Core Java Servlets i JavaServer Pages. Tom II. Wydanie II

Autor: Marty Hall,
Larry Brown, Yaakov Chaikin
Tłumaczenie: Daniel Kaczmarek
ISBN: 978-83-246-1726-5
Tytuł oryginału: Core Servlets and Javaserver Pages: Advanced Technologies, Vol. 2 (2nd Edition)
Format: 172x245, stron: 632

- Jak wykorzystać strumienie?
- Jak stworzyć efektowny interfejs użytkownika?
- Jak zapewnić bezpieczeństwo w tworzonych aplikacjach?

Co spowodowało, że język programowania Java zyskał tak wielką popularność? Przyczyn jest kilka: możliwość przenoszenia kodu między programami, wydajność i to, co programiści lubią najbardziej – mechanizm automatycznego oczyszczania pamięci. Nie bez znaczenia jest również to, że Java jest językiem zorientowanym obiektowo, udostępnia obsługę programowania rozproszonego oraz świetną dokumentację. Ponadto liczne publikacje oraz pomocna społeczność sprawiają, że Java zajmuje poczesne miejsce wśród innych języków programowania.

Kolejne wydanie książki zostało zaktualizowane o wszystkie te elementy, które pojawiły się w wersji szóstej platformy Java Standard Edition. Dzięki tej książce dowiesz się, w jaki sposób wykorzystać strumienie, jak parsować dokumenty XML czy też w jaki sposób tworzyć aplikacje sieciowe. Poznasz interfejs JDBC, sposób wykorzystania transakcji oraz wykonywania zapytań SQL. Autorzy w szczegółowy sposób pokażą Ci, jak tworzyć aplikacje z wykorzystaniem biblioteki Swing. Dodatkowo przedstawią, w jaki sposób zapewnić bezpieczeństwo w tworzonych przez Ciebie aplikacjach. Wszystkie te - oraz wiele innych – zagadnienia zostaną przedstawione w przystępny i sprawdzony sposób!

- Wykorzystanie strumieni
- Dokumenty XML i ich wykorzystanie w języku Java
- Programowanie aplikacji sieciowych
- Wykorzystanie interfejsu JDBC
- Tworzenie aplikacji wielojęzycznych
- Możliwości pakietu Swing
- Wykorzystanie biblioteki AWT
- Bezpieczeństwo w aplikacjach
- Zastosowanie podpisu cyfrowego
- Sposoby wykorzystania obiektów rozproszonych (RMI)

Wykorzystaj zaawansowane możliwości języka Java w swoich projektach!

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

Spis treści

1

Używanie i wdrażanie aplikacji internetowych

W tym rozdziale:

- Cele aplikacji internetowych
- Struktura aplikacji internetowych
- Rejestrowanie aplikacji internetowych
- Strategie rozwoju i wdrażania aplikacji internetowych
- Pliki WAR
- Współużytkowanie danych przez aplikacje internetowe

W ramach aplikacji internetowych można tworzyć zwarte kolekcje serwletów, stron Java Server Pages (JSP), bibliotek znaczników, dokumentów HTML (*Hypertext Markup Language*), obrazków, arkuszy stylów i innych treści przeznaczonych dla internetu. Z kolekcji tych można korzystać na dowolnym serwerze zgodnym ze specyfikacją serwletów. Prawidłowo zaprojektowane aplikacje internetowe można przenosić z jednego serwera na inny albo umieszczać w innej lokalizacji na tym samym serwerze bez konieczności wprowadzania zmian do serwletów, stron JSP czy plików HTML wchodzących w skład aplikacji.

W ten sposób nawet złożone aplikacje można przenosić bez większego wysiłku, ułatwiając ich wielokrotne wykorzystywanie. Ponadto dzięki temu, że każda aplikacja internetowa posiada własną strukturę katalogów, sesje, kontekst `ServletContext` i moduł ładowania klas, używanie aplikacji internetowej upraszcza nawet początkowe etapy rozwoju, ponieważ znacznie mniej uwagi trzeba poświęcać na koordynację poszczególnych elementów całego opracowanego rozwiązania.

1.1. Cele aplikacji internetowych

Aplikacje internetowe mają trzy główne zalety — ułatwiają odpowiednie zorganizowanie zasobów, wdrożona aplikacja charakteryzuje się przenośnością, a ponadto łatwiej jest uniknąć konfliktów między różnymi aplikacjami internetowymi. Każda z tych zalet zostanie omówiona nieco szerzej.

Organizacja

Pierwsza zaleta aplikacji internetowych wypływa z faktu, że wszystko ma w nich swoje miejsce — dla każdego rodzaju zawartości przeznaczona jest standardowa lokalizacja. Poszczególne pliki klas Java zawsze znajdują się w katalogu o nazwie *WEB-INF/classes*, pliki JAR (zbiory plików klas Java) zawsze umieszczane są w katalogu *WEB-INF/lib*, plik konfiguracyjny *web.xml* zawsze znajduje się w katalogu *WEB-INF* i tak dalej. Pliki bezpośrednio dostępne dla klientów (na przykład przeglądarek internetowych) umieszczane są w głównym katalogu aplikacji internetowej lub dowolnym jego podkatalogu z wyjątkiem *WEB-INF*.

Często zdarza się, że programista kończy pracę nad jednym projektem i przechodzi do innego zespołu projektowego. Dzięki standardowej organizacji zasobów aplikacji nie trzeba na nowo poznawać struktury aplikacji za każdym razem, gdy rozpoczyna się pracę nad nowym projektem. Nowy programista rozpoczynający pracę nad aplikacją nie musi również tracić czasu na rozpoznanie sposobu organizowania poszczególnych rodzajów plików.

Przenośność

Ponieważ specyfikacja serwletów przewiduje określoną organizację plików, na każdym serwerze zgodnym z tą specyfikacją można właściwie od razu wdrażać i uruchamiać aplikację. Dzięki temu istnieje duża swoboda w wyborze dostawcy docelowego serwera WWW. Jeśli tylko wybrany serwer jest zgodny ze specyfikacją, każdą aplikację można — w większości przypadków bez żadnych zmian — wdrożyć i uruchomić na serwerze pochodzącym od innego producenta. Można na przykład utworzyć aplikację na darmowym serwerze WWW, a następnie jej wersję przedprodukcyjną uruchomić już na bardziej stabilnym serwerze, dla którego producent świadczy odpowiednie wsparcie techniczne.

Separacja

Poszczególne aplikacje internetowe wdrożone na tym serwerze nie wchodzą ze sobą w konflikt. Każda aplikacja ma własny adres URL, za pomocą którego można do niej uzyskać dostęp, własny obiekt `ServletContext` i tak dalej. Dwie aplikacje wdrożone na tym samym serwerze działają tak, jakby były uruchamiane na dwóch oddzielnych serwerach, żadna z tych aplikacji nie musi również mieć dostępu do drugiej.

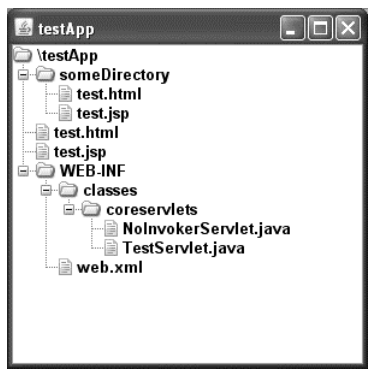
Fakt ten jeszcze bardziej upraszcza proces tworzenia i wdrażania aplikacji internetowych. Programista nie musi bowiem w ogóle zastanawiać się nad tym, w jaki sposób powinno się integrować tworzoną aplikację z aplikacjami już działającymi na tym samym serwerze. Obecnie istnieje kilka sposobów, dzięki którym aplikacje mogą ze sobą współpracować — zostaną one opisane w dalszej części rozdziału. Jednak w znakomitej większości przypadków aplikacje wdraża się niezależnie od siebie.

1.2. Struktura aplikacji internetowych

Jak już wcześniej wspomniano, aplikacja internetowa ma standardowy format i jest przenośna do wszystkich serwerów WWW lub serwerów aplikacji zgodnych ze specyfikacją. Główny katalog aplikacji to zwykły katalog o dowolnej nazwie. Wewnątrz tego katalogu znajdują się lokalizacje przeznaczone dla poszczególnych elementów. W tym punkcie opisane zostaną rodzaje elementów, z jakich może składać się aplikacja, oraz lokalizacje, w których elementy powinny się znajdować.

Lokalizacje poszczególnych rodzajów plików

Na rysunku 1.1 przedstawiono standardową strukturę przykładowej aplikacji internetowej. Aby wykonać kolejne kroki procesu tworzenia własnej aplikacji internetowej, należy pobrać z serwera *ftp://ftp.helion.pl/przyklady/jsp2w2.zip* aplikację *app-blank* i wykonać instrukcje opisane w punkcie 1.6 („Tworzenie prostej aplikacji internetowej").



Rysunek 1.1. *Struktura standardowej aplikacji internetowej*

Strony JSP

Strony JSP należy umieszczać w głównym katalogu aplikacji internetowej lub w jego podkatalogu o dowolnej nazwie, innej niż *WEB-INF* i *META-INF*. Serwery nigdy nie udostępniają klientom plików znajdujących się w katalogach *WEB-INF* i *META-INF*. Gdy aplikacja internetowa zostanie już zarejestrowana (więcej na ten temat w punkcie 1.3), należy wskazać serwerowi prefiks URL wskazujący aplikację oraz zdefiniować lokalizację, w której znajduje się katalog aplikacji internetowej. Zazwyczaj — choć nie jest to obowiązkowe — jako

prefiks URL wskazuje się po prostu nazwę głównego katalogu aplikacji. Po zarejestrowaniu prefiksu strony JSP staną się dostępne za pośrednictwem adresów URL w postaci *http://*
↪komputer/prefiksAplikacji/nazwapliku.jsp (jeżeli strony JSP znajdują się w głównym katalogu aplikacji) albo *http://komputer/prefiksAplikacji/podkatalog/nazwapliku.jsp* (gdy strony znajdują się w podkatalogu).

Od ustawień serwera zależy, czy plik domyślny, taki jak *index.jsp*, będzie dostępny po wpisaniu adresu URL wskazującego jedynie katalog (na przykład *http://komputer/prefiksAplikacji/*) bez zapisania odpowiedniego ustawienia w pliku konfiguracyjnym *WEB-INF/web.xml*. Jeżeli *index.jsp* ma być domyślnym plikiem aplikacji, zalecane jest dodanie w pliku *web.xml* tej aplikacji odpowiedniego elementu `welcome-file-list`. Na przykład poniższy fragment pliku *web.xml* wskazuje, że jeżeli adres URL zawiera nazwę katalogu, a nie zawiera nazwy pliku, wówczas serwer powinien w pierwszej kolejności zwrócić plik *index.jsp*, a w drugiej kolejności plik *index.html*. Jeżeli na serwerze nie ma żadnego z tych dwóch plików, odpowiedź będzie zależeć od serwera (na przykład wyświetlona zostanie zawartość katalogu).

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Więcej informacji na temat sposobu używania pliku *web.xml* znajduje się w rozdziale 2., „Kontrolowanie działania aplikacji przy użyciu deskryptora web.xml”.

Dokumenty HTML, obrazki i inne standardowe elementy aplikacji internetowej

Moduł obsługi serwletów i stron JSP stosuje identyczne reguły dla stron JSP oraz dla plików HTML, obrazków GIF i JPEG, arkuszy stylów i innych dokumentów internetowych. Wszystkie wymienione elementy są umieszczane dokładnie w tych samych lokalizacjach i udostępniane za pośrednictwem adresów URL w identycznej formie.

Pojedyncze serwlety, komponenty JavaBeans i klasy pomocnicze

Serwlety oraz inne pliki z rozszerzeniem *.class* umieszcza się w katalogu *WEB-INF/classes* albo w podkatalogu katalogu *WEB-INF/classes* o nazwie odpowiadającej nazwie pakietu.

Aby uzyskać dostęp do jednego z serwletów trzeba wskazać dla niego odpowiedni adres URL. W tym celu w deskryptorze wdrożenia *web.xml*, znajdującym się w katalogu *WEB-INF*, trzeba zdefiniować element `servlet-mapping`. Więcej informacji na ten temat znajduje się w punkcie 1.3, „Rejestrowanie aplikacji internetowych na serwerze”.

Dostęp do serwletów można także uzyskiwać w inny sposób, który nie wymaga definiowania odpowiedniego adresu URL. Do tego celu można użyć adresu o postaci *http://komputer/prefiks*
↪Aplikacji/servlet/nazwaPakietu.NazwaSerwletu. Używanie adresu o takiej postaci przydaje się w trakcie testowania możliwości aplikacji, natomiast *nie zaleca się* jego używania w aplikacjach produkcyjnych. Powodów jest kilka. Po pierwsze, jeżeli dla tego samego serwletu zostanie również zdefiniowany element `servlet-mapping`, będzie można go wywoływać na dwa różne sposoby, a to z kolei szybko zacznie utrudniać utrzymywanie aplikacji.

Po drugie, ponieważ deklaratywne zabezpieczenia aplikacji zależą od adresu URL, przy użyciu którego zasób jest udostępniany, może w ten sposób powstać luka w zabezpieczeniach. Po trzecie, użytkownik jest zmuszony wpisać adres URL, w którym znajduje się nazwa serwletu ze wszystkimi jego kwalifikatorami, a takie adresy wyglądają mało elegancko i są trudne do zapamiętania. Zatem w kategorii użyteczności aplikacji internetowej taki sposób wywoływania serwletów zyskałby bardzo niskie noty. Po czwarte, jeżeli kiedykolwiek zajdzie potrzeba, by zmienić nazwę klasy albo umieścić klasy w nowym pakiecie, zmienić trzeba będzie również sam adres URL. To z kolei pociągnie za sobą wymóg zmiany wszystkich odwołań znajdujących się w aplikacji, w których używano dotychczasowego adresu URL. Oprócz tego, że sama ta czynność będzie dość uciążliwa, to jej wynik spowoduje zamieszanie również wśród samych użytkowników, którzy przecież mogli zapisać adres wśród swoich ulubionych zakładek, a po zmianie adresu serwletu adres ten przestanie działać i użyteczność aplikacji znowu ucierpi.

Zalecamy jawne blokowanie dostępu do serwletów wchodzących w skład aplikacji internetowej bez odpowiednio odwzorowanego adresu URL. Odwzorowanie takie można zdefiniować za pomocą elementu `servlet-mapping` pliku *web.xml*. Przykładowe odwzorowanie znajduje się w pliku *web.xml* aplikacji *app-blank*, którą można pobrać z witryny *ftp://ftp.helion.pl/*
↪przyklady/jsp2w2.zip.

Serwlety, komponenty JavaBeans i klasy pomocnicze znajdujące się w pakietach JAR

Jeżeli serwlety lub inne pliki z rozszerzeniem *.class* wchodzą w skład pakietów JAR, wówczas pliki JAR powinny się znajdować w katalogu *WEB-INF/lib*. Jeżeli klasy znajdują się w pakietach, wówczas wewnątrz pliku JAR klasy te powinny znajdować się w katalogu o nazwie odpowiadającej nazwie ich pakietu. Większość serwerów umożliwia współużytkowanie plików JAR przez różne aplikacje internetowe. Funkcja ta nie jest jednak standardem, a konkretny sposób jej działania zależy od rodzaju serwera. Na serwerze Tomcat współużytkowane pliki JAR umieszcza się w katalogu *katalog_tomcat/shared/lib*.

Deskryptor wdrożenia

Deskryptor wdrożenia, czyli plik *web.xml*, powinien znajdować się w podkatalogu *WEB-INF* katalogu głównego aplikacji internetowej. Szczegółowe informacje na temat sposobu używania pliku *web.xml* znajdują się w rozdziale 2., „Kontrolowanie działania aplikacji przy użyciu pliku web.xml". Warto zwrócić uwagę, że niektóre serwery mogą obsługiwać globalny plik *web.xml*, dotyczący wszystkich aplikacji internetowych. Na przykład serwer Tomcat wykorzystuje plik *katalog_tomcat/conf/web.xml*, w którym znajdują się globalne ustawienia konfiguracyjne. Plik taki jest jednak cechą charakterystyczną tylko tego serwera. Standardem jest *wyłącznie* plik *web.xml* definiowany dla każdej aplikacji oddzielnie i umieszczany w katalogu *WEB-INF* aplikacji internetowej.

Deskryptory bibliotek znaczników

Pliki będące deskryptorami bibliotek znaczników (ang. *Tag Library Descriptor* — TLD) umieszcza się w katalogu *WEB-INF* lub jego dowolnym podkatalogu. Zaleca się jednak, by deskryptory te umieszczać w podkatalogu *tlds* katalogu *WEB-INF*. Grupowanie deskryptorów

w jednym katalogu (na przykład o nazwie *tlds*) upraszcza proces zarządzania nimi. Strony JSP uzyskują dostęp do plików TLD znajdujących się w *WEB-INF* za pomocą dyrektywy `taglib` o następującej postaci:

```
<%@ taglib uri="/WEB-INF/tlds/mójPlikTaglib.tld" … %>
```

Ponieważ nie mamy do czynienia z klientem (takim jak choćby przeglądarka internetowa), lecz z serwerem, który wykorzystuje plik TLD, blokada dostępu do zawartości znajdującej się w katalogu *WEB-INF* w tym przypadku nie obowiązuje.

Gdy plik z rozszerzeniem *.tld* jest umieszczany wewnątrz pliku JAR, powinien się on znajdować w katalogu *META-INF* lub w którymś z jego podkatalogów. Konieczność zmiany lokalizacji z *WEB-INF* na *META-INF* wynika z tego, że pliki JAR nie są archiwami aplikacji WWW, a więc nie zawierają katalogu *WEB-INF*. Więcej informacji na temat plików TLD znajduje się w rozdziale 7., „Biblioteki znaczników. Zagadnienia podstawowe".

Pliki znaczników

Pliki znaczników powinny znajdować się w katalogu *WEB-INF/tags* lub jego podkatalogu. Podobnie jak w przypadku plików TLD, pliki znaczników pozostają dostępne dla stron JSP nawet wówczas, gdy znajdują się w katalogu *WEB-INF*. Pliki znaczników również deklaruje się na stronie JSP przy użyciu dyrektywy `taglib`. Jednak zamiast `uri` należy w ich przypadku zdefiniować atrybut `tagdir`. Jeżeli na przykład plik *mójPlikZnaczników.tag* umieszczony zostanie w katalogu *WEB-INF/tags* aplikacji internetowej, dyrektywa `taglib` strony JSP powinna mieć następującą postać:

```
<%@ taglib tagdir="/WEB-INF/tags" …%>
```

W takim przypadku serwer automatycznie wygeneruje plik TLD dla plików znaczników, zatem nie trzeba samodzielnie definiować odpowiedniego odwzorowania.

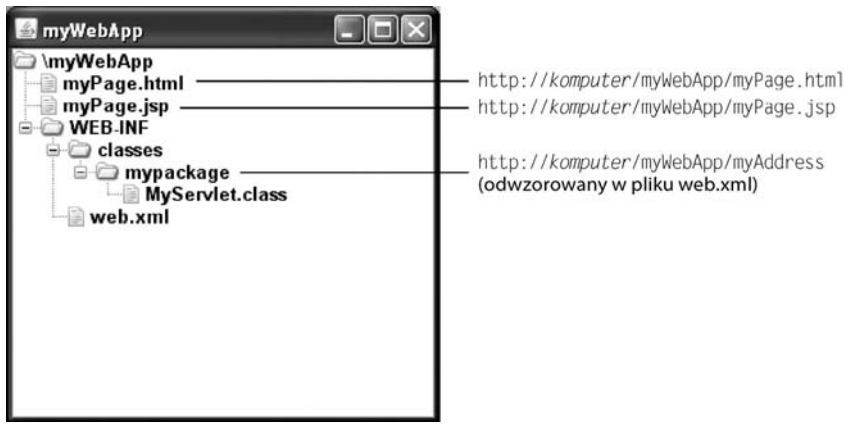
Pliki znaczników można także dołączać do plików JAR. Sam plik JAR powinien zostać umieszczony w katalogu *WEB-INF/lib*, zgodnie z tym, co wspomniano już wcześniej. Jednak wewnątrz pliku JAR pliki znaczników powinny znajdować się w katalogu *META-INF/tags*. W takim przypadku serwer nie wygeneruje automatycznie TLD, dlatego konieczne jest zadeklarowanie plików znaczników i ścieżki dostępu w pliku z rozszerzeniem *.tld*. Warto zauważyć, że plik z rozszerzeniem *.tld* może zawierać również deklaracje własnych znaczników innych typów. Więcej informacji na temat plików znaczników znajduje się w rozdziale 7., „Biblioteki znaczników. Zagadnienia podstawowe".

Plik manifestu WAR

Gdy tworzony jest plik WAR (więcej na ten temat w punkcie 1.5), plik *MANIFEST.MF* zostaje umieszczony w podkatalogu *META-INF*. Zazwyczaj narzędzie jar automatycznie tworzy plik *MANIFEST.MF* i umieszcza go w katalogu *META-INF*, a w przypadku rozpakowywania pliku WAR plik manifestu jest ignorowany. Czasami jednak trzeba jawnie zmodyfikować plik *MANIFEST.MF*, dlatego warto wiedzieć, gdzie się on znajduje.

1.3. Rejestrowanie aplikacji internetowych na serwerze

Jak wspominano już wcześniej, aplikacje internetowe są przenośne. Bez względu na to, jaki serwer jest używany, pliki aplikacji są przechowywane w takiej samej strukturze katalogów, a dostęp do nich uzyskuje się za pośrednictwem adresów URL w takiej samej postaci. Na rysunku 1.2 przedstawiono strukturę adresów URL, których należałoby użyć dla najprostszej aplikacji o nazwie *myWebApp*. W tym punkcie opisane zostaną sposoby instalowania i wykonywania prostej aplikacji internetowej na różnych platformach.



Rysunek 1.2. *Struktura aplikacji internetowej myWebApp*

Same aplikacje internetowe są w pełni przenośne, natomiast już przebieg procesu rejestracji aplikacji zależy od serwera. Aby na przykład przenieść aplikację *myWebApp* z jednego serwera na drugi, nie trzeba wprowadzać żadnych zmian w strukturze katalogów widocznej na rysunku 1.2, natomiast umiejscowienie katalogu głównego aplikacji (w tym przypadku jest to *myWebApp*) zależy od rodzaju używanego serwera. Podobnie od rodzaju serwera zależy sam sposób wskazywania serwerowi, że aplikacja powinna być wywoływana przy użyciu adresu URL w postaci *http://komputer/myWebApp/*.

W tym punkcie przyjęto założenie, że wszystkie kroki zmierzające do zainstalowania i skonfigurowania serwera zostały już wykonane. Więcej informacji na temat konfiguracji serwera można znaleźć w jego dokumentacji, w pierwszym rozdziale pierwszego tomu niniejszej książki lub (w przypadku użytkowników serwera Tomcat) w uaktualnianym na bieżąco przewodniku po instalacji i konfiguracji serwera, dostępnym w witrynie *http://www.coreservlets.*
➥*com*. W tym miejscu zaprezentowany zostanie tylko krótki przykład, a następnie w jednym z kolejnych podpunktów opisana zostanie procedura rejestracji na serwerze Tomcat. Szczegółowy opis procesu wdrożenia przykładowej aplikacji internetowej na serwerze Tomcat zostanie przedstawiony w punkcie 1.6, „Tworzenie prostej aplikacji internetowej”.

Zgodnie z informacjami zawartymi w punkcie 1.4, „Strategie rozwoju i wdrażania aplikacji internetowych”, standardową strategią jest tworzenie aplikacji w środowisku programistycznym i regularne kopiowanie jej do poszczególnych katalogów wdrożeniowych w celu przetestowania rozwiązania na różnych serwerach. Odradza się umieszczanie katalogu rozwojowego

bezpośrednio w katalogu wdrożeniowym serwera, ponieważ znacznie utrudni to wdrożenie aplikacji na kilku serwerach, trudno będzie pracować nad aplikacją w czasie jej działania, a także utrudnione zostanie nadanie odpowiedniej organizacji plików. Lepszym rozwiązaniem jest użycie oddzielnego katalogu rozwojowego i zastosowanie jednej ze strategii opisanych w punkcie 1.4, „Strategie rozwoju i wdrażania aplikacji internetowych". Najprostszym rozwiązaniem jest utworzenie skrótu (w systemie Windows) lub dowiązania symbolicznego (w środowisku UNIX/Linux), prowadzącego do katalogów wdrożeniowych poszczególnych serwerów, oraz kopiowanie całego katalogu rozwojowego za każdym razem, gdy przyjdzie czas na wdrożenie aplikacji. Na przykład w systemie Windows przy użyciu prawego przycisku myszy można przeciągnąć folder rozwojowy do utworzonego skrótu, zwolnić przycisk i wybrać polecenie *Kopiuj*.

Rejestrowanie aplikacji internetowych na serwerze Tomcat

Tworzenie aplikacji internetowej na serwerze Tomcat polega na utworzeniu odpowiedniej struktury katalogów i skopiowanie jej do katalogu *katalog_tomcat/webapps*. Za całą resztę odpowiada już sam serwer. Możliwość wdrażania aplikacji internetowych przez kopiowanie struktury katalogów do jednego z katalogów serwera to tak zwane **wdrażanie na gorąco** (ang. *hot-deployment*) albo **wdrażanie automatyczne** (ang. *auto-deployment*). Katalog w strukturze katalogów serwera, który odpowiada za tę funkcję, to tak zwany **katalog wdrażania na gorąco** (ang. *hot-deploy directory*) albo **katalog wdrażania automatycznego** (ang. *auto-deploy directory*). Większość serwerów WWW, jeśli nie wszystkie współczesne serwery WWW, obsługuje taki właśnie sposób wdrażania aplikacji. Aby zwiększyć kontrolę nad procesem wdrożenia, można odpowiednio zmodyfikować zawartość pliku *katalog_*
↪tomcat/conf/server.xml (jest to plik charakterystyczny tylko dla serwera Tomcat), tak aby odwoływał się on do aplikacji.

Poniżej przedstawiono kolejne czynności, jakie trzeba wykonać, aby utworzyć aplikację internetową dostępną za pośrednictwem adresu URL rozpoczynającego się od *http://komputer/*
↪myWebApp.

1. **Utworzyć strukturę katalogów dla aplikacji internetowej, w której katalog główny będzie nosić nazwę *myWebApp*.** Ponieważ jest to nasza własna struktura rozwojowa, można ją umieścić w dowolnie wybranej lokalizacji. Struktura katalogów powinna być zgodna ze strukturą przedstawioną w punkcie 1.2, „Struktura aplikacji internetowych". Aby zmniejszyć liczbę czynności, jakie należy wykonać w ramach tego kroku, z *ftp://ftp.helion.pl/przyklady/jsp2w2.zip* można pobrać aplikację *app-blank*. Zawiera ona wszystkie niezbędne katalogi oraz przykładowy deskryptor wdrożenia *web.xml*. Pozostaje wówczas tylko zmienić nazwę katalogu głównego *app-blank* na *myWebApp*.

   Jeżeli natomiast zapadnie decyzja o ręcznym utworzeniu wymaganych katalogów, należy postępować zgodnie z dalszym opisem. Najpierw trzeba utworzyć katalog o nazwie *myWebApp* — może się on znajdować w dowolnej lokalizacji w systemie poza katalogiem instalacyjnym serwera. W katalogu tym trzeba utworzyć podkatalog o nazwie *WEB-INF*, w nim zaś utworzyć kolejny katalog *classes*. Następnie należy utworzyć deskryptor wdrożenia *web.xml* i umieścić go w katalogu *WEB-INF*. Szczegółowe informacje na temat deskryptora wdrożenia znajdują się w rozdziale 2., „Kontrolowanie działania aplikacji przy użyciu deskryptora web.xml". Na razie

wystarczy tylko skopiować plik *web.xml* znajdujący się w katalogu *katalog_tomcat/*
➥*webapps/ROOT/WEB-INF* albo użyć wersji deskryptora znajdującej się w aplikacji *app-blank*.

Po utworzeniu odpowiedniej struktury katalogów należy w katalogu *myWebApp* umieścić prostą stronę JSP, a do katalogu *WEB-INF/classes* skopiować prosty serwlet o nazwie *MyServlet.class*.

2. **Zadeklarować serwlet i skierować go na odpowiedni adres URL przez wprowadzenie odpowiednich zmian w pliku *web.xml* będącym deskryptorem wdrożenia.** W odróżnieniu od stron JSP serwlety muszą być deklarowane jawnie. Należy zatem podać pełną ścieżkę dostępu do klasy serwletu, aby wskazać serwerowi, że serwlet ten istnieje. Dodatkowo konieczne jest wskazanie serwerowi adresów URL, które po wpisaniu przez użytkownika powinny wywołać serwlet *MyServlet.class*. Obydwie wspomniane czynności można zrealizować przez wstawienie w pliku *web.xml* poniższego kodu:

```
<servlet>
  <servlet-name>MyName</servlet-name>
  <servlet-class>mypackage.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MyName</servlet-name>
  <url-pattern>/MyAddress</url-pattern>
</servlet-mapping>
```

Element `servlet` oraz jego podelementy wskazują serwerowi nazwę, za pomocą której serwlet będzie deklarowany, a także pełną ścieżkę dostępu do klasy serwletu. Element `servlet-mapping` oraz jego podelementy wskazują serwerowi serwlet, który ma być wywoływany, gdy klient wywoła adres URL odpowiadający wzorcowi wskazanemu jako wartość elementu `url-pattern`. Zatem serwlet zadeklarowany jako *MyName* będzie wywoływany przez adres w postaci *http://komputer/*
➥*myWebApp/MyAddress*.

3. **Skopiować katalog *myWebApp* do katalogu *katalog_tomcat/webapps*.** Załóżmy, że serwer Tomcat został zainstalowany w katalogu *C:\katalog_tomcat*. Katalog *myWebApp* powinno się zatem skopiować do katalogu *webapps* i uzyskać w ten sposób następujące ścieżki dostępu: *C:\katalog_tomcat\webapps\myWebApp*
➥*HelloWebApp.jsp*, *C:\katalog_tomcat\webapps\myWebApp\WEB-INF\classes*
➥*HelloWebApp.class* oraz *C:\katalog_tomcat\webapps\myWebApp\WEB-INF*
➥*web.xml*. Katalog ten można by następnie zawrzeć w pliku WAR (więcej na ten temat w punkcie 1.5) i umieścić go w katalogu *C:\katalog_tomcat\webapps*.

4. **Opcjonalnie — w pliku *server.xml* dopisać element `Context`.** Domyślnie Tomcat skonfiguruje aplikację internetową w taki sposób, by prefiks jej adresu URL dokładnie odpowiadał nazwie głównego katalogu tej aplikacji. Jeżeli takie rozwiązanie będzie wystarczające, można pominąć krok opisywany w tym punkcie. Jeżeli natomiast konieczne jest zmodyfikowanie tego etapu rejestracji aplikacji, w pliku *katalog_tomcat/*
➥*conf/server.xml* można dodać element `Context`. Przed rozpoczęciem edycji zawartości pliku *server.xml* należy utworzyć jego kopię bezpieczeństwa, ponieważ nawet drobny błąd składniowy uniemożliwi prawidłowe uruchomienie serwera Tomcat. Natomiast w nowszych wersjach serwera Tomcat zaleca się, by element `Context` (oraz jego podelementy) umieszczać w pliku *context.xml*. Plik *context.xml* trzeba wówczas umieścić obok pliku *web.xml*, w katalogu *WEB-INF* aplikacji internetowej.

Element `Context` ma kilka atrybutów, które są udokumentowane na stronie pod adresem *http://tomcat.apache.org/tomcat-5.5-doc/config/context.html*. Za pomocą tego elementu można decydować, czy należy używać plików *cookies* albo czy należy przepisywać adresy URL w celu śledzenia sesji; można także włączać i wyłączać możliwość przeładowywania serwletów (to znaczy monitorować klasy pod kątem zachodzących w nich zmian i przeładowywać serwlety tych klas, których pliki na dysku uległy zmianie). Można również ustawiać poziomy debugowania. Jednak w przypadku prostych aplikacji internetowych wystarczy użycie tylko dwóch wymaganych atrybutów: `path` (prefiks adresu URL) oraz `docBase` (bazowy katalog instalacyjny dla aplikacji internetowych względem katalogu *katalog_tomcat/webapps*). Definicja elementu `Context` powinna wyglądać podobnie do poniższej:

```
<Context path="/jakaś-aplikacja" docBase="myWebApp" />
```

Należy pamiętać, by jako prefiks adresu URL nie podawać prefiksu */examples*, ponieważ jest on już używany przez Tomcat jako prefiks dla przykładowych aplikacji internetowych.


Na serwerze Tomcat nie należy używać prefiksu */examples* w roli prefiksu adresów URL aplikacji internetowych.

5. **Wywołać stronę JSP i serwlet.** Adres URL *http://komputer/myWebApp/*
   ↪*myPage.jsp* wywoła stronę JSP, zaś adres *http://komputer/myWebApp/MyAddress* spowoduje wywołanie serwletu. W trakcie rozwoju aplikacji prawdopodobnie jako nazwa komputera używana będzie nazwa *localhost*. W podawanych adresach URL założono, że plik konfiguracyjny serwera Tomcat (*katalog_tomcat/conf/server.xml*) zmodyfikowano w taki sposób, by używany był port numer 80, zgodnie z zaleceniami w instrukcji instalacji i konfiguracji serwera, znajdującej się w witrynie *http://www.coreservlets.com/*. Jeżeli takiej zmiany nie wprowadzono, należy użyć adresów w postaci *http://komputer:**8080**/myWebApp/myPage.jsp* oraz *http://komputer:**8080**/myWebApp/MyAddress*.

Rejestrowanie aplikacji internetowych na innych serwerach

Pierwsze dwie czynności opisane w poprzednim podrozdziale dotyczą tworzenia przenośnej części aplikacji internetowej i będą przebiegać identycznie na każdym serwerze zgodnym ze specyfikacją serwletów. Natomiast czynności dotyczące już samego procesu rejestracji są charakterystyczne dla serwera Tomcat. Nie zmienia to jednak faktu, że rejestrowanie aplikacji na innych serwerach przebiega bardzo podobnie. W kolejnych podpunktach krótko zostaną opisane sposoby rejestrowania aplikacji na kilku najbardziej popularnych obecnie serwerach.

Serwer BEA WebLogic

Podobnie jak Tomcat, serwer WebLogic udostępnia katalog automatycznego wdrażania, służący do rejestrowania i wdrażania aplikacji internetowych. Najpierw, jeżeli jeszcze nie zostało to zrobione, trzeba utworzyć nową domenę o nazwie *myDomain*. Domenę można utworzyć przy użyciu kreatora konfiguracji serwera WebLogic *config.cmd* w systemie Windows lub *config.sh* w środowiskach UNIX/Linux (obydwie wersje znajdują się w katalogu *bea/katalog_weblogic/common/bin*). Gdy domena zostanie już utworzona, należy skopiować całą strukturę katalogów aplikacji internetowej (w tym również jej katalog główny) lub plik WAR do katalogu *bea/user_projects/domains/myDomain/applications*. Do wdrożenia aplikacji można także użyć konsoli WWW serwera WebLogic. W tym celu należy zalogować się do konsoli pod adresem *http://localhost:7001/console/* (przy założeniu, że nie zmieniono domyślnych ustawień portu oraz że serwer jest uruchomiony na komputerze lokalnym). W panelu konsoli po lewej stronie należy rozwinąć węzeł *Deployments* i kliknąć pozycję *Web Application Modules*, po czym kliknąć polecenie *Deploy a New Web Application Module*. W ten sposób uruchomiony zostanie kreator wdrożenia działający w przeglądarce, w którym wystarczy wykonywać kolejne wyświetlane instrukcje. Po wdrożeniu aplikacji internetowej strona JSP będzie dostępna pod adresem *http://localhost:7001/myWebApp/myPage.jsp*, zaś serwlet będzie dostępny pod adresem *http://localhost:7001/myWebApp/MyAddress*.

JBoss

Zarejestrowanie aplikacji internetowej na serwerze JBoss jest niemal równie proste, jak w przypadku serwera Tomcat. Tak naprawdę JBoss domyślnie używa osadzonego serwera WWW Tomcat. Aby zarejestrować aplikację, należy najpierw zmienić nazwę katalogu głównego *myWebApp* na *myWebApp.war*. W istocie nie jest tworzony plik WAR, lecz jedynie nazwa katalogu głównego jest zmieniana w taki sposób, by kończyła się słowem *.war*. JBoss wymaga, by słowem *.war* kończyły się nie tylko nazwy plików WAR (co jest zresztą wymagane na mocy specyfikacji serwletów), lecz również nazwa głównego katalogu aplikacji internetowej. Po zmianie nazwy katalogu katalog *myWebApp.war* należy skopiować do katalogu *katalog_jboss/server/default/deploy*. Jeżeli w trakcie instalacji serwera JBoss żadne domyślne ustawienia nie uległy zmianie, strona JSP będzie dostępna pod adresem *http://localhost:8080/myWebApp/myPage.jsp*, natomiast serwlet będzie wywoływany adresem *http://localhost:8080/myWebApp/MyAddress*. Jeżeli aplikacja została spakowana w pliku WAR, w celu jej wdrożenia plik *myWebApp.war* trzeba umieścić w tym samym katalogu serwera JBoss.

Caucho Resin

Aby skorzystać z funkcji automatycznego wdrażania na serwerze Caucho Resin, należy skopiować całą strukturę katalogów aplikacji internetowej (włącznie z jej katalogiem głównym) lub plik WAR do katalogu *katalog_resin/webapps*. Jeżeli domyślne ustawienia serwera nie zostały zmienione, strona JSP stanie się dostępna pod adresem *http://localhost:8080/myWebApp/*
↪myPage.jsp, zaś serwlet będzie dostępny pod adresem *http://localhost:8080/myWebApp/*
↪MyAddress.

1.4. Strategie rozwoju i wdrażania aplikacji internetowych

Gdy wszystko będzie już przygotowane do rozpoczęcia procesu tworzenia nowej aplikacji internetowej, należy wykonać następujące trzy czynności:

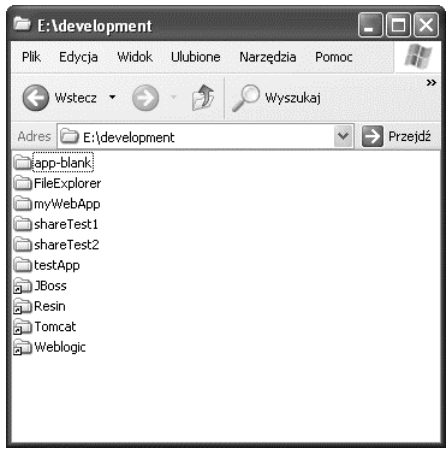
1. **Utworzyć strukturę katalogów aplikacji.** W katalogu rozwojowym trzeba utworzyć nową strukturę katalogów, zgodną ze strukturą aplikacji internetowej (włączając w to plik *web.xml* znajdujący się w katalogu *WEB-INF*), opisaną we wcześniejszej części tego rozdziału. Najprostszym sposobem wykonania tej czynności jest skopiowanie i zmiana nazwy aplikacji *app-blank*. (Jak pamiętamy, aplikację *app-blank* oraz wszystkie inne przykładowe kody źródłowe prezentowane w książce można pobrać z *ftp://ftp.helion.pl/przyklady/jsp2w2.zip*).
2. **Napisać kod źródłowy aplikacji.** Strony HTML i JSP trzeba umieścić w katalogu głównym lub jego podkatalogach, z wyjątkiem podkatalogu *WEB-INF* i *META-INF*. Poszczególne pliki klas języka Java należy umieścić w katalogu *WEB-INF/classes/*
↪*podkatalog_o_nazwie_zgodnej_z_nazwą_pakietu*. Pliki JAR powinny trafić do katalogu *WEB-INF/lib*, pliki z rozszerzeniami *.tag* i *.tagx* do katalogu *WEB-INF/tags* i tak dalej.
3. **Wdrożyć aplikację.** Całą strukturę katalogów aplikacji internetowej (w tym również jej katalog główny) trzeba skopiować do katalogu wdrażania automatycznego serwera. Istnieje kilka rozwiązań upraszczających to zadanie, a najczęściej stosowanymi są:
   - Skopiowanie struktury do skrótu lub dowiązania symbolicznego.
   - Wykorzystanie funkcji wdrażania udostępnianych przez zintegrowane środowisko programistyczne (IDE).
   - Wykorzystanie narzędzia Ant lub podobnego.
   - Użycie środowiska IDE wraz z narzędziem Ant.

Użytkownicy dopiero poznający działanie serwletów i stron JSP mogą pozostać przy pierwszej opcji, zaś rozpoznawanie sposobu działania narzędzia Ant lub konkretnego środowiska IDE można rozpocząć po zaznajomieniu się przynajmniej z podstawami serwletów i stron JSP. Trzeba jednak zaznaczyć, że *nie* wspomniano o opcji polegającej na umieszczeniu kodu bezpośrednio w katalogu wdrożeniowym serwera. Wprawdzie to właśnie rozwiązanie najczęściej wybierają początkujący programiści, wachlarz jego zastosowań w bardziej zaawansowanych zadaniach jest tak skromny, że zaleca się, by nie stosować go już na samym początku.

W kolejnych podpunktach opisane zostaną wszystkie cztery wspomniane opcje.

Kopiowanie struktury do skrótu lub dowiązania symbolicznego

W systemie Windows należy przejść do katalogu nadrzędnego wobec katalogu automatycznego wdrożenia serwera. W przypadku serwera Tomcat będzie to główny katalog instalacyjny *katalog_tomcat*. Katalog automatycznego wdrożenia należy kliknąć prawym przyciskiem myszy (na serwerze Tomcat będzie to katalog *webapps*) i wybrać polecenie *Kopiuj*. Następnie trzeba przejść do katalogu bezpośrednio nadrzędnego wobec głównego katalogu rozwojowego (czyli katalogu o jeden wyższego w hierarchii niż *myWebApp*), kliknąć go prawym przyciskiem myszy i wybrać polecenie *Wklej skrót* (a nie zwykłe polecenie *Wklej*). Od teraz za każdym razem, gdy przyjdzie już czas na wdrożenie aplikacji internetowej, należy kliknąć i przytrzymać wciśnięty prawy przycisk myszy na katalogu rozwojowym (na przykład *myWebApp*) i przeciągnąć go do skrótu do katalogu wdrożenia, po czym zwolnić przycisk myszy. Pojawi się wówczas menu podręczne, w którym należy wybrać polecenie *Kopiuj tutaj*. Na rysunku 1.3 przedstawiono przykładową konfigurację, która ułatwi wdrażanie i testowanie aplikacji prezentowanych w tym rozdziale na serwerach Tomcat, WebLogic, JBoss i Resin. W systemie UNIX można użyć dowiązań symbolicznych (tworzonych poleceniem `ln -s`) w sposób podobny do sposobu używania skrótów w systemie Windows.



Rysunek 1.3. *Wykorzystanie skrótów w celu uproszczenia procesu wdrażania aplikacji*

Zaletą tego rozwiązania jest to, że jest ono bardzo proste. Jest ono zatem dobre dla osób początkujących, które chcą się skupić na poznawaniu działania serwletów i stron JSP, a nie na narzędziach do wdrażania aplikacji czy środowiskach IDE.

Wadą przedstawionego rozwiązania jest to, że gdy używanych jest kilka serwerów, konieczne jest powtarzanie operacji kopiowania. Załóżmy na przykład, że w środowisku rozwojowym funkcjonuje kilka różnych serwerów (Tomcat, Resin i tak dalej), na których regularnie trzeba testować implementowany kod źródłowy. Druga wada polega na tym, że w takim rozwiązaniu na serwer kopiowane są zarówno pliki z kodem źródłowym Javy, jak i pliki klas, choć tak naprawdę niezbędne są tylko te ostatnie. Na serwerze biurkowym nie ma to większego znaczenia, lecz gdy przyjdzie do wdrożenia aplikacji na „prawdziwym" serwerze produkcyjnym, pliki z kodem źródłowym nie powinny na niego trafić.

Wykorzystanie funkcji wdrażania udostępnianych przez IDE

Większość środowisk programistycznych do tworzenia serwletów i stron JSP (takich jak IBM WebSphere Studio Application Developer, Sun ONE Studio, NetBeans, Oracle JDeveloper, Borland JBuilder, Eclipse z MyEclipseIDE czy wtyczkami NitroX) pozwala na stosowanie takich konfiguracji, w których jednym kliknięciem myszy możliwe jest wdrożenie aplikacji internetowej na serwerze testowym, rozwojowym lub produkcyjnym.

Mimo iż środowiska IDE mają wiele niekwestionowanych zalet, nie są one jednak wolne od wad. Większość prawdziwych środowisk IDE trzeba najpierw poznać, co zajmuje trochę czasu. Nie można się więc skoncentrować na zadaniach stricte programistycznych, przynajmniej na początku. Ponadto często się zdarza, że programiści są przerzucani z jednego projektu na drugi, a programiści pracujący wspólnie nad daną aplikacją muszą korzystać z takiego samego środowiska IDE. Gdy więc programista przejdzie do innego zespołu projektowego, w którym kompilacja i wdrażanie aplikacji będą wykonywane w innym środowisku IDE niż to, którego dotąd używał programista, konieczne będzie poznanie nowego narzędzia i przyzwyczajenie się do pracy w nim. To z kolei będzie wymagać poświęcenia dodatkowego czasu.

Używanie narzędzi ant, maven i im podobnych

Narzędzie `ant` zaimplementowane przez Apache Foundation działa podobnie jak `make` w środowisku UNIX. Jednak `ant` został napisany w Javie (dzięki czemu jest przenośny) i powszechnie uważa się, że w porównaniu z `make` jest prostszy w użyciu i bardziej rozbudowany. Wielu programistów serwletów i stron JSP kompiluje i wdraża aplikacje właśnie przy użyciu `ant`. Sposób używania narzędzia `ant` został opisany w dodatku „Tworzenie Aplikacji przy użyciu narzędzia Apache ant".

Ogólne informacje o `ant` są dostępne na stronie pod adresem *http://ant.apache.org/manual/*, natomiast wskazówki dotyczące używania `ant` z serwerem Tomcat znajdują się pod adresem *http://tomcat.apache.org/tomcat-5.5-doc/appdev/processes.html*.

Główną zaletą takiego rozwiązania jest elastyczność — `ant` jest tak rozbudowany, że obsługuje wszystkie etapy od kompilacji kodu źródłowego Javy, przez kopiowanie plików aż po generowanie plików archiwum WAR (więcej na ten temat w punkcie 1.5).

Kolejnym narzędziem, które przyciąga uwagę programistów Javy, jest narzędzie `maven`. Narzędzie to rozszerza możliwości `ant`, dlatego w niektórych przypadkach bardzo je przypomina, a w niektórych zasadniczo się od niego różni. Wprawdzie `maven` jest równie elastyczny jak `ant`, lecz jego twórcy skupili się przede wszystkim na prostocie jego używania. Dlatego też ważną rolę odgrywają w nim konwencje. `maven` może na przykład skompilować kod źródłowy nawet wówczas, gdy programista nie wskaże go w folderze projektowym. Jest to możliwe dzięki temu, że w `maven` przyjęto, iż zgodnie z konwencją kod źródłowy języka Java znajduje się w folderze *src/main/java*. Oczywiście, założenie to można zmieniać, ale po co? Dzięki zastosowaniu konwencji pliki konfiguracyjne programu `maven` są, w odróżnieniu od `ant`, bardzo krótkie i zrozumiałe. Więcej informacji na temat tego narzędzia można znaleźć na stronie *http://maven.apache.org*.

Wadą zarówno `ant`, jak i `maven` jest konieczność poświęcenia określonego czasu na naukę pracy z nimi. Zasadniczo krzywa uczenia się `ant` i `maven` jest bardziej stroma niż w przypadku dwóch pierwszych technik prezentowanych w tym podrozdziale. Natomiast istotną różnicą między poświęcaniem czasu na naukę konkretnego środowiska IDE a poznawaniem narzędzia `ant` albo `maven` jest to, że w coraz większej liczbie środowisk `ant` i `maven` są używane jako rozwiązanie standardowe, zamiast narzędzi charakterystycznych tylko dla tych środowisk. Można się więc spodziewać, że wiadomości na temat `ant` lub `maven` będzie można wykorzystywać również w dalszej przyszłości.

Używanie IDE wraz z narzędziem Ant

Środowiska IDE zwiększają efektywność pracy programistów, ponieważ pomagają tworzyć kod źródłowy, natomiast ograniczają możliwości jego przenoszenia. Narzędzie `ant` z kolei pozwala na tworzenie aplikacji przenośnych, lecz w żadnej mierze nie pomaga pisać kodu źródłowego. Co z tym zrobić?

Przepaść dzielącą `ant` i IDE można nieco zmniejszyć przez używanie środowiska IDE zintegrowanego z `ant`. Dzięki temu nadal można pracować z ulubionym środowiskiem programistycznym, które wspiera proces pisania kodu źródłowego, a jednocześnie jednym kliknięciem myszy wywołuje skrypty `ant` odpowiadające za skompilowanie i wdrożenie aplikacji. Takiego rozwiązania można używać nawet w środowiskach, które z `ant` nie są zintegrowane (choć większość współczesnych środowisk taką integrację już posiada), lecz będzie to wymagać przełączania się między wierszem poleceń a środowiskiem programistycznym.

Takie podejście doskonale sprawdza się w prawdziwych projektach. Programiści używają ulubionych IDE, w których pisanie kodu jest bardziej efektywne, zaś sam projekt nie jest narażony na powstawanie niespójności w trakcie wdrożenia, ponieważ kompilacja i wdrożenie są przeprowadzane przez ten sam przenośny skrypt `ant`. Uczestniczyliśmy w projektach, w których poszczególni programiści należący do zespołu projektowego tworzącego jedną aplikację internetową korzystali z różnych IDE; niektórzy z nich pracowali nawet w odmiennych systemach operacyjnych. Proces wdrażania aplikacji natomiast pozostawał spójny, a sami programiści byli zadowoleni z możliwości używania swoich ulubionych narzędzi programistycznych. A czyż zadowolenie programisty nie jest najważniejsze?

1.5. Umieszczanie aplikacji internetowych w plikach WAR

Pliki archiwum WWW (ang. *Web archive* — WAR) doskonale nadają się do łączenia w pojedynczym archiwum całych aplikacji internetowych. Jeden duży plik z aplikacją zastępujący wiele małych plików znacznie łatwiej jest przenosić z serwera na serwer.

Archiwum WAR jest tak naprawdę plikiem JAR z rozszerzeniem *.war* i tworzy się go zwykłym poleceniem `jar`. Aby na przykład umieścić całą aplikację *aplikacja_internetowa* w pliku WAR o nazwie *aplikacja_internetowa.war*, należy przejść do katalogu *aplikacja_internetowa* i wykonać następujące polecenie:

```
jar cvf aplikacja_internetowa.war *
```

Pliku WAR nie łączy z poleceniem `jar` żadna specjalna zależność. Polecenie `jar` jest tak naprawdę jednym z wielu narzędzi, dzięki któremu można tworzyć archiwa plików. Takie samo archiwum można utworzyć na przykład przy użyciu programu WinZip (albo `tar` w systemie UNIX). Wystarczy w tym celu wskazać nazwę pliku docelowego z rozszerzeniem *.war* zamiast rozszerzenia *.zip*.

Oczywiście, nic nie stoi na przeszkodzie, by w trakcie tworzenia archiwów WAR wykorzystać inne opcje narzędzia `jar` (na przykład cyfrowo podpisywać klasy), tak samo jak wobec standardowych plików JAR. Więcej szczegółowych informacji na ten temat można znaleźć pod adresem *http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/jar.html* (dotyczy systemu Windows) oraz *http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/jar.html* (dotyczy systemów UNIX/Linux).

1.6. Tworzenie prostej aplikacji internetowej

Czas już przejść od słów do czynów, czyli utworzyć pierwszą aplikację internetową. Jako serwer użyty zostanie serwer Tomcat, lecz opisywane czynności będą przebiegać podobnie również w przypadku wykorzystania innych serwerów. Więcej informacji na ten temat przedstawiono w punkcie 1.3, „Rejestrowanie aplikacji internetowych na serwerze".

W celu utworzenia aplikacji należy wykonać następujące kroki:

1. Z *ftp://ftp.helion.pl/przyklady/jsp2w2.zip* pobrać aplikację *app-blank* i zmienić jej nazwę na *testApp*.
2. Z tej samej witryny pobrać pliki *test.html*, *test.jsp* oraz *TestServlet.java*.
3. Dodać do tej aplikacji pliki *test.html* i *test.jsp*.
4. Umieścić plik *TestServlet.java* w katalogu *testApp/WEB-INF/classes/coreservlets*.
5. Skompilować plik *TestServlet.java*.
6. Zadeklarować w pliku *web.xml* klasę *TestServlet.class* oraz adres URL, który będzie ją wywoływać.
7. Skopiować aplikację *testApp* do katalogu *katalog_tomcat/webapps*.
8. Uruchomić serwer Tomcat.
9. Wywołać aplikację *testApp*, wpisując adres URL w postaci *http://localhost/testApp/nazwaZasobu*.

Wymienione czynności zostaną szczegółowo opisane w kolejnych sekcjach.

Pobranie aplikacji app-blank i zmiana jej nazwy na testApp

Ten krok jest bardzo prosty. Wystarczy z *ftp://ftp.helion.pl/przyklady/jsp2w2.zip* pobrać plik *app-blank.zip*. Zawiera on odpowiednią strukturę katalogów dla każdej aplikacji internetowej zgodnej ze specyfikacją J2EE. Archiwum to zawiera także wyjściową wersję deskryptora wdrożenia *web.xml*, w którym znajduje się odwzorowanie wyłączające wywoływacz serwletu. Więcej na temat odwzorowywania serwletów powiemy przy okazji odwzorowywania adresu URL na serwlet `TestServlet`. Na razie wystarczy rozpakować archiwum *app-blank.zip* do wybranego katalogu i zmienić nazwę katalogu na *testApp*. Trzeba również pamiętać, by katalog *testApp* trafił do katalogu, który będzie się znajdował poza strukturą katalogów serwera Tomcat.

Pobranie plików test.html, test.jsp oraz TestServlet.java

Podobnie jak w poprzednim kroku, wymagane pliki można pobrać z *ftp://ftp.helion.pl/*
➥przyklady/jsp2w2.zip. Pliki można pobrać oddzielnie jeden po drugim bądź też w postaci pojedynczego archiwum *testAppFiles.zip*, a następnie rozpakować je w wybranym katalogu.

Dodanie plików test.html i test.jsp do aplikacji internetowej testApp

Pliki *test.html* i *test.jsp* należy umieścić w katalogu *testApp*, a następnie utworzyć w nim katalog *someDirectory* i wstawić w nim kopię plików *test.html* i *test.jsp*. Plik *test.html* zawiera treść statyczną, natomiast *test.jsp* zawiera skrypt zwracający adres URL, z którego wywołano stronę. Na listingach 1.1 i 1.2 przedstawiono kompletne kody źródłowe obydwóch plików.

Listing 1.1. *test.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Test HTML</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>Test HTML</H1>
Witaj.
</BODY></HTML>
```

Listing 1.2. *test.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Test JSP</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>Test JSP</H1>
Użyto następującego adresu URL:
<%= request.getRequestURL() %>
</BODY></HTML>
```

Umieszczenie pliku TestServlet.java w katalogu testApp/WEB-INF/classes/coreservlets

W pliku *TestServlet.java* znajduje się deklaracja, według której serwlet należy do pakietu `coreservlets`. Zatem plik *TestServlet.java* musi znaleźć się w tym pakiecie, zanim będzie można go skompilować. Podobnie jak plik *test.jsp*, plik *TestServlet.class* zawiera kod zwracający adres URL, za pomocą którego nastąpiło wywołanie serwletu. Pełny kod źródłowy serwletu *TestServlet.java* znajduje się na listingu 1.3.

Listing 1.3. *TestServlet.java*

```
package coreservlets;

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

public class TestServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    response.setCharacterEncoding("UTF-8");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println
      (docType +
       "<HTML>\n" +
       "<HEAD><TITLE>Test serwletu</TITLE></HEAD>\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1>Test serwletu</H1>\n" +
       "Użyto następującego adresu URL: " + request.getRequestURL() + "\n" +
       "</BODY></HTML>");
  }
}
```

Kompilacja pliku TestServlet.java

Należy pamiętać, by w zmiennej środowiskowej `CLASSPATH` wskazać interfejs programistyczny (API) serwletów. Serwer Tomcat udostępnia interfejs w postaci pliku *servlet-api.jar*, znajdującym się w katalogu *katalog_tomcat/common/lib*. W systemie Windows zmienną `CLASSPATH` można zdefiniować w wierszu poleceń systemu operacyjnego, przez wykonanie następującego polecenia:

```
set CLASSPATH=katalog_tomcat\common\lib\servlet-api.jar
```

Natomiast w środowiskach zgodnych z UNIX/Linux zmienną `CLASSPATH` można zdefiniować przy użyciu konsoli, przez wykonanie polecenia o następującej treści:

```
CLASSPATH=katalog_tomcat/common/lib/servlet-api.jar
```

Po odpowiednim zdefiniowaniu zmiennej `CLASSPATH` można już skompilować plik *TestServlet.*
➥*java*. W tym celu należy przejść do katalogu *testApp/WEB-INF/classes/coreservlets* i wpisać następujące polecenie:

```
javac TestServlet.java
```

Gdy kompilacja dobiegnie końca, w katalogu *testApp/WEB-INF/classes/coreservlets* powinna pojawić się klasa *TestServlet.class*.

Zadeklarowanie w pliku web.xml klasy TestServlet.class oraz adresu URL, który ją wywołuje

Należy przejść do katalogu *testApp/WEB-INF* i otworzyć w ulubionym edytorze języka XML (Extensible Markup Language) albo edytorze tekstu plik *web.xml*. Aby zadeklarować klasę *TestServlet.class* należy bezpośrednio za komentarzem `<!-- Tutaj należy umieścić własne wpisy -->` umieścić poniższe wiersze kodu:

```
<servlet>
  <servlet-name>Test</servlet-name>
  <servlet-class>coreservlets.TestServlet</servlet-class>
</servlet>
```

Przedstawiony fragment zawiera deklarację serwletu o nazwie `Test`, który reprezentuje klasę *TestServlet.class*. Warto zwrócić uwagę, że element `<servlet-class>` wskazuje w pełną nazwę klasy serwletu w postaci *nazwaPakietu.nazwaKlasy* (bez rozszerzenia *.class*).

W następnym kroku trzeba wskazać serwerowi Tomcat adresy URL, które mają wywoływać serwlet `Test`. W tym celu w pliku *web.xml* należy bezpośrednio po zamknięciu elementu `</servlet>` umieścić następujące wiersze kodu:

```
<servlet-mapping>
  <servlet-name>Test</servlet-name>
  <url-pattern>/test</url-pattern>
</servlet-mapping>
```

Powyższe wiersze wskazują serwerowi Tomcat, że gdy klient wywoła aplikację internetową *testApp* przy użyciu adresu URL w postaci *http://komputer/testApp/test*, powinien zostać wywołany wcześniej zadeklarowany serwlet *Test*. Zawartość pliku *web.xml* przedstawiono na listingu 1.4.

Listing 1.4. *web.xml*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Plik web.xml z szablonu aplikacji WWW app-blank,
   z witryny http://courses.coreservlets.com/Course-Materials/.
   Zawiera dwa standardowe elementy: welcome-file-list
   oraz servlet-mapping, który wyłącza wywoływacz serwletu.
-->
<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
            http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
```

```
    <!-- Tutaj należy umieścić własne wpisy -->
  <servlet>
    <servlet-name>Test</servlet-name>
    <servlet-class>coreservlets.TestServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Test</servlet-name>
    <url-pattern>/test</url-pattern>
  </servlet-mapping>


  <!-- Wyłączenie wywoływacza serwletu -->
  <servlet>
    <servlet-name>NoInvoker</servlet-name>
    <servlet-class>coreservlets.NoInvokerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>NoInvoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- Jeżeli adres URL zawiera katalog bez nazwy pliku, nastąpi próba wywołania pliku index.jsp,
       a następnie index.html. Jeżeli żaden z tych plików nie zostanie znaleziony,
       efekt będzie zależeć od serwera (np. może zostać wyświetlona zawartość katalogu).
  -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Więcej informacji na temat pliku *web.xml* znajduje się w rozdziale 2., „Kontrolowanie działania aplikacji przy użyciu deskryptora web.xml".

Skopiowanie aplikacji testApp do katalogu katalog_tomcat/webapps

W tym kroku należy skopiować całą strukturę katalogów, począwszy od katalogu głównego aplikacji *testApp*, do katalogu automatycznego wdrożenia na serwerze Tomcat *katalog_tomcat/*
↪webapps. Alternatywnie można spakować aplikację *testApp* do pliku WAR (przy użyciu polecenia `jar`, programu WinZip albo narzędzia `tar` lub innego, podobnego programu), a następnie skopiować sam plik WAR do katalogu *katalog_tomcat/webapps*. Bez względu na to, czy do katalogu docelowego trafi aplikacja w swojej „rozdrobnionej" wersji, czy jako plik WAR, ostateczny efekt będzie identyczny.

Uruchomienie serwera Tomcat

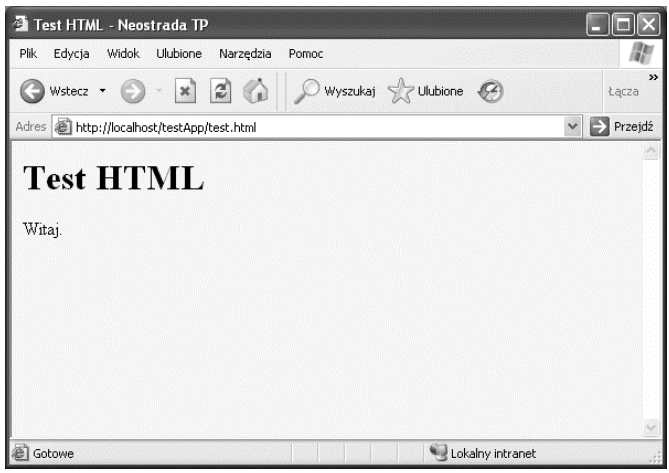
Serwer Tomcat można uruchomić w systemie Windows przez wywołanie pliku *katalog_tomcat/*
↪bin/startup.bat. W systemie UNIX/Linux należy w tym celu wywołać plik *katalog_tomcat/*
↪bin/startup.sh. Tomcat odnajdzie nowy katalog w swym katalogu automatycznego wdrażania i automatycznie zarejestruje i wdroży aplikację internetową *testApp*.

Wywołanie aplikacji testApp przy użyciu adresu URL w postaci http://localhost/testApp/zasób

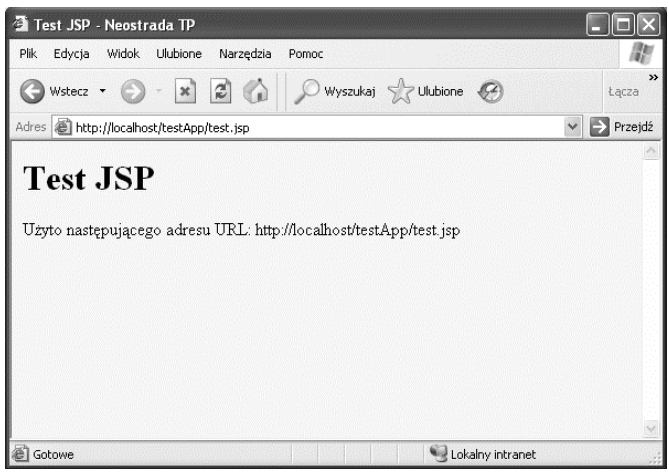
Wywołanie adresów URL w postaci *http://localhost/testApp/test.html* oraz *http://localhost/testApp/someDirectory/test.html* spowoduje wyświetlenie w przeglądarce pliku *test.html*. Wywołanie adresów *http://localhost/testApp/test.jsp* oraz *http://localhost/testApp/someDirectory/test.jsp* spowoduje wywołanie pliku *test.jsp*, natomiast adres URL w postaci *http://localhost/testApp/test* wywoła serwlet *TestServlet.class*.

Wszystkie przytoczone adresy zadziałają przy założeniu, że zgodnie z zaleceniem przedstawionym w pierwszym tomie niniejszej książki w pliku konfiguracyjnym serwera Tomcat (*katalog_tomcat/conf/server.xml*) wskazany zostanie port numer 80. Jeżeli zmiana ta nie została wprowadzona, wystarczy we wszystkich adresach URL zastąpić słowo *localhost* słowem *localhost:8080*. Na rysunkach od 1.4 do 1.8 przedstawiono zrzuty ekranów będących wynikiem wywołania poszczególnych zasobów.

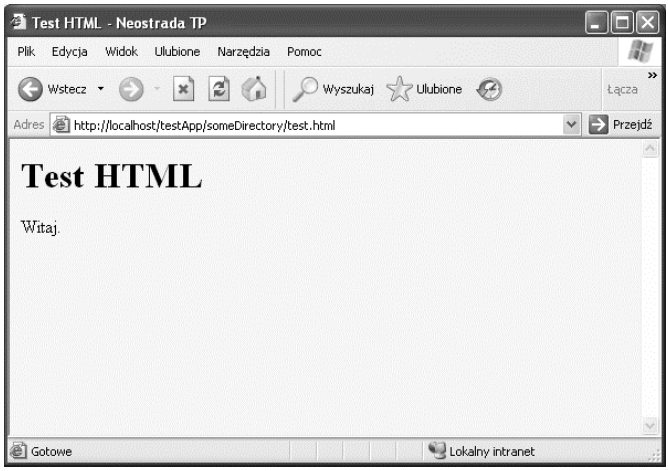
Rysunek 1.4. *Wynik wywołania adresu http://localhost/testApp/test.html*



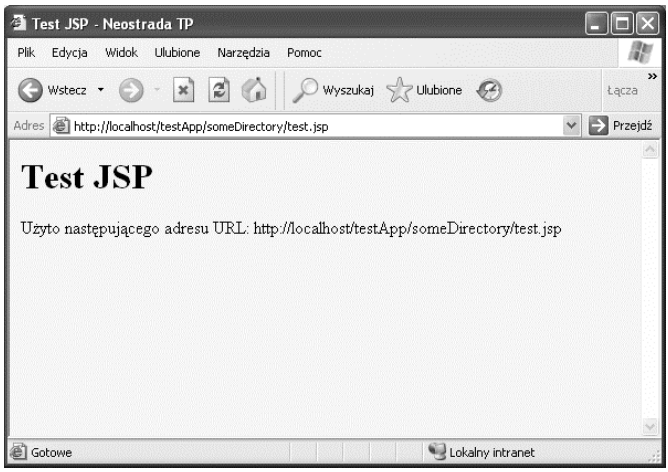
Rysunek 1.5. *Wynik wywołania adresu http://localhost/testApp/test.jsp*



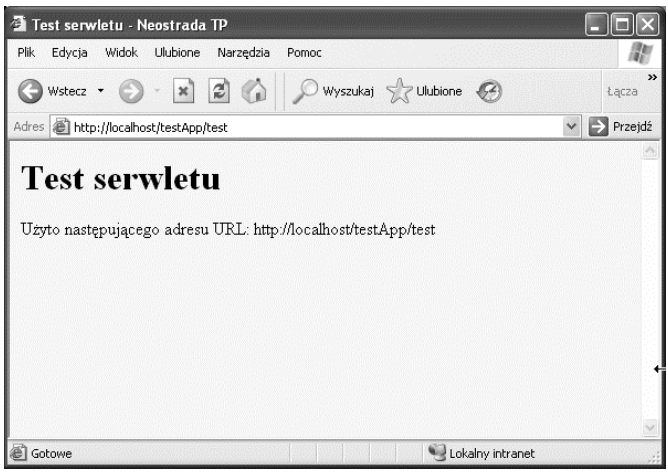
Rysunek 1.6. *Wynik wywołania adresu http://localhost/ ↪testApp/ ↪someDirectory/ ↪test.html*



Rysunek 1.7. *Wynik wywołania adresu http://localhost/ ↪testApp/ ↪someDirectory/ ↪test.jsp*



Rysunek 1.8. *Wynik wywołania adresu http://localhost/ ↪testApp/test*



1.7. Współużytkowanie danych przez aplikacje internetowe

Jednym z głównych celów aplikacji internetowych jest oddzielenie danych od udostępnianych funkcji. Każda aplikacja internetowa utrzymuje własną tabelę sesji oraz własny kontekst serwletów. Ponadto każda aplikacja używa własnego modułu ładowania klas. Dzięki temu wyeliminowano problemy związane z konfliktami nazw, lecz z drugiej strony oznacza to, że do współużytkowania danych między aplikacjami nie można używać metod i pól statycznych. Nadal jednak współużytkowanie danych *jest* możliwe dzięki plikom *cookies* albo przy użyciu obiektów `ServletContext` powiązanych z konkretnymi adresami URL. Obydwa podejścia całkiem dobrze sprawdzają się wówczas, gdy ilość danych wymienianych między aplikacjami jest niewielka, lecz gdy wolumen wymienianych danych będzie duży i wymiana będzie zachodzić stosunkowo często, należy rozważyć możliwość połączenia tych aplikacji w jedną większą aplikację internetową. Poniżej krótko opisano obydwie wymienione powyżej metody współużytkowania danych między aplikacjami.

- **Pliki *cookies*.** Pliki *cookies* są utrzymywane przez przeglądarkę, a nie przez serwer. Dzięki temu *cookies* mogą być współużytkowane przez wiele aplikacji internetowych, jeśli tylko ustawiono je w taki sposób, by dotyczyły dowolnej ścieżki na serwerze. Domyślnie przeglądarka wysyła pliki *cookies* jedynie do tych adresów URL, których prefiks jest taki sam, jak prefiks adresu URL, z którego one pochodzą. Na przykład jeżeli plik *cookie* zostanie przesłany ze strony o adresie *http://komputer/ścieżka1/JakiśPlik.jsp*, wówczas przeglądarka będzie przesyłać *cookie* do stron o adresach *http://komputer/ścieżka1/JakiśInnyPlik.jsp* oraz *http://komputer/ścieżka1/ścieżka2/DowolnyAdres*, natomiast do adresu *http://komputer/ścieżka3/DowolnyAdres cookie* już nie zostanie wysłane. Ponieważ aplikacje internetowe zawsze mają unikatowe prefiksy adresu URL, takie zachowanie przeglądarek spowoduje, że plik *cookie* nigdy nie będzie użytkowany przez dwie różne aplikacje.

  Jednak zgodnie z informacjami zawartymi w rozdziale 8. pierwszego tomu tej książki to standardowe ustawienie można zmienić za pomocą metody `setPath` klasy `Cookie`. Wystarczy wywołać tę metodę z argumentem `"/"`, dzięki czemu przeglądarka zacznie wysyłać pliki *cookie* do *wszystkich* adresów URL znajdujących się na komputerze, z którego otrzymany został oryginalny plik:

  ```
  Cookie c = new Cookie("nazwa", "wartość");
  c.setMaxAge(…);
  c.setPath("/");
  response.addCookie(c);
  ```

- Obiekty `ServletContext` powiązane z określonym adresem URL. W ramach serwletu kontekst serwletu aplikacji internetowej odczytuje się przez wywołanie jego metody `getServletContext` (dziedziczonej z klasy `GenericServlet`). Na stronie JSP używa się natomiast do tego celu predefiniowanej zmiennej `application`. W obydwóch przypadkach zwracaną wartością jest kontekst serwletu powiązany z serwletem lub stroną JSP, z których pochodziło wywołanie. Dodatkowo można również użyć metody `getContext` obiektu `ServletContext`, która zwróci kontekst serwletu

— niekoniecznie własnego — powiązanego ze wskazanym adresem URL. Podejście to przedstawiono w poniższym fragmencie kodu:

```
ServletContext myContext = getServletContext();
String url = "/prefiksAplikacjiWWW";
ServletContext otherContext = myContext.getContext(url);
Object someData = otherContext.getAttribute("jakiśKlucz");
```

Ani jedno, ani drugie przedstawione rozwiązanie nie jest idealne.

Wadą plików *cookie* jest to, że można w nich przechowywać tylko ograniczoną ilość danych. Każda wartość znajdująca się w pliku *cookie* jest ciągiem znaków, a długość każdego ciągu znaków jest ograniczona do czterech kilobajtów. Zatem współużytkowanie większych ilości danych wymaga użycia bazy danych, a w plikach *cookie* przechowywane będą wówczas jedynie wartości pełniące rolę kluczy do tej bazy, a więc do danych rzeczywistych.

Słabą stroną rozwiązania bazującego na kontekstach serwletów jest to, że z góry trzeba w nich znać prefiksy adresów URL używanych przez inne aplikacje internetowe. Zwykle oczekuje się możliwości dowolnego zmieniania prefiksu adresu aplikacji bez konieczności wprowadzania jakichkolwiek zmian w kodzie aplikacji. Używanie metody `getContext` tę elastyczność ogranicza. Drugą wadą jest to, że ze względów bezpieczeństwa na serwerach można blokować dostęp do kontekstów serwletów `ServletContext` niektórych aplikacji internetowych. Gdy coś takiego nastąpi, metoda `getContext` będzie zwracać wartość `null`. Na przykład w niektórych wersjach serwera Tomcat możliwość współużytkowania kontekstów jest domyślnie włączona, natomiast w innych wersjach serwera możliwość tę trzeba jawnie udostępnić. Choćby na serwerze Tomcat w wersji 5.5.7 jako część elementu `Context` w pliku *katalog_tomcat/conf/context.xml* można dodać atrybut `crossContext="true"`, który włącza możliwość współużytkowania kontekstów i czyni ją rozwiązaniem domyślnym dla wszystkich wdrożonych aplikacji. Z kolei całkowite pominięcie atrybutu `crossContext` powoduje, że Tomcat będzie się zachowywał w sposób domyślny, to znaczy współużytkowanie kontekstów `ServletContext` między aplikacjami będzie zabronione.

Obydwie metody współużytkowania danych zaimplementowano w serwletach `SetSharedInfo` oraz `ShowSharedInfo`, których kody źródłowe przedstawiono odpowiednio na listingach 1.5 i 1.6. Obydwa te serwlety zostały odwzorowane na adresy URL w deskryptorze wdrożenia widocznym na listingu 1.7. Serwlet `SetSharedInfo` tworzy własne wpisy w obiekcie sesji, a także kontekst serwletu. Ponadto serwlet ten tworzy dwa pliki *cookie*. Pierwszy z nich zawiera ścieżkę domyślną wskazującą, że *cookie* powinno być stosowane jedynie dla adresów URL z prefiksem identycznym jak URL, z którego nadeszło pierwotne żądanie. Drugi plik *cookie* zawiera ścieżkę w postaci `"/"`, co oznacza, że *cookie* powinno być stosowane do wszystkich adresów URL na komputerze. Na końcu serwlet `SetSharedInfo` przekierowuje klienta do serwletu `ShowSharedInfo`, który z kolei wyświetla nazwy wszystkich atrybutów sesji, wszystkich atrybutów w bieżącym kontekście serwletu, wszystkich atrybutów w kontekście serwletu dotyczącego adresów URL z prefiksem */shareTest1* oraz nazwy wszystkich plików *cookie*.

Listing 1.5. *SetSharedInfo.java*

```
package coreservlets;

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

```
public class SetSharedInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    session.setAttribute("sessionTest", "Sesja - wpis pierwszy");
    ServletContext context = getServletContext();
    context.setAttribute("servletContextTest",
                         "Kontekst serwletu - wpis pierwszy");
    Cookie c1 = new Cookie("cookieTest1", "Cookie numer jeden");
    c1.setMaxAge(3600);      // jedna godzina
    response.addCookie(c1); // ścieżka domyślna
    Cookie c2 = new Cookie("cookieTest2", "Cookie numer dwa");
    c2.setMaxAge(3600);      // jedna godzina
    c2.setPath("/");         // bezpośrednie wyznaczenie ścieżki: wszystkie adresy URL
    response.addCookie(c2);
    String url = request.getContextPath() +
                 "/showSharedInfo";
    // na wypadek, gdyby śledzenie sesji bazowało na przepisywaniu adresów URL
    url = response.encodeRedirectURL(url);
    response.sendRedirect(url);
  }
}
```

Listing 1.6. *ShowSharedInfo.java*

```
package coreservlets;

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

public class ShowSharedInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    response.setCharacterEncoding("UTF-8");
    PrintWriter out = response.getWriter();
    String title = "Dane współużytkowane";
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">" +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI>Sesja:");
    HttpSession session = request.getSession(true);
    Enumeration attributes = session.getAttributeNames();
    out.println(getAttributeList(attributes));
    out.println("  <LI>Bieżący kontekst serwletu:");
    ServletContext application = getServletContext();
    attributes = application.getAttributeNames();
    out.println(getAttributeList(attributes));
    out.println("  <LI>Kontekst serwletu /shareTest1:");
```

```
    application = application.getContext("/shareTest1");
    if (application == null) {
       out.println("Współużytkowanie kontekstów wyłączone");
    } else {
       attributes = application.getAttributeNames();
       out.println(getAttributeList(attributes));
    }
    out.println("  <LI>Pliki cookies:<UL>");
    Cookie[] cookies = request.getCookies();
    if ((cookies == null) || (cookies.length == 0)) {
      out.println("    <LI>Nie znaleziono plików cookie.");
    } else {
      Cookie cookie;
      for(int i=0; i<cookies.length; i++) {
        cookie = cookies[i];
        out.println("    <LI>" + cookie.getName());
      }
    }
    out.println("    </UL>\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }

  private String getAttributeList(Enumeration attributes) {
    StringBuffer list = new StringBuffer("  <UL>\n");
    if (!attributes.hasMoreElements()) {
      list.append("    <LI>Nie znaleziono atrybutów.");
    } else {
      while(attributes.hasMoreElements()) {
        list.append("    <LI>");
        list.append(attributes.nextElement());
        list.append("\n");
      }
    }
    list.append("  </UL>");
    return(list.toString());
  }
}
```

Listing 1.7. *web.xml*

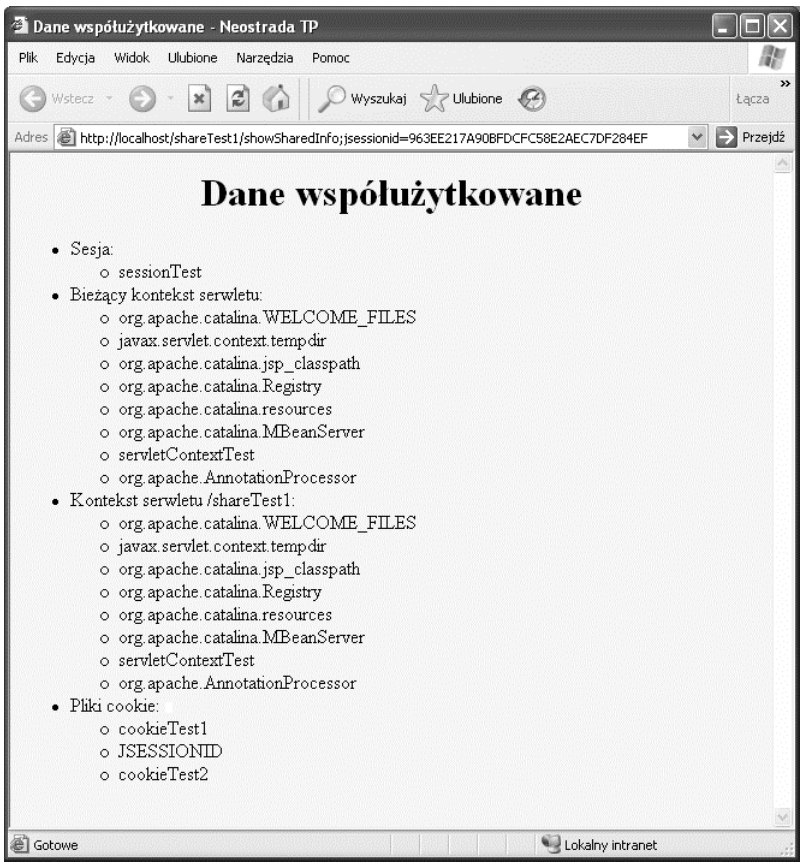
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Plik web.xml z szablonu aplikacji WWW app-blank,
   z witryny http://courses.coreservlets.com/Course-Materials/.
   Zawiera dwa standardowe elementy: welcome-file-list
   oraz servlet-mapping, który wyłącza wywoływacz serwletu.
-->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
           "http://java.sun.com/xml/ns/j2ee
            http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">

  <!-- Tutaj należy umieścić własne wpisy -->
  <servlet>
    <servlet-name>setSharedInfoServlet</servlet-name>
```

```
    <servlet-class>coreservlets.SetSharedInfo</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>setSharedInfoServlet</servlet-name>
    <url-pattern>/setSharedInfo</url-pattern>
  </servlet-mapping>
  <servlet>
    <servlet-name>showSharedInfoServlet</servlet-name>
    <servlet-class>coreservlets.ShowSharedInfo</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>showSharedInfoServlet</servlet-name>
    <url-pattern>/showSharedInfo</url-pattern>
  </servlet-mapping>

  <!-- Wyłączenie wywoływacza serwletu -->
  <servlet>
    <servlet-name>NoInvoker</servlet-name>
    <servlet-class>coreservlets.NoInvokerServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>NoInvoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- Jeżeli URL zawiera katalog bez nazwy pliku, nastąpi próba wywołania pliku index.jsp,
       a następnie index.html. Jeżeli żaden z tych plików nie zostanie znaleziony,
       efekt będzie zależeć od serwera (np. może zostać wyświetlona zawartość katalogu).
  -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

W celu sprawdzenia działania przedstawionej aplikacji należy uruchomić tylko serwlet `SetSharedInfo` z aplikacji *shareTest1*, a po nim wywołać serwlet `ShowSharedInfo` aplikacji *shareTest1* oraz *shareTest2*. Nie należy natomiast wywoływać serwletu `ShowSharedInfo` z aplikacji *shareTest2*, ponieważ nie zilustruje on mechanizmu współużytkowania danych przez te dwie aplikacje.

Na rysunku 1.9 przedstawiono wynik wywołania przez użytkownika serwletów `SetSharedInfo` i `ShowSharedInfo` z aplikacji internetowej, której przypisano prefiks adresu URL */shareTest1*. Dla serwletu `ShowSharedInfo` dostępne są:

- Własny atrybut sesji.
- Atrybuty znajdujące się w domyślnym kontekście serwletu — własny (utworzony jawnie przez serwlet `ShowSharedInfo`) oraz standardowy (utworzony automatycznie przez serwer).
- Atrybuty własny i standardowy, znajdujące się w kontekście serwletu odczytanym przez metodę `getContext("/shareTest1")`. W tym przypadku kontekst serwletu jest tożsamy z domyślnym kontekstem serwletu.
- Dwa jawnie utworzone pliki *cookie* oraz *cookie* utworzone przez system i używane przez mechanizm śledzenia sesji.



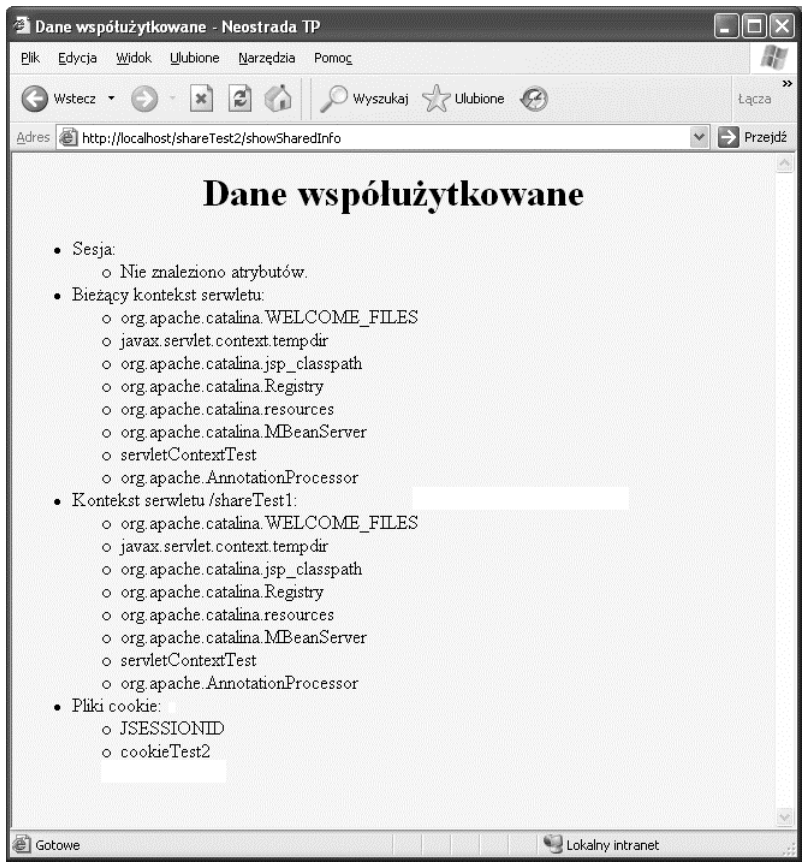
Rysunek 1.9. *Wynik wywołania serwletów SetSharedInfo i ShowSharedInfo z tej samej aplikacji internetowej*

Na rysunku 1.10 z kolei przedstawiono wynik późniejszego wywołania przez użytkownika identycznego serwletu `ShowSharedInfo` zainstalowanego w aplikacji internetowej, której przypisano prefiks adresu URL `/shareTest2`. Dla serwletu dostępne są:

- Standardowe atrybuty zawarte w domyślnym kontekście serwletu.
- Atrybuty własny i standardowy znajdujące się w kontekście serwletu odczytanym przez metodę `getContext("/shareTest1")`; w tym przypadku jest to kontekst inny niż domyślny kontekst serwletu.
- Dwa pliki *cookie*: jeden utworzony jawnie, którego ścieżką jest `"/"`, oraz drugi, utworzony przez system, używany przez mechanizm śledzenia sesji (w nim również używana jest własna ścieżka `"/"`).

Z kolei następujące elementy są dla serwletu *niewidoczne*:

- Atrybuty znajdujące się w obiekcie sesji.
- Własne atrybuty znajdujące się w domyślnym kontekście serwletu.
- Jawnie utworzone pliki *cookie*, w których użyto ścieżki domyślnej.



Rysunek 1.10. *Wynik wywołania serwletu SetSharedInfo w jednej aplikacji internetowej oraz serwletu ShowSharedInfo z drugiej aplikacji internetowej*