Head First Servlets & JSP. Edycja polska. Wydanie II

Autor: Bryan Basham,
Kathy Sierra, Bert Bates
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 978-83-246-1814-9
Tytuł oryginału: Head First Servlets and JSP, 2 ed.
Format: 200x230, stron: 920

Wykorzystaj innowacyjne metody nauki i zacznij tworzyć dynamiczne aplikacje internetowe!

- Jak korzystać z technologii JSP?
- Jak działają serwlety?
- Jak zastosować wzorzec projektowy MCV?

Statyczna strona internetowa dziś już nikogo nie zachwyca. Czas na zmianę! Pora sprawić, aby tworzone przez Ciebie aplikacje stały się wyjątkowo dynamiczne, elastyczne i interaktywne. Poznaj i wykorzystaj w tym celu nowoczesną technologię serwletów i stron JSP. Dzięki niej zbudujesz złożony serwis z zastosowaniem języka Java wplecionego w kod HTML danej strony oraz serwletu, który przetwarza informacje otrzymane z serwera, aby dostarczyć użytkownikowi gotowy obraz witryny. Aby Twoja aplikacja działała idealnie, potrzebujesz jeszcze tylko zabezpieczeń, kontenerów serwletów i modelu MCV. Jak to wszystko złożyć i uruchomić? Tego dowiesz się właśnie z naszego nowatorskiego podręcznika!

Książka „Head First Servlets & JSP. Edycja polska. Wydanie II" napisana została w oparciu o innowacyjne metody przekazywania wiedzy, polegające na włączaniu w proces nauki elementów zabawy oraz oddziaływaniu na wiele zmysłów. To pomaga szybko zrozumieć tę technologię i osiągnąć biegłość w tworzeniu dynamicznych stron oraz aplikacji. Z bogato ilustrowanego i napisanego lekkim językiem podręcznika dowiesz się m.in., co to jest serwlet, kontener i protokół HTTP, do czego służą strony JSP oraz jak tworzyć środowisko wdrożeniowe aplikacji i zabezpieczać przesyłane informacje. Architektura aplikacji internetowej nie będzie już miała przed Tobą żadnych tajemnic!

- Przegląd technologii serwletów i stron JSP
- Architektura aplikacji internetowej
- Wdrażanie aplikacji internetowej
- Strony bezskryptowe
- Znaczniki niestandardowe
- Zabezpieczenia serwletów
- Filtry
- Korporacyjne wzorce projektowe
- Protokół HTTPS

Książka przygotowuje do najnowszej wersji egzaminu SCWCD z platformy J2EE 1.5!

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

Spis treści (skrócony)

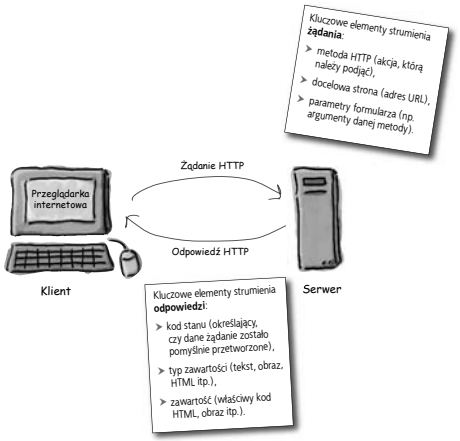
Spis treści

Wprowadzenie

Twój mózg koncentruje się na serwletach. W tym rozdziale *Ty* próbujesz się czegoś *nauczyć*, a Twój *mózg* robi Ci przysługę i nie przykłada się do *zapamiętywania* zdobytej wiedzy. Twój mózg myśli sobie: „Lepiej zachowam miejsce na bardziej istotne informacje, na przykład: jakich dzikich zwierząt należy unikać bądź czy jazda nago na snowboardzie jest dobrym pomysłem". Jak w takim razie można przekonać swój mózg, że nasze życie zależy od opanowania serwletów?

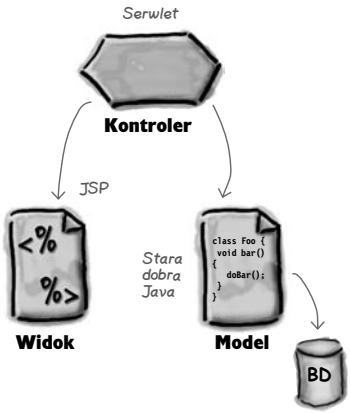
1 Do czego służą serwlety i strony JSP?

Aplikacje internetowe są super. Ile tradycyjnych aplikacji z graficznym interfejsem użytkownika używanych przez miliony osób na całym świecie potrafisz wymienić? Jako programista aplikacji internetowych możesz uwolnić się od problemów wdrażania będących udziałem wszystkich standardowych aplikacji i udostępniać swoje aplikacje każdemu, kto dysponuje przeglądarką internetową. Będziesz jednak potrzebował serwletów i stron JSP. Będziesz ich potrzebował, ponieważ zwykłe, statyczne strony HTML były dobre... w latach 90. ubiegłego wieku. Zatem dowiedz się, jak przekształcić *witrynę WWW* w *aplikację internetową*.



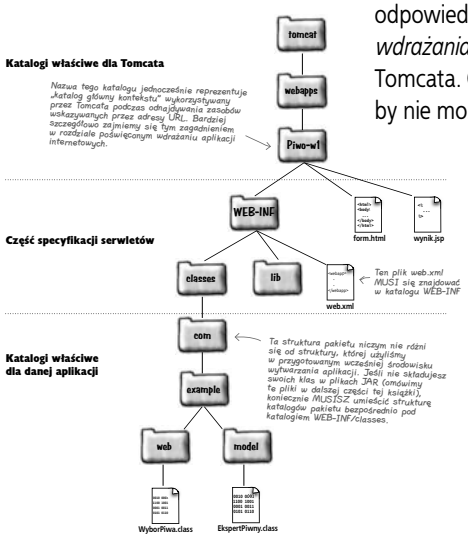
2 Architektura aplikacji internetowej

Serwlety potrzebują pomocy. Kiedy do naszej aplikacji dociera żądanie, ktoś musi utworzyć obiekt serwletu lub przynajmniej wątek, który to żądanie obsłuży. Ktoś musi wywołać metodę `doPost()` lub `doGet()` serwletu. Ktoś musi przekazać żądanie do serwletu oraz odebrać to, co serwlet wygeneruje w odpowiedzi. Ktoś musi decydować o życiu, śmierci i zasobach niezbędnych do pracy serwletu. W tym rozdziale przyjrzymy się koncepcji kontenera i po raz pierwszy zwrócimy uwagę na wzorzec projektowy MVC.



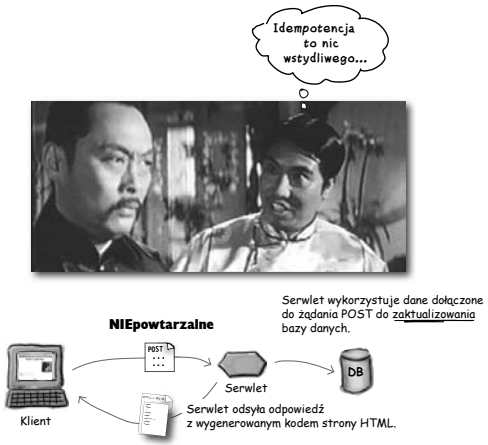
3 Minipodręcznik MVC

Tworzenie i wdrażanie aplikacji internetowych MVC. Nadszedł czas, aby utrudzić nasze dłonie pisaniem formularzy HTML, kontrolerów serwletów, modeli (zwykłych, tradycyjnych klas Javy), deskryptorów wdrożenia w formacie XML oraz widoków opartych na stronach JSP. Najwyższa pora zbudować, wdrożyć i przetestować taką aplikację. Najpierw jednak musimy przygotować odpowiednie środowisko *wytwarzania* aplikacji. Następnie musimy przygotować środowisko *wdrażania*, postępując przy tym zgodnie ze specyfikacją serwletów i JSP oraz wymaganiami Tomcata. Owszem... tworzymy małą aplikację, jednak niemal żadna aplikacja nie jest na tyle mała, by nie można w niej było wykorzystać wzorca MVC.



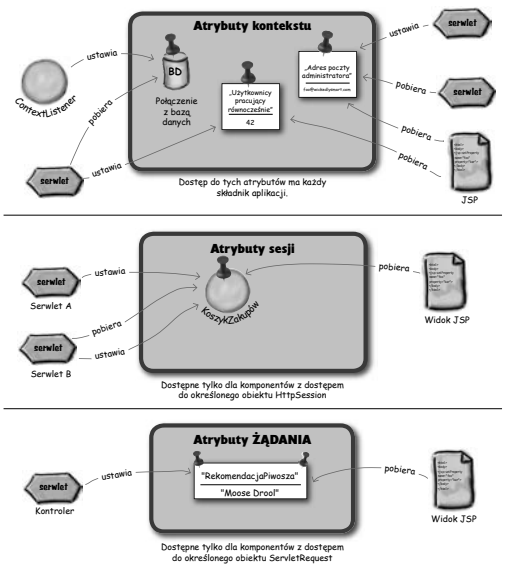
4 Być serwletem

Serwlety potrzebują pomocy. Zadaniem serwletu jest obsługa ***żądań*** klientów i odsyłanie do klienta właściwych ***odpowiedzi***. Żądanie może być zupełnie proste, np. *prześlij mi stronę powitalną*, lub znacznie bardziej skomplikowane, np. *wygeneruj zamówienie na podstawie zawartości mojego koszyka*. **Żądanie** obejmuje kluczowe dane, a kod Twojego serwletu musi wiedzieć, jak należy te dane *odszukać* i jak ich *użyć*. Co więcej, kod serwletu musi wiedzieć, jak odesłać **odpowiedź**. A jeśli *nie*...



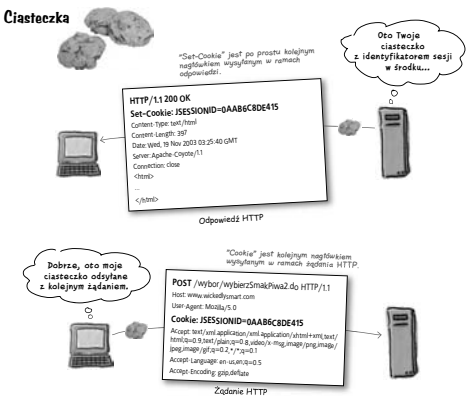
5 Być aplikacją internetową

Żaden serwlet nie działa samodzielnie. We współczesnych aplikacjach internetowych osiąganie zamierzonego celu jest możliwe dzięki współpracy wielu komponentów. Stosujemy komponenty modelu, widoku oraz kontrolera. Wykorzystujemy także rozmaite klasy pomocnicze. Jednak w jaki sposób należy łączyć wszystkie te elementy, aby tworzyły jedną aplikację internetową? W jaki sposób komponenty mogą korzystać z tych samych informacji? Jak *ukrywać* pewne informacje? *Jak zapewniać bezpieczeństwo informacji podczas przetwarzania wielowątkowego?* Od odpowiedzi na te pytania może zależeć Twoja praca.



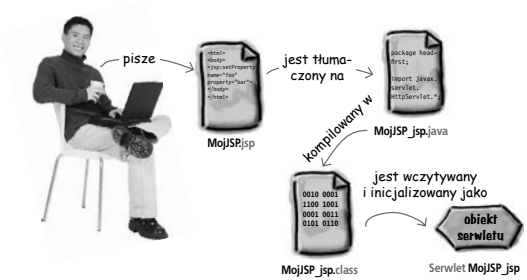
6 Stan konwersacyjny

Serwery WWW nie mają pamięci krótkotrwałej. Zaraz po odesłaniu do nas odpowiedzi serwery WWW zapominają, kim jesteśmy. Kiedy wysyłamy kolejne żądanie, docelowy serwer WWW już nas nie rozpoznaje. Innymi słowy, serwery WWW nie pamiętają ani tego, czego żądaliśmy w przeszłości, ani tego, co do nas wysłały w ramach odpowiedzi. Zupełnie nic! Jednak czasami przechowywanie informacji o stanie konwersacji z klientem i korzystanie z nich podczas obsługi *wielu żądań* jest konieczne. Koszyk w sklepie internetowym nie mógłby działać, gdyby użytkownik musiał wybierać wszystkie towary i realizować zamówienie w ramach *jednego żądania*.



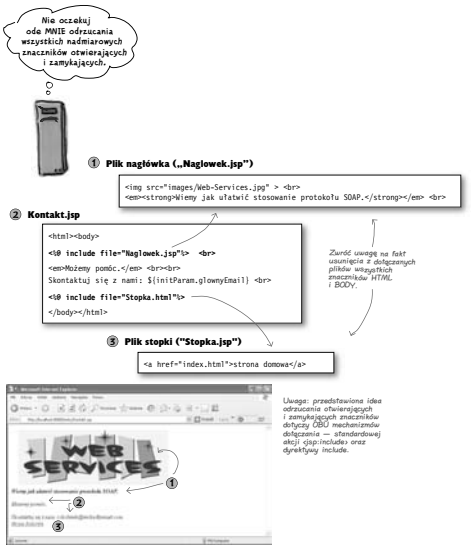
7 Być stroną JSP

Strona JSP staje się serwletem. Serwletem, którego *nie* musisz tworzyć. Kontener przegląda kod Twojej strony JSP, tłumaczy go na kod źródłowy języka programowania Java i kompiluje tak przetłumaczony kod na postać pełnowartościowej klasy serwletu Javy. Warto jednak wiedzieć, co dzieje się w czasie konwertowania Twojego kodu strony JSP na kod Javy. W ramach kodu JSP *można* co prawda umieszczać kod Javy, ale czy na pewno powinniśmy to robić? A jeśli w kodzie stron JSP nie umieścimy żadnych wyrażeń Javy, co *znajdzie* się w kodzie naszego serwletu? Przyjrzymy się sześciu rodzajom elementów JSP — z których każdy ma swoje przeznaczenie i, niestety, *odmienną składnię*. Dowiesz się, co i dlaczego można umieszczać w kodzie stron JSP. Dowiesz się także, czego *nie* powinieneś umieszczać w tym kodzie.



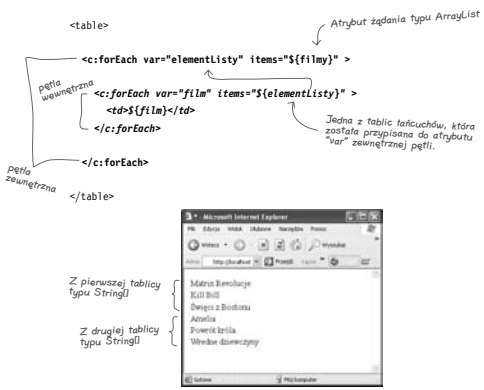
8 Strony bezskryptowe

Porzuć skrypty. Czy współpracujący z Tobą projektanci stron internetowych naprawdę muszą znać Javę? Czy sami oczekują od programistów Javy, aby byli jednocześnie np. grafikami? A jeśli nawet przyjmiemy, że jesteś *jedynym* członkiem zespołu, czy rzeczywiście chciałbyś umieszczać rozbudowane fragmenty kodu Javy w swoich stronach JSP? Czyż nie nasuwa Ci się określenie „koszmar konserwacji oprogramowania"? Pisanie stron bezskryptowych jest nie tylko możliwe, ale stało się znacznie *prostsze* i bardziej elastyczne dzięki nowej specyfikacji JSP 2.0, a przede wszystkim wskutek wprowadzenia nowego języka wyrażeń (EL). Ponieważ język EL bazuje na językach JavaScript i XPATH, projektanci stron mogą go stosować bez najmniejszych problemów; zresztą także Ty go polubisz (kiedy już się do niego przyzwyczaisz). Jednak EL stwarza też pewne pułapki. Jego wyrażenia *wyglądają* co prawda jak wyrażenia Javy, jednak w rzeczywistości nimi nie są. Niektóre wyrażenia EL działają nawet inaczej niż wyrażenia Javy o identycznej składni, zatem warto mieć się na baczności!



9 Potęga znaczników niestandardowych

Czasami potrzebujemy czegoś więcej niż tylko języka wyrażeń (EL) i akcji standardowych. Co będzie, jeśli zechcesz użyć pętli do przeszukania danych składowanych tablicy i wyświetlenia po jednym elemencie w każdym wierszu generowanej dynamicznie tabeli HTML? Oczywiście doskonale *zdajesz* sobie sprawę z możliwości błyskawicznego skonstruowania odpowiedniej pętli w skryptlecie. Z drugiej strony, staramy się nie używać kodu skryptowego. To żaden problem. Kiedy język wyrażeń (EL) i akcje standardowe okazują się niewystarczające, zawsze można wykorzystać *znaczniki niestandardowe*. Ich stosowanie w kodzie stron JSP jest równie proste jak korzystanie z akcji standardowych. Co więcej, znaczniki, których najprawdopodobniej będziesz potrzebował, ktoś już napisał i umieścił w standardowej bibliotece znaczników JSP (ang. *JSP Standard Tag Library*, w skrócie *JSTL*). W *tym* rozdziale dowiesz się, jak *używać* znaczników niestandardowych, a w następnym — jak tworzyć własne znaczniki.

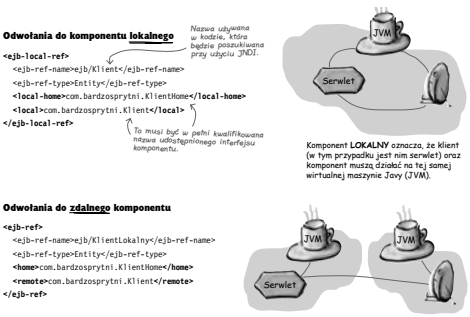


10 Kiedy JSTL nie wystarcza...

Czasami JSTL i standardowe akcje nie wystarczają. Kiedy trzeba zrobić coś niestandardowego, a nie chcesz uciekać się do stosowania skryptu, możesz stworzyć *własne* procedury obsługi znaczników. Dzięki temu projektanci stron będą mogli używać w projektowanych stronach Twoich *znaczników*, a całą „czarną robotę" będzie, w niewidoczny sposób, realizować Twoja *klasa* obsługująca. Z drugiej strony, takie rozwiązanie wymaga od Ciebie sporo nauki, ponieważ własne, niestandardowe znaczniki można tworzyć na aż trzy różne sposoby. Dwa spośród nich (chodzi o *znaczniki proste* oraz *pliki znaczników*) wprowadzono z myślą o naszej wygodzie dopiero w specyfikacji JSP 2.0.

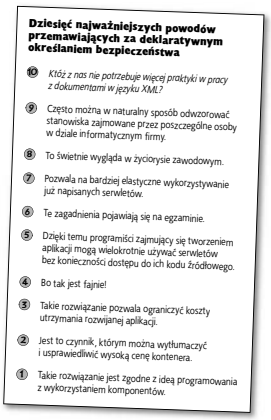
11 Wdrażanie aplikacji internetowych

W końcu Twoja aplikacja jest gotowa. Strony zostały dopracowane w najdrobniejszych szczegółach, kod jest przetestowany i zoptymalizowany, a termin... minął dwa tygodnie temu. Ale gdzie to wszystko należy umieścić? Jest tyle różnych katalogów, tyle niezrozumiałych reguł. Jak *powinieneś* nazwać swoje katalogi? Jakich nazw użyłbyś *klient*? Do jakich zasobów tak naprawdę będzie się odwoływać klient i skąd kontener ma wiedzieć, gdzie należy ich szukać?



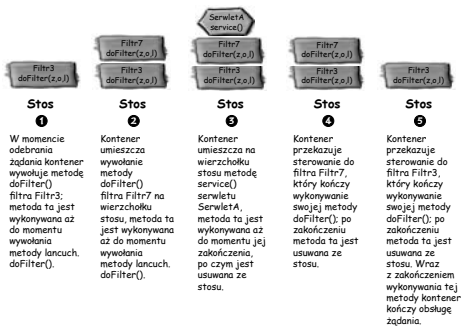
12 Zachowaj to w tajemnicy, ukryj w bezpiecznym miejscu

Twoja aplikacja internetowa jest w *niebezpieczeństwie*. Problemy czyhają w każdym zakamarku sieci. Nie chcesz chyba, aby ci źli faceci podsłuchiwali transakcje realizowane w Twoim sklepie internetowym i przechwytywali podawane numery kart kredytowych? Nie chcesz też, by byli w stanie przekonać Twój serwer, iż tak naprawdę są Bardzo Ważnymi Klientami Liczącymi Na Bardzo Duże Upusty. I w końcu nie chcesz, by *ktokolwiek* (niezależnie od zamiarów) miał dostęp do poufnych informacji o pracownikach. Czy Janek z działu marketingu naprawdę musi wiedzieć, że Lucyna z działu technicznego zarabia trzy razy więcej od niego?



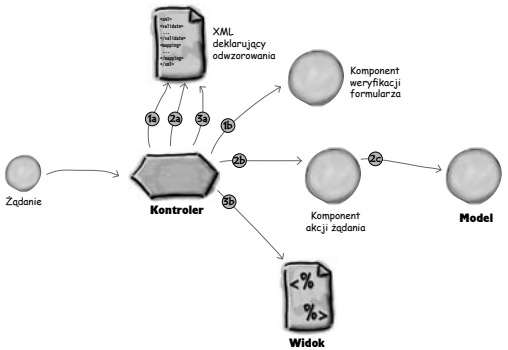
13 Potęga filtrów

Filtry umożliwiają przechwytywanie żądań. A skoro można przechwycić *żądanie*, można także kontrolować *odpowiedź*. Ale najlepsze w tym wszystkim jest to, że serwlet nie ma o tym **najmniejszego pojęcia**. Serwlet nigdy nie wie, czy coś się zdarzyło w czasie dzielącym odebranie żądania przez kontener od wywołania metody `service()` serwletu. Co to oznacza dla Ciebie? Dłuższe wakacje. Ponieważ czas, który musiałbyś poświęcić na modyfikowanie tylko *jednego* z istniejących serwletów, możesz poświęcić na napisanie i skonfigurowanie filtra, który będzie miał wpływ na *wszystkie* Twoje serwlety. Może chciałbyś dodać opcję śledzenia żądań we *wszystkich* serwletach tworzących aplikację? Nie ma żadnego problemu. A może chciałbyś w określony sposób modyfikować wyniki generowane przez wszystkie *serwlety* wchodzące w skład aplikacji? To także żaden problem. A co najlepsze — nie musisz przy tym w *żaden sposób* modyfikować kodu serwletów.



14 Korporacyjne wzorce projektowe

Ktoś to już wcześniej zrobił. Jeśli właśnie zaczynasz tworzyć aplikacje internetowe w języku Java, masz dużo szczęścia. Możesz czerpać z wiedzy dziesiątek tysięcy programistów, którzy od dawna się tym zajmują i mają już swoje firmowe koszulki. Wykorzystując wzorce projektowe, zarówno te związane z platformą J2EE, jak i wszelkie *inne*, możesz uprościć swój kod i swoje życie. W świecie aplikacji internetowych najważniejszym wzorcem projektowym jest MVC, który zastosowano między innymi w bardzo popularnym frameworku Struts (stworzonym z myślą o wsparciu programistów tworzących elastyczne i łatwe w utrzymaniu kontrolery frontonów serwletów). Wykorzystanie pracy *innych* jesteś winien samemu sobie, dzięki temu będziesz mógł poświęcić więcej czasu na ważniejsze sprawy.



A BAR KAWOWY

Końcowy Egzamin Próbny Baru Kawowego. To jest to. 69 pytań. Charakter, zagadnienia i poziom trudności jest niemal taki sam jak na *prawdziwym egzaminie. Możesz nam wierzyć.*

S Skorowidz

Architektura aplikacji internetowej

Serwlety potrzebują pomocy. Kiedy żądanie HTTP dociera do serwera WWW, ktoś musi jeszcze stworzyć egzemplarz serwletu lub przynajmniej utworzyć nowy wątek obsługujący to żądanie. Ktoś musi przecież wywołać metodę `doPost()` lub `doGet()` serwletu. Warto też pamiętać o otrzymywaniu przez wspomniane metody dwóch kluczowych argumentów — obiektów reprezentujących żądanie i odpowiedź protokołu HTTP. Ktoś te obiekty musi oczywiście przekazać do serwletu. Ktoś musi zarządzać życiem, śmiercią i zasobami serwletu. Tym kimś jest kontener (ang. *container*). W tym rozdziale przyjrzymy się zasadom funkcjonowania aplikacji internetowych w ramach kontenerów i rzucimy okiem na strukturę aplikacji internetowych budowanych z wykorzystaniem wzorca projektowego MVC (ang. *Model View Controller*).

Wysokopoziomowa architektura aplikacji internetowych

1.1. Dla każdej z metod przesyłania żądań protokołu HTTP (takich jak GET, POST, HEAD itp.) opisz jej przeznaczenie i charakterystyki techniczne, wymień czynniki, które mogą decydować o wyborze danej metody przez klienta (zazwyczaj przeglądarkę internetową), oraz zidentyfikuj metodę `HttpServlet`, która odpowiada danej metodzie protokołu HTTP.

1.4. Opisz znaczenie i sekwencję zdarzeń składających się na cykl życia serwletu: (1) załadowanie klasy serwletu, (2) konkretyzacja serwletu, (3) wywołanie metody `init()`, (4) wywołanie metody `service()` oraz (5) wywołanie metody `destroy()`.

2.1. Skonstruuj strukturę plików i katalogów dla aplikacji internetowej zawierającej (a) statyczną treść, (b) strony JSP, (c) klasy serwletów, (d) deskryptor wdrożenia, (e) biblioteki znaczników, (f) pliki JAR oraz (g) pliki klas Javy. Opisz techniki ochrony plików zasobów przed dostępem za pośrednictwem protokołu HTTP.

2.2. Opisz przeznaczenie i semantykę każdego z wymienionych dalej elementów deskryptora wdrożenia: egzemplarz serwletu, nazwa serwletu, klasa serwletu, parametry inicjalizacji serwletu, adres URL nazwanego odwzorowania serwletu.

Uwagi wyjaśniające:

Wszystkie cele wymienione w tym podrozdziale zostaną dogłębnie przeanalizowane w pozostałych rozdziałach, zatem niniejszy rozdział należy traktować jak pierwsze spojrzenie na podstawy tego, czym szczegółowo zajmiemy się w kolejnych częściach. Innymi słowy, nie powinieneś się przejmować, jeśli po przeczytaniu tego rozdziału nie będziesz potrafił odpowiedzieć na postawione tutaj pytania (lub jeśli nie będziesz tych pytań nawet pamiętał).

Na końcu tego rozdziału nie będziemy analizowali żadnych przykładowych pytań egzaminacyjnych, ponieważ ich analiza będzie możliwa dopiero po zapoznaniu się z bardziej szczegółowym materiałem zawartym w kolejnych rozdziałach.

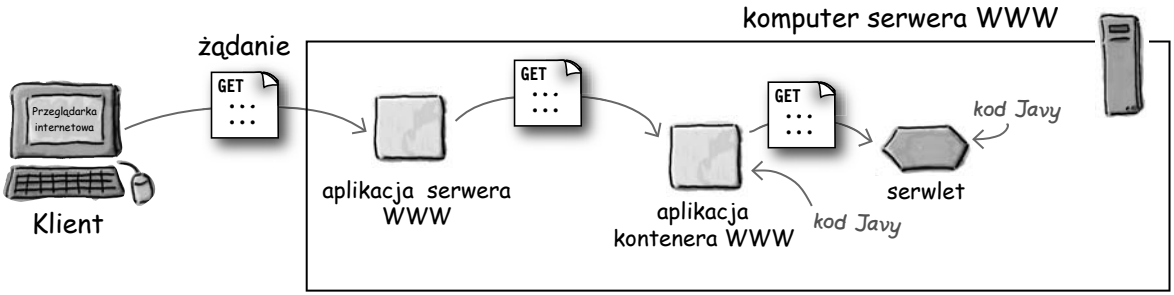
Ciesz się z prostego, wprowadzającego materiału, póki możesz!

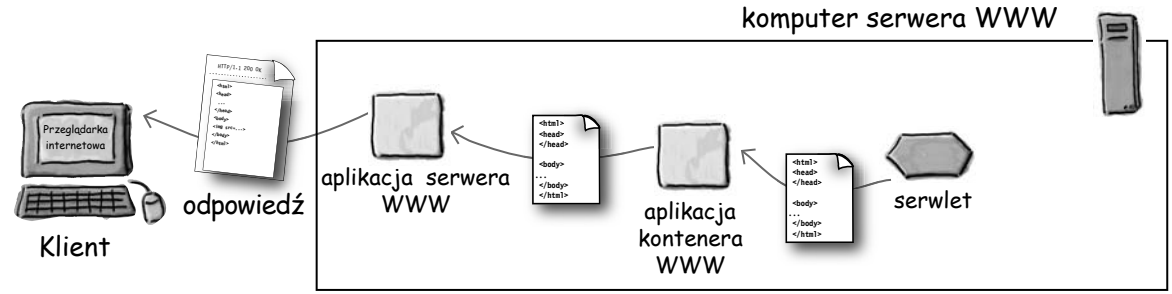
PAMIĘTAJ JEDNAK, że prezentowane w tym rozdziale zagadnienia będą nam potrzebne w kolejnych rozdziałach. Jeśli masz już pewne doświadczenie w kwestii serwletów, prawdopodobnie możesz ten rozdział jedynie przekartkować (najlepiej analizując tylko najważniejsze rysunki i wykonując ćwiczenia) i przejść do rozdziału 3.

Czym jest kontener?

Serwlety nie zawierają metody main(). Serwlety są kontrolowane przez inną aplikację Javy nazywaną *kontenerem*.

Przykładem kontenera (ang. *container*) jest Tomcat. Kiedy nasza aplikacja serwera WWW (np. Apache) otrzymuje żądanie dotyczące *serwletu* (w przeciwieństwie np. do przestarzałej, płaskiej, statycznej strony HTML), serwer nie przekazuje tego żądania bezpośrednio do wskazywanego serwletu, tylko do kontenera, w którym ten serwlet wcześniej *umieszczono*. Od tej pory to kontener odpowiada za przekazanie do serwletu obiektów żądania i odpowiedzi protokołu HTTP oraz za wywołanie właściwych metod serwletu (takich jak `doPost()` lub `doGet()`).





Co zrobić, jeśli mam odpowiedni kod Javy, ale nie mam ani serwletów, ani kontenerów?

Jakie rozwiązanie należy zastosować w sytuacji, gdy musimy napisać program Javy, który ma obsługiwać dynamiczne żądania przychodzące do aplikacji serwera WWW (np. Apache'a), ale bez dodatkowego kontenera (takiego jak wspominany już Tomcat)? Innymi słowy, wyobraźmy sobie, że nie istnieje pojęcie serwletu i że dysponujemy jedynie standardowymi bibliotekami J2SE. W takim przypadku należy oczywiście przyjąć, że mamy możliwość takiego skonfigurowania aplikacji serwera WWW, by wywoływał naszą aplikację napisaną w Javie. Stosowanie takiego rozwiązania byłoby uzasadnione, gdyby nie była nam znana koncepcja kontenera. Wystarczy sobie wyobrazić sytuację, w której musimy stworzyć aplikację internetową, dysponując tylko klasyczną Javą.

Wymień kilka funkcji, które musielibyśmy zaimplementować w aplikacji J2SE, gdybyśmy nie mogli korzystać z aplikacja kontenera WWW:

* *Stwórz gniazdo połączenia z serwerem i odpowiedni obiekt nasłuchujący (odbiornik) dla nowego gniazda.*

Możliwe odpowiedzi: stwórz obiekt zarządzający wątkami, zaimplementuj techniki zabezpieczeń, coś do filtrowania takich elementów jak zapisy dziennika, obsługa JSP (o Boże!), zarządzanie pamięcią…

Co daje nam kontener?

Wiemy już, że to właśnie kontener uruchamia serwlety i nimi zarządza, ale właściwie *dlaczego* tak się dzieje? Czy warto wprowadzać dodatkową warstwę i godzić się na związane z tym opóźnienia?

Obsługa komunikacji Kontener zapewnia prosty mechanizm komunikacji pomiędzy naszymi serwletami a serwerem WWW. Dzięki kontenerom nie musimy budować gniazd serwera, nasłuchiwać komunikacji na porcie, tworzyć strumieni itp. Kontener zna odpowiedni protokół porozumiewania się z serwerem WWW, zatem nasze serwlety nie muszą się martwić o zapewnienie interfejsu API pomiędzy np. serwerem Apache a kodem naszej aplikacji internetowej. W tej sytuacji programista aplikacji internetowej musi jedynie zadbać o logikę biznesową umieszczoną w serwlecie (odpowiedzialną na przykład za przyjmowanie i przetwarzanie zamówień składanych w sklepie internetowym).

Zarządzanie cyklem życia Kontener jest panem życia i śmierci naszych serwletów. Właśnie kontener odpowiada za wczytywanie niezbędnych klas, tworzenie egzemplarzy i inicjalizację serwletów, wywoływanie ich metod oraz przystosowywanie serwletów do współpracy z mechanizmami odśmiecania pamięci. Dzięki kontenerom *nie* musimy tracić naszego cennego czasu na zarządzanie zasobami.

Obsługa wielowątkowości Kontener automatycznie tworzy nowy wątek Javy dla każdego otrzymanego żądania dotyczącego serwletu. Kiedy serwlet kończy wykonywanie metody obsługującej przysłane przez klienta żądanie HTTP, odpowiedni wątek jest zamykany (zabijany). Nie oznacza to jednak, że programista jest zwolniony z obowiązku zapewniania bezpieczeństwa wątków — nadal musi pamiętać o ich właściwej synchronizacji. Z drugiej strony, samo przejęcie przez serwer odpowiedzialności za tworzenie i zarządzanie wątkami obsługującymi równocześnie wiele żądań pozwala mu zaoszczędzić mnóstwo pracy.

Bezpieczeństwo deklaratywne Korzystanie z kontenera wiąże się ze stosowaniem deskryptora wdrożenia (w formacie XML), który konfiguruje (i modyfikuje) mechanizmy zabezpieczeń bez konieczności trwałego umieszczania odpowiednich instrukcji w kodzie klasy serwletu (lub kodzie dowolnej innej klasy Javy). Aż trudno w to uwierzyć! Możesz zarządzać i wprowadzać zmiany w zabezpieczeniach bez konieczności modyfikowania i ponownego kompilowania swojego kodu źródłowego Javy.

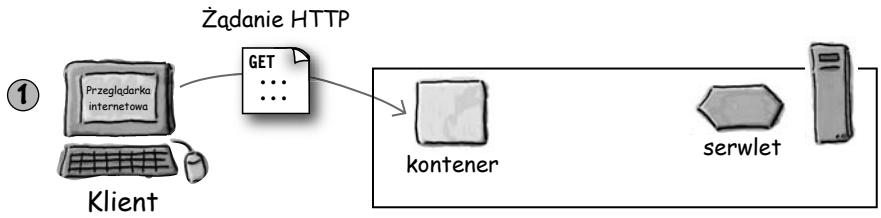
Obsługa JSP Wiesz już, jak wygodne są strony JSP. Kto Twoim zdaniem zajmuje się tłumaczeniem kodu JSP do postaci właściwego kodu języka programowania Java? To oczywiste — *kontener*.

Dzięki kontenerowi MOŻESZ się w większym stopniu skoncentrować na własnej logice biznesowej, zamiast martwić się pisaniem kodu zarządzającego wątkami, zapewniającego bezpieczeństwo oraz obsługującego komunikację sieciową.

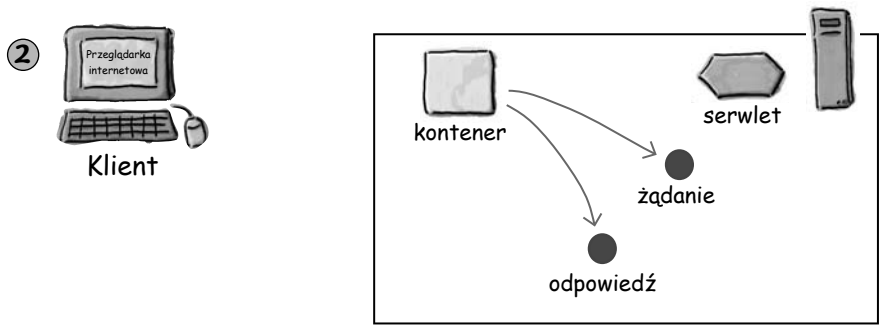
Możesz skupić całą swoją energię na opracowaniu bajecznego sklepu internetowego z folią z bąbelkami i pozostawić usługi pracujące w tle (w tym zapewnianie bezpieczeństwa oraz przetwarzanie stron JSP) kontenerowi.

Jak kontener obsługuje żądanie HTTP?

Co prawda najsmakowitsze kąski pozostawimy sobie na kolejne rozdziały tej książki, jednak warto już teraz dokonać krótkiego przeglądu sposobu funkcjonowania kontenerów:

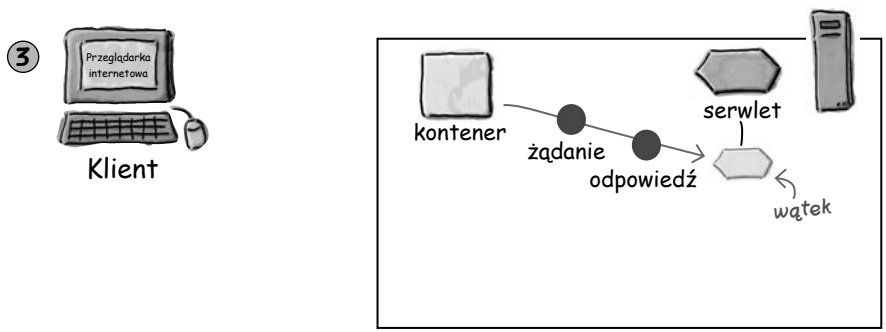


Użytkownik klika łącze zawierające adres URL serwletu (zamiast adresu URL wskazującego na statyczną stronę WWW).

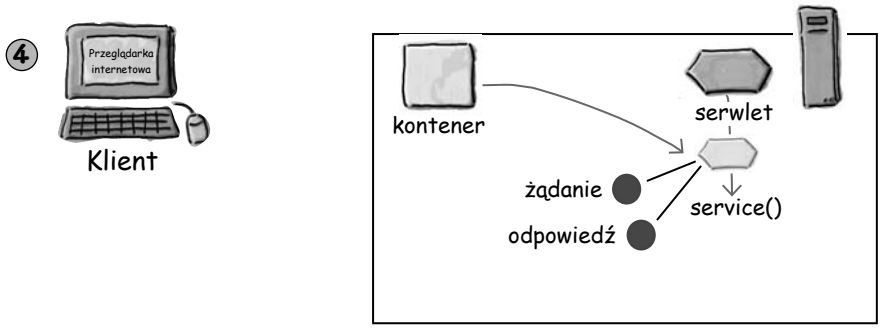


Kontener „widzi", że żądanie jest kierowane do serwletu, zatem tworzy dwa niezbędne obiekty:

1. HttpServletResponse
2. HttpServletRequest

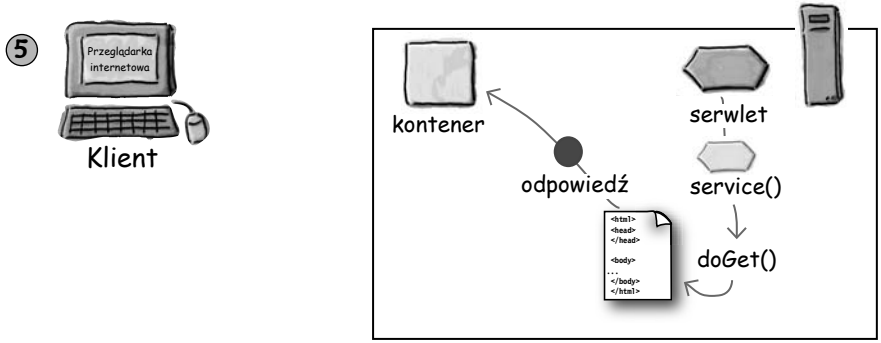


Kontener odnajduje odpowiedni serwlet na podstawie adresu URL dołączonego do żądania, tworzy lub przydziela pamięć dla wątku obsługującego to żądanie oraz przekazuje do nowego wątku serwletu obiekty reprezentujące żądanie i odpowiedź.

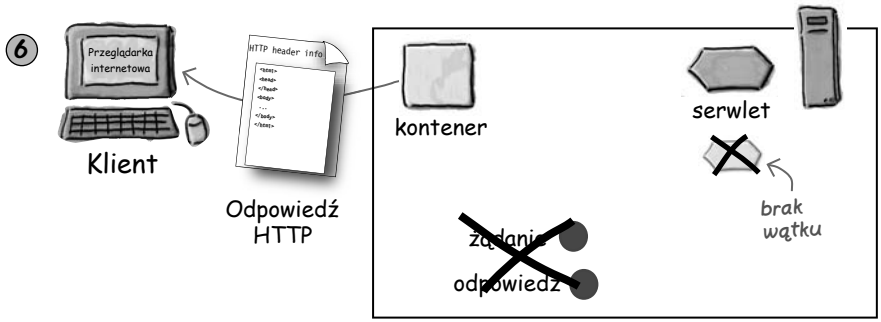


Kontener wywołuje metodę `service()` serwletu. W zależności od typu żądania metoda `service()` wywołuje albo metodę `doGet()`, albo metodę `doPost()`.

W tym przypadku zakładamy, że do kontenera dotarło żądanie protokołu HTTP typu GET.



Metoda `doGet()` generuje dynamiczną stronę HTML i umieszcza ją w obiekcie odpowiedzi. Pamiętaj, że kontener cały czas utrzymuje referencję do obiektu odpowiedzi!



Działanie wątku się kończy, kontener przekształca obiekt odpowiedzi w odpowiedź protokołu HTTP, odsyła tę odpowiedź do klienta, po czym usuwa obiekty żądania i odpowiedzi.

Jak to wszystko wygląda w kodzie (co sprawia, że serwlet jest serwletem)?

W świecie rzeczywistym 99,9% wszystkich serwletów nadpisuje metodę doGet() bądź metodę doPost().

Zwróć uwagę na brak metody main(). Metody związane z cyklem życia serwletu (w tym doGet()) są wywoływane przez kontener.

99,9999% serwletów dziedziczy właśnie po klasie HttpServlet.

W tym miejscu nasz serwlet otrzymuje referencje do utworzonych przez kontener obiektów reprezentujących żądanie i odpowiedź.

Obiekt klasy PrintWriter możemy uzyskać za pośrednictwem obiektu odpowiedzi przekazanego naszemu serwletowi przez kontener. Obiekt PrintWriter służy do zapisywania w obiekcie odpowiedzi tekstu HTML (można też użyć innych obiektów umożliwiających zapisywanie zamiast tekstu HTML, np. obrazów).

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

public class Ch2Servlet extends HttpServlet {

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

    PrintWriter out = response.getWriter();
    java.util.Date dzisiaj = new java.util.Date();
    out.println("<html> " +
                "<body> " +
                "<h1 style=\"text-align:center>" +
                "Nasz drugi serwlet: Ch2Servlet</h1>" +
                "<br>" + dzisiaj +
                "</body>" +
                "</html>");
  }
}
```

Nie ma niemądrych pytań

P: Pamiętam o metodach `doGet()` oraz `doPost()`, ale na poprzedniej stronie była przecież mowa o metodzie `service()`. Skąd się wzięła ta metoda?

O: Twój serwlet odziedziczył ją po klasie HttpServlet, która z kolei odziedziczyła tę metodę po klasie GenericServlet, która odziedziczyła ją po... zresztą, hierarchię klas opiszemy wyczerpująco w rozdziale „Być serwletem", więc musisz się wykazać odrobiną cierpliwości.

P: Do tej pory unikałeś wyjaśnienia, skąd kontener wie, gdzie *szukać* właściwego serwletu i jak adres URL jest odwzorowywany na odpowiedni serwlet. Czy użytkownik musi podawać kompletną ścieżkę i nazwę pliku klasy serwletu?

O: Dobre pytanie, ale odpowiedź brzmi „nie". Poruszyłeś jednak bardzo istotny problem (odwzorowywania serwletu oraz wzorców URL), którym krótko zajmiemy się na kolejnych kilku stronach i który dogłębnie omówimy w rozdziale poświęconym wdrażaniu aplikacji internetowych.

Zastanawiasz się pewnie, jak kontener znajduje odpowiedni serwlet...

Adres URL, który dociera do serwera WWW jako część żądania klienta, jest w jakiś sposób *odwzorowywany* na odwołanie do konkretnego serwera. Takie odwzorowywanie adresów URL na serwlety może się odbywać na wiele różnych sposobów i jest jednym z kluczowych problemów, przed którymi stajemy w procesie wytwarzania aplikacji internetowych. Żądanie użytkownika musi zostać przekształcone w prawidłowe odwołanie do konkretnego serwletu — zrozumienie i (często) *konfiguracja* tego typu odwzorowań należy do programisty aplikacji internetowej. Co o tym myślisz?

WYTĘŻ UMYSŁ

Jak kontener powinien odwzorowywać serwlety na adresy URL?

Kiedy użytkownik robi *coś* w swojej przeglądarce (klika łącze, naciska przycisk *Wyślij zapytanie*, wpisuje adres URL itp.), należy podjąć *jakieś* kroki, które spowodują wysłanie żądania do *konkretnego* serwletu (lub innego zasobu aplikacji internetowej, np. strony JSP). Jak to działa w praktyce?

Wymień zalety i wady każdego z poniższych rozwiązań.

1. *Zapisanie odwzorowania na stałe w kodzie strony HTML. Innymi słowy, klient wykorzystuje dokładną ścieżkę i nazwę pliku (klasy) serwletu:*

   ZALETY:

   WADY:

2. *Wykorzystanie do odwzorowania narzędzia dostarczonego przez producenta kontenera:*

   ZALETY:

   WADY:

3. *Użycie swoistej tabeli właściwości reprezentującej odwzorowania:*

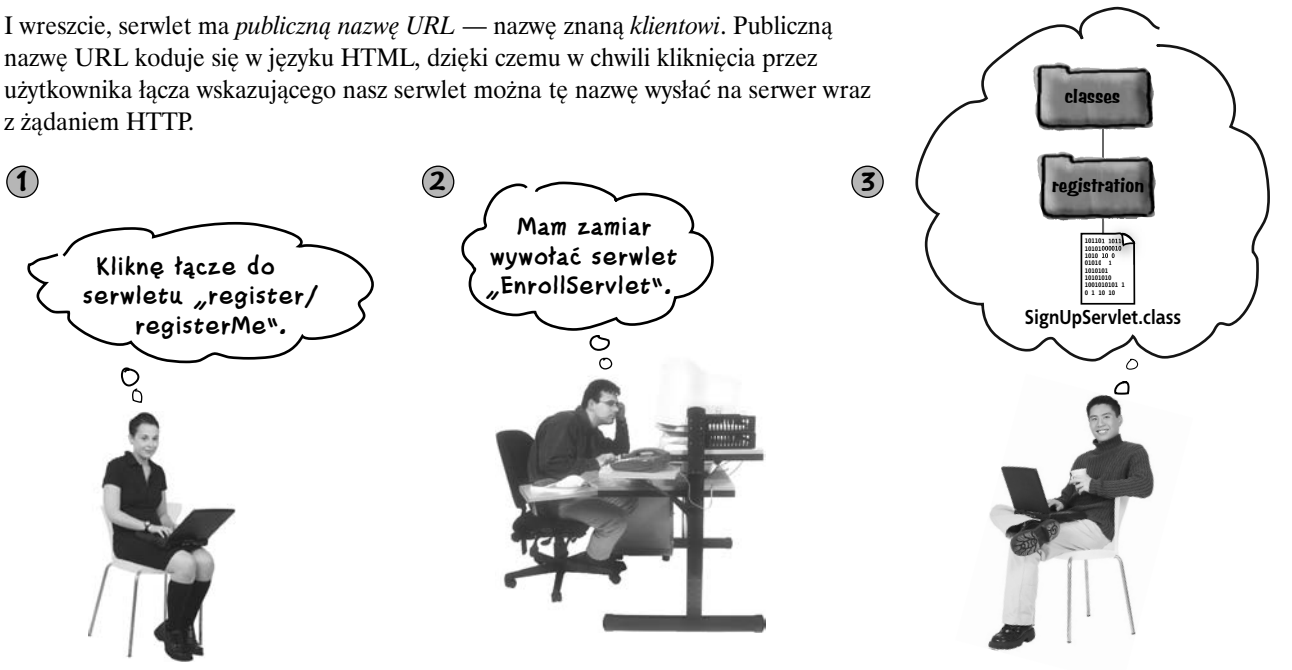
   ZALETY:

   WADY:

Serwlet może mieć aż TRZY nazwy

Serwlet musi oczywiście mieć *nazwę ścieżki* do pliku klasy (np. *classes/registration/SignUpServlet.class*), czyli ścieżkę do faktycznego pliku klasy. Twórca oryginalnej *klasy* serwletu wybiera jej nazwę (i nazwę pakietu, która definiuje odpowiednią część struktury katalogów), natomiast pełna nazwa ścieżki jest uzależniona od położenia pliku klasy na serwerze. Każda osoba wdrażająca serwlet na serwerze WWW może mu dodatkowo nadać specjalną *nazwę wdrożenia*. Nazwa wdrożenia jest po prostu *poufną nazwą wewnętrzną*, która nie musi być taka sama jak nazwa klasy lub nazwa pliku. Nazwa wdrożenia *może* co prawda odpowiadać nazwie klasy serwletu (`registration.SignUpServlet`) lub względnej ścieżce do pliku klasy (*classes/registration/SignUpServlet.class*), ale też może być zupełnie inna (np. *EnrollServlet*).

I wreszcie, serwlet ma *publiczną nazwę URL* — nazwę znaną *klientowi*. Publiczną nazwę URL koduje się w języku HTML, dzięki czemu w chwili kliknięcia przez użytkownika łącza wskazującego nasz serwlet można tę nazwę wysłać na serwer wraz z żądaniem HTTP.



Znana klientowi nazwa *URL*

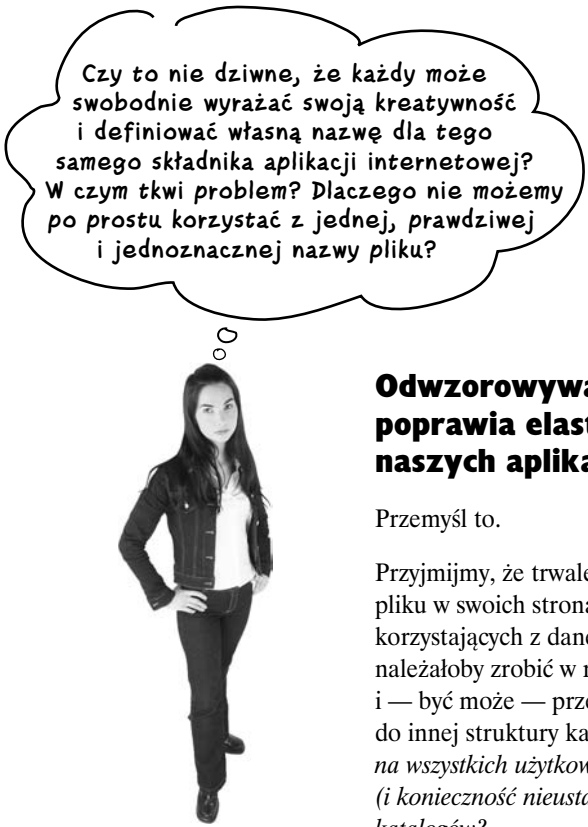
Klient widzi adres URL serwletu (zakodowany w języku HTML), ale nie wie, jak ta nazwa serwletu zostanie w praktyce odwzorowana na rzeczywiste katalogi i pliki składowane na serwerze. Publiczna nazwa URL jest swoistą podróbką stworzoną specjalnie dla klientów.

Znana wdrożeniowcowi *poufna nazwa wewnętrzna*

Wdrożeniowiec może stworzyć nazwę, która będzie znana tylko jemu i innym użytkownikom pracującym w rzeczywistym środowisku operacyjnym. Także ta nazwa jest fałszywką stworzoną wyłącznie na potrzeby procesu wdrażania serwletu. Poufna nazwa wewnętrzna nie musi odpowiadać ani wykorzystywanej przez klienta publicznej nazwie URL, ani rzeczywistej nazwie pliku czy ścieżki do klasy serwletu.

Faktyczna nazwa *pliku*

Opracowana przez programistę *klasa* serwletu ma w pełni kwalifikowaną nazwę, która obejmuje zarówno nazwę klasy, jak i nazwę pakietu. *Plik* klasy serwletu ma oczywiście rzeczywistą ścieżkę i swoją nazwę (zależną od miejsca składowania struktury katalogów pakietu na serwerze WWW).



Odwzorowywanie nazw serwletów poprawia elastyczność i bezpieczeństwo naszych aplikacji internetowych

Przemyśl to.

Przyjmijmy, że trwale zakodowałeś rzeczywistą ścieżkę i nazwę pliku w swoich stronach JSP i pozostałych stronach HTTP korzystających z danego serwletu. Świetnie. Co w takim razie należałoby zrobić w razie konieczności reorganizacji aplikacji i — być może — przeniesienia niektórych jej składników do innej struktury katalogów? *Czy naprawdę chcesz wymusić na wszystkich użytkownikach swojego serwletu znajomość (i konieczność nieustannej weryfikacji) tej samej struktury katalogów?*

Jeśli zamiast zapisywania w kodzie źródłowym rzeczywistej nazwy pliku i ścieżki zastosujemy mechanizm odwzorowań, będziemy mogli swobodnie przenosić składniki aplikacji bez konieczności koszmarnie czasochłonnego wyszukiwania i aktualizowania kodu klienta, który sztywno odwołuje się do starej lokalizacji plików serwletu.

A co z bezpieczeństwem? Czy naprawdę zależy nam na tym, aby klient dokładnie znał strukturę plików i katalogów w naszym serwerze? Czy chcemy, by na przykład próbował bezpośrednio przeglądać pliki serwletów bez kontroli zapewnianej przez właściwe strony lub formularze? Nie ma wątpliwości, że użytkownik końcowy dysponujący wiedzą o *rzeczywistej* ścieżce do pliku klasy serwletu będzie mógł tę ścieżkę wpisać w swojej przeglądarce, aby przynajmniej spróbować uzyskać bezpośredni dostęp do tego serwletu.

Odwzorowania adresów URL na serwlety za pośrednictwem deskryptora wdrożenia

Podczas wdrażania naszego serwletu w kontenerze WWW musimy opracować stosunkowo prosty dokument XML nazywany deskryptorem wdrożenia (ang. *Deployment Descriptor — DD*), który na potrzeby kontenera określa sposób, w jaki należy wykonywać nasze serwlety i (lub) strony JSP. Deskryptory wdrożenia nie służą co prawda wyłącznie odwzorowywaniu nazw, ale zawierają dwa elementy języka XML odpowiedzialne właśnie za definiowanie odwzorowań adresów URL na serwlety — jeden odwzorowuje znaną klientowi *publiczną nazwę URL* na naszą *nazwę wewnętrzną*, drugi odwzorowuje naszą *nazwę wewnętrzną* do postaci w pełni kwalifikowanej *nazwy klasy*.

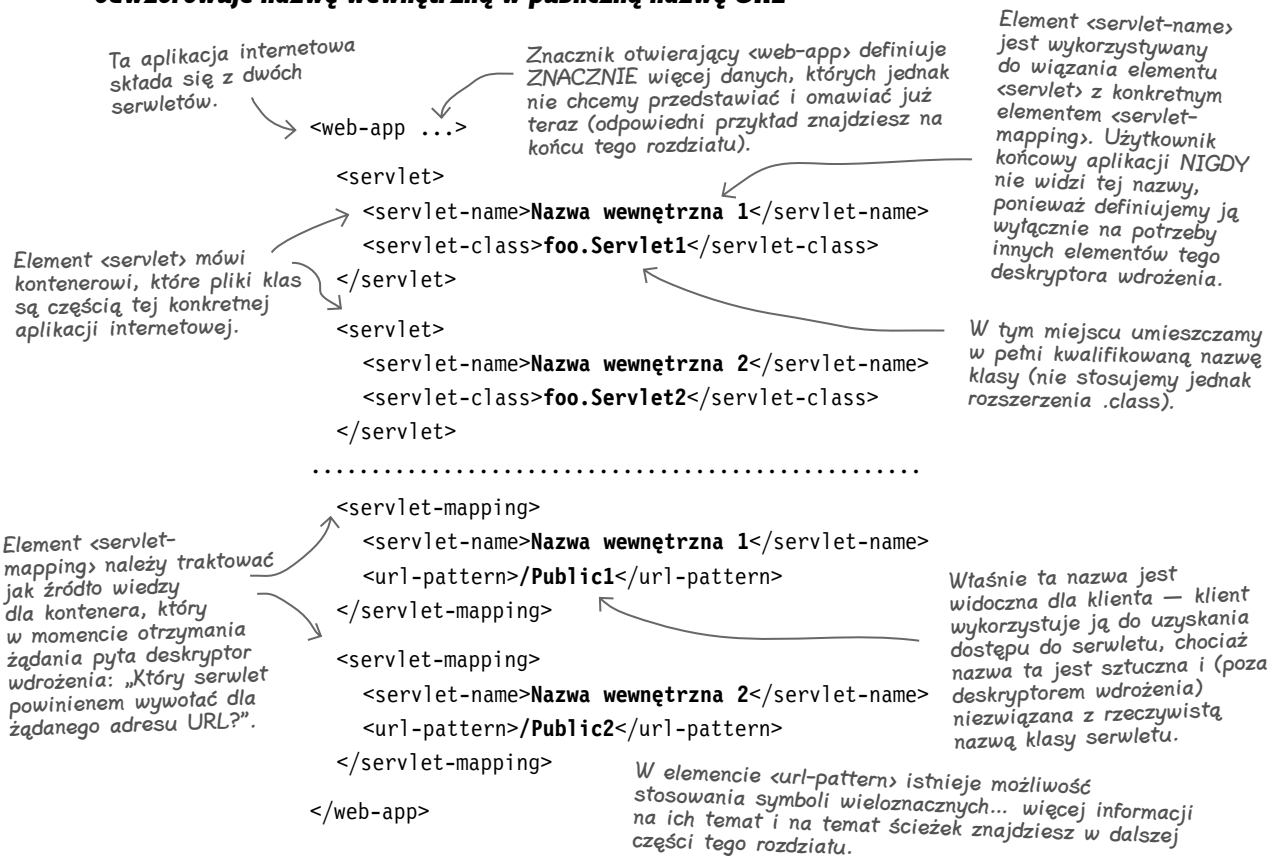
Dwa elementy deskryptora wdrożenia wykorzystywane do odwzorowywania adresów URL:

① <servlet>

odwzorowuje nazwę wewnętrzną do postaci w pełni kwalifikowanej nazwy klasy

② <servlet-mapping>

odwzorowuje nazwę wewnętrzną w publiczną nazwę URL




```
<web-app ...>
  <servlet>
    <servlet-name>Nazwa wewnętrzna 1</servlet-name>
    <servlet-class>foo.Servlet1</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Nazwa wewnętrzna 2</servlet-name>
    <servlet-class>foo.Servlet2</servlet-class>
  </servlet>
  ..................................................
  <servlet-mapping>
    <servlet-name>Nazwa wewnętrzna 1</servlet-name>
    <url-pattern>/Public1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Nazwa wewnętrzna 2</servlet-name>
    <url-pattern>/Public2</url-pattern>
  </servlet-mapping>
</web-app>
```

Zaczekaj! W deskryptorze wdrożenia możemy zrobić znacznie więcej

Poza definiowaniem odwzorowań adresów URL na rzeczywiste nazwy serwletów, deskryptory wdrożenia można wykorzystywać do dostosowywania do naszych potrzeb także innych aspektów aplikacji internetowej, jak role bezpieczeństwa, strony o błędach, biblioteki znaczników, informacje o konfiguracji początkowej czy wręcz (jeśli korzystamy z pełnowartościowego serwera J2EE) deklaracje dostępu do konkretnych komponentów Enterprise JavaBeans (EJB).

Nie przejmuj się na razie szczegółami. Póki co zasadnicze znaczenie ma dla nas możliwość deklaratywnego modyfikowania naszej aplikacji na poziomie deskryptora wdrożenia zamiast wprowadzania zmian w kodzie źródłowym (i ponownego kompilowania tego kodu)!

Pomyśl tylko… oznacza to, że nawet osoby niebędące programistami Javy mogą dostosowywać do swoich potrzeb napisane w tym języku aplikacje internetowe bez konieczności zawracania nam głowy i zakłócania wakacji w tropikach.

Deskryptor wdrożenia (DD) oferuje nam mechanizm „deklaratywnego" dostosowywania aplikacji internetowych do potrzeb użytkownika bez konieczności modyfikowania ich kodu źródłowego!

Nie ma niemądrych pytań

P: Mam kłopot. W przedstawionym deskryptorze wdrożenia nadal nie widzę niczego, co mogłoby wskazywać na rzeczywistą nazwę ścieżki do serwletu! Deskryptor wspomina tylko o nazwie klasy. Nadal nie ma więc odpowiedzi na pytanie, w jaki sposób kontener wykorzystuje tę nazwę do odnajdywania konkretnych plików klas serwletów. Może istnieje gdzieś INNE odwzorowanie, które określa, że taka i taka nazwa klasy jest odwzorowywana na taki i taki plik w takiej i takiej lokalizacji?

O: Wiedziałem, że zwrócisz na to uwagę. Masz rację — w elemencie `<servlet-class>` w deskryptorze wdrożenia umieszczamy jedynie nazwę klasy (w pełni kwalifikowaną, a więc z nazwą pakietu). Wynika to z faktu, iż kontener dysponuje określonym miejscem, w którym odnajduje wszystkie te serwlety, dla których w deskryptorze wdrożenia zdefiniowano odpowiednie odwzorowania.

W praktyce kontener stosuje wyszukany zbiór reguł do odnajdywania pasujących do siebie adresów URL (pochodzących z żądań klienta) i rzeczywistych klas Javy (przechowywanych gdzieś w serwerze). Omówimy to zagadnienie bardziej szczegółowo w dalszej części tej książki (w rozdziale poświęconym wdrażaniu aplikacji internetowych). Na razie w zupełności wystarczy, jeśli zapamiętasz, że definiowanie tego typu odwzorowań nie wymaga żadnych dodatkowych czynności.

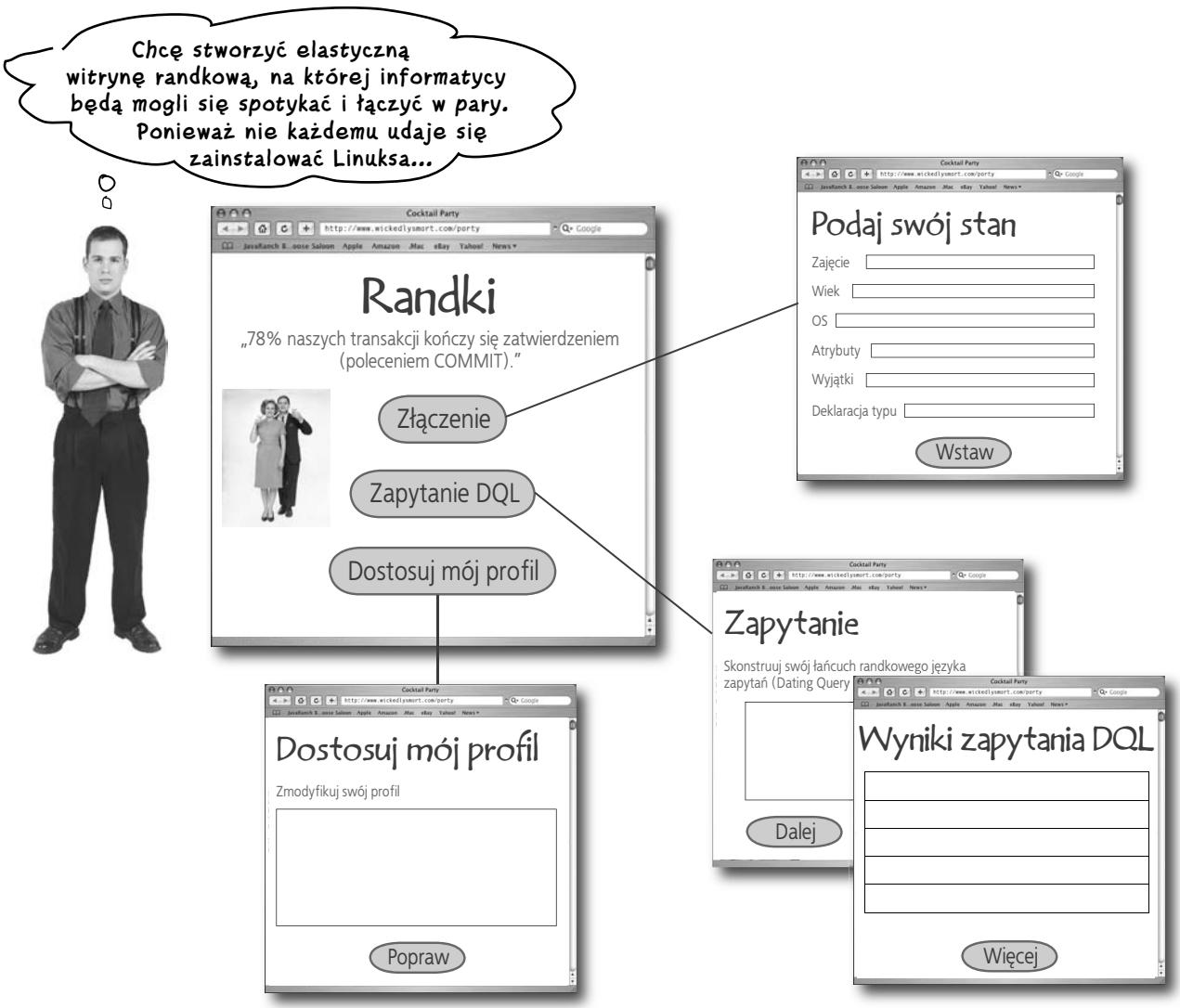
Zalety deskryptorów wdrożenia

- Minimalizują liczbę operacji na kodzie źródłowym, który przeszedł już niezbędne testy.
- Umożliwiają nam dostosowywanie możliwości naszej aplikacji, nawet jeśli *nie mamy* dostępu do kodu źródłowego.
- Dają możliwość przystosowywania naszej aplikacji do korzystania z różnych zasobów (np. baz danych) bez konieczności ponownego kompilowania i testowania kodu źródłowego.
- Ułatwiają zarządzanie dynamicznymi regułami bezpieczeństwa, w tym listami kontrolnymi i rolami bezpieczeństwa.
- Osobom niebędącym programistami umożliwiają modyfikowanie i wdrażanie naszych aplikacji internetowych w czasie, gdy *my* możemy się koncentrować na ciekawszych zajęciach (np. na doborze najlepszych ciuchów przed wyjazdem nad morze).

Opowiadanie: Bob buduje witrynę swatającą

Umawianie się na randki jest w dzisiejszych czasach dosyć trudne. Kto ma na to czas, skoro zawsze jest jakiś dysk do zdefragmentowania? Bob, który chce wejść w interes *dot.com* (załóżmy na chwilę, że rynek tego typu witryn nie jest jeszcze nasycony), wierzy, że stworzenie specjalnej witryny randkowej dla pasjonatów informatyki będzie jego przepustką do wielkiej kariery i ostatecznego porzucenia dotychczasowej roli komiksowego Dilberta.

Problem w tym, że Bob był menedżerem projektów informatycznych tak długo, że nie jest już zbyt biegły we współczesnych technikach inżynierii oprogramowania. Zna jednak kilka trudnych pojęć oraz niektóre konstrukcje Javy; poświęcił też trochę czasu na pobieżną lekturę materiałów o serwletach, zatem błyskawicznie opracował niezbędny projekt i przystąpił do kodowania...

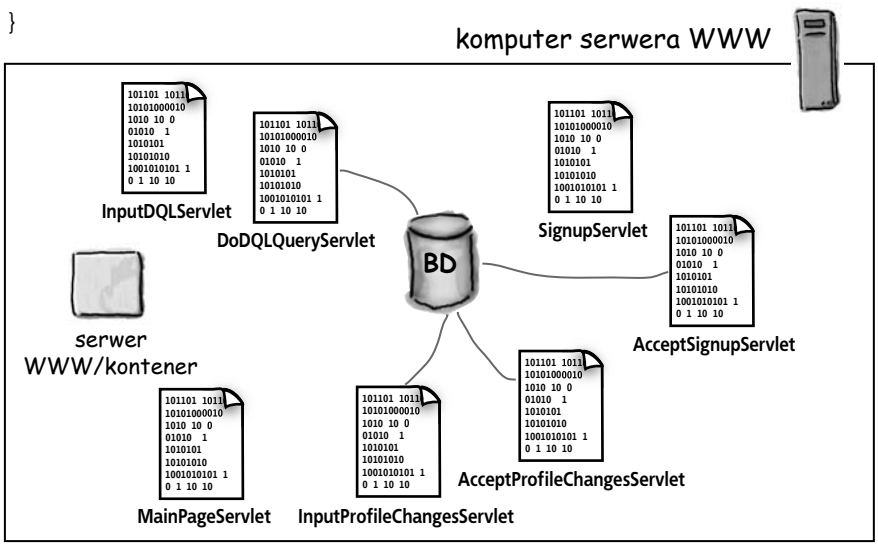


Bob przystąpił do tworzenia garści serwletów... po jednym dla każdej strony

Bob rozważał zastosowanie tylko jednego serwletu pełnego niezbędnych wyrażeń warunkowych, ale ostatecznie zdecydował, że podział funkcjonalności pomiędzy wiele osobnych serwletów będzie rozwiązaniem właściwszym — każdy serwlet powinien odpowiadać za generowanie i obsługę pojedynczej strony (strony zapytania, strony zapisywania do serwisu, strony wyników wyszukiwania itp.).

Każdy serwlet obejmuje kompletną logikę biznesową niezbędną do zmodyfikowania lub odczytania zawartości bazy danych oraz zapisania w strumieniu odpowiedzi (a więc odesłania do klienta) odpowiedniego kodu HTML.

```
// polecenia importowania

public class DatingServlet extends HttpServlet {

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

    // w tym miejscu jest zakodowana logika biznesowa
    // bieżącego serwletu; wykonywany kod zależy od zakresu
    // działań danego serwletu (zapis w bazie danych,
    // wykonanie zapytania itp.)

    PrintWriter out = response.getWriter();

    // wygeneruj dynamiczną stronę HTML
    out.println("w tym miejscu znajduje się coś naprawdę okropnego");
  }
}
```



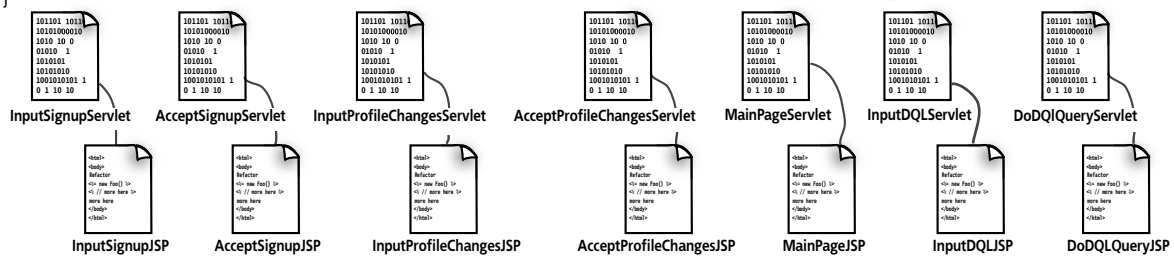
Serwlet podejmuje wszystkie działania niezbędne do przetworzenia otrzymanego żądania (jak wstawienie rekordu do bazy danych lub jej przeszukanie), po czym zwraca stronę HTML za pośrednictwem obiektu reprezentującego odpowiedź protokołu HTTP.

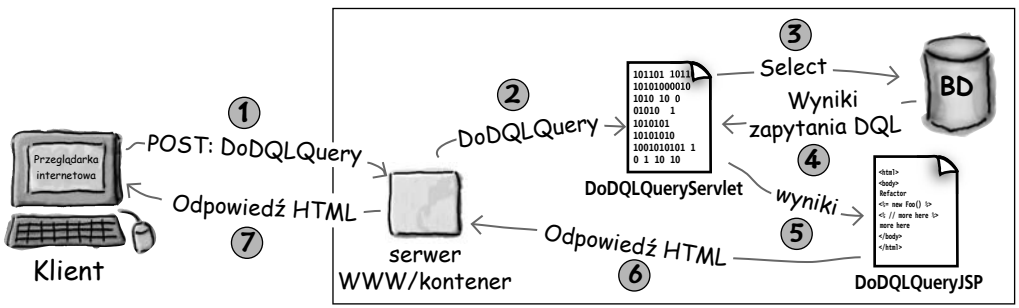
Cała logika biznesowa ORAZ odpowiedź ze stroną HTML dla klienta znajduje się wewnątrz kodu serwletu.

Po jakimś czasie okazało się jednak, że taki model będzie niepraktyczny, zatem Bob dodał kilka stron JSP

Te irytujące wywołania metody `println()`, które zapisywały wynikową stronę HTML w obiekcie odpowiedzi, były na tyle nieporęczne, że Bob zdecydował się poczytać trochę o stronach JSP i przemodelować swoją aplikację. Od tej pory każdy serwlet będzie realizował odpowiedni fragment logiki biznesowej (wykonywał zapytanie na bazie danych, wstawiał lub aktualizował rekord bazy danych itp.), *po czym będzie przekazywał żądanie do odpowiedniej strony JSP*, która wygeneruje stronę HTML dla klienta. W ten sposób Bob oddzielił *logikę biznesową* od *prezentacji*… a ponieważ czytał trochę o projektowaniu oprogramowania, dobrze wie, że *podział odpowiedzialności* jest dobrym posunięciem.

```
// polecenia importowania

public class DatingServlet extends HttpServlet {

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

    // w tym miejscu jest zakodowana logika biznesowa
    // bieżącego serwletu; wykonywany kod zależy od zakresu
    // działań danego serwletu (zapis w bazie danych,
    // wykonanie zapytania itp.)

    // przekaż żądanie do konkretnej strony JSP
    // zamiast próbować samodzielnie zapisywać kod
    // HTML w strumieniu wyjściowym
  }
}
```





Klient wypełnia formularz zapytania DQL i klika przycisk Dalej. W efekcie wysyłane jest żądanie POST protokołu HTTP dla serwletu DoDQLQuery. Serwer WWW wywołuje żądany serwlet, który z kolei na bazie otrzymanych danych wykonuje odpowiednie zapytanie i przekazuje żądanie dalej do odpowiedniej strony JSP. Strona JSP buduje dokument HTML i odsyła go z powrotem do klienta.

Ale dopiero wtedy kolega zapytał go: „STOSUJESZ wzorzec projektowy MVC, prawda?"

Kim chciał wiedzieć, czy usługa umawiania na randki będzie dostępna z poziomu aplikacji z graficznym interfejsem użytkownika (GUI) opartym na komponentach Swing. Bob odrzekł: „No cóż, nie myślałem o tym". Wówczas Kim odpowiedział: „Nie przejmuj się, to żaden problem — pewnie zastosowałeś wzorzec MVC nieświadomie, więc będziemy mogli bez trudu stworzyć klienta Swing GUI, który uzyska dostęp do Twoich klas logiki biznesowej".

Co zrobisz, kiedy zaistnieje potrzeba zastosowania dla Twojej usługi randkowej aplikacji z graficznym interfejsem użytkownika opartym na komponentach Swing; aplikacji, która będzie korzystała z tej samej logiki biznesowej?

Bob przełknął ślinę.

Kim odpowiedział: „Tylko nie mów, że… ***nie*** stosowałeś MVC?".

Bob powiedział: „Cóż, oddzieliłem prezentację od logiki biznesowej…".

Kim odrzekł: „To dopiero początek… ale, niech zgadnę… pewnie umieściłeś całą logikę biznesową w *serwletach*!?".

Bob nagle uświadomił sobie, dlaczego na pewnym etapie swojej kariery zrezygnował z programowania na rzecz zarządzania.

Jest jednak na tyle zdeterminowany by doprowadzić swoją aplikację do właściwego końca, że poprosił Kima o błyskawiczny przegląd tajemniczego wzorca MVC.

Stosowanie wzorca projektowego MVC oznacza, że logika biznesowa jest nie tylko oddzielona od prezentacji... logika biznesowa nie powinna nawet wiedzieć, że ISTNIEJE jakaś prezentacja.

Istotą wzorca *MVC* (ang. *Model View Controller*) jest nie tylko oddzielenie logiki biznesowej od prezentacji, ale także umieszczenie czegoś *pomiędzy* nimi, dzięki czemu możliwe będzie utrzymywanie logiki biznesowej w samodzielnej klasie Javy wielokrotnego użytku, która nie musi dysponować żadną wiedzą o działaniach widoku.

Bob był całkiem blisko, ponieważ sam wpadł na pomysł rozdzielenia logiki biznesowej i prezentacji, ale w jego aplikacji logika biznesowa pozostawała w *ścisłym związku* z widokiem. Innymi słowy, Bob *włączył logikę biznesową do serwletu* i — tym samym — wykluczył możliwość ponownego wykorzystania tego kodu z innym widokiem (np. aplikacją Swing GUI czy nawet aplikacją mobilną). Jego logika biznesowa utknęła w serwletach, a powinna się znaleźć w osobnych klasach Javy, których Bob mógłby używać w przyszłości.

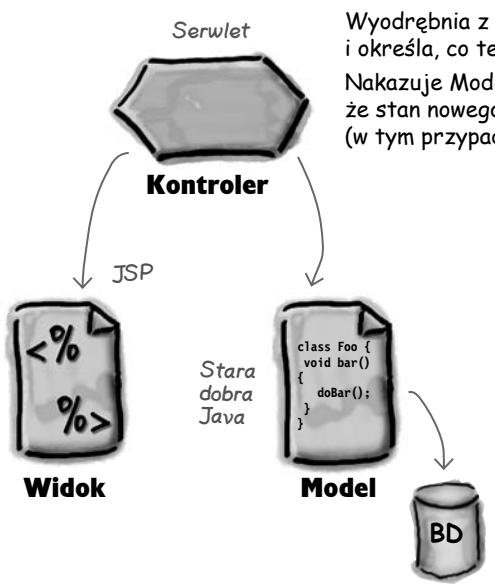
Rozwiązaniem zaistniałego problemu jest zastosowanie wzorca projektowego model-widok-kontroler (MVC)

Gdyby Bob znał i rozumiał wzorzec projektowy MVC, wiedziałby, że logika biznesowa nie powinna być osadzana w ramach serwletu bądź serwletów. Zdawałby sobie sprawę, że umieszczanie logiki biznesowej w kodzie serwletu powoduje ogromne komplikacje w razie konieczności uzyskania dostępu do serwisu randkowego z poziomu innej aplikacji (np. aplikacji z interfejsem GUI opartym na komponentach Swing). Wzorcowi projektowemu MVC (i innym wzorcom) poświęcimy co prawda znacznie więcej uwagi w dalszej części tej książki, jednak teraz powinieneś przejść krótki kurs na ten temat, ponieważ na końcu tego rozdziału zaprezentujemy przykładową aplikację zbudowaną właśnie w oparciu o wzorzec MVC.

Jeśli znasz już wzorzec projektowy MVC, zapewne wiesz, że wzorzec ten nie dotyczy wyłącznie serwletów i stron JSP — jednoznaczne oddzielenie logiki biznesowej od prezentacji jest równie istotne we wszelkiego rodzaju aplikacjach. Okazuje się jednak, że właśnie w przypadku aplikacji internetowych taki podział jest *szczególnie* istotny, ponieważ nie można zakładać, że nasza logika biznesowa będzie udostępniana *wyłącznie* za pośrednictwem stron WWW! Jesteśmy przekonani, że Twoje doświadczenie w pracy programisty w zupełności wystarczy, aby rozumieć, że jedynym pewnym aspektem procesu wytwarzania oprogramowania jest ***stale modyfikowana specyfikacja***.

Model*Widok*Kontroler (MVC) wyciąga logikę biznesową poza serwlet i umieszcza ją w „Modelu", czyli tradycyjnych, zwykłych klasach Javy wielokrotnego użytku. Model jest kombinacją danych biznesowych (np. stanu koszyka z zakupami) i metod (reguł), które na tych danych operują.

Wzorzec projektowy MVC w świecie serwletów i stron JSP



KONTROLER

Wyodrębnia z żądania dane wejściowe użytkownika i określa, co te dane oznaczają dla bieżącego Modelu.

Nakazuje Modelowi samoaktualizację i sprawia, że stan nowego Modelu jest dostępny dla widoku (w tym przypadku strony JSP).

WIDOK

Odpowiada za prezentację. Otrzymuje od Kontrolera stan Modelu (przekazywanie stanu nie odbywa się jednak bezpośrednio — Kontroler umieszcza dane Modelu w miejscu, z którego Widok może je odczytać). Widok jest także tą częścią aplikacji, która pobiera dane wejściowe użytkownika (przekazywane następnie Kontrolerowi).

MODEL

Zawiera właściwą logikę biznesową i stan aplikacji. Innymi słowy, zna reguły uzyskiwania i aktualizowania tego stanu.

Częścią Modelu we wzorcu projektowym MVC będzie np. zawartość koszyków z zakupami (wraz z regułami określającymi możliwe działania na tej zawartości).

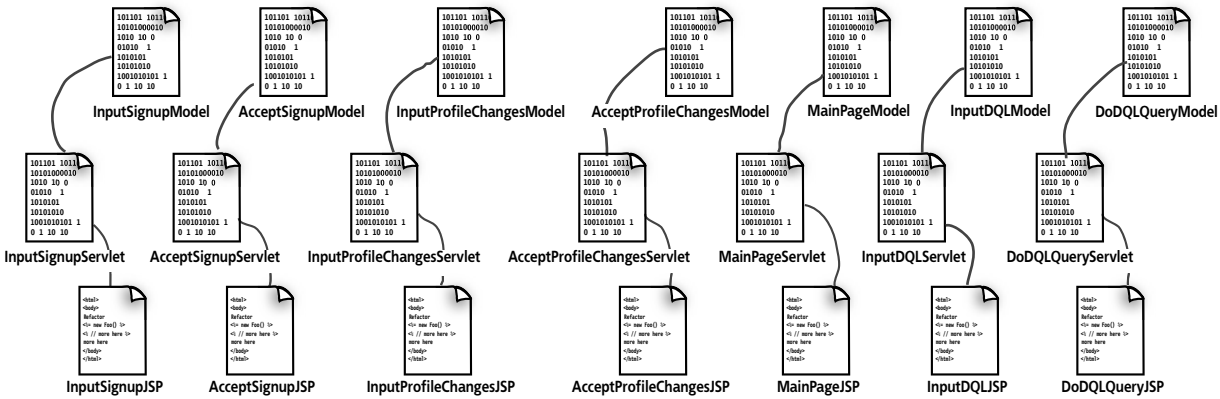
Model jest jedyną częścią systemu współpracującym z bazą danych (chociaż w wielu przypadkach wykorzystuje inny obiekt do faktycznej komunikacji z bazą danych — omówienie tego wzorca zostawimy sobie jednak na później...).

Stosowanie wzorca projektowego MVC dla aplikacji internetowej Boba

Bob wie już, co musi zrobić. Powinien oddzielić logikę biznesową od serwletów i dla każdego z nich stworzyć zwykłą klasę Javy — klasy te będą tworzyły Model.

Po wprowadzeniu tych zmian oryginalny serwlet będzie pełnił funkcję Kontrolera, nowa klasa logiki biznesowej będzie występowała w roli Modelu, a użyte strony JSP będą tworzyły Widok aplikacji.

Dla każdej strony swojej aplikacji Bob dysponuje teraz serwletem (Kontroler), klasą Javy (Model) oraz stroną JSP (Widok).



Ale wtedy na całą aplikację rzucił okiem jego przyjaciel Kim

Kim odwiedził Boba, spojrzał na jego aplikację i stwierdził, że JEST to co prawda projekt MVC, ale jego architektura jest zupełnie nieprzemyślana. To fakt, logika biznesowa została wyodrębniona i umieszczona w Modelu, a serwlety pełnią funkcję Kontrolerów działających pomiędzy Modelami a Widokami, zatem Modele teoretycznie mogą nie mieć pojęcia o istnieniu Widoków. Taki podział jest oczywiście pożądany, warto jednak zwrócić uwagę na wszystkie te małe serwlety.

Co tak naprawdę te serwlety *robią*? Skoro logika biznesowa znajduje się już w bezpiecznej odległości od serwletów (Kontrolerów) — w osobnych klasach tworzących Model — same serwlety nie mają zbyt wiele do roboty poza ogólną obsługą aplikacji i (tak, pamiętam o tym) aktualizacją Modelu i wywoływaniem stron Widoku.

Zasadniczą wadą Twojego rozwiązania jest to, że ogólna logika aplikacji jest powielana w każdym z tych cholernych serwletów! Jeśli będziesz musiał zmienić choć jeden drobiazg, odpowiednie modyfikacje będziesz musiał wprowadzać we wszystkich serwletach. Konserwacja tego rodzaju aplikacji jest koszmarem.

„No tak, czułem, że z tym powielanym kodem coś jest nie tak” — odpowiedział Bob — „ale cóż innego mogłem w tej sytuacji zrobić? Nie chcesz chyba, żebym jeszcze raz umieszczał wszystko w jednym serwlecie? Czy *takie* rozwiązanie miałoby w ogóle sens?”.

Czy istnieje rozwiązanie?

Czy Bob powinien wrócić do jednego serwletu Kontrolera, aby uniknąć powtarzania kodu? Czy takie rozwiązanie będzie zgodne z koncepcją programowania obiektowego — przecież każdy ze stosowanych do tej pory serwletów w praktyce robi coś innego? Czy Keanu Reeves rzeczywiście zna Kung Fu?

WYTĘŻ UMYSŁ

Zostaw ten problemy na później — my też tak zrobimy.

Jak sądzisz? Czy znasz odpowiedź? Czy w ogóle ISTNIEJE jedna odpowiedź? Czy zgodnie z propozycją Boba pozostawiłbyś istniejące serwlety, czy może umieściłbyś ten kod w jednym serwlecie Kontrolera? A gdybyś faktycznie użył jednego Kontrolera dla wszystkich działań, skąd taki Kontroler wiedziałby, który Model i Widok wywoływać?

To pytanie pozostanie bez odpowiedzi niemal do samego *końca* tej książki, zatem nie trać na nie zbyt dużo czasu i przekaż je wątkowi pracującemu w tle Twojego umysłu...

Zaostrz ołówek

1. Jeśli zastosujemy wzorzec projektowy MVC w świecie serwletów i stron JSP, każdy z trzech wykorzystywanych komponentów (strona JSP, klasa Javy oraz serwlet) będzie odgrywał jedną z trzech ról MVC. Oznacz kółkami litery *M*, *V* i *C* w zależności od tego, jaką funkcję w ramach wzorca MVC odgrywa dany komponent. Dla każdego komponentu powinieneś oznaczyć w ten sposób tylko jedną literę.

JSP

M V C

klasa Javy niebędąca serwletem

M V C

serwlet

M V C

2. Które elementy wzorca projektowego MVC reprezentują poszczególne litery składające się na akronim MVC?

***M* oznacza** _____

***V* oznacza** _____

***C* oznacza** _____

KLUCZOWE ZAGADNIENIA

- Kontener zapewnia naszej aplikacji internetowej obsługę komunikacji, zarządzanie cyklem życia, obsługę przetwarzania wielowątkowego, bezpieczeństwo deklaratywne oraz pełną obsługę stron JSP — dzięki temu możemy się skoncentrować na tworzeniu logiki biznesowej tej aplikacji.
- Kontener tworzy obiekty żądania i odpowiedzi, które mogą być wykorzystywane przez serwlety (i inne składniki aplikacji internetowej) do uzyskiwania niezbędnych informacji o żądaniu i odsyłania odpowiednich danych do klienta.
- Typowy serwlet jest klasą dziedziczącą po klasie `HttpServlet` (rozszerzającą klasę `HttpServlet`) i nadpisującą przynajmniej jedną z metod odpowiadających metodom protokołów HTTP wywoływanym przez przeglądarkę internetową (`doGet()`, `doPost()` itp.).
- Wdrożeniowiec może odwzorować klasę serwletu na odpowiedni adres URL, który będzie dostępny dla klientów zainteresowanych wysyłaniem żądań do danego serwletu. Nazwa użyta w adresie URL może nie mieć nic wspólnego z faktyczną nazwą *pliku* klasy.

Kto za co odpowiada?

Wypełnij poniższą tabelę. Określ, czy serwer WWW, kontener, a może serwlet odpowiada w największym stopniu za realizację wymienionych zadań. W kilku przypadkach prawidłowa może być więcej niż jedna odpowiedź. Jako zadanie dodatkowe umieść w tabeli krótkie komentarze opisujące poszczególne procesy.

Zadanie	Serwer WWW	Kontener	Serwlet
Tworzenie obiektów żądania i odpowiedzi			
Wywoływanie metody `service()`			
Uruchamianie nowych wątków obsługujących przychodzące żądania			
Konwersja obiektu odpowiedzi do postaci odpowiedzi protokołu HTTP			
Znajomość protokołu HTTP			
Dodawanie kodu HTML do obiektu odpowiedzi			
Utrzymywanie referencji do obiektów odpowiedzi			
Odnajdywanie adresów URL w deskryptorze wdrożenia			
Usuwanie obiektów żądania i odpowiedzi			
Koordynowanie tworzenia dynamicznej zawartości stron			
Zarządzanie cyklami życia			
Posiadanie właściwej dla elementu `<servlet-class>` nazwy z deskryptora wdrożenia			

Ćwiczenie

Magnesiki metod

Fragmenty kodu działającego serwletu i odpowiadającego mu deskryptora wdrożenia wymieszano na drzwiach lodówki. Czy potrafisz prawidłowo połączyć te fragmenty, aby z powrotem otrzymać właściwe listingi serwletu i deskryptora wdrożenia, których adres URL kończy się słowem **/Dice?** Pamiętaj, że na lodówce mogą się znajdować dodatkowe magnesy, których w ogólne nie będziesz potrzebował.

Serwlet

```
public class
```

```
extends HttpServlet {
```

```
public void doGet(
```

```
throws IOException {
```

```
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

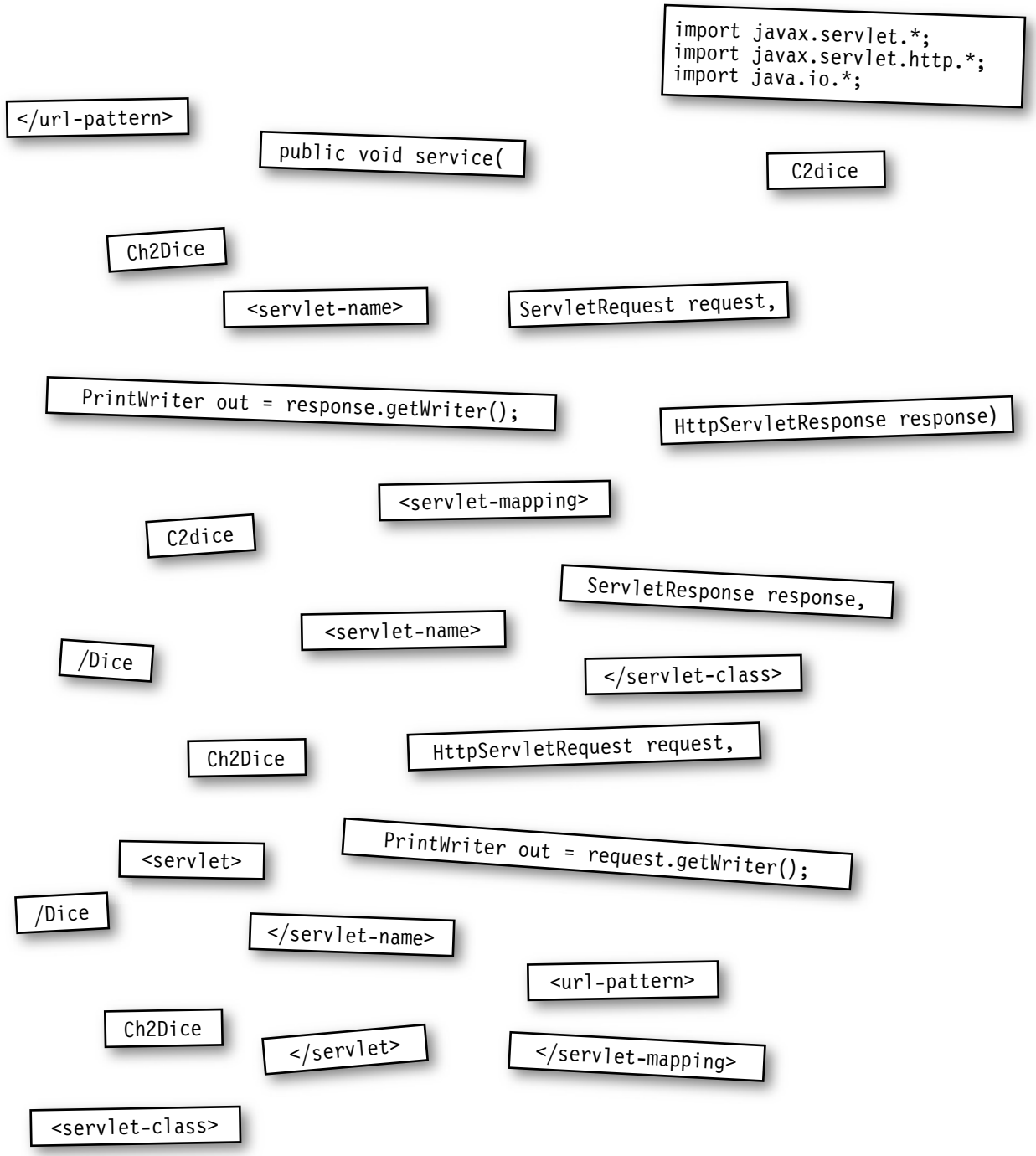
```
<web-app ... >
```

(Pamiętaj, że nie jest to kompletny znacznik otwierający <web-app>; prawidłowy przykład tego znacznika przedstawimy na końcu tego rozdziału. Nie ma to oczywiście wpływu na rozwiązanie tego ćwiczenia.)

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

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

Magnesiki z kodem, kontynuacja...



ROZWIĄZANIA ĆWICZEŃ

Zadanie	Serwer WWW	Kontener	Serwlet
Tworzenie obiektów żądania i odpowiedzi		*Bezpośrednio przed uruchomieniem wątku.*	
Wywoływanie metody `service()`		*Metoda service() wywołuje następnie metodę doGet() lub doPost().*	
Uruchamianie nowych wątków obsługujących przychodzące żądania		*Uruchamia wątek serwletu.*	
Konwersja obiektu odpowiedzi do postaci odpowiedzi protokołu HTTP		*Generuje strumień odpowiedzi HTTP na podstawie danych zawartych w obiekcie odpowiedzi.*	
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		*Przekazuje referencję serwletowi.*	*Wykorzystuje referencję do przekazania odpowiedzi.*
Odnajdywanie adresów URL w deskryptorze wdrożenia		*W ten sposób odnajduje serwlet właściwy 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 service() (i, jak się przekonamy, nie tylko).*	
Przechowywanie właściwej dla elementu `<servlet-class>` z deskryptora wdrożenia			*public class Cokolwiek*

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 wdrożenia (DD)

Na razie nie musisz się martwić o rzeczywiste znaczenia poszczególnych fragmentów deskryptora wdrożenia (w *dalszych* rozdziałach zdobędziesz odpowiednią wiedzę i zostaniesz z tej wiedzy rozliczony). W tym miejscu chcemy Ci tylko pokazać deskryptor wdrożenia *web.xml*, który faktycznie *działa*. W pozostałych przykładach prezentowanych w tym rozdziale pominiemy sporo fragmentów otwierającego znacznika `<web-app>`. (Teraz już wiesz, dlaczego nie rezygnowaliśmy z jego każdorazowego powielania).

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

```
<web-app ...>

  <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>
```

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

RZECZYWISTA postać deskryptora wdrożenia:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsl="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>
```

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

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 obowiązują kontenery WWW). Warto jednak pamiętać, że platforma J2EE 1.4 obejmuje także specyfikację komponentów Enterprise JavaBeans (w skrócie EJB) w wersji 2.1, która z kolei obowiązuje twórców kontenerów EJB. Innymi słowy, kontener WWW ma na celu obsługę komponentów *WWW* (serwletów i stron JSP), natomiast zadaniem kontenera EJB jest obsługa komponentów *biznesowych*.

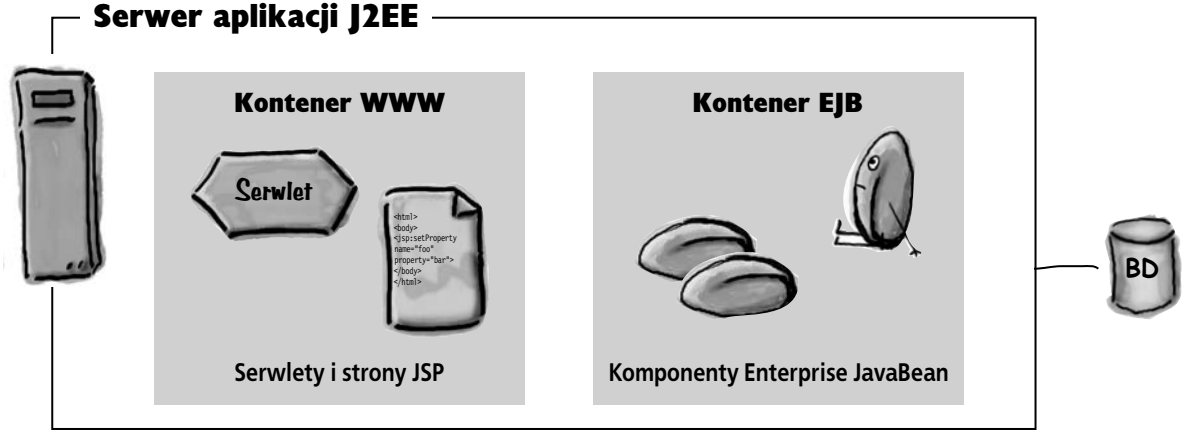
Serwer aplikacji w pełni zgodny ze specyfikacją J2EE musi zawierać *zarówno* kontener WWW, jak i kontener EJB (a także inne mechanizmy, jak implementacja standardów JNDI i JMS). Warto pamiętać, że np. Tomcat jest jedynie kontenerem WWW! Jego zgodność ze specyfikacją J2EE dotyczy wyłącznie elementów składających 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 pełnowartościowych serwerów J2EE, choć istnieje jeszcze kilka autonomicznych 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 pozbawionych komponentów EJB wykorzystuje odpowiednio skonfigurowane serwery Apache i Tomcat — gdzie Apache pełni funkcję *serwera WWW*, natomiast Tomcat występuje w roli *kontenera*.

Do najbardziej popularnych serwerów J2EE należą: WebLogic firmy BEA, JBoss AS typu open source oraz WebSphere firmy IBM.