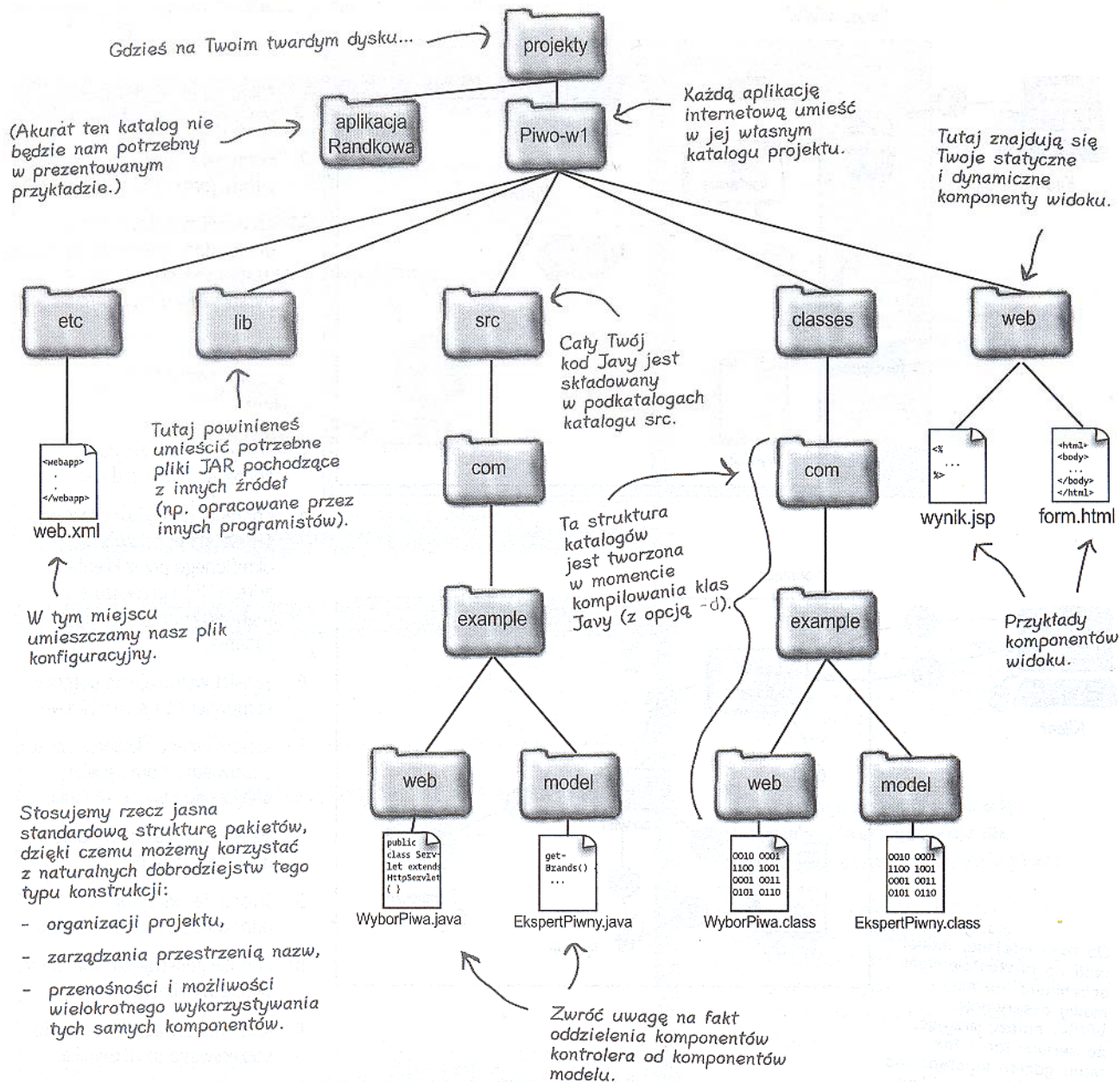


6. Serwlet wywołuje do pomocy komponent EkspertPiwny.
7. Klasa piwnego eksperta zwraca odpowiedź, którą serwlet dołącza do obiektu żądania.
8. Serwlet przekazuje żądanie dalej do strony JSP.
9. Strona JSP odczytuje odpowiedź z żądania obiektu.
10. Kod JSP generuje dla kontenera stronę wynikową.
11. Kontener zwraca tę stronę do szczęśliwego użytkownika.

Od tego miejsca, nawet jeśli na prezentowanym schemacie nie ma mowy o serwerze WWW, należy przyjąć, że serwer ten i tak musi gdzieś występować w architekturze aplikacji internetowej.

Tworzenie środowiska wytwarzania aplikacji

Istnieje wiele sposobów organizowania struktury katalogów stanowiącej środowisko wytwarzania naszej aplikacji internetowej — poniżej przedstawiliśmy naszą propozycję tego typu struktury dla małych i średnich projektów. Kiedy nadejdzie czas wdrażania naszej aplikacji internetowej, po prostu skopiujemy część tej struktury w miejsce wyznaczone do tego celu przez nasz kontener (w tym minipodręczniku wykorzystujemy kontener Tomcat w wersji 5.).



Tworzenie środowiska wdrażania aplikacji

Z wdrażaniem aplikacji internetowej wiąże się szereg charakterystycznych dla konkretnego kontenera reguł oraz wymagań zdefiniowanych w specyfikacjach serwetów i stron JSP (jeśli nie używasz Tomcata, będziesz musiał sprawdzić, w jaki sposób należy powiązać *Twój* kontener z Twoją aplikacją internetową). W naszym przypadku wszystko, co znajduje się poniżej katalogu *Piwo-w1* jest całkowicie niezależne od stosowanego kontenera!

Katalogi specyficzne dla Tomcata

Wszystko POD tą kropkowaną linią STANOWI już aplikację internetową, zatem będzie miało taką samą postać niezależnie od producenta wykorzystywanego kontenera WWW.

Ta nazwa katalogu reprezentuje także katalog główny kontekstu, który jest wykorzystywany przez Tomcata podczas odnajdywania zasobów wskazywanych przez adresy URL. Bardziej szczegółowo zajmiemy się tym zagadnieniem w rozdziale poświęconym wdrażaniu aplikacji internetowych.

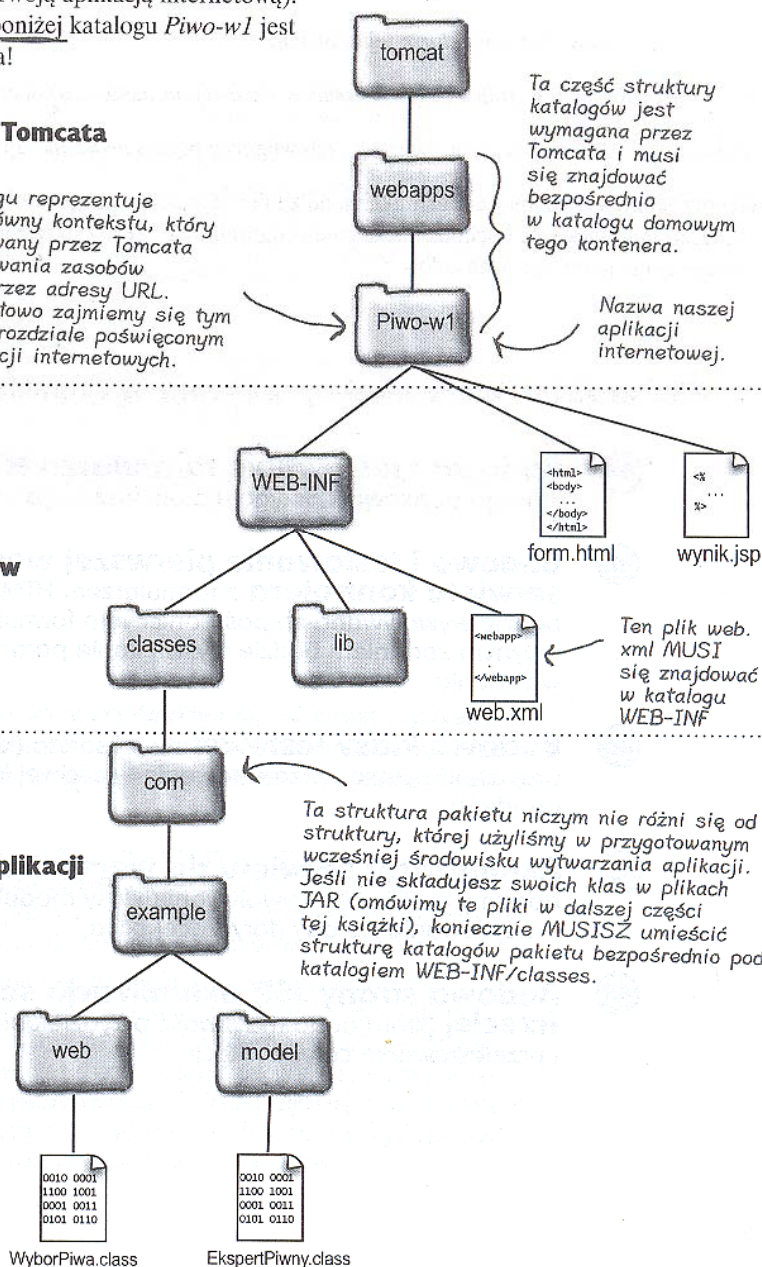
To jest katalog domowy Tomcata; jego nazwa może być w ogólności zupełnie inna, np. jakarta-tomcat-5.0.19.

Ta część struktury katalogów jest wymagana przez Tomcata i musi się znajdować bezpośrednio w katalogu domowym tego kontenera.

Nazwa naszej aplikacji internetowej.

Część specyfikacji serwetów

Katalogi specyficzne dla danej aplikacji



Nasz plan procesu budowy aplikacji

Na początku tego rozdziału przedstawiliśmy w skrócie czteroetapowy proces wytwarzania naszej aplikacji internetowej. Do tej pory zrealizowaliśmy następujące zadania:

1. Przeanalizowaliśmy *widoki* użytkownika dla naszej aplikacji internetowej.
2. Przyjrzelśmy się *architekturze* tej aplikacji.
3. Przygotowaliśmy środowiska *wytwarzania* i *wdrażania* naszej nowej aplikacji.

Nadszedł czas, by omówić krok czwarty, czyli właściwy proces *tworzenia* aplikacji internetowej.

Wykorzystamy w tym miejscu techniki znane z kilku popularnych metodologii wytwarzania oprogramowania (część pochodzi z extreme programming, część z iteracyjnego wytwarzania aplikacji) i wymieszamy je do własnych celów...

Pięć kroków, które musimy wykonać w ramach kroku 4.:

- 4a **Budowa i testowanie formularza HTML**, za pomocą którego użytkownik przygotowuje pierwsze żądanie.
- 4b **Budowa i testowanie pierwszej wersji testowej serwletu kontrolera** z formularzem HTML. Ta wersja będzie wywoływana za pośrednictwem formularza HTML, a jej jedynym zadaniem będzie wyświetlanie parametrów otrzymanych w żądaniu.
- 4c **Budowa klasy testowej** dla eksperta (klasy modelu) oraz zbudowanie i przetestowanie właściwej klasy dla eksperta (modelu).
- 4d **Aktualizacja serwletu do wersji drugiej**. W tej wersji dodamy możliwość wywoływania klasy modelu i — tym samym — uzyskiwania porady dotyczącej piwa.
- 4e **Budowa strony JSP, aktualizacja serwletu do wersji trzeciej** (która doda możliwość przydzielania stron JSP) i przetestowanie całej aplikacji.

Kod HTML dla początkowej strony formularza

Nasz pierwszy kod HTML jest prosty — wyświetlamy tekst nagłówka, listę rozwijaną, za pomocą której użytkownik może wybrać odpowiedni kolor piwa, oraz przycisk akceptacji formularza.

```
<html><body>
<h1 align="center">Strona wyboru piwa</h1>
<form method="POST"
  action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option>jasny
    <option>bursztynowy
    <option>brązowy
    <option>ciemny
  </select>
  <br><br>
  <center>
    <input type="SUBMIT">
  </center>
</form></body></html>
```

Dlaczego wybraliśmy metodę POST zamiast metody GET?

To jest nazwa, którą wykorzystujemy w kodzie HTML do wywołania odpowiedniego serwletu. W naszej strukturze katalogów nie ma nic, co miałyby nazwę WybierzPiwo.do! Jest to jedynie nazwa logiczna...

W ten sposób tworzymy listę rozwijaną (opcje w Twoim przykładzie mogą się oczywiście różnić od tych, które tutaj przedstawiliśmy) (Czy zwróciłeś uwagę na atrybut size="1"?)

P: Dlaczego nasz formularz odwołuje się do serwletu **WybierzPiwo.do**, skoro nasza aplikacja **NIE** zawiera serwletu z taką nazwą? W analizowanych przed chwilą strukturach katalogów nie widziałem pliku nazwanego **WybierzPiwo.do**. Nie wiem nawet, co oznacza samo rozszerzenie **.do**.

O: **WybierzPiwo.do** jest jedynie nazwą logiczną, która nie ma nic wspólnego z rzeczywistą nazwą pliku. Użyliśmy tej nazwy, ponieważ chcemy, aby była ona znana dla klientów naszej aplikacji! W praktyce klient **NIGDY** nie powinien mieć bezpośredniego dostępu do pliku klasy serwletu, zatem nie należy np. tworzyć stron HTML z łączami lub akcjami zawierającymi rzeczywiste ścieżki do odpowiednich plików naszej aplikacji. W powyższym przykładzie zastosowaliśmy sztuczkę polegającą na użyciu rozmieszczenia wdrożenia w formacie XML (wspominanego już pliku *web.xml*) do odwzorowania żądań klientów (**WybierzPiwo.do**) na faktyczną nazwę klasy serwletu, która będzie wykorzystywana za każdym razem, gdy kontener otrzyma żądanie dotyczące nieistniejącego serwletu **WybierzPiwo.do**. Na razie możesz traktować rozszerzenie **.do** po prostu jak część logicznej nazwy serwletu (nie jak *rzeczywisty* typ pliku). W dalszej części tej książki dowiesz się nieco więcej na temat innych sposobów wykorzystywania rozszerzeń (zarówno tych rzeczywistych, jak i tych tworzonych na potrzeby klientów, czyli logicznych) w odwzorowaniach swoich serwletów.

Wdrażanie i testowanie strony otwierającej

Aby przetestować pierwszą stronę HTML naszej aplikacji internetowej, musimy ją umieścić w strukturze katalogów kontenera WWW (w naszym przypadku będzie to Tomcat), uruchomić ten kontener oraz wyświetlić tę stronę w oknie przeglądarki internetowej.

1 Stwórz stronę HTML w swoim środowisku wytwarzania aplikacji

Stwórz nowy plik HTML, nazwij go *form.html* i zapisz w katalogu *piwoWI\web* przygotowanego wcześniej środowiska wytwarzania.

2 Skopiuj nowy plik do środowiska wdrażania aplikacji

Umieść kopię pliku *form.html* w katalogu *tomcat\webapps\Piwo-w1* (pamiętaj, że katalog domowy Twojego kontenera Tomcat może mieć zupełnie inną nazwę).

3 Uruchom Tomcata

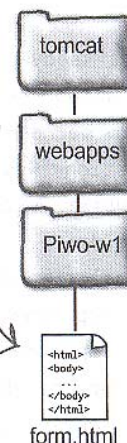
We wszystkich rozdziałach tej książki będziemy wykorzystywali Tomcata zarówno w roli serwera WWW, jak i w roli kontenera WWW. W praktyce najczęściej stosuje się bardziej wyszukany serwer WWW (np. Apache) skonfigurowany w sposób umożliwiający współpracę z jednym z istniejących kontenerów WWW (np. z Tomcatem). Tomcat oferuje jednak funkcjonalność przyzwoitego (choć skromnego) serwera WWW, który w zupełności wystarcza do analizy przykładów z tej książki. Aby uruchomić Tomcata, przejdź do jego katalogu domowego i uruchom polecenie *bin/startup.sh*.

4 Przetestuj nową stronę

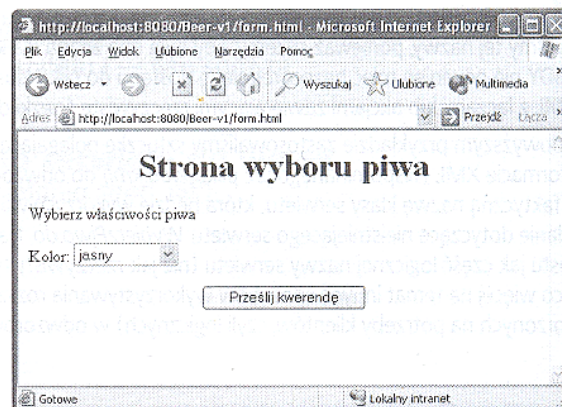
Otwórz okno przeglądarki internetowej i wpisz adres:

<http://localhost:8080/Piwo-w1/form.html>

Na ekranie powinna zostać wyświetlona strona podobna do tej na widocznym obok zrzucie ekranu.



```
Wiersz polecenia
> cd jakarta-tomcat-5.0.27
> bin\startup.bat
```



Tworzenie deskryptora rozmieszczenia (DD)

Głównym zadaniem niniejszego deskryptora rozmieszczenia jest zdefiniowanie odwzorowania pomiędzy wykorzystywaną przez klienta nazwą logiczną dla żądań (w tym przypadku *WybierzPiwo.do*) oraz rzeczywistą nazwą pliku klasy serwletu (w tym przypadku *com.example.web.WyborPiwa*).

1 Stwórz w swoim środowisku wytwarzania aplikacji deskryptor rozmieszczenia

Opracuj nowy dokument XML, nadaj mu nazwę *web.xml* i zapisz go w katalogu *piwoW1\etc* swojego środowiska wytwarzania.

Nie musisz wiedzieć, co znaczą te tajemnicze zapisy; po prostu wklep je w swoim pliku XML.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
```

```
<servlet>
  <servlet-name>R3 Piwo</servlet-name>
  <servlet-class>com.example.web.WyborPiwa</servlet-class>
</servlet>
```

Jest to przykład sztucznej nazwy, która będzie wykorzystywana WYŁĄCZNIE w innych fragmentach tego samego deskryptora wdrożenia.

W pełni kwalifikowana nazwa pliku klasy serwletu.

```
<servlet-mapping>
  <servlet-name>R3 Piwo</servlet-name>
  <url-pattern>/WybierzPiwo.do</url-pattern>
</servlet-mapping>
```

```
</web-app>
```

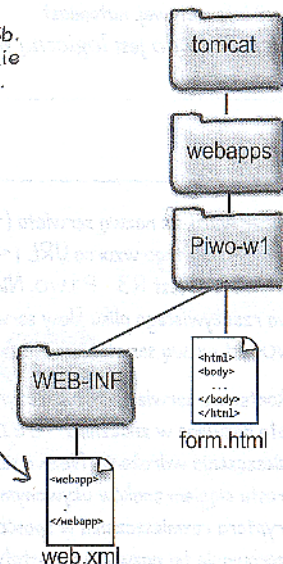
Nie zapomnij o ukośniku na początku.

Chcemy, aby klient odwoływał się do naszego serwletu właśnie w ten sposób. Rozszerzenie *.do* zastosowano wyłącznie dla zachowania zgodności z konwencją.

2 Skopiuj nowy plik do środowiska wdrażania aplikacji

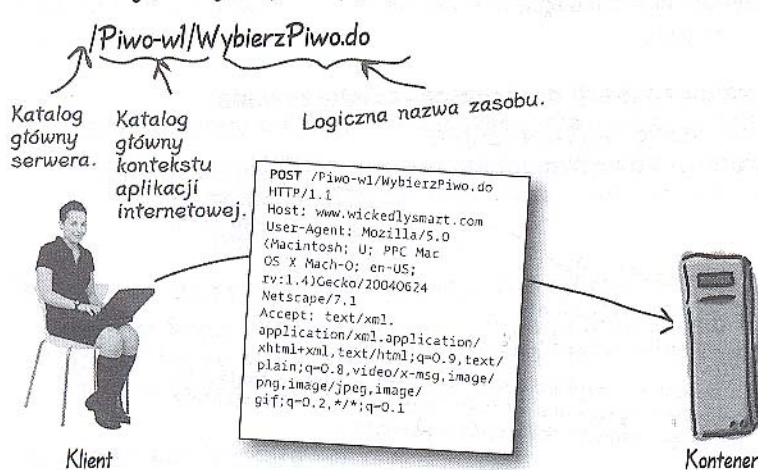
Umieść kopię pliku *web.xml* w katalogu *tomcat\webapps\piwo-w1\WEB-INF*.

Plik *web.xml* MUSI znaleźć się w odpowiednim miejscu; w przeciwnym przypadku kontener tego pliku nie znajdzie i nic nie zadziała, a Ty popadniesz w głęboką depresję.



Odwzorowywanie nazw logicznych w pliki klas serwletów

- ① Dłanie wypełnia formularz i klika przycisk jego akceptacji. Przeglądarka generuje następujący adres URL żądania:



W naszym kodzie HTML katalog /Piwo-w1/ nie był częścią ścieżki do serwletu. Użyliśmy tam jedynie odwołania w postaci:

```
<form method="POST"
action="WybiezPiwo.do">
```

Okazuje się jednak, że przeglądarka sama dodaje katalog /Piwo-w1/ na początek żądania, ponieważ właśnie stamtąd pochodzi żądanie klienta. Innymi słowy, użyte w kodzie HTML odwołanie WybiezPiwo.do jest nazwą względną w stosunku do adresu URL bieżącej strony. W tym przypadku jest to ścieżka interpretowana względem katalogu głównego naszej aplikacji internetowej, czyli właśnie /Piwo-w1/.

- ② Kontener przeszukuje deskryptor wdrożenia i odnajduje element `<servlet-mapping>` z podelementem `<url-pattern>`, który pasuje do użytej w żądaniu nazwy /WybiezPiwo.do (gdzie ukośnik reprezentuje katalog główny kontekstu aplikacji internetowej, natomiast WybiezPiwo.do jest logiczną nazwą żadanego zasobu).



Kontener

- ③ Kontener widzi, że nazwą serwletu (`<servlet-name>`) dla tego wzorca URL (`<url-pattern>`) jest R3 Piwo. Nie jest to jednak nazwa rzeczywistego pliku klasy serwletu. R3 Piwo jest nazwą serwletu, nie nazwą klasy!

Dla kontenera serwlet jest tylko czymś, co zostało nazwane w znaczniku `<servlet>` rozmieszczenia wdrożenia. Nazwa serwletu jest po prostu ciągiem znaków używanym w ramach deskryptora rozmieszczenia w sposób umożliwiający odwzorowanie tej nazwy w pozostałych częściach tego samego deskryptora.



Kontener

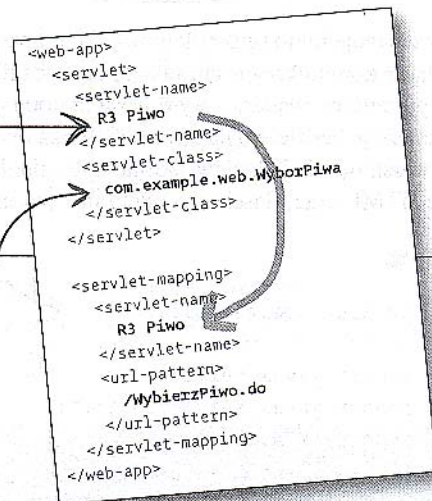
```
<web-app>
<servlet>
<servlet-name>
R3 Piwo
</servlet-name>
<servlet-class>
com.example.web.WyborPiwa
</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>
R3 Piwo
</servlet-name>
<url-pattern>
/WybiezPiwo.do
</url-pattern>
</servlet-mapping>
</web-app>
```

- ④ Kontener zagląda do znacznika `<servlet>`, aby znaleźć zapis R3 Piwo w którymś z elementów `<servlet-name>`.



Kontener

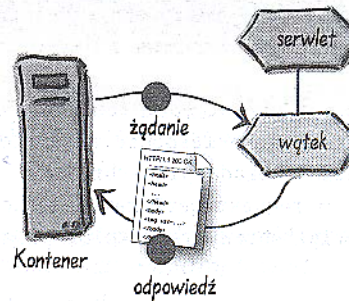


- ⑤ Kontener wykorzystuje podznacznik `<servlet-class>` znacznika `<servlet>` do określania, która klasa serwletu odpowiada za obsługę danego żądania. Jeśli odpowiedni serwet nie został jeszcze zainicjalizowany, klasa jest wczytywana, a sam serwet jest inicjalizowany.



Kontener

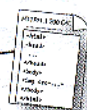
- ⑥ Kontener uruchamia nowy wątek, który obsłuży otrzymane żądanie, po czym przekaże je do tego wątku (do metody `service()` tego serwletu).



- ⑦ Kontener odsyła do klienta odpowiedź (oczywiście za pośrednictwem serwera WWW).



Klient



Kontener



serwet

Pierwsza wersja serwletu kontrolera

Nasz plan przewiduje budowę serwletu w kilku etapach, które będą obejmowały testowanie na bieżąco rozmaitych łączy komunikacyjnych. Jak zapewne pamiętasz, w swojej ostatecznej wersji serwlet będzie przyjmował parametry z żądania, wywoływał metodę należącą do modelu, zapisywał informacje w miejscu, w którym znaleźć je będzie mogła strona JSP oraz przekazywał żądanie do strony JSP. Naszym celem w pierwszej wersji będzie jednak wyłącznie upewnienie się co do możliwości prawidłowego wywołania serwletu przez stronę HTML oraz właściwego odebrania parametrów przekazanych z tej strony HTML.

Kod serwletu

```
package com.example.web;

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

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Porada piwna<br>");
        String c = request.getParameter("kolor");
        out.println("<br>Wybrany kolor piwa:" + c);
    }
}
```

Użyjemy metody `doPost` do obsługi żądania HTTP, ponieważ w formularzu HTML użyliśmy atrybutu: `method=POST`.

Upewnij się, że zadeklarowany pakiet odpowiada stworzonym wcześniej strukturom wytwarzania i wdrażania aplikacji.

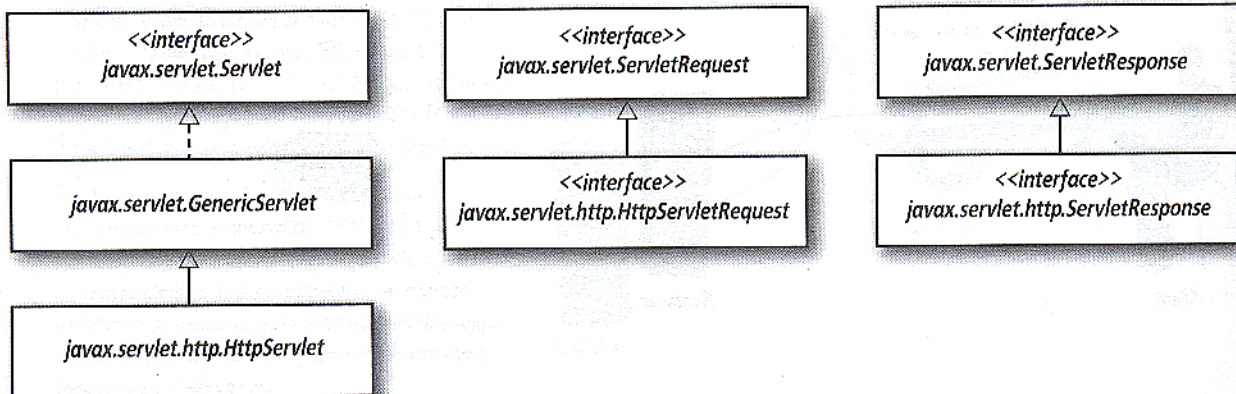
Klasa `HttpServlet` dziedziczy z klasy `GenericServlet`, która z kolei implementuje interfejs `Servlet`.

Ta metoda pochodzi z implementowanego interfejsu `ServletResponse`.

Ta metoda pochodzi z implementowanego interfejsu `ServletRequest`. Zwróć uwagę na fakt, iż argument tej metody odpowiada użytej w kodzie HTML wartości atrybutu `name` znacznika `<select>`.

W tym miejscu odsyłamy poradę do klienta (wyświetlamy prostą informację tekstową).

Kluczowe interfejsy API



Kompilowanie, wdrażanie i testowanie serwletu kontrolera

No dobrze, zbudowaliśmy, wdrożyliśmy i przetestowaliśmy naszą stronę HTML oraz zbudowaliśmy i wdrożyliśmy nasz deskryptor rozmieszczenia (po prostu umieściliśmy plik *web.xml* w środowisku wdrażania, ale z technicznego punktu widzenia deskryptor ten nie będzie w pełni wdrożony aż do chwili ponownego uruchomienia Tomcata). Nadszedł czas skompilowania, wdrożenia i przetestowania (za pośrednictwem już istniejącego formularza HTML) pierwszej wersji naszego serwletu. Uruchomimy teraz kontener Tomcat, aby upewnić się, że „widzi” on zarówno deskryptor *web.xml*, jak i klasę serwletu.

Kompilowanie serwletu

Skompiluj serwlet z flagą *-d*, aby umieścić gotową klasę w środowisku wdrażania.

Zmodyfikuj ten fragment w taki sposób, aby odpowiadał strukturze katalogów istniejącej w Twoim systemie! Ścieżka za katalogiem *tomcat/* powinna pozostać niezmienną.

```
Wiersz polecenia
> cd projekty\piwo-w1
> javac -classpath d:\java\jakarta-tomcat-5.0.27\common\lib\servlet-api.jar:classes:.
-d src\com\example\web\WyborPiwa.java
```

Użyj opcji *-d*, aby wymusić na kompilatorze umieszczenie pliku klasy w katalogu klas w ramach prawidłowej struktury pakietów. Twój plik klasy (z rozszerzeniem *.class*) powinien się znaleźć w katalogu *piwoW1\classes\com\example\web*.

Wdrażanie serwletu

Aby wdrożyć serwlet, stwórz kopię wygenerowanego w poprzednim kroku pliku *class* i przenieś ją do katalogu *Piwo-w1\WEB-INF\classes\com\example\web* w przygotowanej wcześniej strukturze wdrożenia.

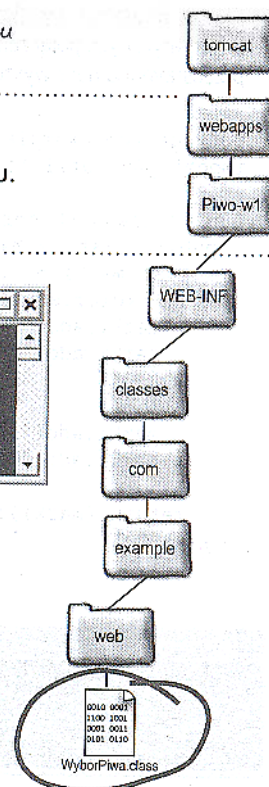
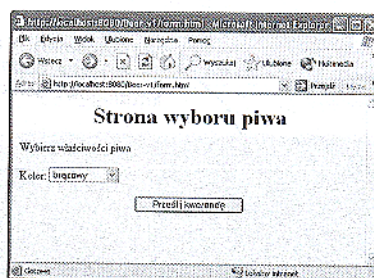
Testowanie serwletu

1. Uruchom ponownie Tomcata!
2. Uruchom swoją przeglądarkę i przejdź na stronę *http://localhost:8080/Piwo-w1/form.html*.
3. Wybierz kolor piwa i kliknij przycisk *Prześlij kwerendę*.
4. Jeśli Twój serwlet działa, powinieneś zobaczyć w oknie przeglądarki odpowiedź serwletu w następującej postaci:

Porada piwna

Wybrany kolor piwa: brązowy

```
Wiersz polecenia
> cd jakarta-tomcat-5.0.27
> bin\shutdown.bat
> bin\startup.bat
```



Budowa i testowanie klasy modelu

We wzorcu projektowym MVC model jest traktowany jako „zaplecze” aplikacji. Model często ma postać istniejącego od dawna systemu informatycznego, który dopiero teraz jest udostępniany za pośrednictwem stron WWW. W większości przypadków jest to tradycyjny kod Javy, który został opracowany bez żadnej wiedzy o możliwości jego wywoływania z poziomu serwetów. Model nigdy nie powinien być ściśle wiązany z pojedynczą aplikacją internetową, zatem jego kod należy umieszczać w jego własnych pakietach.

Specyfikacja modelu

- kod modelu powinien należeć do pakietu `com.example.model`,
- struktura katalogów powinna być składowana w katalogu `WEB-INF\classes\com\example\model`,
- model powinien udostępniać pojedynczą metodę `getMarki()`, która pobiera preferowany kolor piwa (w postaci łańcucha) i która zwraca obiekt klasy `ArrayList` z rekomendowanymi markami piwa (także w postaci łańcuchów).

Budowa i testowanie klasy dla danego modelu

Opracuj klasę testową dla danego modelu (tak, klasę testową należy przygotować *przed* zbudowaniem samego modelu). Teraz wszystko zależy tylko od Ciebie — nie musisz dalej realizować zaleceń tego skróconego podręcznika. Pamiętaj, że kiedy będziesz po raz pierwszy testował swój model, nadal będzie on pozostawał w środowisku wytwarzania — tak jak każdą inną klasę Javy, można ten model przetestować bez Tomcata.

Budowa i testowanie samego modelu

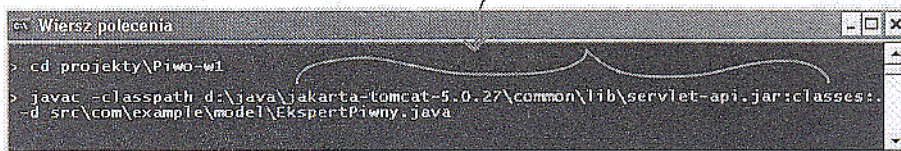
Modele mogą być wyjątkowo skomplikowane. Często zawierają odziedziczone po istniejących systemach informatycznych połączenia z bazami danych i odwołania do skomplikowanej logiki biznesowej. Poniżej przedstawiono nasz wyszukany i oparty na zaawansowanych regułach system ekspercki w zakresie porad piwnych:

```
package com.example.model;
import java.util.*;

public class EkspertPiwny {
    public List getMarki(String kolor) {
        List marki = new ArrayList();
        if (kolor.equals("bursztynowy")) {
            marki.add("Jack Amber");
            marki.add("Red Moose");
        }
        else {
            marki.add("Jail Pale Ale");
            marki.add("Gout Stout");
        }
        return (marki);
    }
}
```

Zwróć uwagę na sposób, w jaki wyraziliśmy skomplikowaną wiedzę ekspercką na temat piwa za pomocą zaawansowanych wyrażeni warunkowych.

Nie zapomnij o zmianie tego fragmentu na ścieżkę odpowiadającą TWOJEMU katalogowi domowemu Tomcata.



Rozszerzanie kodu serwletu o wywołanie modelu — zapewnienie klientom RZECZYWISTYCH porad...

W tej (*drugiej*) wersji serwletu dodamy do opracowanej wcześniej metody `doPost()` wywołanie modelu zapewniającego porady piwne (w *trzeciej* wersji porady będą przychodziły z odpowiedniej strony JSP). Zmiany wprowadzane w kodzie są zupełnie trywialne — najważniejsze jest w tym momencie dobre zrozumienie procesu ponownego wdrażania rozszerzonej aplikacji internetowej. Możesz teraz albo spróbować przygotować i ponownie skompilować kod aplikacji, po czym wdrożyć go w swoim środowisku, albo przejść na kolejną stronę i wykorzystać zaproponowane tam rozwiązanie...



Przygotuj ołówek

Rozszerzenie serwletu — druga wersja

Zapomnij na chwilę o serwletach i pomyśl wyłącznie o klasach języka programowania Java. Jakie kroki powinniśmy podjąć, aby zrealizować następujące zadania?

1. Rozszerzenie metody `doPost()` o wywołanie modelu.
2. Skompilowanie zmienionego serwletu.
3. Wdrożenie i przetestowanie zaktualizowanej aplikacji internetowej.

```
public class WyborPiwa extends HttpServlet {
```


Druga wersja kodu serwletu

Pamiętaj, że model jest jedynie tradycyjną klasą języka Java, zatem wywołujemy go dokładnie w taki sam sposób, w jaki wywołujemy wszelkie inne metody Javy
— tworzymy egzemplarz klasy modelu i wywołujemy jej metodę!

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Porada piwna<br>");
        String c = request.getParameter("kolor");

        EkspertPiwny be = new EkspertPiwny();
        List wynik = be.getMarki(c);
        Iterator it = wynik.iterator();
        while (it.hasNext()) {
            out.println("<br>Spróbuj:" + it.next());
        }

    }
}
```

← Nie zapomnij o zaimportowaniu pakietu, do którego należy klasa EkspertPiwny.

← Modyfikujemy oryginalny serwlet, ale wcale nie tworzymy nowej klasy.

← Tworzymy egzemplarz klasy EkspertPiwny i wywołujemy metodę getMarki().

Wyświetlamy poradę (elementy zwróconego przez model obiektu klasy ArrayList, które reprezentują marki piwa). W ostatecznej (trzeciej) wersji porady będą wyświetlane przez stronę JSP, a nie (jak w tej wersji) przez serwlet.

Kluczowe kroki związane z finalizacją drugiej wersji serwletu

Pozostały nam do zrobienia jeszcze dwie rzeczy: ponowne skompilowanie serwletu oraz wdrożenie klasy modelu.

Kompilowanie serwletu

Użyjemy tego samego polecenia kompilatora, którego użyliśmy podczas kompilacji pierwszej wersji naszego serwletu.

```
Wiersz polecenia
> cd projekty\piwo-w1
> javac -classpath d:\java\jakarta-tomcat-5.0.27\common\lib\servlet-api.jar:classes:.
-d src\com\example\web\WyborPiwa.java
```

Wdrażanie i testowanie aplikacji internetowej

Poza samym serwletem musimy teraz wdrożyć także zbudowany model. Do kluczowych kroków tego procesu należą:

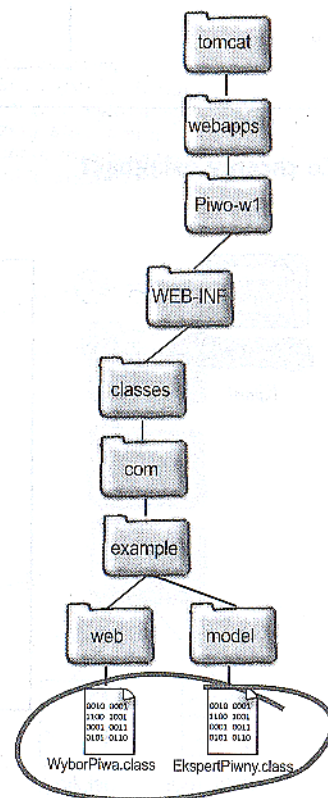
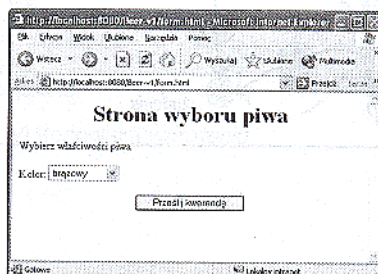
1. Przeniesienie kopii pliku.class serwletu do katalogu *Piwo-w1\WEB-INF\classes\com\example\web*. W ten sposób zastąpimy pierwszą wersję pliku klasy serwletu!
2. Przeniesienie kopii pliku.class modelu do katalogu *Piwo-w1\WEB-INF\classes\com\example\model*.
3. Zatrzymanie i uruchomienie ponownie Tomcata.
4. Przetestowanie aplikacji za pośrednictwem strony internetowej *form.html*, w oknie przeglądarki ostatecznie powinna zostać wyświetlona następująca porada:

Porada piwna

Spróbuj: Jack Amber

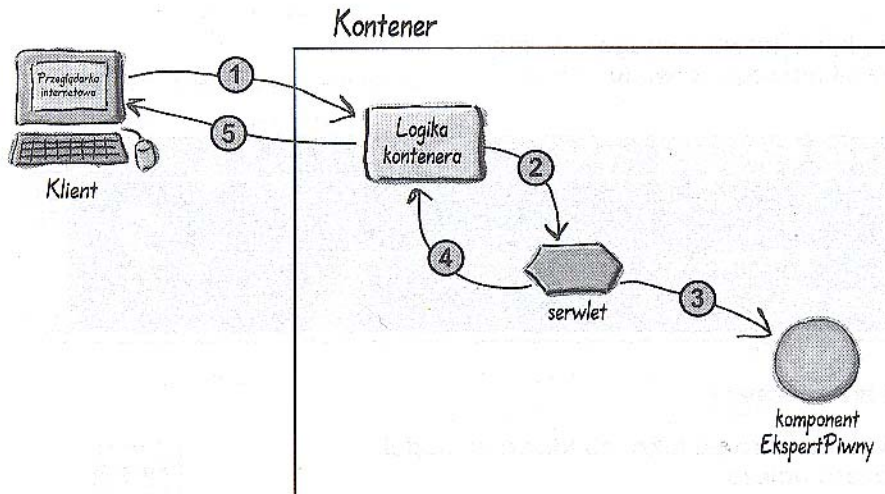
Spróbuj: Red Moose

```
Wiersz polecenia
> cd jakarta-tomcat-5.0.27
> bin\shutdown.bat
> bin\startup.bat
```



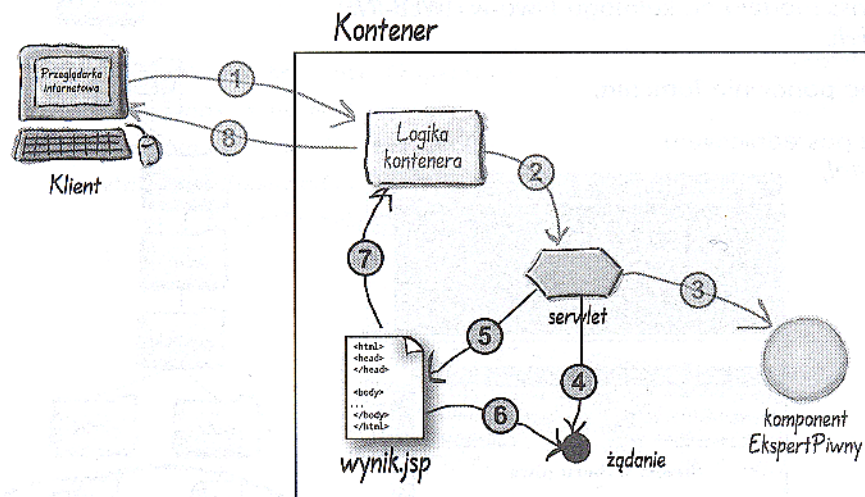
Przegląd prawie kompletnej aplikacji internetowej MVC zapewniającej porady piwne

Co już działa?



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o otrzymany adres URL) właściwy serwlet i przekazuje do tego serwletu otrzymane wcześniej żądanie.
3. Serwlet wywołuje do pomocy komponent EkspertPawny.
4. Serwlet zwraca na wyjściu odpowiedź (która ma postać porady piwnej).
5. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Co chcemy osiągnąć?



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o otrzymany adres URL) właściwy serwlet i przekazuje do tego serwletu otrzymane wcześniej żądanie.
3. Serwlet wywołuje do pomocy komponent EkspertPawny.
4. Klasa eksperta zwraca odpowiedź, którą serwlet dołącza do obiektu żądania.
5. Serwlet przekazuje to żądanie dalej do odpowiedniej strony JSP.
6. Strona JSP pobiera odpowiedź z obiektu żądania.
7. Strona JSP generuje stronę HTML dla kontenera.
8. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Opracowanie „widoku” JSP, który będzie przekazywał poradę

Nie trać nadziei. Będziesz się musiał wykazać cierpliwością jeszcze przez kilka rozdziałów, zanim rzeczywiście zaczniemy analizować technologię stron JSP. Przedstawiony poniżej kod JSP nie jest szczególnie dobrym przykładem (przede wszystkim z powodu kodu skryptletu, któremu poświęcimy trochę uwagi w dalszej części tej książki). Na razie prezentowany kod strony JSP powinien być dosyć łatwy do odczytania, zatem jeśli chcesz trochę poeksperymentować we własnym zakresie, nie powinieneś napotykać żadnych przeszkód. Chociaż *moglibyśmy* już teraz przetestować tę stronę JSP w przeglądarce, wstrzymamy się z tą czynnością do czasu zmodyfikowania serwletu (do wersji trzeciej), aby dopiero wtedy przekonać się, jak działa cała nasza aplikacja.

Oto nasz kod JSP:

```
<% page import="java.util.*" %>
```

← To jest „dyrektywa strony” (myślimy, że znaczenie tego wiersza jest dosyć oczywiste).

```
<html>
```

```
<body>
```

```
<h1 align="center">JSP z rekomendacjami dotyczącymi piwa</h1>
```

← Trochę standardowego kodu HTML (który w świecie JSP jest znany jako „tekst szablonu”)

```
<p>
```

```
<%
```

```
    list styles = (List)request.getAttribute("styles");
```

```
    Iterator it = styles.iterator();
```

```
    while (it.hasNext()) {
```

```
        out.print("<br>Spróbuj:" + it.next());
```

```
    }
```

```
%>
```

Trochę standardowego kodu Javy umieszczonego w znacznikach <%> (zawartość tych znaczników często określa się mianem kodu skryptletu).

```
</body>
```

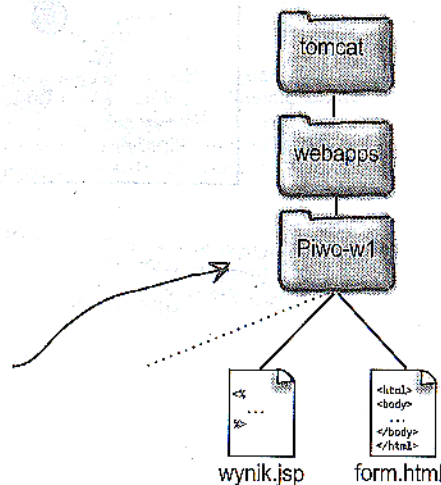
```
</html>
```

← W tym miejscu pobieramy atrybut z obiektu żądania. Nieco później w tej książce wyjaśnimy wszystkie aspekty stosowania atrybutów i technik uzyskiwania dostępu do obiektu żądania...

Wdrażanie strony JSP

Nie kompilujemy strony JSP (zrobi to kontener zaraz po otrzymaniu pierwszego żądania dotyczącego tej strony). *Musimy* jednak wykonać następujące kroki:

1. Nadać jej nazwę *wynik.jsp*.
2. Zapisać ją w katalogu *web* środowiska *wytwarzania*.
3. Przenieść kopię tego pliku do katalogu *Piwo-w1* środowiska *wdrożenia*.

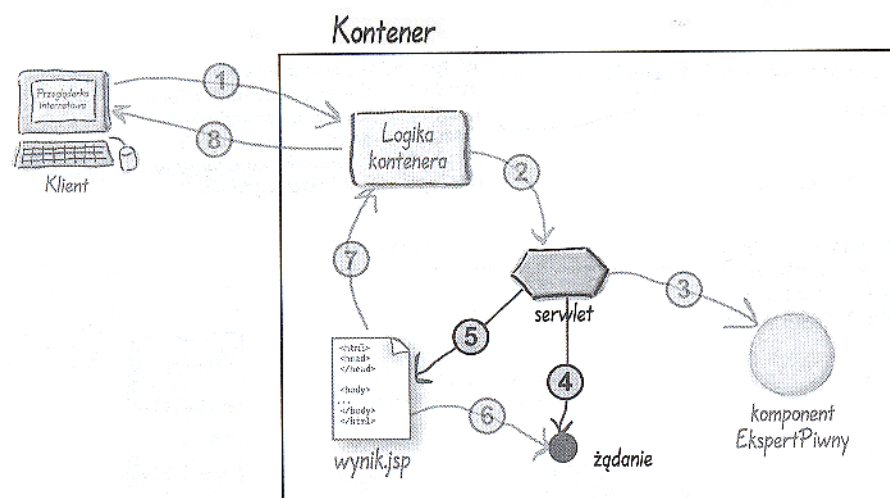


Rozszerzenie serwletu o „wywołanie” strony JSP (wersja trzecia)

W tym kroku mamy zamiar zmodyfikować nasz serwer w taki sposób, aby „wywoływał” stronę JSP, która zwróci dane wyjściowe (widok). Kontener zapewnia mechanizm nazywany *przekazywaniem żądań*, który umożliwia komponentom zarządzanym przez kontener wywoływanie innych komponentów. Właśnie taki jest nasz cel w tym podrozdziale — serwlet pobierze niezbędne informacje z modelu, zapisze je w obiekcie żądania i przekaże to żądanie do odpowiedniej strony JSP.

Najważniejsze zmiany, które musimy wprowadzić w serwlecie:

1. Dodać odpowiedź uzyskaną z komponentu modelu do obiektu żądania, aby odpowiednia strona JSP mogła uzyskać dostęp do tej odpowiedzi (patrz krok 4.).
2. Wymusić na kontenerze przekazanie żądania do strony *wynik.jsp* (patrz krok 5.).



1. Przeglądarka wysyła do kontenera dane żądania.
2. Kontener odnajduje (w oparciu o otrzymany adres URL) właściwy serwlet i przekazuje do tego serwletu otrzymane wcześniej żądanie.
3. Serwlet wywołuje do pomocy komponent EkspertPrawny.
4. Klasa eksperta zwraca odpowiedź, którą serwlet dołącza następnie do obiektu żądania.
5. Serwlet przekazuje żądanie do strony JSP.
6. Strona JSP pobiera odpowiedź z obiektu żądania.
7. Strona JSP generuje stronę HTML dla kontenera.
8. Kontener zwraca szczęśliwemu użytkownikowi gotową stronę.

Trzecia wersja kodu serwletu

Poniżej przedstawiono zmodyfikowany kod serwletu, który od tej pory będzie dodawał do obiektu żądania odpowiedź uzyskaną z komponentu modelu (a więc także przekazywał tę odpowiedź do odpowiedniej strony JSP) oraz wymuszał na kontenerze przekazywanie obiektu żądania do właściwej strony JSP.

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class WyborPiwa extends HttpServlet {

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {

        // response.setContentType("text/html");

        // usuwamy stary kod testowy
        // PrintWriter out = response.getWriter();
        // out.println("Porada piwna<br>");

        String c = request.getParameter("kolor");

        // out.println("<br>Wybrany kolor piwa:" + c);

        EkspertPiwny be = new EkspertPiwny();
        List wynik = be.getMarki(c);

        request.setAttribute("styles", wynik);

        RequestDispatcher view =
            request.getRequestDispatcher("wynik.jsp");

        view.forward(request, response);
    }
}
```

Skoro teraz to strona JSP ma generować dane wyjściowe, możemy usunąć z kodu serwletu wyrażenia związane z testowym wyświetlaniem danych wyjściowych. Oznaczyliśmy te wyrażenia komentarzami, aby nadal były widoczne w kodzie naszego serwletu.

Dodajemy do obiektu żądania atrybut, który będzie wykorzystywany przez docelową stronę JSP. Zwróć uwagę na fakt, iż wspomniana strona JSP będzie szukała atrybutu styles.

Tworzymy egzemplarz klasy przekazującej żądanie do właściwej strony JSP.

Używamy obiektu klasy RequestDispatcher do wymuszania na kontenerze zaangażowania wskazanej strony JSP (włącznie z przekazaniem do tej strony obiektów żądania i odpowiedzi).

Kompilacja, wdrożenie i przetestowanie ostatecznej wersji aplikacji!

W tym rozdziale zbudowaliśmy całą (choć bardzo niewielką) aplikację MVC składającą się ze strony HTML, serwletów oraz stron JSP. Możesz tę aplikację uwzględnić w swoim CV.

Kompilowanie serwletu

Użyjemy tego samego polecenia kompilatora, którego użyliśmy podczas kompilacji dwóch poprzednich wersji naszego serwletu.

```
Wiersz polecenia
> cd projekty\piwo-w1
> javac -classpath d:\java\jakarta-tomcat-5.0.27\common\lib\servlet-api.jar:classes:..
-d src\com\example\web\WyborPiwa.java
```

Wdrażanie i testowanie aplikacji internetowej

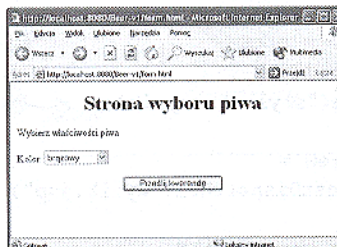
Po skompilowaniu serwletu możemy go ponownie wdrożyć w naszym środowisku wdrożenia.

1. Przenieś kopię pliku *.class* serwletu do katalogu *Piwo-w1\WEB-INF\classes\com\example\web* (także teraz zastępujesz poprzednią wersję pliku klasy serwletu).

2. Zatrzymaj i uruchom ponownie Tomcata.

```
Wiersz polecenia
> cd jakarta-tomcat-5.0.27
> bin\shutdown.bat
> bin\startup.bat
```

3. Przetestuj aplikację za pośrednictwem strony *form.html*.



Taką stronę powinieneś zobaczyć w swojej przeglądarce!

