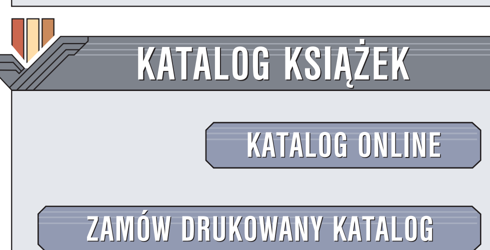
helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

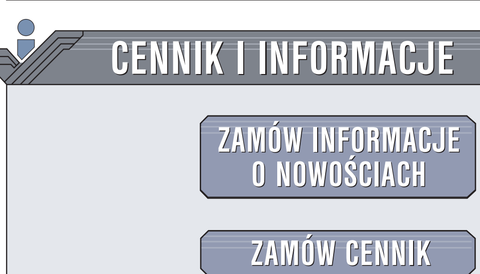
IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

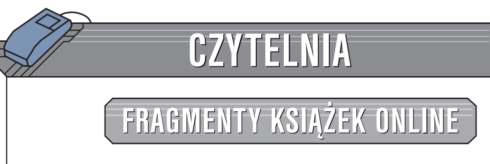


TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK



CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE



Head First Servlets & JSP. Edycja polska

Autorzy: Bryan Basham, Kathy Sierra, Bert Bates
Tłumaczenie: Piotr Rajca, Mikołaj Szczepaniak
ISBN: 83-7361-810-4
Tytuł oryginału: Head First Servlets & JSP
Format: B5, stron: 888

Poznaj w niekonwencjonalny sposób nowoczesną technologię tworzenia stron WWW

- Dowiedz się, czym są serwlety i jak działają
- Poznaj model MVC
- Zastosuj serwlety i JSP w praktyce
- Naucz się projektować aplikacje internetowe

Otwórz swój umysł. Poznaj wszystko, co jest związane z serwletami i JSP, w sposób gwarantujący szybkie i skuteczne opanowanie zasad ich stosowania. Zapomnij o listingach liczących tysiące linii, długich i nużących opisach teoretycznych oraz rozbudowanych diagramach. Czytając książkę „Head First Servlets & JSP. Edycja polska", poznasz jedną z najnowocześniejszych metod tworzenia aplikacji WWW w inny sposób. Serwlety i JSP to technologia pozwalająca na budowanie zarówno pojedynczych stron WWW, jak i złożonych dynamicznych serwisów z wykorzystaniem języka Java połączonego z kodem HTML. Aby ją prawidłowo stosować, należy poznać założenia, w oparciu o które została stworzona, oraz nauczyć się tworzyć elementy aplikacji we właściwy sposób.

Dzięki książce „Head First Servlets & JSP. Edycja polska" serwlety i technologia Java Server Pages przestaną być dla Ciebie wiedzą z pogranicza magii. Autorzy książki, wykorzystując najnowsze elementy teorii uczenia, przedstawią Ci wszystkie zagadnienia niezbędne do rozpoczęcia projektowania i tworzenia aplikacji internetowych oraz serwisów WWW z wykorzystaniem JSP. Poznasz typowe elementy aplikacji i zasady ich budowania. Jednak, co najważniejsze, nauczysz się stosować tę wiedzę w praktyce.

- Serwlety i strony JSP
- Architektura aplikacji internetowych i model MVC
- Zasady tworzenia serwletów i aplikacji internetowych
- Budowanie obiektów wchodzących w skład aplikacji
- Tworzenie stron JSP
- Stosowanie niestandardowych znaczników
- Wdrażanie aplikacji internetowych
- Bezpieczeństwo serwletów
- Wykorzystywanie wzorców projektowych

Naucz się stosowania nowoczesnej technologii tworzenia aplikacji, wykorzystując nowoczesną technologię uczenia.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści (skrócony)

	Wprowadzenie	15
1.	Do czego służą serwlety i strony JSP? *Wprowadzenie i przegląd najważniejszych zagadnień*	29
2.	Architektura aplikacji internetowej. *Bardziej szczegółowy przegląd zagadnień*	65
3.	Minipodręcznik MVC. *Omówienie MVC*	95
4.	Być serwletem. *Żądanie i odpowiedź*	121
5.	Być aplikacją internetową. *Atrybuty i obiekty nasłuchujące*	175
6.	Stan konwersacyjny. *Zarządzanie sesjami*	249
7.	Być stroną JSP. *Stosowanie technologii JSP*	307
8.	Strony bezskryptowe. *Bezskryptowe strony JSP*	369
9.	Potęga znaczników niestandardowych. *Stosowanie biblioteki JSTL*	461
10.	Kiedy JSTL nie wystarcza. *Tworzenie znaczników niestandardowych*	517
11.	Jak wdrożyć aplikację internetową? *Wdrażanie aplikacji internetowych*	597
12.	Zachowaj je w tajemnicy, ukryj je w bezpiecznym miejscu. *Bezpieczeństwo aplikacji internetowych*	645
13.	Potęga filtrów. *Filtry i opakowania*	697
14.	Korporacyjne wzorce projektowe. *Wzorce i Struts*	733
A	Ostateczny egzamin próbny	787
	Skorowidz	865

Spis treści (na serio)

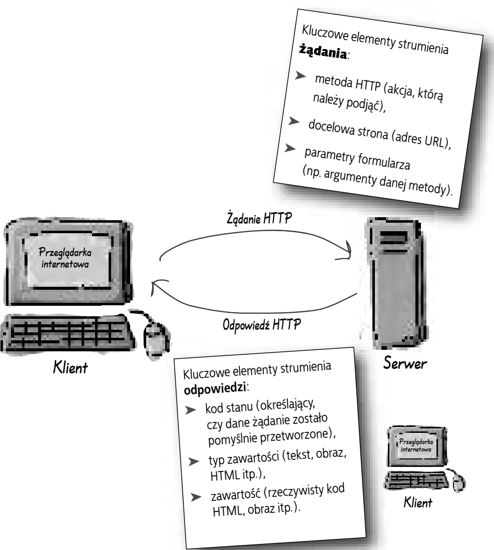
Wprowadzenie

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 zostawię miejsce w pamięci na bardziej istotne informacje, na przykład: jakich dzikich zwierząt należy unikać bądź czy jeżdżenie nago na snowboardzie jest dobrym pomysłem". A zatem w jaki sposób możesz przekonać swój mózg, że Twoje życie zależy od poznania serwletów?

Dla kogo jest ta książka?	16
Wiemy, co sobie myśli Twój mózg	17
Metapoznanie	19
Zmuś swój mózg do posłuszeństwa	21
Czego potrzebujesz, aby skorzystać z tej książki?	22
Zdajemy egzamin certyfikujący	24
Redaktorzy techniczni	26
Podziękowania	28

1 Do czego służą serwlety i strony JSP?

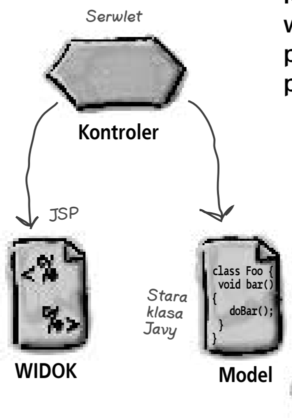
Aplikacje internetowe są super. Ile znasz normalnych aplikacji o graficznym interfejsie użytkownika, których używają miliony osób na całym świecie? Jako programista aplikacji internetowych możesz uwolnić się od problemów z wdrażaniem, jakie występują w przypadku tworzenia wszystkich standardowych aplikacji, i udostępniać swoje aplikacje wszystkim, którzy posiadają przeglądarki WWW. Jednak do tego będziesz potrzebować serwletów i stron JSP. Będziesz ich potrzebować, ponieważ zwykłe, statyczne strony HTML były dobre… w latach 90. Zatem dowiedz się, jak zmienić *witrynę WWW* w *aplikację internetową*.



Cele egzaminu	30
Czym zajmuje się serwer WWW i klient oraz w jaki sposób porozumiewają się ze sobą	34
Dwuminutowy kurs języka HTML	35
Czym jest protokół HTTP?	38
Anatomia żądań `GET` i `POST` oraz odpowiedzi protokołu HTTP	43
Lokalizacja stron WWW przy użyciu adresów URL	48
Serwery WWW, strony statyczne i CGI	52
Serwlety bez tajemnic: pisanie, wdrażanie i uruchamianie serwletów	58
Technologia JSP jest efektem wprowadzenia języka Java do kodu HTML	62

2 Architektura aplikacji internetowej

Serwlety potrzebują pomocy. Kiedy jest odbierane żądanie, ktoś musi utworzyć egzemplarz serwletu albo przynajmniej utworzyć wątek, który będzie umiał obsłużyć żądanie. Ktoś musi wywołać metodę `doPost()` lub `doGet()` serwletu. Ktoś musi przekazać żądanie do serwletu oraz odebrać to, co serwlet wygeneruje w odpowiedzi. Ktoś musi zarządzać życiem, śmiercią oraz zasobami używanymi przez serwlet. W tym rozdziale będziemy pisać o kontenerze i po raz pierwszy przyjrzymy się wzorcowi MVC.



Cele egzaminu	66
Czym jest kontener oraz co nam daje?	67
Jak to wszystko wygląda w kodzie (co sprawia, że serwlet jest serwletem)?	72
Określanie nazw serwletów i kojarzenie ich z adresami URL w deskryptorze wdrożenia	74
Opowiadanie: Bob buduje witrynę swatającą (wprowadzenie do wzorca MVC)	78
Ogólne informacje i przykład wzorca model-widok-kontroler (MVC)	82
„Działający” deskryptor wdrożenia (DD)	92
Jaka w tym wszystkim jest rola platformy J2EE?	93

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), deskryptora rozmieszczenia w formacie XML oraz widoku opartego 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.

Katalogi specyficzne dla Tomcata

tomcat
webapps
Piwo-w1

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

Część specyfikacji serwletów

WEB-INF
form.html
wynik.jsp
classes
lib
web.xml

Ten plik web.xml MUSI się znajdować w katalogu WEB-INF

Twoja aplikacja

com
example
web
model

Ta struktura pakietu niczym nie różni się od struktury, której użyliśmy w przygotowanym wcześniej środowisku wytwarzania aplikacji. Jeśli nie składujesz swoich klas w plikach JAR (omówimy te pliki w dalszej części tej książki), koniecznie MUSISZ umieścić strukturę katalogów pakietu bezpośrednio pod katalogiem WEB-INF/classes.

Cele egzaminu	96
Zbudujmy aplikację internetową MVC — pierwszy projekt	97
Tworzenie środowisk wytwarzania i wdrażania aplikacji	100
Tworzenie i testowanie kodu HTML początkowej strony formularza	103
Tworzenie deskryptora wdrożenia (DD)	105
Tworzenie, kompilacja, wdrażanie i testowanie serwletu kontrolera	108
Projektowanie, tworzenie i testowanie komponentu modelu	110
Rozszerzenie kontrolera o wywołania modelu	111
Tworzenie i wdrożenie komponentów widoku (to jest JSP)	115
Rozszerzenie serwletu o wywołanie strony JSP	116

4 Być serwletem

Serwlety żyją, by obsługiwać klientów. 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 dane kluczowe, a kod Twojego serwletu musi wiedzieć, jak należy te dane *odszukać* i jak można ich *użyć*. Co więcej, kod serwletu musi wiedzieć, jak odesłać **odpowiedź**. A jeśli *nie*…

Idempotencja to nic wstydliwego...

NIEpowtarzalne

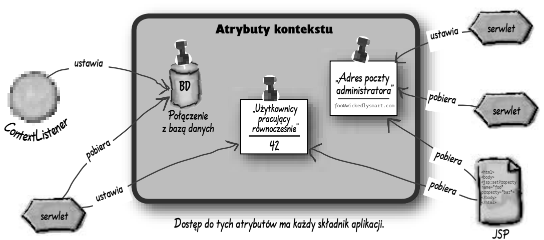
Serwlet wykorzystuje dane dołączone do żądania POST do zaktualizowania bazy danych.

GET
Serwlet
BD
Klient
Serwer odsyła odpowiedź z wygenerowanym kodem strony HTML.

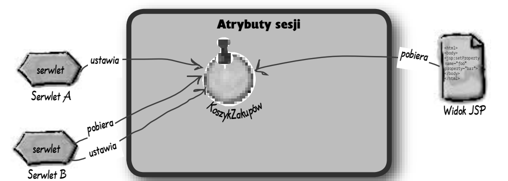
Cele egzaminu	122
Życie serwletu w kontenerze	123
Inicjalizacja i wątki serwletu	129
RZECZYWISTYM celem serwletu jest obsługa żądań `GET` i `POST`	133
Historia pewnego niepowtarzalnego żądania	140
Co sprawia, że przeglądarka wysyła żądanie `GET` albo żądanie `POST`?	145
Wysyłanie i stosowanie parametrów	147
Dobrze, wiemy już, do czego służy klasa `Request`… przyjrzyjmy się teraz klasie `Response`	154
Możesz ustawiać nagłówki odpowiedzi, możesz dodawać nagłówki odpowiedzi	161
Przekierowania a przydzielanie żądania	164
Przypomnienie: `HttpServletResponse`	168

Być aplikacją internetową

Żaden serwlet nie działa samodzielnie. We współczesnych aplikacjach internetowych osiąganie zamierzonego celu jest możliwe dzięki współpracy wielu komponentów. Stosowane są komponenty modelu, widoku oraz kontrolera. Wykorzystuje się także różne klasy pomocnicze. Jednak w jaki sposób wszystkie te elementy wspólnie tworzą jedną aplikację internetową? W jaki sposób komponenty mogą *wspólnie* korzystać z pewnych informacji? Jak *ukrywać* pewne informacje? *Jak zapewnić bezpieczeństwo informacji podczas wykonywania wątków?* Od odpowiedzi na te pytania może zależeć Twoja praca.



Dostęp do tych atrybutów ma każdy składnik aplikacji.



Dostępne tylko dla komponentów z dostępem do określonego obiektu `HttpSession`

Dostępne tylko dla komponentów z dostępem do określonego obiektu `ServletRequest`

Cele egzaminu 176

Wybawieniem są parametry inicjalizacji oraz obiekt `ServletConfig` 178

Jak strona JSP może uzyskać dostęp do parametrów inicjalizacji serwletu? 183

Wybawieniem są parametry inicjalizacji kontekstu 185

Porównanie obiektów `Servlet`***`Config`*** oraz `Servlet`***`Context`*** 187

Chcemy obiektu `ServletContextListener` 192

Przewodnik: prosty obiekt `ServletContextListener` 196

Kompilacja, wdrażanie i testowanie obiektu nasłuchującego 204

Cała historia — obiekt nasłuchujący kontekstu 206

Osiem interfejsów obiektów nasłuchujących — nie tylko zdarzenia kontekstu 208

Czym dokładnie *jest* atrybut i jakie są dla niego rodzaje zasięgów 213

Interfejs API dla atrybutów — ciemna strona atrybutów 217

Zasięg kontekstu nie zapewnia bezpieczeństwa wątków! 220

Jak można zapewnić bezpieczeństwo wątków podczas przetwarzania atrybutów kontekstu? 222

Próba wykorzystania synchronizacji 223

Czy atrybuty sesji są bezpieczne z punktu widzenia wielowątkowości? 226

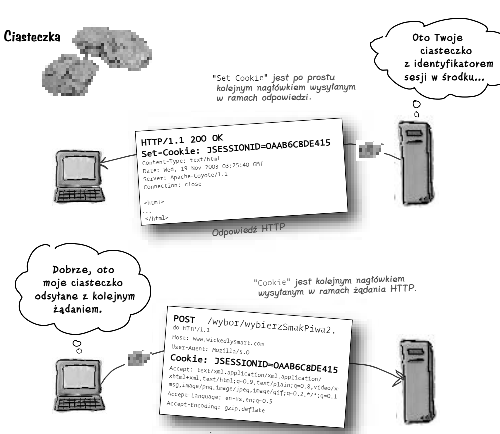
Zły model jednowątkowy 229

Tylko atrybuty żądania i zmienne lokalne są bezpieczne z punktu widzenia wielowątkowości! 230

Atrybuty żądania i przydzielanie żądań 231

6 Stan konwersacyjny

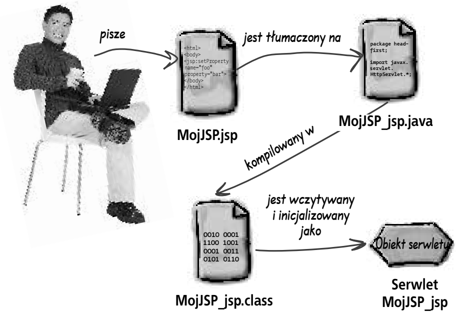
Serwery WWW nie mają pamięci krótkoterminowej. Zaraz po odesłaniu do nas odpowiedzi serwery WWW zapominają, kim jesteśmy. Kiedy wysyłamy kolejne żądanie, docelowy serwer WWW już nas nie rozpoznaje. Innymi słowy, serwery WWW nie pamiętają ani tego, czego żądaliśmy w przeszłości, ani tego, co do nas wysłały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



Cele egzaminu	250
To ma być konwersacja (jak działają sesje?)	252
Identyfikatory sesji, ciasteczka oraz pozostałe podstawy działania sesji	257
Przepisywanie adresów URL, sposób rozwiązania problemu	263
Kiedy sesje stają się nieaktualne — usuwanie sesji błędnych	267
Czy ciasteczka mogą mieć także *inne* zastosowania oprócz obsługi sesji?	276
Kluczowe momenty w życiu obiektu `HttpSession`	280
Nie zapominaj o interfejsie `HttpSessionBindingListener`	282
Migracja sesji	283
Przykłady klas nasłuchujących	287

7 Być stroną JSP

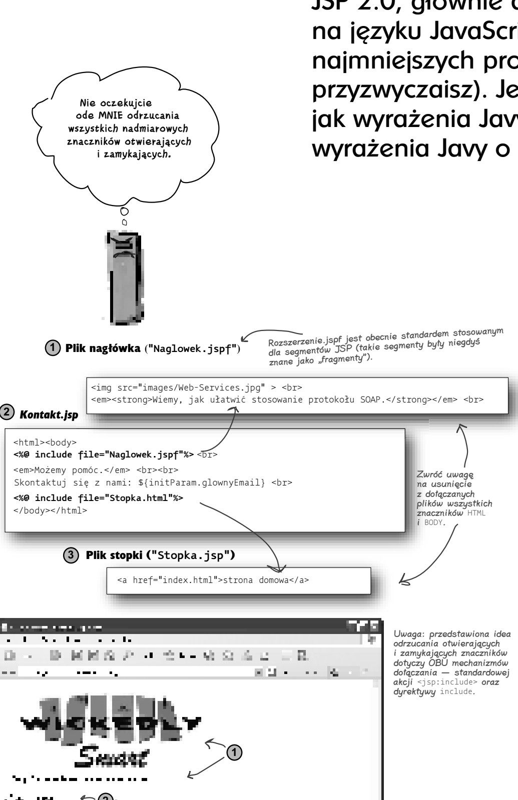
Strona JSP staje się serwletem. Serwletem, którego *nie* musisz tworzyć. Kontener przegląda kod Twojej strony JSP, tłumaczy go na kod źródłowy języka programowania Java i kompiluje tak przetłumaczony kod na postać pełnowartościowej klasy serwletu Javy. Musisz jednak wiedzieć, co dzieje się w czasie konwertowania Twojego kodu strony JSP na kod Javy. W ramach kodu JSP *możesz* co prawda umieszczać kod Javy, ale czy na pewno powinieneś? A jeśli w kodzie stron JSP nie umieszczamy żadnych wyrażeń języka Java, to co *znajdzie* się w tym kodzie? Przyjrzymy się sześciu rodzajom elementów JSP — z których każdy ma swoje przeznaczenie i, no niestety, *odmienną składnię*. Dowiesz się, co i dlaczego można umieszczać w kodzie stron JSP. Dowiesz się także, czego *nie* należy umieszczać w kodzie stron JSP.



Cele egzaminu	308
Tworzymy prostą stronę JSP wykorzystującą zmienną `out` i dyrektywę `page`	309
Wyrażenia JSP, zmienne oraz deklaracje	314
Czas się zapoznać z wygenerowanym serwletem	322
Zmienna `out` nie jest jedynym obiektem domyślnym…	324
Cykl życia i inicjalizacja stron JSP	332
Skoro już poruszyliśmy ten temat… porozmawiajmy o trzech *dyrektywach*	340
Czy skryptlety można uznać za niebezpieczne? Oto EL	343
Ale zaczekaj… nie widzieliśmy jeszcze akcji	349

8 Strony bezskryptowe

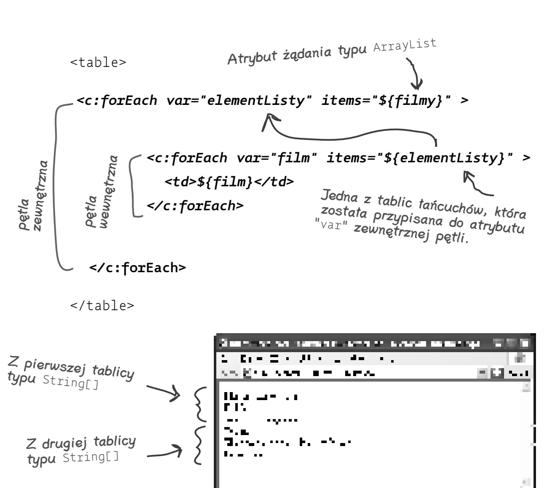
Porzuć skrypty. Czy współpracujący z Tobą projektanci stron internetowych naprawdę muszą znać Javę? Czy oczekują oni od programistów Javy, aby byli jednocześnie np. grafikami? A jeśli przyjmiemy nawet, że Ty jesteś *jedynym* członkiem zespołu, czy rzeczywiście chciałbyś umieszczać rozbudowane fragmenty 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 w specyfikacji JSP 2.0, głównie dzięki nowemu językowi wyrażeń. Dzięki temu, iż bazuje on na języku JavaScript oraz XPATH, projektanci stron mogą go stosować bez najmniejszych problemów; zresztą także Ty go polubisz (kiedy już się do niego przyzwyczaisz). Jednak EL ma także pewne pułapki. Jego wyrażenia *wyglądają* jak wyrażenia Javy, jednak nimi nie są. Czasami wyrażenia EL działają inaczej niż wyrażenia Javy o identycznej składni, a zatem musisz uważać!



Cele egzaminu	370
Które atrybuty są *komponentami JavaBean*	371
Standardowe akcje: useBean, getProperty oraz setProperty	374
Czy można tworzyć polimorficzne referencje do komponentów?	380
Rozwiązaniem jest użycie atrybutu ***param***	386
Konwersja właściwości	389
Uratował nas język wyrażeń (EL)	394
Stosowanie operatora kropki (.) do uzyskiwania dostępu do właściwości oraz odwzorowywania wartości	396
Operator [] daje nam więcej możliwości (listy, tablice…)	397
Więcej szczegółów o operatorach kropki (.) oraz []	402
Obiekty domyślne języka EL	409
Funkcje języka EL i obsługa wartości „null”	415
Szablony wielokrotnego użytku — dwa rodzaje „dołączania”	426
Standardowa akcja <jsp:forward>	437
Ona nie wie jeszcze o znacznikach JSTL (zapowiedź)	441
Przegląd standardowych akcji i dołączania	442

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 *zdajesz* sobie sprawę z tego, że można w tym celu błyskawicznie skonstruować odpowiednią pętlę w skryptlecie. Jednak staramy się nie używać kodu skryptowego. To żaden problem. Gdyby język wyrażeń (EL) oraz akcje standardowe okazały się niewystarczające, to zawsze można wykorzystać *znaczniki niestandardowe*. W kodzie stron JSP stosowanie ich jest równie proste co stosowanie akcji standardowych. Najlepsze jest to, że ktoś już napisał takie znaczniki, których najprawdopodobniej będziesz potrzebował, i umieścił je 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.



Cele egzaminu	462
Pętle bez skryptów: `<c:forEach>`	464
Kontrola warunkowa przy wykorzystaniu znaczników `<c:if>` oraz `<c:choose>`	469
Zastosowanie znaczników `<c:set>` oraz `<c:remove>`	473
Dzięki znacznikowi `<c:import>` mamy do dyspozycji aż *trzy* metody dołączania treści	478
Dostosowywanie dołączanej zawartości	480
Realizacja tego samego zadania za pomocą znacznika `<c:param>`	481
Stosowanie znacznika `<c:url>` do realizacji wszystkich zadań związanych z obsługą hiperłączy	483
Tworzenie własnych stron o błędach	486
Znacznik `<c:catch>`, który przypomina trochę… konstrukcję `try-catch`	490
Co będzie, jeśli uznamy za niezbędne użycie znacznika SPOZA biblioteki JSTL?	493
Zwróć uwagę na element `<rtexprvalue>`	498
Co może się znaleźć w ciele znacznika	500
Klasa obsługująca znacznik, deskryptor TLD i strona JSP	501
Podelement `<uri>` elementu `taglib` jest tylko nazwą, nie lokalizacją	502
Kiedy strona JSP wykorzystuje więcej niż jedną bibliotekę znaczników	505

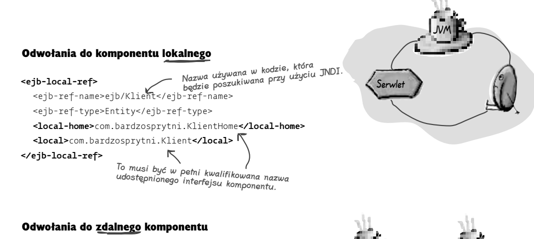
Kiedy JSTL nie wystarcza...

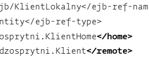
Czasami JSTL i standardowe akcje nie wystarczają. Kiedy trzeba zrobić coś niestandardowego, a nie chcesz uciekać się do stosowania skryptu, możesz stworzyć *własne* procedury obsługi znaczników. Dzięki temu projektanci stron będą mogli używać w projektowanych stronach Twoich *znaczników*, a całą „czarną robotę" będzie, w niewidoczny sposób, realizować Twoja *klasa* obsługi znacznika. Niemniej jednak czeka Cię dużo nauki, gdyż własne niestandardowe znaczniki można tworzyć na trzy różne sposoby. Dwa spośród nich zostały wprowadzone w JSP 2.0, aby ułatwić nam życie (chodzi o *znaczniki proste* oraz *pliki znaczników*).

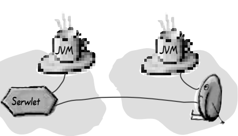
Cele egzaminu	518
Pliki znaczników — podobnie jak znacznik *include*, tylko lepiej	520
Gdzie kontener szuka plików znaczników?	527
Tworzenie prostej klasy obsługi znacznika	531
A co, jeśli w zawartości znacznika pojawi się wyrażenie?	537
Wciąż musimy znać klasyczne klasy obsługi znaczników niestandardowych	547
Interfejs programowy klas obsługi znaczników	548
Bardzo prosta klasyczna klasa obsługi znacznika niestandardowego	549
Cykl istnienia znacznika klasycznego zależy od wartości zwracanych	554
Interfejs `IterationTag` pozwala na wielokrotne przetwarzanie zawartości znacznika	555
Domyślne wartości zwracane przez metody klasy `TagSupport`	557
Interfejs `BodyTag` udostępnia dwie kolejne metody	561
A co, jeśli znaczniki współpracują ze sobą?	565
Wykorzystanie interfejsu programowego klasy `PageContext`	575

Jak wdrożyć aplikację internetową?

W końcu Twoja aplikacja jest skończona. 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 zasad. Jakie nazwy *powinieneś* nadać katalogom? Jakie nazwy nadałby *klient*? Do jakich zasobów tak naprawdę będzie się odwoływać klient i skąd kontener ma wiedzieć, gdzie należy ich szukać?







Cele egzaminu	598
Podstawowe zagadnienia związane z wdrażaniem różnych elementów aplikacji	599
Pliki WAR	607
Jak NAPRAWDĘ działają odwzorowania serwletów?	612
Konfiguracja plików powitalnych w deskryptorze rozmieszczenia	618
Konfiguracja stron błędów w deskryptorze rozmieszczenia	622
Konfigurowanie inicjalizacji serwletu w deskryptorze rozmieszczenia	624
Tworzenie stron JSP zgodnych z zasadami konstrukcji dokumentów XML — dokumenty JSP	625

12 Zachowaj je w tajemnicy, ukryj je w bezpiecznym miejscu

Twoja aplikacja internetowa jest w niebezpieczeństwie. Problemy czyhają w każdym zakamarku sieci. Nie chcesz, aby ci źli faceci podsłuchiwali transakcje realizowane w Twoim sklepie internetowym albo przechwytywali numery kart kredytowych. Nie chcesz, by byli w stanie przekonać Twój serwer, iż tak naprawdę są Bardzo Ważnymi Klientami Posiadającymi Bardzo Duże Upusty. I w końcu nie chcesz, by *ktokolwiek* (niezależnie od tego, czy dobry czy zły) miał dostęp do ważnych informacji na temat pracowników. Czy Janek z działu marketingu naprawdę musi wiedzieć, że Liza z działu technicznego zarabia trzy razy więcej?

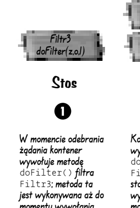
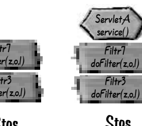
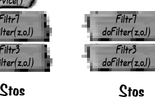
Dziesięć najważniejszych powodów przemawiających za *deklaratywnym* określaniem bezpieczeństwa

10. *Kto nie potrzebuje lepszego opanowania sposobów wykorzystania języka XML?*
9. Często można w naturalny sposób odwzorować stanowiska zajmowane przez poszczególne osoby w dziale informatycznym firmy.
8. To świetnie wygląda w życiorysie zawodowym.
7. Pozwala na bardziej elastyczne wykorzystywanie już napisanych serwletów.
6. Te zagadnienia pojawiają się na egzaminie.
5. Dzięki temu programiści zajmujący się tworzeniem aplikacji mogą wielokrotnie używać serwletów bez konieczności dostępu do ich kodu źródłowego.
4. Bo tak jest fajnie!
3. Rozwiązanie to pozwala na ograniczenie kosztów utrzymania wraz z rozwojem aplikacji.
2. Jest to czynnik, którym można wytłumaczyć i usprawiedliwić wysoką cenę kontenera.
1. Rozwiązanie to jest zgodne z ideą programowania bazującego na stosowaniu komponentów.

Cele egzaminu	646
Wielka czwórka bezpieczeństwa serwletów	649
Jak realizować uwierzytelnianie w świecie protokołu HTTP?	652
Dziesięć najważniejszych powodów przemawiających za *deklaratywnym* określaniem bezpieczeństwa	655
Kto zajmuje się zabezpieczeniem aplikacji internetowej?	656
Autoryzacja: role i ograniczenia	659
Uwierzytelnianie: cztery odmiany	673
CZTERY typy uwierzytelniania	673
Zabezpieczanie przesyłanych informacji — protokół HTTPS śpieszy z pomocą	678
W jaki sposób wybiórczo i deklaratywnie zaimplementować poufność i integralność danych?	680

13 Potęga filtrów

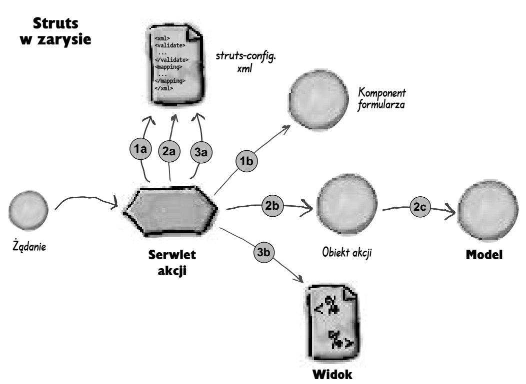
Filtry pozwalają na przechwytywanie żądań. A jeśli można przechwycić *żądanie*, to można także kontrolować *odpowiedź*. Ale najlepsze w tym wszystkim jest to, że serwlet nie ma o tym **najmniejszego pojęcia**. Serwlet nigdy nie wie, czy coś się zdarzyło w czasie pomiędzy odebraniem żądania przez kontener a wywołaniem metody `service()` serwletu. A 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? Nie ma żadnego problemu. A co najlepsze — nie musisz przy tym w *żaden sposób* modyfikować kodu serwletów.

Cele egzaminu	698
Tworzenie filtra służącego do śledzenia żądań	703
Cykl istnienia filtrów	704
Deklarowanie i określanie kolejności filtrów	706
Kompresja wyników przy wykorzystaniu filtra operującego na odpowiedzi	709
Opakowania są świetne!	715
Prawdziwy kod filtra kompresji odpowiedzi	718
Kod opakowania kompresji odpowiedzi	720

Korporacyjne wzorce projektowe

Ktoś to już wcześniej zrobił. Jeśli właśnie zaczynasz tworzyć aplikacje internetowe w języku Java, to masz dużo szczęścia. Możesz wykorzystać wspólną mądrość dziesiątków tysięcy programistów, którzy już od dawna się tym zajmują i mają już firmową koszulkę. Wykorzystując wzorce projektowe — zarówno te związane z platformą J2EE, jak i wszelkie inne — możesz uprościć swój kod i swoje życie. W przypadku aplikacji internetowych najważniejszym wzorcem projektowym jest wzorzec MVC. Został on wykorzystany w bardzo popularnym szkielecie aplikacji, o nazwie Struts, który pomaga przy tworzeniu elastycznego i łatwego w utrzymaniu serwletu Kontrolera Frontowego. Wykorzystanie pracy *innych* jesteś winien samemu sobie — dzięki temu będziesz mógł poświęcić więcej czasu na ważniejsze sprawy.



Cele egzaminu	734
Sprzętowe i programowe siły związane z wzorcami projektowymi	735
Przegląd zasad związanych z projektowaniem oprogramowania	739
Wzorce wspomagają zdalne komponenty modelu	741
JNDI oraz RMI — krótka prezentacja	743
Delegat Biznesowy jest obiektem pośredniczącym	749
Uproszczenie używanych delegatów biznesowych poprzez zastosowanie lokalizatora usług	750
Nadszedł czas na przedstawienie obiektu transferowego?	755
Nasz pierwszy wzorzec po raz wtóry — MVC	758
Tak! To Struts (i Kontroler Frontowy) w zarysie	763
Przystosowanie aplikacji piwnej do korzystania ze szkieletu Struts	766
Przegląd wzorców projektowych	774

Ostateczny Egzamin Próbny Baru Kawowego. To jest to. 69 pytań. Charakter, zagadnienia i poziom trudności jest niemal identyczny jak na *prawdziwym egzaminie. My to wiemy*.

Ostateczny egzamin próbny	787
Odpowiedzi	826

Skorowidz 865

Rozdział 4. Żądanie i odpowiedź

Być serwletem

Serwlety żyją dla klientów korzystających z usług. Zadaniem serwletu jest obsługa ***żądań*** klientów i odsyłanie do klienta odpowiednich ***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 można ich *użyć*. Odpowiedź musi zawierać informacje niezbędne do wizualizowania strony (lub pobranych bajtów) przez przeglądarkę, zatem kod Twojego serwletu musi wiedzieć, jak te informacje *wysłać*. Okazuje się jednak, że może być *inaczej*… Twój serwlet może decydować o przekazaniu żądania zupełnie *gdzie indziej* — do innej strony, serwletu lub JSP.

Wdrażanie 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 klasy `HttpServlet`, która odpowiada danej metodzie protokołu HTTP.

1.2. Wykorzystując interfejs `HttpServletRequest`, napisz kod odczytujący z żądania protokołu HTTP parametry formularza HTML, odczytujący informacje nagłówka protokołu HTTP lub odczytujący z żądania ciasteczka klienta.

1.3. Wykorzystując interfejs `HttpServletResponse`, napisz kod, który ustawia nagłówek odpowiedzi protokołu HTTP, ustawia typ zawartości odpowiedzi, uzyskuje dostęp do strumienia tekstowego odpowiedzi, uzyskuje dostęp do strumienia binarnego odpowiedzi, przekierowuje żądanie HTTP na inny adres URL lub dodaje do odpowiedzi ciasteczka dla klienta[1].

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

Uwagi wyjaśniające:

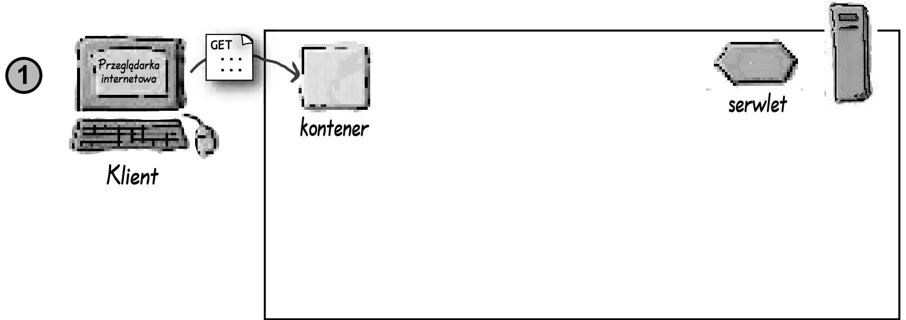
Wszystkie wymienione obok cele egzaminu zostaną dokładnie omówione jeszcze w tym rozdziale (wyjątkiem jest fragment celu 1.3 poświęcony ciasteczkom). O znacznej części zagadnień opisywanych w tym rozdziale wspominaliśmy już w rozdziale 2., jednak w tamtym rozdziale zasugerowaliśmy, iż Czytelnik nie musi zapamiętywać prezentowanych treści.
W tym rozdziale MUSISZ nieco zwolnić i przystąpić do rzeczywistego studiowania przedstawianego materiału włącznie z jego zapamiętywaniem. Pamiętaj, że do szczegółów wymienionych obok celów egzaminu nie będziemy już wracali w kolejnych rozdziałach, zatem jest to Twoja ostatnia szansa na ich przeanalizowanie.
Wykonuj wszystkie ćwiczenia opisywane w tym rozdziale, dokładnie analizuj cały jego materiał i w skupieniu spróbuj odpowiedzieć na pytania pierwszego egzaminu próbnego na końcu rozdziału. Jeśli odsetek prawidłowych odpowiedzi nie przekroczy 80%, ZANIM przystąpisz do lektury kolejnego (piątego) rozdziału, wróć do treści rozdziału i jeszcze raz zapoznaj się z materiałem, którego nieznajomość została ujawniona w trakcie testu.
Niektóre z próbnych pytań egzaminacyjnych, które dotyczą wymienionych obok celów, zostały przeniesione do rozdziałów 5. i 6., ponieważ odpowiedź na nie wymaga dodatkowej wiedzy w zakresie omawianym dopiero w tamtych rozdziałach. Oznacza to, że po przeczytaniu tego rozdziału będziesz musiał odpowiedzieć na mniejszą liczbę pytań egzaminacyjnych niż w późniejszych rozdziałach (w ten sposób uniknęliśmy testowania wiedzy, której nie miałeś jeszcze okazji posiąść).

Ważna uwaga: o ile w pierwszych trzech rozdziałach omawialiśmy przede wszystkim informacje stanowiące tło dla właściwej treści tej książki, o tyle już od następnej strony niemal cały materiał będzie bezpośrednio związany z konkretnymi fragmentami egzaminu.

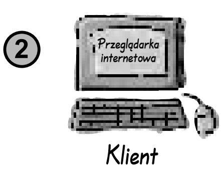
[1] Celom egzaminu związanym z ciasteczkami klienta poświęcimy więcej czasu dopiero w rozdziale dotyczącym sesji.

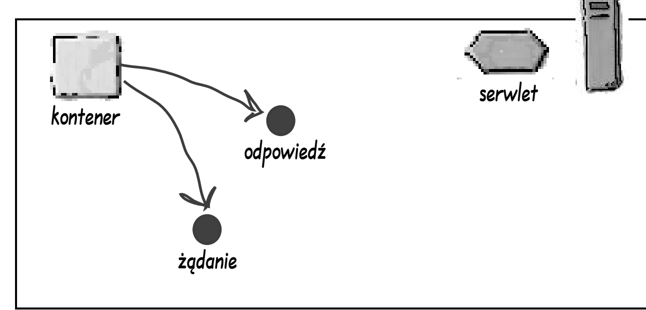
Serwlety są kontrolowane przez kontener

W rozdziale drugim pobieżnie omówiliśmy rolę kontenera w życiu serwletu — tworzy on obiekty żądania i odpowiedzi, tworzy lub przydziela pamięć dla nowego wątku serwletu oraz wywołuje metodę `service()` serwletu (przekazując w postaci argumentów tej funkcji referencje do obiektów żądania i odpowiedzi). Oto krótkie przypomnienie…



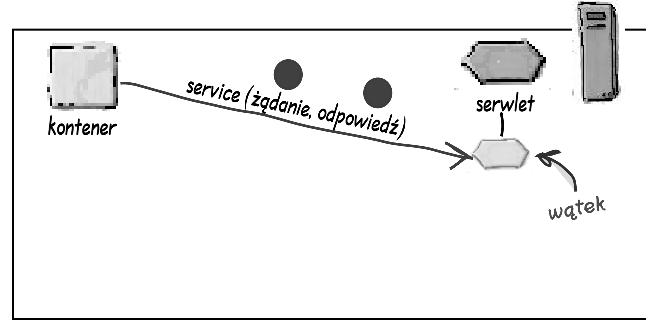
Użytkownik klika łącze obejmujące adres URL do serwletu.





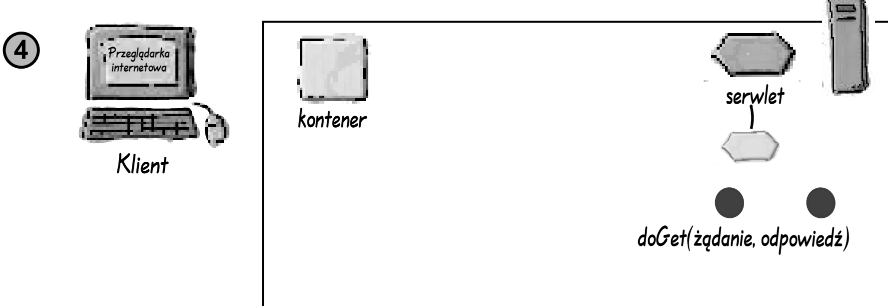
Kontener „widzi”, że żądanie jest kierowane do serwletu, zatem tworzy dwa niezbędne obiekty:

1. HttpServletResponse
2. HttpServletRequest



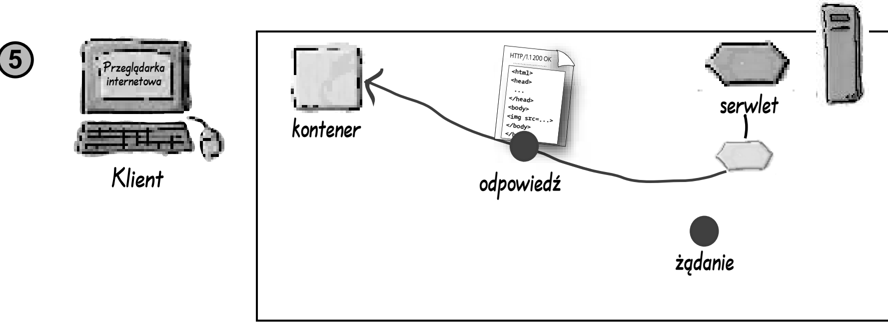
Kontener odnajduje odpowiedni serwlet na podstawie adresu URL dołączonego do żądania, tworzy lub przydziela pamięć dla wątku obsługującego to żądanie oraz wywołuje metodę `service()` serwletu i przekazuje w postaci argumentów obiekty reprezentujące żądanie i odpowiedź.

Dalszy ciąg historii życia serwletu...

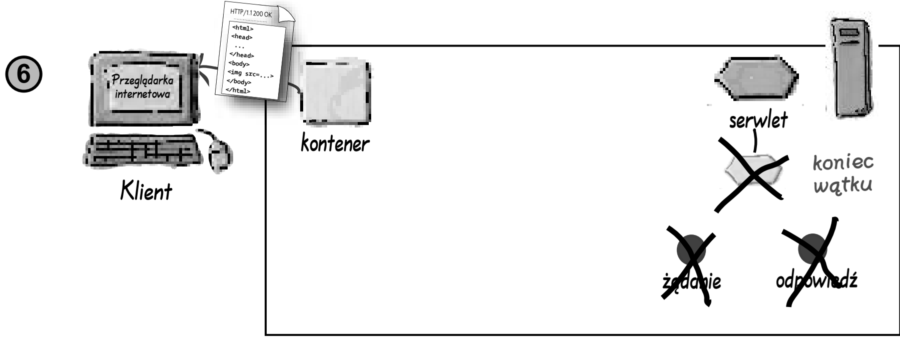


Metoda `service()` określa na postawie przysłanej przez klienta metody protokołu HTTP (`GET`, `POST` itp.), którą metodę serwletu należy wykonać.

Klient wysłał żądanie HTTP `GET`, zatem metoda `service()` wywołuje metodę `doGet()` z obiektem żądania i obiektem odpowiedzi przekazanymi w formie argumentów.



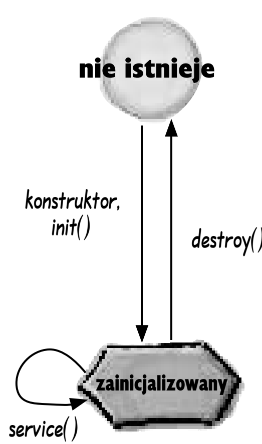
Serwlet wykorzystuje obiekt odpowiedzi do zapisania swojej odpowiedzi dla klienta. Odpowiedź jest odsyłana do klienta za pośrednictwem kontenera.



Wykonywanie metody `service()` kończy się, zatem odpowiedni wątek jest albo zabijany, albo zwracany do zarządzanej przez kontener puli wątków. Referencje do obiektów reprezentujących żądanie i odpowiedź wypadają poza bieżący zakres, zatem zajmowana przez nie pamięć może zostać zwolniona (w procesie odśmiecania pamięci).

Klient otrzymuje wygenerowaną odpowiedź.

Życie serwletu toczy się jednak nadal



W naszych rozważaniach gładko przeszliśmy do środka życia serwletu, nadal jednak pozostawiliśmy bez odpowiedzi wiele istotnych pytań. Kiedy klasa serwletu jest wczytywana? Kiedy jest wywoływany konstruktor serwletu? Jak długo żyje obiekt serwletu? Kiedy powinniśmy inicjalizować zasoby naszego serwletu? Kiedy należy te zasoby zwolnić?

Cykl życia serwletu jest prosty; istnieje tylko jeden stan — ***zainicjalizowany***. Jeśli serwlet nie jest zainicjalizowany, musi się znajdować w stanie *jest inicjalizowany* (jeśli w danej chwili wykonywany jest jego konstruktor lub metoda `init()`), w stanie *jest niszczony* (jeśli w danym momencie wykonywana jest jego metoda `destroy()`) lub w stanie ***nie istnieje***.

Kontener WWW **Klasa serwletu** **Obiekt serwletu**

Kontener

Wczytaj klasę

Serwlet.class

Stwórz egzemplarz serwletu (wywołaj konstruktor)

Wykonywany jest bezargumentowy konstruktor Twojej klasy serwletu (NIE powinieneś pisać kodu konstruktora, należy raczej korzystać z obsługiwanego przez kompilator konstruktora domyślnego).

init()

Metoda wywoływana tylko RAZ w całym cyklu życia serwletu — jej wykonanie musi się zakończyć, zanim kontener będzie mógł wywołać metodę `service()`

W tym miejscu serwlet spędza większość swojego życia.

service()

zainicjalizowany

obsługuje żądania klienta

doGet(), doPost() etc.

(Każde żądanie jest obsługiwane w osobnym wątku.)

destroy()

zainicjalizowany

Kontener wywołuje metodę `destroy()`, aby umożliwić serwletowi zwolnienie wszelkich przydzielonych mu zasobów przed jego zniszczeniem (czyli przygotować go do procesu odśmiecania pamięci). Podobnie jak metoda `init()` także metoda `destroy()` jest wywoływana tylko raz.

Twój serwlet dziedziczy metody cyklu życia

UWAGA: NIE próbuj zapamiętywać wszystkich wymienionych na tej stronie informacji! Zapoznaj się tylko ze sposobem funkcjonowania tego interfejsu API...

<<interface>>
Servlet

service(ServletRequest, ServletResponse)
init(ServletConfig)
destroy()
getServletConfig()
getServletInfo()

Interfejs Servlet

(javax.servlet.Servlet)

Interfejs Servlet *określa, że wszystkie serwlety muszą zawierać te pięć metod (trójka metod oznaczona pogrubieniem należy do cyklu życia serwletu).*

GenericServlet

service(ServletRequest, ServletResponse)
init(ServletConfig)
init()
destroy()
getServletConfig()
getServletInfo()
getInitParameter(String)
getInitParameterNames()
getServletContext()
log(String)
log(String, Throwable)

Klasa GenericServlet

(javax.servlet.GenericServlet)

GenericServlet *jest klasą abstrakcyjną, która implementuje większość potrzebnych nam metod serwletów, włącznie z tymi zadeklarowanymi w interfejsie* Servlet. *Prawdopodobnie NIGDY nie będziesz samodzielnie rozszerzać tej klasy. „Zachowanie" większości Twoich serwletów będzie pochodziło właśnie z klasy* GenericServlet.

HttpServlet

service(HttpServletRequest, HttpServletResponse)
service(ServletRequest, ServletResponse)
doGet(HttpServletRequest, HttpServletResponse)
doPost(HttpServletRequest, HttpServletResponse)
doHead(HttpServletRequest, HttpServletResponse)
doOptions(HttpServletRequest, HttpServletResponse)
doPut(HttpServletRequest, HttpServletResponse)
doTrace(HttpServletRequest, HttpServletResponse)
doDelete(HttpServletRequest, HttpServletResponse)
getLastModified(HttpServletRequest)

Klasa HttpServlet

(javax.servlet.http.HttpServlet)

Klasa HttpServlet *(także abstrakcyjna) implementuje metodę* service()*, która jest niezbędna do zapewnienia zgodności serwletu z protokołem* HTTP *— metoda* service() *nie pobiera ŻADNYCH żądań i odpowiedzi serwletu, tylko właśnie żądania i odpowiedzi* HTTP.

MojSerwlet

doPost(HttpServletRequest, HttpServletResponse)
mojaMetodaBiznesowa()

Klasa MojSerwlet

(com.wickedlysmart.foo)

Większość serwletowości Twojej aplikacji jest obsługiwana przez metody pochodzące z rozszerzanych nadklas. W swojej klasie serwletu musisz przykryć tylko te metody HTTP, których potrzebujesz.

Trzy najważniejsze momenty w cyklu życia serwletu

init()

Kiedy jest wywoływana?

Kontener wywołuje metodę `init()` należącą do egzemplarza serwletu *po* utworzeniu tego egzemplarza, ale *zanim* dany serwlet będzie mógł obsłużyć jakiekolwiek żądania klientów.

Do czego służy?

Daje programiście możliwość inicjalizowania serwletu jeszcze przed przystąpieniem do obsługi żądań klientów.

Czy będziesz ją przykrywał?

Być może.

Jeśli dysponujesz kodem inicjalizującym aplikację internetową (np. przez utworzenie połączenia z bazą danych lub zarejestrowanie innych obiektów), musisz przykryć metodę `init()` w klasie swojego serwletu.

---

service()

Kiedy jest wywoływana?

Kiedy do serwera dociera żądanie klienta, kontener uruchamia nowy wątek serwletu lub przydziela do nowego zadania istniejący wątek z dostępnej puli wątków, po czym wymusza wywołanie metody `service()` danego serwletu.

Do czego służy?

Metoda analizuje otrzymane żądanie, określa metodę protokołu HTTP (`GET`, `POST` itp.) i wywołuje odpowiednią metodę (`doGet()`, `doPost()` itp.) udostępnianą przez dany serwlet.

Czy będziesz ją przykrywał?

Nie, to bardzo mało prawdopodobne.
NIE powinieneś przykrywać metody `service()`.
Twoim zadaniem jest przykrycie metody `doGet()` i (lub) metody `doPost()` oraz pozostawienie odpowiedzialności za wywołanie właściwej części kodu metodzie `service()` klasy `HttpServlet`.

---

doGet()

i (lub)

doPost()

Kiedy są wywoływane?

Metoda `service()` wywołuje metodę `doGet()` lub `doPost()` w zależności od metody protokołu HTTP (`GET`, `POST` itp.) otrzymanej w ramach żądania od klienta.
(W tym miejscu wspominamy wyłącznie o metodach `doGet()` i `doPost()`, ponieważ są to prawdopodobnie jedyne tego typu metody, z których kiedykolwiek będziesz korzystał.)

Do czego służą?

Właśnie w tym miejscu zaczyna się *Twój* kod! Metoda `doGet()` lub `doPost()` jest odpowiedzialna za realizację właściwych ZADAŃ Twojej aplikacji internetowej.
Możesz oczywiście wywoływać inne metody należące do innych obiektów, jednak wszelkie tego typu działania rozpoczynają się właśnie od tego miejsca.

Czy będziesz je przykrywał?

ZAWSZE, przynajmniej jedną z nich (doGet() lub doPost())!
To, którą metodę (lub które metody) przykrywasz, jest sygnałem dla kontenera, które typy żądań będą obsługiwane w Twoim serwlecie. Przykładowo, jeśli nie przykryjesz metody `doPost()`, tym samym dasz kontenerowi jasno do zrozumienia, ze Twój serwlet nie obsługuje żądań `POST` protokołu HTTP.

Myślę, że już wiem, o co chodzi... zacznijmy od początku: kontener wywołuje należącą do mojego serwletu metodę `init()`, jeśli jednak tej metody nie przykryję w swoim kodzie, zostanie wykonana domyślna metoda `init()` z klasy `GenericServlet`. Następnie, kiedy do serwera dociera żądanie klienta, kontener uruchamia nowy wątek lub wykorzystuje jeden z wątków dostępnych w puli, po czym wywołuje metodę `service()`, której nie przykrywam (zatem faktycznie wykonywana jest metoda `service()` z klasy `HttpServlet`). Należąca do klasy `HttpServlet` metoda `service()` wywołuje następnie przykrytą przeze mnie metodę `doGet()` lub `doPost()`. Oznacza to, że za każdym razem, gdy wywoływana jest moja metoda `doGet()` lub `doPost()`, metoda ta jest wykonywana w osobnym wątku.

Metoda `service()` zawsze jest wywoływana w ramach jej własnego stosu ...

Inicjalizacja serwletu

Wątek A

Kontener wywołuje metodę `init()` egzemplarza serwletu, oczywiście już *po* utworzeniu tego egzemplarza, ale *zanim* serwlet będzie mógł obsłużyć jakiekolwiek żądania klienta.

Jeśli dysponujesz kodem inicjalizującym aplikację internetową (np. przez utworzenie połączenia z bazą danych lub zarejestrowanie innych obiektów), musisz przykryć metodę `init()` w klasie swojego serwletu. W przeciwnym razie wykonywana jest metoda `init()` należąca do klasy `GenericServlet`.

Pierwsze żądanie klienta

Wątek B

Kiedy do serwera WWW dociera pierwsze żądanie klienta, kontener uruchamia (lub znajduje w puli) wątek i wymusza wywołanie metody `service()` serwletu wskazanego w żądaniu.

Zazwyczaj NIE będziesz przykrywał metody `service()`, zatem w większości przypadków wykonywana będzie metoda `service()` należąca do klasy bazowej `HttpServlet`. Metoda `service()` określa, której metody protokołu HTTP (`GET`, `POST` itp.) użyto w żądaniu, i wywołuje odpowiednią metodę serwletu (odpowiednio `doGet()` lub `doPost()`). Zdefiniowane w klasie `HttpServlet` metody `doGet()` i `doPost()` nie wykonują żadnych działań, zatem przynajmniej jedną z tych metod musimy przykryć w kodzie naszego serwletu. Kiedy działanie metody `service()` się kończy, odpowiedni wątek jest zabijany lub zwracany do puli zarządzanej przez kontener.

Drugie żądanie klienta

Wątek C

Kiedy do serwera WWW dotrze drugie (i każde kolejne) żądanie klienta, kontener ponownie utworzy lub wyszuka w puli wątek i wymusi wywołanie metody `service()` należącej do odpowiedniego serwletu.

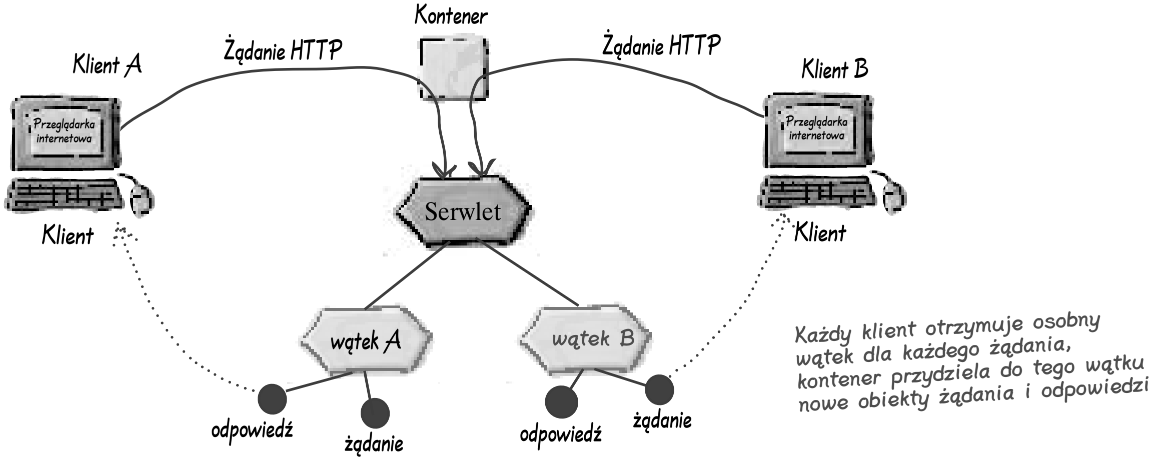
Zatem sekwencja metod `servlet()` i `doGet()` jest wykonywana za każdym razem, gdy do serwera dociera żądanie klienta. W dowolnym momencie będziesz miał dokładnie tyle wykonywalnych wątków, ile żądań klientów będzie obsługiwanych przez Twoją aplikację internetową — ich liczba może być ograniczana przez dostępne zasoby lub strategie (ustawienia konfiguracyjne) zastosowane w kontenerze (możesz np. wykorzystywać kontener, który umożliwia określanie maksymalnej liczby jednocześnie utrzymywanych wątków i oferuje mechanizm kolejkowania żądań oczekujących na przydział własnych wątków).

Każde żądanie jest realizowane w osobnym wątku!

Być może miałeś okazję usłyszeć wyrażenie „każdy egzemplarz serwletu…”, takie określenie jest jednak *niepoprawne*. Nie istnieje wiele *egzemplarzy* żadnej klasy serwletu z wyjątkiem bardzo specyficznych przypadków (pracujących w rzadko stosowanym i niezalecanym modelu jednowątkowym), których w tej książce nie będziemy omawiali.

Kontener uruchamia wiele wątków, których zadaniem jest przetworzenie wielu żądań adresowanych do pojedynczego serwletu.

Każde żądanie klienta generuje także nową parę obiektów reprezentujących żądanie i odpowiedź.



Nie ma niemądrych pytań

P: Wciąż mam pewne wątpliwości… na powyższym rysunku przedstawiono dwóch klientów, z których każdy korzysta z własnego wątku. Co się stanie, jeśli ten sam klient wygeneruje wiele żądań? Czy kontener stworzy wówczas po jednym wątku dla każdego klienta czy po jednym wątku dla każdego żądania?

O: Po jednym wątku dla każdego żądania. Kontenera w ogóle nie interesuje, kto przysłał dane żądanie — każde otrzymane żądanie wiąże się z koniecznością stworzenia (przydzielenia) nowego wątku-stosu.

P: Co się stanie, jeśli kontener wykorzystuje technikę łączenia w klastry i rozprasza aplikację internetową na więcej niż jednej wirtualnej maszynie Javy?

O: Wyobraź sobie, że przedstawiony powyżej rysunek dotyczy pojedynczej wirtualnej maszyny Javy oraz że taki sam rysunek istnieje dla każdej wirtualnej maszyny należącej do naszej aplikacji. W przypadku rozproszonej aplikacji internetowej musiałby istnieć jeden egzemplarz konkretnego serwletu w każdej wirtualnej maszynie Javy, ale pojedyncza wirtualna maszyna Javy nadal może utrzymywać tylko jeden egzemplarz tego serwletu.

P: Zauważyłem, że klasa HttpServlet należy do innego pakietu niż klasa GenericServlet… ile wobec tego istnieje pakietów dla serwletów?

O: Wszystko, co wiąże się z serwletami (poza elementami technologii JSP), jest obsługiwane albo w pakiecie `javax.servlet`, albo w pakiecie `javax.servlet.http`. Nietrudno także opisać różnice pomiędzy tymi pakietami… kod związany z obsługą żądań protokołu HTTP należy do pakietu `javax.servlet.http`, cała reszta (ogólne klasy i interfejsy serwletów) jest zakodowana w pakiecie `javax.servlet`. Technologii JSP poświęcimy kilka rozdziałów w dalszej części tej książki.

Etap pierwszy. Wczytywanie i inicjalizacja

Życie serwletu rozpoczyna się w momencie, w którym kontener (odpowiedzialny za realizację żądań klientów) odnajduje plik klasy serwletu. Klasa serwletu jest niemal zawsze odnajdywana już w chwili rozpoczęcia pracy kontenera (np. w momencie uruchomienia Tomcata). Uruchamiany kontener przegląda strukturę katalogów w poszukiwaniu wdrożonych aplikacji internetowych, po czym przystępuje do odnajdywania wszystkich plików klas serwletów, które składają się na znalezione aplikacje (proces odnajdywania serwletów omówimy bardziej szczegółowo w rozdziale poświęconym wdrażaniu aplikacji internetowych).

Samo *znalezienie* pliku klasy jest tylko pierwszym krokiem.

Wczytanie klasy jest drugim krokiem, którego realizacja ma miejsce albo podczas *uruchamiania kontenera*, albo w czasie *obsługi pierwszego żądania klienta*. Twój kontener może Ci umożliwić wybór odnośnie czasu wczytywania klas, ale może także wczytywać klasy w dowolnym wybranym przez siebie momencie. Niezależnie od tego, czy Twój kontener z góry przygotowuje serwlety aplikacji internetowych czy wczytuje je na bieżąco, w odpowiedzi na potrzeby zgłaszane przez użytkownika, *metoda* `service()` *serwletu nigdy nie zostanie wykonana przed pełną inicjalizacją serwletu*.

Twój serwlet zawsze musi zostać wczytany i zainicjalizowany ZANIM będzie mógł obsłużyć pierwsze żądania klientów.

Metoda `init()` zawsze kończy swoje działanie przed pierwszym wywołaniem metody `service()`.

WYTĘŻ UMYSŁ

Po co w ogóle istnieje metoda `init()`? Inaczej mówiąc, dlaczego sam *konstruktor* nie wystarczy do zainicjalizowania serwletu?

Jakiego rodzaju kod należy umieścić w ciele metody `init()`?

Wskazówka: Metoda `init()` pobiera w formie argumentu referencję do obiektu. Jak myślisz, jaka jest rola tego argumentu metody `init()` i jak (lub do czego) mógłbyś tego argumentu użyć?

Inicjalizacja serwletu. Kiedy obiekt staje się serwletem

nie istnieje

konstruktor, init()

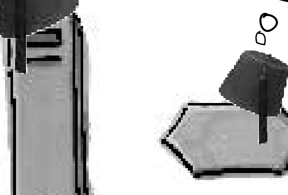
destroy()

zainicjalizowany

service()

Musisz pamiętać, że metoda init() jest wykonywana tylko raz w całym życiu serwletu! Nie próbuj wykonywać pewnych działań zbyt wcześnie... konstruktor nie jest właściwym miejscem dla operacji charakterystycznych dla serwletu.

Najbardziej doniosłe wydarzenie w moim życiu miało miejsce w momencie, w którym Wielki Mistrz Kontener uczynił mnie serwletem przez stworzenie dla mnie obiektu `ServletConfig` i wywołanie metody `init()`. Zanim to nastąpiło, byłem zwykłym obiektem. Jednak dopiero teraz, już jako serwlet, korzystam ze specjalnych przywilejów, włącznie z możliwością rejestrowania zdarzeń, otrzymywania referencji do innych zasobów oraz tworzenia i zapisywania atrybutów dla innych serwletów...



Początek procesu przejścia serwletu ze stanu *nie istnieje* w stan *zainicjalizowany* (który tak naprawdę oznacza *gotowość do obsługi żądań klientów*) wiąże się z wykonaniem konstruktora. Warto jednak pamiętać, że sam konstruktor tworzy jedynie *obiekt*, nie *serwlet*. Aby zostać pełnoprawnym serwletem, obiekt taki musi jeszcze otrzymać podstawowe atrybuty *serwletowości*.

Kiedy obiekt staje się serwletem, otrzymuje jednocześnie wszystkie unikatowe uprawnienia, które wiążą się z funkcjonowaniem wszystkich serwletów, w tym możliwość wykorzystywania referencji do kontekstu `ServletContext`, za pośrednictwem którego serwlet może uzyskiwać potrzebne informacje z kontenera.

Dlaczego w ogóle zajmujemy się szczegółami procesu inicjalizacji?

Ponieważ gdzieś pomiędzy konstruktorem a metodą `init()` serwlet znajduje się w stanie *serwletu Schroedingera*[2]. Wykonywanie kodu inicjalizującego Twój serwlet (np. uzyskującego informacje o konfiguracji aplikacji internetowej lub wyszukującego referencje do innych części aplikacji) może zakończyć się niepowodzeniem, jeśli spróbujesz to zrobić zbyt *wcześnie* w życiu serwletu. Reguła decydująca o właściwym miejscu dla kodu inicjalizującego jest jednak dosyć prosta — wystarczy pamiętać, aby niczego nie umieszczać w konstruktorze serwletu!

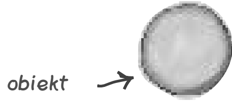
Serwlet nie musi zawierać żadnego kodu, którego wykonanie nie może poczekać na wywołanie metody `init()`.

---

[2] Jeśli Twoja wiedza na temat mechaniki kwantowej była ostatnimi czasy zaniedbywana, możesz w wyszukiwarce *Google* wpisać wyrażenie *Kot Schroedingera* (uwaga: miłośnicy zwierząt domowych powinni unikać tego zagadnienia). Mówiąc o *stanie Schroedingera*, mamy na myśli coś, co nie jest ani w pełni martwe, ani w pełni żywe — znajduje się w zagadkowym punkcie pomiędzy życiem a śmiercią.

Co tak naprawdę oznacza „bycie serwletem”?

Co się stanie, kiedy serwlet przejdzie stąd…



tutaj?

Oglądnij to!

Nie należy mylić parametrów obiektu `ServletConfig` z parametrami obiektu `ServletContext`!

Nie mieliśmy zamiaru omawiać tych zagadnień w tym miejscu, lecz w ***kolejnym*** rozdziale, jednak tak wiele osób myli ze sobą parametry obu obiektów, że warto już teraz położyć nacisk na ***konieczność rozróżniania tych parametrów***.

Zacznijmy od analizy samych nazw:

W nazwie obiektu `ServletConfig` występuje słowo „config”, które reprezentuje „konfigurację” (ang. *configuration*). Tak rozumiana konfiguracja jest po prostu zbiorem wartości (ustawień) definiowanych dla serwletu w czasie wdrażania aplikacji (dla każdego serwletu istnieje osobny zbiór ustawień konfiguracyjnych). Konfiguracja obejmuje ustawienia, które z jednej strony mają być dostępne dla serwletu, ale z drugiej strony nie powinny być trwale kodowane w tym serwlecie — dotyczy to np. nazwy bazy danych.

Parametry obiektu `ServletConfig` nie są zmieniane w całym okresie wykorzystywania serwletu po jego wdrożeniu. Aby je zmienić, będziesz musiał ponownie wdrożyć dany serwlet w swoim środowisku.

Obiekt `ServletContext` w rzeczywistości powinien nosić nazwę `AppContext` (niestety, nikt nie chce uwzględnić naszej opinii na ten temat), ponieważ dla każdej aplikacji internetowej istnieje tylko jeden taki obiekt (nie jak w przypadku obiektu `ServletConfig`, gdzie mamy do czynienia z jednym obiektem dla każdego serwletu).

Tak czy inaczej wszystkie te zagadnienia szczegółowo omówimy w następnym rozdziale — ten tekst należy traktować wyłącznie jak jego zapowiedź.

1 Obiekt `ServletConfig`

- Dla każdego serwletu istnieje jeden obiekt `ServletConfig`.
- Obiektu `ServletConfig` należy używać do przekazywania do serwletu informacji znanych już w czasie wdrażania aplikacji (np. nazw identyfikujących bazy danych lub komponenty EJB), których nie chcemy trwale kodować w samym serwlecie (np. w postaci jego parametrów inicjalizacji).
- Obiektu `ServletConfig` należy używać także do uzyskiwania dostępu do kontekstu serwletu (obiektu `ServletContext`).
- Parametry tego obiektu są konfigurowane w deskryptorze rozmieszczenia (DD).

2 Obiekt `ServletContext`

- Dla każdej aplikacji internetowej istnieje dokładnie jeden obiekt `ServletContext` (należałoby więc nazywać te obiekty kontekstami aplikacji lub np. `AppContext`).
- Obiektu `ServletContext` należy używać do uzyskiwania dostępu do *parametrów* aplikacji internetowej (także skonfigurowanych w deskryptorze rozmieszczenia).
- Obiekt `ServletContext` można również wykorzystywać w roli komunikatora, za pomocą którego możesz tworzyć komunikaty (nazywane atrybutami) udostępniane pozostałym częściom aplikacji internetowej (więcej informacji na temat tego typu rozwiązań znajdziesz w kolejnym rozdziale).
- Obiektu `ServletContext` należy używać także do uzyskiwania informacji o serwerze, włącznie z nazwą i wersją wykorzystywanego kontenera oraz wersją obsługiwanego interfejsu API.

RZECZYWISTYM zadaniem serwletu jest jednak obsługa żądań. Dopiero wtedy życie serwletu ma jakiś sens

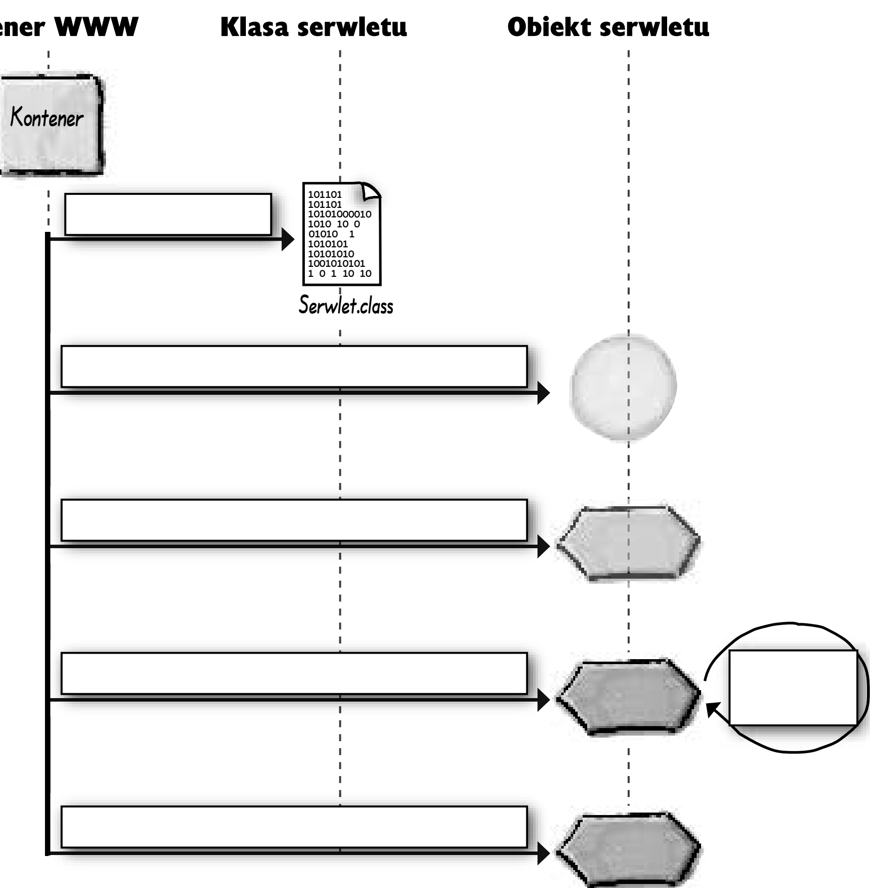
W kolejnym rozdziale zajmiemy się obiektami `ServletConfig` i `ServletContext`, na razie musimy jednak szczegółowo przeanalizować funkcjonowanie mechanizmu obsługującego żądania i odpowiedzi. Warto pamiętać, że obiekty `ServletConfig` i `ServletContext` istnieją wyłącznie po to, by umożliwić naszemu serwletowi realizację jego najważniejszego zadania — obsługę żądań klientów! Zanim przystąpimy do omawiania sposobu, w jaki nasze obiekty kontekstu i konfiguracji mogą pomóc w wykonywaniu tego zadania, musimy wrócić do podstaw przetwarzania żądań i odpowiedzi.

Wiemy już, że obiekty reprezentujące żądanie i odpowiedź są przekazywane do naszego serwletu w postaci argumentów metody `doGet()` lub `doPost()`, warto się jednak zastanowić, co *szczególnego* wynika z dostępu do tego typu obiektów. Co możemy zrobić z tymi obiektami, i dlaczego przywiązujemy do nich tak dużą wagę ?

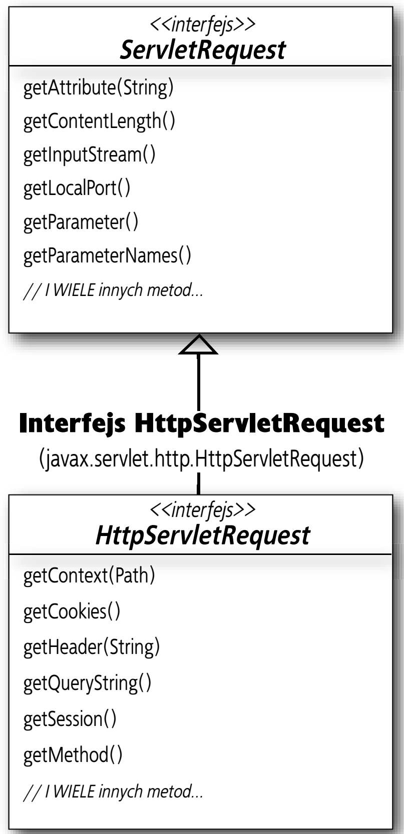
Przygotuj ołówek

Oznacz etykietami brakujące elementy (puste pola) przedstawionego wykresu cyklu życia serwletu (porównaj swoje odpowiedzi z wykresem przedstawionym we wcześniejszej części tego rozdziału).

Dodaj własne komentarze, które pozwolą Ci lepiej zapamiętać szczegóły tego schematu.

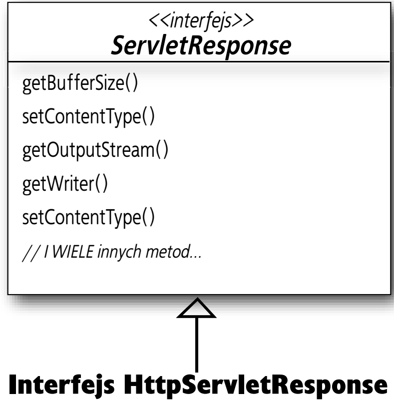


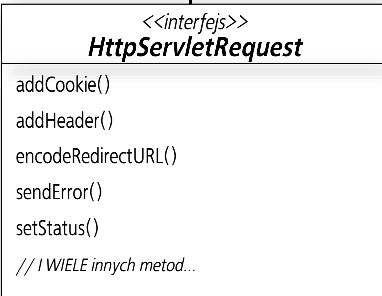
Żądanie i odpowiedź — kluczowe obiekty będące także argumentami metody service()[3]



Metody interfejsu `HttpServletRequest` obsługują takie elementy żądań protokołu HTTP jak ciasteczka, nagłówki i sesje.

Interfejs `HttpServletRequest` dodaje metody związane z protokołem HTTP, które Twój serwlet wykorzystuje do zapewniania komunikacji pomiędzy klientem a serwerem.





Ta para interfejsów z pewnością ma związek z odpowiedzią... interfejs `HttpServletResponse` dodaje metody niezbędne podczas stosowania takich elementów protokołu HTTP jak błędy, ciasteczka czy nagłówki.

---

[3] Obiekty reprezentujące żądanie i odpowiedź są także argumentami *innych* przykrywanych przez nas metod klasy bazowej `HttpServlet` – `doGet()`, `doPost()` itp.

Nie ma niemądrych pytań

P: Kto implementuje interfejsy `HttpServletRequest` i `HttpServletResponse`? Czy odpowiednie klasy należą do interfejsu programowego API?

O: Pierwsza odpowiedź brzmi „kontener"; druga — „nie". Odpowiednie klasy nie należą do interfejsu programowego API, ponieważ ich implementację pozostawiono producentom oprogramowania kontenerów. Niewątpliwie dobrą wiadomością dla Ciebie jest to, że sam nie musisz się martwić o implementację tych klas. Możesz przyjąć, że w momencie wywołania metody `service()` Twojego serwletu otrzymasz referencje do niezawodnych obiektów, które *implementują* interfejsy `HttpServletRequest` i `HttpServletResponse`. W żadnym przypadku nie musisz się zajmować faktyczną nazwą lub typem klasy — Twoim zadaniem jest wyłącznie właściwe wykorzystanie funkcjonalności oferowanej przez obiekty implementujące interfejsy `HttpServletRequest` i `HttpServletResponse`.
Innymi słowy, do programisty serwletu należy jedynie znajomość *metod, które może wywoływać* za pośrednictwem obiektów przekazanych przez kontener w postaci części żądania! Tworzenie rzeczywistej klasy implementującej te metody nie należy do Twoich zadań — odwołujesz się do obiektów reprezentujących żądanie i odpowiedź *wyłącznie za pośrednictwem odpowiedniego typu interfejsu*.

P: Czy właściwie odczytałem przedstawione diagramy UML? Czy wspomniane interfejsy rozszerzają interfejsy bazowe?

O: Tak. Pamiętaj, że interfejsy mogą tworzyć własne drzewa dziedziczenia. Jeśli jeden interfejs rozszerza inny interfejs (to wszystko, co *może* robić interfejs — interfejsy z natury rzeczy nie mogą *implementować* innych interfejsów), każda klasa implementująca jakiś interfejs musi implementować *wszystkie* metody zdefiniowane zarówno w danym interfejsie, jak i we *wszystkich* jego nadinterfejsach. Oznacza to, że każda klasa implementująca np. interfejs `HttpServletRequest` musi zawierać implementacje wszelkich metod, które zadeklarowano zarówno w tym interfejsie, jak i w rozszerzanym przez niego nadinterfejsie `ServletRequest`.

P: Nadal nie jest dla mnie jasne, dlaczego istnieje zarówno `GenericServlet`, interfejs `ServletRequest`, jak i interfejs `ServletResponse`? Skoro poza serwletami HTTP nikt nic nie robi, po co aż tyle tych interfejsów?

O: Nie powiedzieliśmy, że *nikt* nic nie robi. Można sobie wyobrazić, że ktoś gdzieś wykorzystuje model technologii serwletów bez protokołu HTTP. Chodzi nam wyłącznie o to, że nigdy nikogo takiego nie spotkaliśmy ani nigdy o nikim takim nie czytaliśmy. To wszystko.
Model oparty na serwletach został zaprojektowany w sposób na tyle elastyczny, że istnieje możliwość komunikowania się z serwletami np. za pośrednictwem protokołu SMTP lub nawet protokołu opracowanego specjalnie z myślą o danej aplikacji internetowej. Warto jednak pamiętać, że omawiany interfejs programowy API ma wbudowaną wyłącznie obsługę protokołu HTTP i że właśnie z tego protokołu korzystają niemal wszyscy programiści serwletów.

Relax

Na egzaminie nikt nie będzie od Ciebie wymagał opracowywania serwletów innych niż serwlety HTTP.

Nikt nie oczekuje od Ciebie pełnej wiedzy na temat technik tworzenia i wykorzystywania serwletów z protokołem innym niż HTTP. Nadal jednak powinieneś mieć pewne rozeznanie w sposobie funkcjonowania istniejącej hierarchii klas. Przykładowo, MUSISZ wiedzieć, że interfejsy `HttpServletRequest` i `HttpServletResponse` rozszerzają odpowiednio interfejsy `ServletRequest` i `ServletResponse`, oraz że większa część implementacji interfejsu `HttpServlet` faktycznie pochodzi z abstrakcyjnej klasy `GenericServlet`. To wszystko — twórcy egzaminu zakładają, że jesteś programistą serwletów dziedziczących po klasie `HttpServlet`.

Metoda żądania protokołu HTTP określa, czy zostanie wywołana metoda doGet() czy metoda doPost()

Jak zapewne pamiętasz, żądanie klienta zawsze zawiera konkretną metodę protokołu HTTP. Jeśli tą metodą jest `GET`, metoda `service()` wywoła metodę `doGet()`. Jeśli natomiast wskazaną metodą protokołu HTTP jest `POST`, metoda `service()` wywoła metodę `doPost()`.

```
GET /wybor/wybierzSmakPiwa.do?kolor
=ciemny&smak=slodkawy HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh;
U; PPC Mac OS X Mach-O; en-US;
rv:1.4)Gecko/200
Accept: text/xm
application/xhtm
html;q=0.9,tex
x-msg,image/
gif;q=0.2,*/*
```

```
POST /wybor/wybierzSmakPiwa2.do
HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh;
U; PPC Mac OS X Mach-O; en-US;
rv:1.4)Gecko/20040624 Netscape/7.1
Accept: text/xml.application/xml.
application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,video/
x-msg,image/png,image/jpeg,image/
gif;q=0.2,*/*;q=0.1
```

HTTP requests

Cały czas mówimy tylko o metodach doGet() i doPost() tak jakby nie istniały żadne inne... WIEM przecież, że w sumie istnieje aż osiem metod standardu HTTP 1.1.

Prawdopodobnie nie będą Cię interesowały żadne metody protokołu HTTP poza najpopularniejszymi GET i POST

To prawda, poza `GET` i `POST` istnieją także inne metody protokołu HTTP 1.1. Protokół ten obsługuje metody `HEAD`, `TRACE`, `OPTIONS`, `PUT`, `DELETE` oraz `CONNECT`.

Okazuje się, że aż siedem spośród tych metod protokołu HTTP ma odpowiednie metody `doXXX()` w klasie `HttpServlet`, zatem poza wielokrotnie wspominanymi metodami `doGet()` i `doPost()` mamy do dyspozycji metody `doOptions()`, `doHead()`, `doTrace()`, `doPut()` oraz `doDelete()`. Interfejs programowy API dla serwletów nie przewiduje mechanizmów obsługujących metodę `doConnect()`, zatem metoda ta nie jest częścią klasy `HttpServlet`.

O ile jednak wymienione metody protokołu HTTP mogą być interesujące np. dla programistów *serwerów* WWW, o tyle programiści *serwletów* bardzo rzadko korzystają z czegokolwiek innego niż popularne metody `GET` i `POST`.

W większości (być może nawet we *wszystkich*) procesów wytwarzania serwletów będziesz używał albo metody `doGet()` (dla prostych żądań), albo metody `doPost()` (do przyjmowania i przetwarzania danych z formularza), zatem nie będziesz nawet myślał o pozostałych dostępnych metodach.

Jeśli pozostałe metody protokołu HTTP nie są dla mnie ważne... Z PEWNOŚCIĄ część pytań egzaminacyjnych będzie poświęcona właśnie tym metodom.

W rzeczywistości jeden lub więcej spośród pozostałych metod protokołu HTTP może przelotnie pojawić się na egzaminie...

Jeśli przygotowujesz się do egzaminu, powinieneś potrafić prawidłowo rozpoznać wszystkie wymienione metody i choćby w skrócie wyjaśnić, do czego służą, i co je różni. Nie trać jednak zbyt wiele czasu na omawiane tutaj zagadnienia — ich występowanie w pytaniach egzaminacyjnych będzie marginalne.

W świecie rzeczywistych serwletów interesują nas wyłącznie metody `GET` i `POST`

W świecie egzaminów powinniśmy nieco uwagi poświęcić także pozostałym metodom protokołu HTTP.

GET Żąda *zwrócenia* zasobu (pliku) znajdującego się pod podanym adresem URL.

POST Żąda od serwera *przyjęcia* informacji dołączonych do żądania i przekazania ich pod wskazany adres URL. Metodę `POST` można traktować jak rozszerzoną metodę `GET`... jak metodę `GET` z dodatkowymi informacjami wysyłanymi wraz z żądaniem.

HEAD Żąda zwrócenia wyłącznie części *nagłówkowej* tego, co zostałoby zwrócone w odpowiedzi na analogiczne żądanie `GET`. Metoda `HEAD` jest więc żądaniem `GET` pomijającym właściwe informacje w zwracanym komunikacie. Pozwala uzyskać informacje na temat wskazanego adresu URL bez faktycznego zwrócenia *zawartości* generowanej odpowiedzi.

TRACE Żąda odesłania komunikatu żądania, dzięki czemu klient może się łatwo przekonać, co dotarło do drugiego końca komunikacji — tego typu funkcjonalność wykorzystuje się najczęściej podczas testowania aplikacji i usuwania błędów

PUT Żąda *umieszczenia* osadzonych informacji (ciała) pod wskazanym adresem URL.

DELETE Żąda *usunięcia* zasobu (pliku) znajdującego się pod wskazanym adresem URL.

OPTIONS Żąda *listy* metod protokołu HTTP, na które zasób znajdujący się pod wskazanym adresem URL może odpowiedzieć.

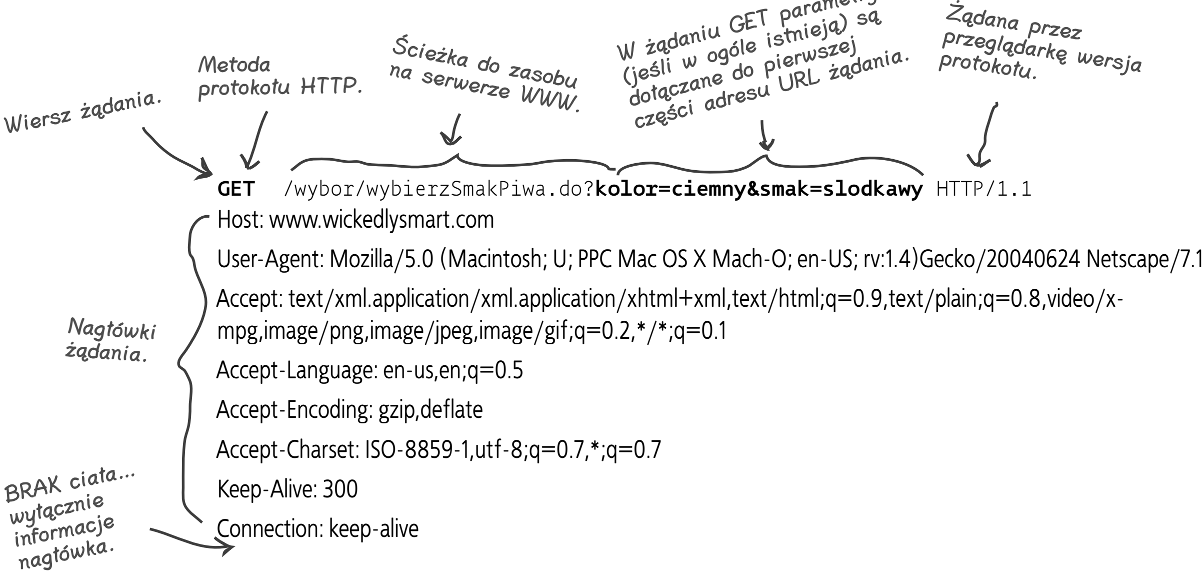
CONNECT Żąda nawiązania *połączenia* celem tunelowania protokołów.

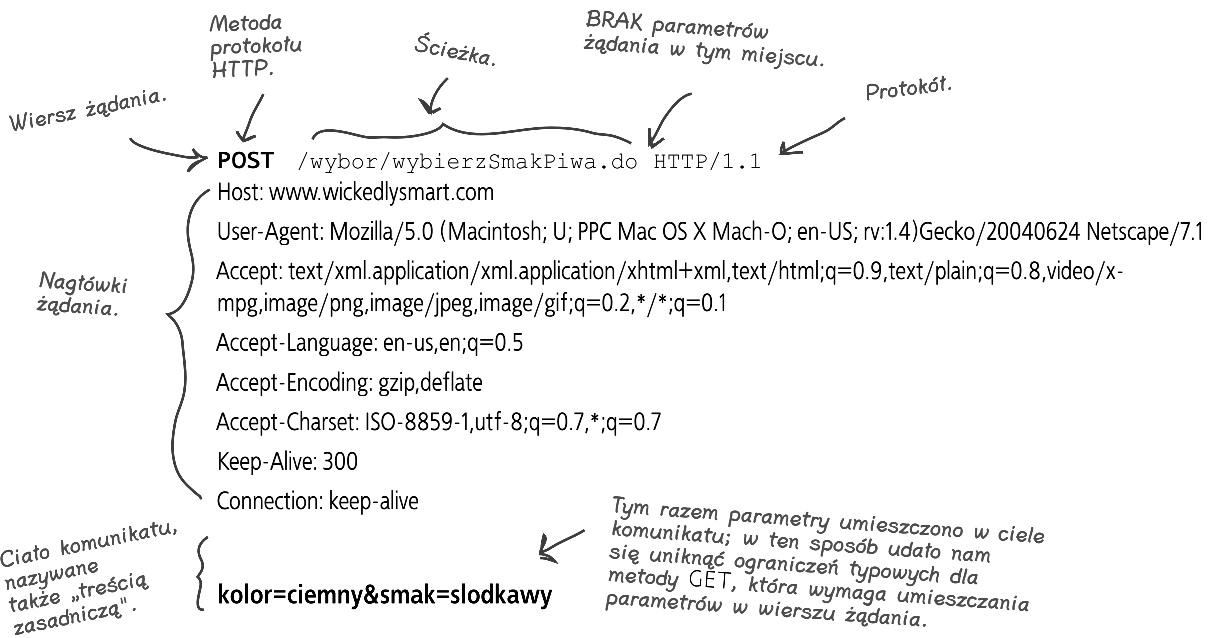
Przykład odpowiedzi dla żądania `OPTIONS` protokołu HTTP.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Date: Thu, 20 Apr 2004
16:20:00 GMT
Allow: OPTIONS, TRACE,
GET, HEAD, POST
Content-Length: 0
```

Różnica pomiędzy metodą GET a metodą POST

Żądanie POST zawiera ciało. To jest kluczowa informacja. Zarówno metoda `GET`, jak i metoda `POST` przewiduje możliwość przesyłania parametrów, ale w przypadku metody `GET` długość danych zawartych w parametrach ogranicza się do zawartości wiersza żądania.



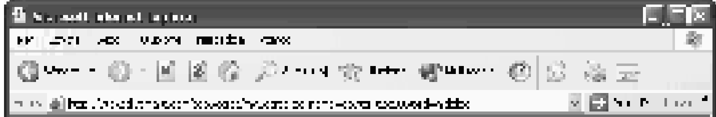


Wygląda na to, że jedyna różnica pomiędzy metodami GET i POST sprowadza się do możliwego rozmiaru przesyłanych danych parametrów.

Nie, nie chodzi wyłącznie o rozmiar

Kiedy używasz metody GET, dane parametrów są wyświetlane na pasku adresu okna przeglądarki internetowej, bezpośrednio za właściwym adresem URL (oba składniki tego adresu są oddzielone jedynie znakiem zapytania). Nietrudno wyobrazić sobie scenariusz, w którym programista nie chce, by parametry były znane użytkownikowi.

Pamiętasz, jak w pierwszym rozdziale wspominaliśmy o pozostałych cechach metody GET?



Można więc przyjąć, że kwestie bezpieczeństwa stanowią kolejną różnicę pomiędzy obiema metodami.

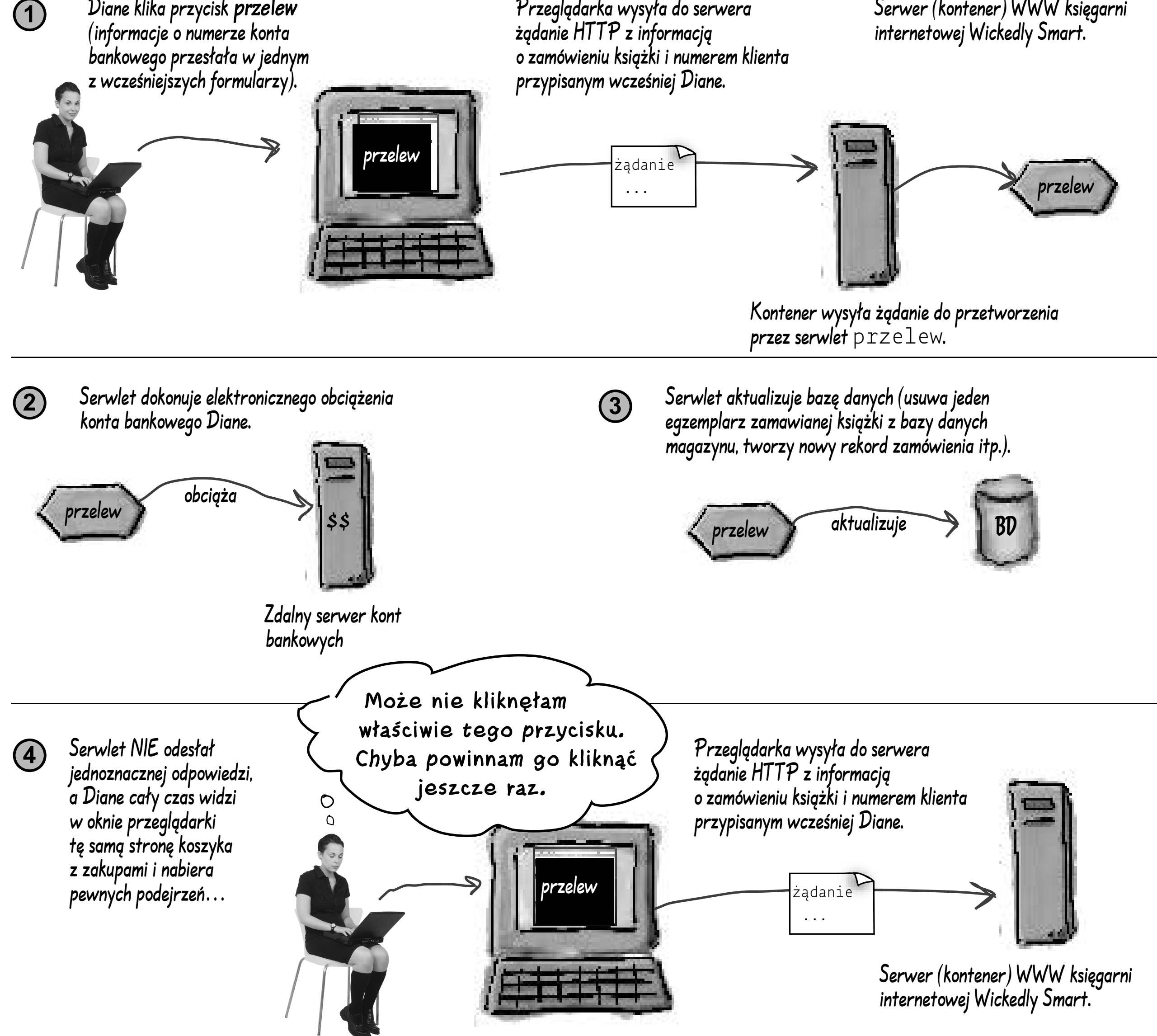
Jeszcze innym elementem stanowiącym o odmienności metod GET i POST jest możliwość dodawania żądanych stron do listy ulubionych adresów użytkownika końcowego. Żądania GET można dodawać do takiej listy, co nie jest możliwe w przypadku żądań POST. Może to być szczególnie istotne w przypadku strony, na której użytkownicy mogą np. określać kryteria przeszukiwania dostępnych zasobów. Użytkownicy mogą w takim przypadku wrócić na stronę po tygodniu i spróbować wykonać tę samą operację przeszukiwania ponownie, aby sprawdzić, czy na serwerze nie pojawiły się nowe dane.

Jednak *poza* rozmiarem, bezpieczeństwem i listą ulubionych adresów istnieje jeszcze inna kluczowa właściwość odróżniająca metodę GET od metody POST — jest to sposób, w jaki obie metody mają *w założeniu* być wykorzystywane. Metoda GET została zaprojektowana z myślą o *uzyskiwaniu* dostępu do żądanych zasobów. Tylko tyle. Jej celem jest proste dostawanie się do zasobów. Możesz oczywiście wykorzystywać dodatkowe parametry, które pomagają w określaniu tego, co serwer powinien Ci odesłać, jednak z reguły żądania GET nie powinny być źródłem zmian po stronie serwera! Metoda POST ma w założeniu służyć do *przesyłania danych przeznaczonych do przetworzenia*. Dane dołączone do żądania POST mogą co prawda mieć postać prostych parametrów zapytania używanych do precyzyjnego określania zasobów interesujących użytkownika (a więc podobnie jak w przypadku metody GET), jednak prawdziwym powodem zaprojektowania tej metody jest *aktualizowanie* danych po stronie serwera. Należy przyjąć, że dane z ciała żądania POST mają na celu *wprowadzenie jakichś zmian na serwerze*.

W naturalny sposób doszliśmy do kolejnego zagadnienia… czy dane żądanie może być *powtarzalne (idempotentne)*. Jeśli nie, możemy popaść w tarapaty, których nie rozwiąże mała, niebieska pigułka. Jeśli po raz pierwszy zetknąłeś się z pojęciem *powtarzalności* w świecie aplikacji internetowych, czytaj dalej…

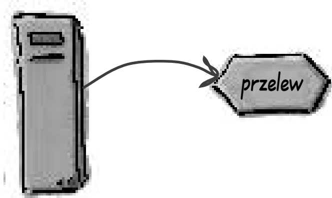
Historia pewnego nie powtarzalnego żądania

Diane ma problem. Próbuje znaleźć w księgarni internetowej książkę z serii *Head First* poświęconą dziewiarstwu; nie wie jednak, że prace nad witryną internetową *Wickedly Smart* wciąż nie zostały ukończone. Diane nie ma zbyt dużo pieniędzy — uzbierała na swoim koncie dokładnie tyle, ile będzie potrzebne na *jedną* książkę. Diane rozważała co prawda zakup książki bezpośrednio w księgarni internetowej *Amazon* lub na witrynie wydawnictwa, jednak ostatecznie zdecydowała się na zakup egzemplarza z autografem autora, który jest dostępny wyłącznie na witrynie *Wickedly Smart*. Tego wyboru już wkrótce będzie żałowała…



Dalszy ciąg naszej historii...

5.
Kontener przekazuje żądanie do przetworzenia przez serwlet `przelew`.

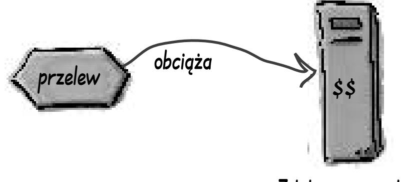


Serwer (kontener) WWW księgarni internetowej Wickedly Smart.

6. Serwlet nie widzi problemu w tym, że Diane kupuje tę samą książkę, którą kupiła przed chwilą.

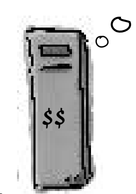
Książka musiała jej się naprawdę spodobać... kupiła aż dwa egzemplarze. Świetnie.

7. Serwlet dokonuje elektronicznego obciążenia konta bankowego Diane.



Zdalny serwer kont bankowych

8. Bank obsługujący konto Diane przyjmuje polecenie przelewu i obciąża ją surową opłatą za spowodowanie salda debetowego.



Zdalny serwer kont bankowych

Pozwolimy jej na zakup tej książki, ale obciążymy ją karą umowną za spowodowanie salda debetowego. Niedobra, oj niedobra Diane!

9. Diane ostatecznie przechodzi do strony stanu zamówienia i przekonuje się, że wysłała aż DWA zamówienia dotyczące tej samej książki…

Coś tu nie gra. Chciałam przecież kupić tylko JEDNĄ książkę. Co ten głupi programista aplikacji internetowej najlepszego zrobił? Powinien przecież wykrywać powtarzające się transakcje...

10. Halo, bank? Ten głupi programista księgarni internetowej wszystko pomylił...

Która z metod protokołu HTTP jest (powinna być) Twoim zdaniem powtarzalna (idempotentna)? Swoją odpowiedź możesz oprzeć na samym znaczeniu słowa powtarzalność i (lub) analizie przedstawionego przed chwilą przykładu podwójnego zakupu dokonanego przez Diane. Prawidłowe odpowiedzi przedstawiono na dole tej strony.

- ❑ `GET`
- ❑ `POST`
- ❑ `PUT`
- ❑ `HEAD`

(Celowo pomięliśmy metodę `CONNECT`, ponieważ nie jest ona obsługiwana w klasie `HttpServlet`.)

WYTĘŻ UMYSŁ

Co było źródłem problemów związanych z transakcją Diane?

(Zaistniałe nieporozumienie nie było efektem tylko JEDNEJ usterki — prawdopodobnie istnieje kilka problemów, które muszą zostać wyeliminowane przez programistę.)

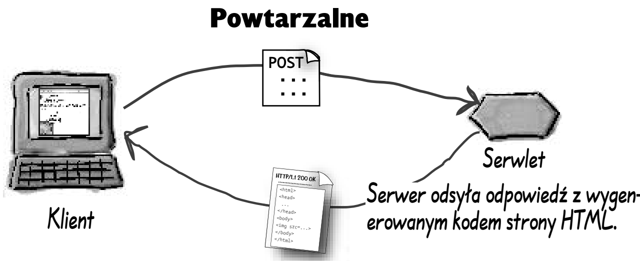
Jakie są możliwości ograniczania przez programistę ryzyka związanego z podobnymi niedociągnięciami w kodzie aplikacji?

(Wskazówka: nie wszystkie rozwiązania w tym zakresie muszą mieć charakter działań *programistycznych*.)

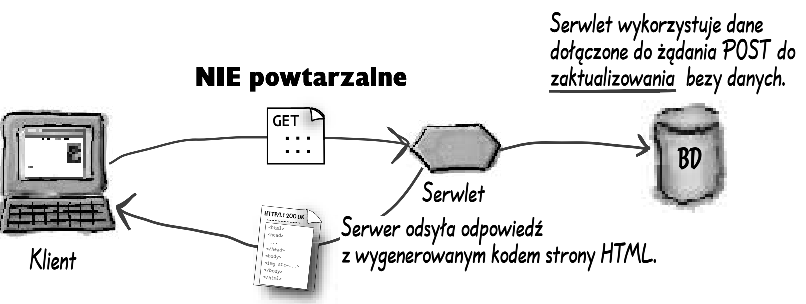
Specyfikacja standardu HTTP 1.1 deklaruje metody `GET`, `HEAD` i `PUT` jako powtarzalne, chociaż teoretycznie ISTNIEJE możliwość samodzielnego napisania niepowtarzalnej metody `doGet()` (nie należy jednak z tej możliwości korzystać). Specyfikacja HTTP 1.1 nie przewiduje powtarzalności metody `POST`.

Bycie powtarzalnym ma swoje PLUSY. Oznacza bowiem, że możesz wykonywać te same czynności wielokrotnie bez niepożądanych efektów ubocznych!

Powtarzalne



NIE powtarzalne



Żądanie POST <u>nie</u> jest idempotentne

Żądanie GET protokołu HTTP ma w założeniu służyć do *uzyskiwania* dostępu do zasobów, a nie do *zmieniania* czegokolwiek na serwerze. Żądanie GET jest więc (zgodnie z definicją i specyfikacją standardu HTTP) powtarzalne (idempotentne). Można identyczne żądanie tego typu wykonać więcej niż raz bez obaw o jakiekolwiek niekorzystne efekty uboczne.

Żądanie POST *nie* jest powtarzalne — dane osadzone w ciele żądania POST mogą być kierowane do nieodwracalnej transakcji. Oznacza to, że musisz bardzo uważać podczas opracowywania funkcjonalności swojej metody `doPost()`.

Żądanie GET jest, żądanie POST nie jest idempotentne. To, czy logika Twojej aplikacji internetowej w każdej sytuacji będzie przygotowana do właściwego obsłużenia scenariusza podobnego do zakupów Diane (gdzie do serwera dociera więcej niż jedno identyczne żądanie POST), zależy tylko od Ciebie.

Co może mnie powstrzymać przed stosowaniem parametrów metody GET do aktualizowania danych składowanych na serwerze?

W standardzie HTTP 1.1 metoda GET <u>zawsze</u> jest uważana za powtarzalną...

...nawet jeśli na egzaminie występują żądania GET z parametrami wykorzystywanymi w procesie aktualizowania serwera! Innymi słowy, metoda GET jest idempotentna zgodnie ze specyfikacją protokołu HTTP. Nie ma jednak żadnych ograniczeń, które mogłyby Cię odwieść od implementowania <u>nie</u> idempotentnych metod `doGet()` w Twoich serwletach. Generowane przez klienta żądanie GET zawsze jest traktowane jak działanie powtarzalne, niezależnie od tego, czy zaimplementowany przez CIEBIE mechanizm przetwarzania tych danych aktualizuje serwer. Zawsze musisz mieć na uwadze istotną różnicę pomiędzy metodą GET protokołu HTTP a metodą `doGet()` Twojego serwletu.

*Uwaga: istnieje wiele różnych zastosowań słowa „idempotentny"; w tej książce używamy tego słowa wyłącznie w odniesieniu do protokołu HTTP i technologii serwletów, a więc w takim znaczeniu, że to samo żądanie może być wykonywane dwukrotnie bez negatywnych konsekwencji dla serwera. *Nie* twierdzimy natomiast, że „idempotencja" oznacza zwracanie takich samych odpowiedzi na następujące po sobie identyczne żądania; NIE mówimy także, że przetwarzanie tych żądań nie ma żadnego wpływu na serwer.*

Co sprawia, że przeglądarka wysyła albo żądanie GET, albo żądanie POST?

GET

Proste hiperłącze zawsze reprezentuje żądanie GET.

```
<A HREF="http://www.wickedlysmart.com/index.html/">Kliknij tutaj</A>
```

POST

Jeśli OKREŚLISZ wprost, że method="POST", wtedy o dziwo zostanie użyta metoda POST.

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

Kiedy użytkownik kliknie przycisk Prześlij kwerendę, parametry zostaną przesłane w ciele żądania POST. W tym przypadku będzie istniał tylko jeden taki parametr (nazwany "kolor"), którego wartością będzie wybrany przez użytkownika kolor piwa (jasny, bursztynowy, brązowy lub ciemny).

Co stanie się, jeśli w znaczniku <form> NIE określimy, że interesuje nas metoda POST (method="POST")?

Tym razem nie występuje turaj method="POST".

```
<form   action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option>jasny
    <option>bursztynowy
    <option>brązowy
    <option>ciemny
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

Co TERAZ stanie się z parametrami, jeśli użytkownik kliknie przycisk Prześlij kwerendę (przecież formularz nie zawiera już stwierdzenia method="POST")?

POST NIE jest metodą domyślną!

Jeśli w znaczniku <form> nie umieścisz atrybutu **method="POST"**, zostanie zastosowane domyślne żądanie GET protokołu HTTP. Oznacza to, że przeglądarka wyśle zdefiniowane w formularzu parametry w nagłówku żądania, ale nie to jest naszym zasadniczym problemem. Jeśli bowiem do serwera dotrze żądanie GET, a w swoim serwlecie nie zdefiniujesz metody doGet(), wpadniesz w poważne tarapaty dopiero podczas wykonywania tak skonstruowanej aplikacji internetowej.

Jeśli zrobisz to:

Brak atrybutu "method=POST" w formularzu HTML.

```
<form   action="WybierzPiwo.do">
```

Po czym zrobisz to:

```
public class WyborPiwa extends HttpServlet {

  public void doPost(HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
     // tutaj powinien się znajdować Twój kod
  }
}
```

Brak metody doGet() w serwlecie.

Otrzymasz to:

BŁĄD! Jeśli Twój formularz HTML stosuje metodę GET zamiast metody POST, koniecznie MUSISZ zdefiniować w swojej klasie serwletu metodę doGet(). Domyślną metodą formularzy HTML jest GET.

P: Co powinienem zrobić, jeśli chcę w pojedynczym serwlecie obsługiwać zarówno żądania GET, jak i żądania POST?

O: Programiści, którzy chcą obsługiwać obie metody protokołu HTTP umieszczają zwykle odpowiednią logikę w ciele metody doPost(), po czym — jeśli żądanie nie wymaga przetwarzania parametrów — przekazują wywołanie do metody doGet():

```
public void doPost(...)
                     throws ... {
  doGet(request, response);
}
```

Przesyłanie i wykorzystywanie pojedynczego parametru

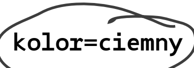
Formularz HTML

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

Przeglądarka wyśle w ciele żądania jedną z czterech dostępnych opcji parametru "kolor". Może to być np. wartość "kolor=bursztynowy".

Żądanie POST protokołu

```
POST /wybor/wybierzPiwo.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1
Accept: text/xml.application/xml.application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-msg,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```



Pamiętaj, że to przeglądarka generuje żądanie, zatem nie musisz się martwić o jego tworzenie; w tym miejscu przedstawiliśmy możliwy kształt takiego żądania docierającego do serwera WWW...

Klasa serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
              throws IOException, ServletException {
  String kolorParam = request.getParameter("kolor");
 // tutaj jest Twój niesamowity kod...
}
```

(W prezentowanym przykładzie łańcuch kolorParam ma wartość "ciemny".)

Ten łańcuch odpowiada nazwie atrybutu formularza HTML.

Przesyłanie i wykorzystywanie DWÓCH parametrów

Formularz HTML

```
<form method="POST" action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Kolor:
  <select name="kolor" size="1">
    <option>jasny
    <option>bursztynowy
    <option>brązowy
    <option>ciemny
  </select>
  Moc:
  <select name="body" size="1">
    <option>lekkie
    <option>ciężkie
    <option>mocne
  </select>
  <center>
    <input type="SUBMIT">
  </center>
</form>
```

Przeglądarka wyśle w ciele żądania jedną z czterech dostępnych opcji związanych z parametrem nazwanym "kolor".

Przeglądarka wyśle w ciele żądania jedną z czterech dostępnych opcji związanych z parametrem nazwanym „body"

Żądanie POST protokołu HTTP

```
POST /wybor/wybierzPiwo.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1
Accept: text/xml.application/xml.application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-msg,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

kolor=ciemny&body=mocne
```

Żądanie POST zawiera teraz oba parametry (oddzielone znakiem &).

Klasa serwletu

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
              throws IOException, ServletException {
  String kolorParam = request.getParameter("kolor");
  String bodyParam = request.getParameter("body");
     // jeszcze więcej kodu w tym miejscu...
}
```

Zmienna łańcuchowa kolorParam zawiera teraz wartość "ciemny", natomiast zmienna bodyParam reprezentuje wartość "mocne".

Dla pojedynczego parametru może istnieć wiele wartości! Oznacza to, że będziemy potrzebowali metody `getParameterValues()`, która zwróci tablicę tych wartości (zamiast metody `getParameter()` zwracającej pojedynczy łańcuch).

Niektóre typy komponentów formularza HTML (np. zbiór pól wyboru) mogą mieć więcej niż jedną wartość. Oznacza to, że pojedynczy parametr (w poniższym przykładzie nazwany `"sizes"`) będzie miał wiele wartości w zależności od wyboru dokonanego przez użytkownika (liczby zaznaczonych pól wyboru). Formularz, w którym użytkownik może wybrać więcej niż jedną pojemność butelki lub puszki piwa (np. aby określić, że jest zainteresowany WSZYSTKIMI rozmiarami), może mieć następującą postać:

```
<form method="POST" action="WybierzPiwo.do">
  Wybierz właściwości piwa<p>
  Rozmiary puszek lub butelek: <p>
  <input type=checkbox name=sizes value="330ml">330 ml<br>
  <input type=checkbox name=sizes value="500ml">500 ml<br>
  <input type=checkbox name=sizes value="700ml">700 ml<br>
  <br><br>

  <center>
    <input type="SUBMIT">
  </center>
</form>
```

W kodzie naszego serwletu użyjemy metody `getParameterValues()`, która zwróci tabelę wartości:

```
String one = request.getParameterValues("sizes")[0];

String [] sizes = request.getParameterValues("sizes");
```

Gdybyśmy chcieli przejrzeć kolejne elementy uzyskanej w ten sposób tablicy (w celach testowych lub tylko dla zabawy), możemy użyć pętli w następującej postaci:

```
String [] sizes = request.getParameterValues("sizes");
for (int x=0; x < sizes.length; x++) {
  out.println("<br>sizes:" + sizes[x]);
}
```

(przyjmijmy, że `out` jest powiązanym z obiektem odpowiedzi obiektem klasy `PrintWriter`)

Co poza parametrami można odczytać z obiektu żądania?

Interfejsy `ServletRequest` i `HttpServletRequest` oferują mnóstwo metod, które możesz wywoływać, ale których nie musisz koniecznie pamiętać. *Z pewnością* powinieneś we własnym zakresie przyjrzeć się pełnemu interfejsowi API dla interfejsów `javax.servlet.ServletRequest` i `javax.servlet.HttpServletRequest` — w tym rozdziale wspominamy tylko o tych metodach, które programiści serwletów stosują najczęściej (i które mogą się pojawić w pytaniach egzaminacyjnych).

W praktyce będziesz miał przyjemność (lub, w zależności od Twojego punktu widzenia, nieprzyjemność) stosowania nieco ponad 15% interfejsu API żądania. ***Nie martw się, jeśli nie dysponujesz jeszcze wystarczającą wiedzą na temat technik wykorzystywania każdego z elementów tego interfejsu,*** z analizą części z nich (w szczególności tych związanych z ciasteczkami klienta) będziesz miał okazję zapoznać się w dalszej części tej książki.

Informacje o platformie i przeglądarce klienta

```
String client = request.getHeader("User-Agent");
```

Ciasteczka klienta związane z danym żądaniem

```
Cookie[] cookies = request.getCookies();
```

Sesja związana z danym klientem

```
HttpSession session = request.getSession();
```

Użyta w żądaniu metoda protokołu HTTP protokołu

```
String theMethod = request.getMethod();
```

Strumień wejściowy z danego żądania

```
InputStream input = request.getInputStream();
```

Interfejs ServletRequest
(javax.servlet.ServletRequest)

<<interfejs>> ***ServletRequest***
getAttribute(String)
getContentLength()
getInputStream()
getLocalPort()
getRemotePort()
getServerPort()
getParameter(String)
getParameterValues(String)
getParameterNames()
// I WIELE innych metod...

Interfejs HttpServletRequest
(javax.servlet.http.HttpServletRequest)

<<interfejs>> ***HttpServletRequest***
getContextPath()
getCookies()
getHeader(String)
getIntHeader(String)
getMethod()
getQueryString()
getSession()
// I WIELE innych metod...

Nie ma niemądrych pytań

P: Po co miałbym kiedykolwiek używać obiektu `InputStream` zawartego w żądaniu?

O: Wraz z żądaniem `GET` otrzymujemy wyłącznie dane przesłane w ramach informacji nagłówka żądania. Innymi słowy, nie mamy do czynienia z ciałem żądania. Okazuje się JEDNAK, że zupełnie inaczej jest w przypadku żądania `POST` protokołu HTTP, gdzie oprócz nagłówka otrzymujemy ciało. Przez większość czasu będzie nas interesowało właśnie wyciąganie wartości parametrów z tego ciała (np. "`kolor=ciemny`") za pomocą prostej metody `request.getParameter()`, ale warto pamiętać, że takie wartości mogą być całkiem duże. Jeśli uznamy, że dobrym rozwiązaniem będzie odczytanie nieprzetworzonych bajtów reprezentujących dane dołączone do żądania, możemy użyć metody `getInputStream()`. Dysponując obiektem strumienia wejściowego, możesz np. odrzucić wszelkie informacje nagłówka i przystąpić do przetwarzania surowych bajtów składających się na właściwe informacje (ciało) zawarte w danym żądaniu — jednym z możliwych rozwiązań jest ich bezpośrednie zapisanie w pliku na serwerze.

P: Jaka jest różnica pomiędzy metodami `getHeader()` i `getIntHeader()`? Z tego co wiem, nagłówki żądań HTTP zawsze są łańcuchami! Nawet metoda `getIntHeader()` pobiera w postaci parametru łańcuch reprezentujący nazwę nagłówka, co więc oznacza fragment *`Int`* w nazwie metody?

O: Nagłówki mają zarówno *nazwę* (jak choćby "`User-Agent`" czy "`Host`"), jak i *wartość* (odpowiednio "`Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4)Gecko/20040624 Netscape/7.1`" i "`www.wickedlysmart.com`"). Co prawda wartości przesyłane w nagłówkach żądań zawsze mają postać łańcuchów, jednak w przypadku kilku z nich łańcuchy reprezentują pewne liczby. Przykładowo, nagłówek "`Content-Length`" zwraca liczbę bajtów zawartych w ciele komunikatu; nagłówek "`Max-Forwards`" zwraca liczbę całkowitą określającą maksymalną liczbę routerów na drodze żądania (możemy wykorzystać ten nagłówek do śledzenia trasy żądania, o którym sądzimy, że mógł utknąć w jakiejś pętli).

Wartość nagłówka "`Max-Forwards`" możemy łatwo uzyskać za pomocą metody `getHeader()`:

```
String forwards = request.getHeader("Max-Forwards");
int forwardsNum = Integer.parseInt(forwards);
```

Takie rozwiązanie działa bez zarzutu. Skoro jednak *wiemy*, że wartość tego nagłówka powinna być reprezentowana w postaci liczby całkowitej, możemy dla wygody od razu użyć metody `getIntHeader()`, dzięki której będziemy mogli uniknąć dodatkowego kroku parsowania łańcucha do postaci liczby całkowitej:

```
int forwardsNum = request.getIntHeader("Max-Forwards");
```

Mylą mi się metody `getServerPort()`, `getLocalPort()` i `getRemotePort()`!

Znaczenie metody `getServerPort()` powinno być oczywiste... przynajmniej do momentu, w którym zapytasz, co oznacza metoda `getLocalPort()`. Przejdźmy więc do wyjaśniania pierwszej niejasności — metody `getRemotePort()`. Twoje pierwsze pytanie powinno brzmieć: „zdalny dla kogo?”. Ponieważ w tym przypadku to pytanie zadaje serwer, zdalny port musi się odnosić do portu po stronie KLIENTA. Klient jest z punktu widzenia serwera zdalnym węzłem, zatem metoda **`getRemotePort()`** musi oznaczać ***„zwróć port klienta”***. Innymi słowy, metoda zwraca numer portu, z którego klient wysłał dane żądanie. Pamiętaj, jeśli jesteś serwerem, określenie zdalny (ang. *remote*) dotyczy ***klienta***.

Różnica pomiędzy metodą `getLocalPort()` a metodą `getServerPort()` jest bardziej subtelna — metoda `get`**`Server`**`Port()` odpowiada na pytanie: „na który port dane żądanie zostało WYSŁANE?”, natomiast metoda `get`**`Local`**`Port()` odpowiada na pytanie: „na który port dane żądania ostatecznie DOTARŁY?”. Tak, istnieje pewna różnica, ponieważ choć żądania są wysyłane na pojedynczy port (na którym serwer nasłuchuje), odbierający je serwer przydziela inny port lokalny dla każdego wątku serwletu, dzięki czemu aplikacja może jednocześnie obsługiwać wiele klientów.

Przypomnienie. Cykl życia serwletu i interfejs API

KLUCZOWE ZAGADNIENIA

- Kontener inicjalizuje serwlet przez wczytanie jego klasy, wywołanie bezargumentowego konstruktora tej klasy i wywołanie metody `init()` serwletu.
- Metoda `init()` (która może zostać przykryta przez programistę) jest wywoływana tylko raz w całym cyklu życia serwletu; ma to miejsce zawsze przed obsługą przez serwlet jakichkolwiek żądań klientów.
- Metoda `init()` zapewnia serwletowi dostęp do obiektów `ServletConfig` i `ServletContext`, które są niezbędne do uzyskiwania informacji na temat konfiguracji serwletu i jego macierzystej aplikacji internetowej.
- Kontener kończy życie serwletu przez wywołanie jego metody `destroy()`.
- Znaczną część swojego życia serwlet poświęca na wykonywaniu metody `service()` obsługującej żądania klientów.
- Każde obsługiwane przez serwlet żądanie jest wykonywane w osobnym wątku! Może istnieć co najwyżej jeden egzemplarz każdej z klas serwletów składających się na aplikację internetową.
- Twoje serwlety niemal zawsze będą rozszerzały klasę bazową `javax.servlet.http.HttpServlet`, po której będą dziedziczyły metodę `service()` pobierającą obiekty klas `HttpServletRequest` i `HttpServletResponse`.
- Klasa `HttpServlet` rozszerza klasę abstrakcyjną `javax.servlet.GenericServlet`, która z kolei implementuje większość podstawowych metod serwletu.
- Klasa `GenericServlet` implementuje interfejs `Servlet`.
- Wszystkie klasy wykorzystywane w serwletach (z wyjątkiem tych związanych ze stronami JSP) należą do jednego z dwóch pakietów: `javax.servlet` lub `javax.servlet.http`.
- W swoim kodzie klasy serwletu możesz przykryć metodę `init()` i musisz przykryć przynajmniej jedną z metod obsługujących żądania (`doGet()`, `doPost()` itp.).

<<interfejs>>
javax.servlet.Servlet

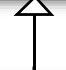
service(ServletRequest, ServletResponse)
init(ServletConfig)
destroy()
getServletConfig()
getServletInfo()

javax.servlet.GenericServlet

service(ServletRequest, ServletResponse)
init(ServletConfig)
init()
destroy()
getServletConfig()
getServletInfo()
getInitParameter(String)
getInitParameterNames()
getServletContext()
log(String)
log(String, Throwable)

javax.servlet.http.HttpServlet

service(HttpServletRequest, HttpServletResponse)
service(ServletRequest, ServletResponse)
doGet(HttpServletRequest, HttpServletResponse)
doPost(HttpServletRequest, HttpServletResponse)
doHead(HttpServletRequest, HttpServletResponse)
doOptions(HttpServletRequest, HttpServletResponse)
doPut(HttpServletRequest, HttpServletResponse)
doTrace(HttpServletRequest, HttpServletResponse)
doDelete(HttpServletRequest, HttpServletResponse)
getLastModified(HttpServletRequest)



com.wickedlysmart.examples.MojSerwlet

doPost(HttpServletRequest, HttpServletResponse)
MojaMetodaBiznesowa()

Przypomnienie. Protokół HTTP i klasa HttpServletRequest

KLUCZOWE ZAGADNIENIA

- Metody `doGet()` i `doPost()` klasy `HttpServlet` pobierają w postaci argumentów po jednym obiekcie klas `HttpServletRequest` i `HttpServletResponse`.
- Metoda `service()` określa na podstawie metody użytej w żądaniu protokołu HTTP (`GET`, `POST` itp.), czy w serwlecie należy wywołać metodę `doGet()` czy metodę `doPost()`.
- Żądania `POST` zawierają ciała, których nie stosuje się w żądaniach `GET`; żądania `GET` mogą jednak zawierać parametry dołączone do adresu URL (nazywane niekiedy *łańcuchami zapytań*).
- Żądania `GET` są z natury rzeczy (zgodnie ze specyfikacją protokołu HTTP) powtarzalne (idempotentne). Powinny umożliwiać wielokrotne wykonywanie tych samych działań bez jakichkolwiek negatywnych skutków w pracy serwletu. Żądania `GET` *nie powinny* zmieniać czegokolwiek w serwlecie, co wcale nie oznacza, że nie możesz napisać złej, nieidempotentnej metody `doGet()`.
- Żądania `POST` są z natury rzeczy niepowtarzalne (nieidempotentne), zatem tylko od Ciebie zależy, czy projektując i kodując swoją aplikację, uwzględniłeś możliwość omyłkowego wysłania dwóch identycznych żądań przez klienta.
- Jeśli formularz HTML nie zawiera zdefiniowanego wprost atrybutu `"method=POST"`, do serwletu zostanie wysłane żądanie `GET`, a nie `POST`. Jeśli Twój serwlet nie zawiera metody `doGet()`, takie żądanie spowoduje błąd.
- Parametry z obiektu reprezentującego żądanie możesz odczytywać za pomocą metody `getParameter("nazwaParametru")`. Metoda zawsze zwraca łańcuch.
- Jeśli dla jednej nazwy parametru może jednocześnie istnieć wiele wartości, powinieneś użyć metody `getParameterValues("nazwaParametru")`, która zwraca tablicę łańcuchów.
- Z obiektu reprezentującego żądanie można odczytywać także inne przydatne dane, włącznie z nagłówkami, ciasteczkami klienta, sesją, łańcuchem zapytania oraz strumieniem wejściowym.

Interfejs ServletRequest
(javax.servlet.ServletRequest)

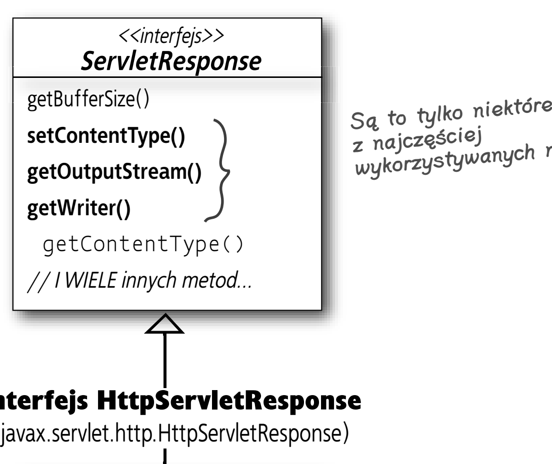
<<interfejs>> *ServletRequest*
getAttribute(String)
getContentLength()
getInputStream()
getLocalPort()
getRemotePort()
getServletPort()
getParameter(String)
getParameterValues(String)
getParameterNames()
// I WIELE innych metod...

Interfejs HttpServletRequest
(javax.servlet.http.HttpServletRequest)

<<interfejs>> *HttpServletRequest*
getContextPath()
getCookies()
getHeader(String)
getIntHeader(String)
getMethod()
getQueryString()
getSession()
// I WIELE innych metod...

Dobrze, wiemy już, do czego służy klasa Request... przyjrzyjmy się teraz klasie Response

Odpowiedź jest tym, co serwlet odsyła klientowi. Odesłana odpowiedź dociera następnie do przeglądarki internetowej, która ją parsuje i wizualizuje dla użytkownika. Obiektu odpowiedzi używa się zwykle do uzyskiwania strumienia wyjściowego (zazwyczaj obiektu klasy `PrintWriter`), w którym serwlet zapisuje odsyłany do użytkownika kod HTML (lub dowolny inny rodzaj zawartości). Poza typowymi operacjami wejścia-wyjścia dla danych wyjściowych obiekt odpowiedzi udostępnia także inne metody — niektóre z nich omówimy bardziej szczegółowo w dalszej części tego rozdziału.



Interfejs HttpServletResponse
(javax.servlet.http.HttpServletResponse)

<<interfejs>>
HttpServletRequest

addCookie()
addHeader()
encodeURL()
sendError()
setStatus()
sendRedirect()
// I WIELE innych metod...

Niekiedy będziesz korzystał także z tych metod.

Przez większość czasu będziesz używał obiektu klasy `Response` wyłącznie do odsyłania danych do klienta.

W pierwszej kolejności musisz wywołać dwie metody obiektu odpowiedzi: `setContentType()` oraz `getWriter()`.

Bezpośrednio potem możesz wykonywać proste operacje wejścia-wyjścia sprowadzające się do zapisywania w strumieniu wyjściowym kodu HTML (lub dowolnej innej zawartości).

Obiektu odpowiedzi można jednak używać także do ustawiania pozostałych nagłówków, wysyłania informacji o błędach oraz dodawania ciasteczek klienta.

Chwileczkę... nie sądziłem, że będziemy wysyłać kod HTML z poziomu naszych serwletów, ponieważ formatowanie tego kodu w strumieniu wyjściowym jest wyjątkowo niepraktyczne.

Używanie obiektu odpowiedzi do operacji wejścia-wyjścia

To prawda, powinniśmy używać stron JSP zamiast wysyłać do klienta kod HTML w konstruowanym w serwlecie strumieniu wyjściowym obiektu odpowiedzi. Formatowanie stron HTML w strumieniu wyjściowym za pomocą metody `println()` faktycznie bywa bardzo kłopotliwe.

Nie oznacza to jednak, że nigdy nie zetkniesz się z koniecznością pracy ze strumieniem wyjściowym na poziomie kodu serwletu.

Dlaczego?

1) Firma prowadząca twój serwer WWW może nie obsługiwać stron JSP. Istnieje wiele starszych serwerów i kontenerów WWW, które co prawda obsługują serwlety, ale nie obsługują stron JSP, co musimy uwzględniać w projektowanych rozwiązaniach.

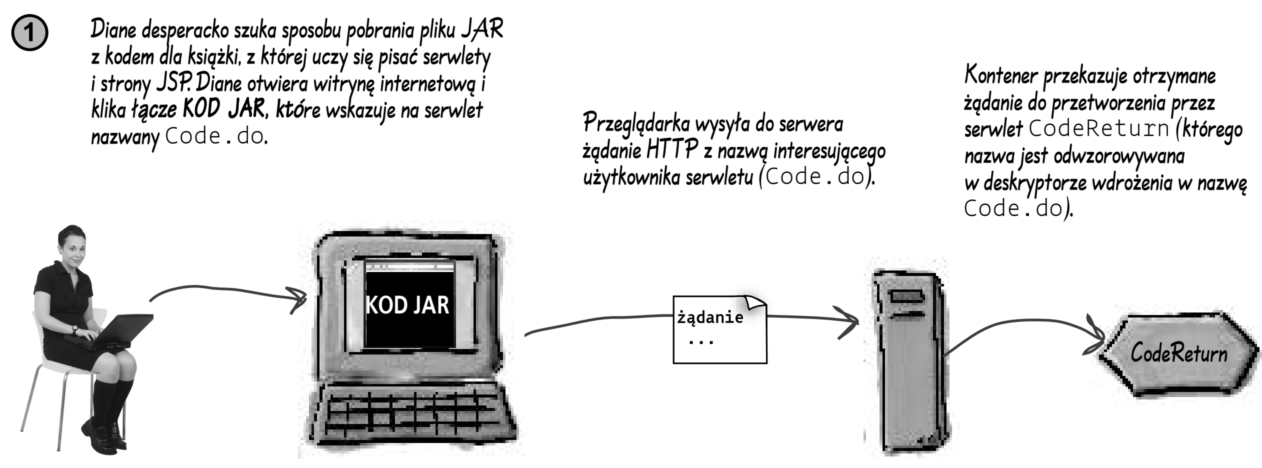
2) Opcja przewidująca użycie stron JSP może nie mieć zastosowania także z innych powodów; np. dlatego, że jesteś zmuszony do współpracy z wyjątkowo głupim kierownikiem projektu, który nie godzi się na stosowanie technologii JSP, ponieważ w 1998 roku szwagier powiedział mu, że miał w swojej pracy niedobre doświadczenia ze stronami JSP.

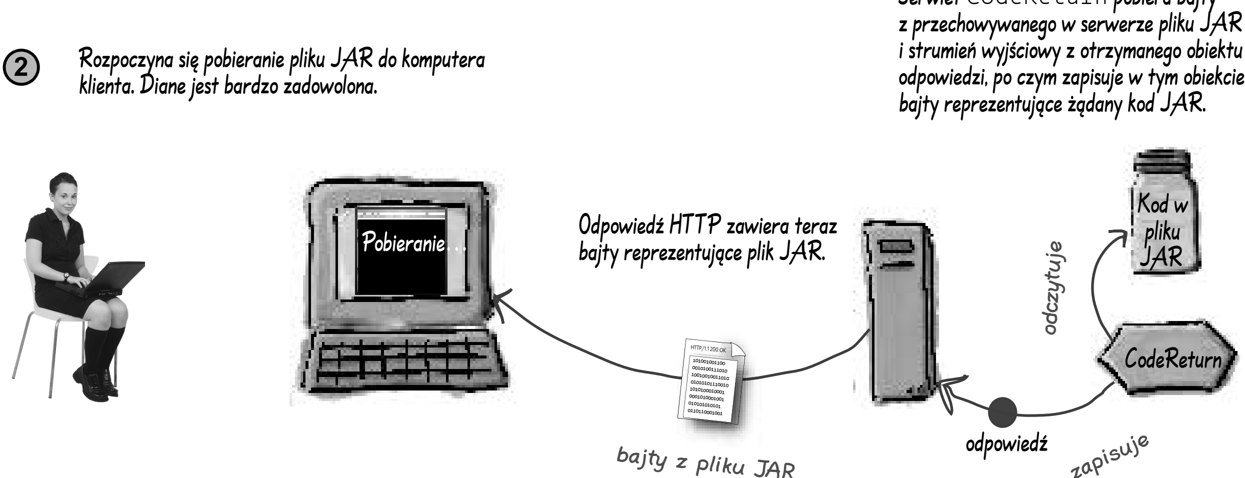
3) Kto powiedział, że kod *HTML* jest jedynym rodzajem zawartości, który możemy odsyłać w ramach odpowiedzi kierowanej do klienta? Moglibyśmy przecież równie dobrze odsyłać do klienta coś zupełnie *innego*. Może to być coś, czego obsługa za pomocą obiektu strumienia wyjściowego jest w pełni uzasadniona.

Na następnej stronie znajdziesz odpowiedni przykład…

Wyobraź sobie, że chcesz odesłać klientowi plik JAR...

Przypuśćmy, że opracowałeś stronę pobierania pliku, gdzie klient ma dostęp do kodu zawartego w plikach JAR. Zamiast odsyłać klientowi stronę HTML, obiekt odpowiedzi powinien wówczas zawierać bajty reprezentujące odpowiednie pliki JAR. *Odczytasz* bajty z pliku JAR, po czym *zapiszesz* je w strumieniu wyjściowym obiektu odpowiedzi.





Kod serwletu obsługującego pobieranie pliku JAR

```
// w tym miejscu znajduje się garść operacji importu
public class CodeReturn extends HttpServlet {

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

    response.setContentType("application/jar");

    ServletContext ctx = getServletContext();
    InputStream is = ctx.getResourceAsStream("/bookCode.jar");

    int read = 0;
    byte[] bytes = new byte[1024];

    OutputStream os = response.getOutputStream();
    while ((read = is.read(bytes)) != -1) {
      os.write(bytes, 0, read);
    }
    os.flush();
    os.close();
  }
}
```

Chcemy, aby przeglądarka wiedziała, że otrzyma bajty kodu JAR, a nie kod HTML, zatem ustawiamy inny niż zazwyczaj typ zawartości: "application/jar".

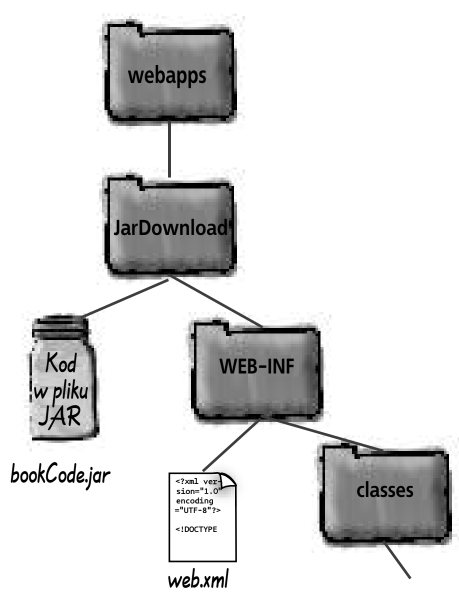
W ten sposób określamy jedynie, że chcemy uzyskać dostęp do strumienia wejściowego dla zasobu nazwanego bookCode.jar.

Oto kluczowa część tego serwletu, ale jak widać, są to tylko tradycyjne, proste operacje wejścia-wyjścia!! Nie ma w tym nic specjalnego, po prostu odczytujemy bajty kodu JAR, po czym zapisujemy bajty w strumieniu wyjściowym, do którego mamy dostęp za pośrednictwem obiektu odpowiedzi.

Nie ma niemądrych pytań

P: Gdzie powinien znajdować się udostępniany plik JAR (bookCode.jar)? Inaczej mówiąc, gdzie metoda `getResourceAsStream()` powinna SZUKAĆ tego pliku? Jak należy interpretować użytą ścieżkę?

O: Metoda `getResourceAsStream()` wymaga od nas użycia znaku ukośnika (/) reprezentującego katalog główny naszej aplikacji internetowej. Ponieważ aplikacja ta została nazwana ***JarDownload***, struktura jej katalogów powinna przypominać strukturę przedstawioną na rysunku obok. Katalog ***JarDownload*** znajduje się wewnątrz katalogu ***webapps*** (pełniącego rolę wspólnego katalogu dla wszystkich aplikacji internetowych), natomiast w samym katalogu ***JarDownload*** umieszczamy katalog ***WEB-INF*** wraz z należącym do niego plikiem JAR. Plik *bookCode.jar* jest więc przechowywany w katalogu głównym aplikacji internetowej ***JarDownload*** (nie przejmuj się, wszelkie szczegóły związane ze strukturą katalogów wdrażanej aplikacji omówimy w rozdziale poświęconym wdrażaniu).

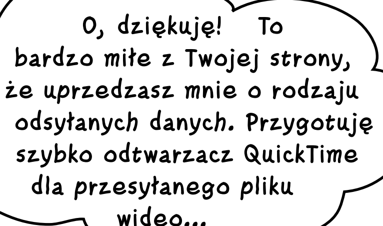


Świetnie, ale do czego służy rodzaj zawartości?

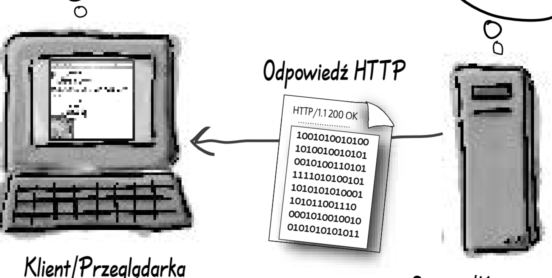
Być może zastanawiasz się, co w przedstawionym kodzie serwletu oznacza następujący wiersz:

```
response.setContentType(„application/jar”);
```

Z pewnością *powinieneś*. Musisz przecież przekazać przeglądarce, co do niej odsyłasz w odpowiedzi HTTP, ponieważ tylko w ten sposób możliwe będzie ***zapewnienie właściwej obsługi tej odpowiedzi***: uruchomienie aplikacji „pomocniczej” (przeglądarki dokumentów PDF lub odtwarzacza plików wideo), wizualizacja kodu HTML, zapisanie bajtów odpowiedzi w postaci pobranego pliku etc. Skoro tak Cię to interesuje, od razu odpowiadamy: owszem, kiedy mówimy o *rodzaju zawartości*, myślimy o typie MIME. Rodzaj zawartości jest jednym z tych nagłówków protokołu HTTP, który *musi* być dołączany do każdej odpowiedzi HTTP.



Oto moja odpowiedź na Twoje żądanie. Rodzajem zawartości tej odpowiedzi jest video/quicktime — w swoim żądaniu wspominałeś, że taka odpowiedź Cię satysfakcjonuje, a ja mam do Ciebie pełne zaufanie.



Najczęściej używane typy MIME:

text/html
application/pdf
video/quicktime
application/java
image/jpeg
application/jar
application/octet-stream
application/x-zip

Nie musisz pamiętać dostępnych rodzajów zawartości.

Relax

Powinieneś wiedzieć, co robi metoda `setContentType()`, i jak należy z niej korzystać, ale za wyjątkiem typu MIME `text/html` nie musisz znać na pamięć nawet najczęściej stosowanych rodzajów zawartości.

Z pewnością należy wiedzieć, jak w praktyce stosuje się metodę `setContentType()`... przykładowo, wywołanie tej metody PO zapisaniu odpowiedzi w strumieniu wyjściowym nic nie zmieni. To oczywiste. Ale wcale nie musi to oznaczać, że nie możesz ustawić rodzaju zawartości, zapisać jakieś dane w strumieniu wyjściowym, po czym zmienić rodzaj zawartości i zapisać w strumieniu wyjściowym inne dane. Warto się jednak zastanowić, jak tak skonstruowana odpowiedź zostanie zinterpretowana przez przeglądarkę internetową. Przeglądarka może jednocześnie obsługiwać tylko jeden rodzaj TREŚCI w otrzymanej odpowiedzi.

Upewnij się, że wszystko działa prawidłowo; dobrą praktyką (a w niektórych przypadkach absolutną koniecznością) jest wywoływanie metody `setContentType()` zawsze jako pierwszej, a więc PRZED wywołaniem metody zwracającej obiekt strumienia wyjściowego (`getWriter()` lub `getOutputStream()`). W ten sposób możemy zagwarantować brak konfliktów pomiędzy rodzajem zawartości a strumieniem wyjściowym.

Nie ma niemądrych pytań

P: Dlaczego musimy ustawiać rodzaj zawartości? Czy serwer nie może sam określić rozszerzenia odsyłanego pliku?

O: Większość serwerów *może*, przynajmniej dla zawartości statycznej. Przykładowo, w serwerze Apache możemy ustawić typy MIME przez odwzorowanie określonych rozszerzeń plików (*.txt*,*.jar*, etc.) na konkretne rodzaje zawartości — serwer Apache będzie wówczas automatycznie wykorzystywał te odwzorowania do ustawiania nagłówka rodzaju zawartości w odpowiedzi protokołu HTTP. Mówimy jednak o tym, co dzieje się w ramach serwletu, gdzie NIE wysyłamy pliku! Odpowiadamy wyłącznie za odesłanie odpowiedzi, a obsługujący nasz serwlet kontener nie ma pojęcia, co się w tej odpowiedzi znajduje.

P: Jak wobec tego wygląda sytuacja w ostatnim przykładzie, w którym odczytywaliśmy konkretny plik JAR? Czy kontener nie może wykryć, że odczytujemy plik JAR?

O: Nie. W naszym serwlecie jedynie odczytujemy bajty z pliku (w tym przypadku jest to akurat plik JAR), obracamy się i zapisujemy odczytane bajty w strumieniu wyjściowym. Kontener nie ma pojęcia, co robiliśmy w czasie, gdy byliśmy zajęci odczytywaniem bajtów z naszego pliku JAR. Wiedza kontenera jest bardzo ograniczona — odczytujesz pewien rodzaj danych i zapisujesz coś zupełnie innego w obiekcie odpowiedzi.

P: Gdzie mogę znaleźć nazwy najbardziej popularnych rodzajów zawartości?

O: Poszukaj ich w wyszukiwarce *Google*. Poważnie. Co prawda stale są dodawane nowe typy MIME, jednak ich listę można bez trudu znaleźć w internecie. Możesz także zajrzeć do ustawień swojej przeglądarki internetowej, gdzie powinna się znajdować lista tych typów MIME, które skonfigurowano w przeglądarce; źródłem wiedzy na temat obsługiwanych rodzajów zawartości są również pliki konfiguracyjne Twojego serwera WWW. Znajomość tych typów nie powinna być weryfikowana na egzaminie i jest mało prawdopodobne, żebyś kiedykolwiek jej potrzebował podczas pracy nad rzeczywistymi aplikacjami internetowymi.

P: Zaczekaj… dlaczego użyliśmy serwletu do odsyłania zawartości pliku JAR, skoro mogliśmy po prostu wykorzystać serwer WWW do udostępniania i odsyłania tego pliku jako jednego z zasobów? Innymi słowy, dlaczego nie użyliśmy klikanego przez użytkownika łącza, które wskazywałoby na dany plik JAR zamiast na serwlet? Czy nie moglibyśmy tak skonfigurować naszego serwera WWW, aby odsyłał plik JAR bezpośrednio (a więc BEZ angażowania serwletu)?

O: Tak. Dobre pytanie. MOGLIBYŚMY skonfigurować serwer WWW w taki sposób, aby użytkownik klikał łącze HTML wskazujące np. na plik JAR składowany na serwerze (tak jak inne zasoby, włącznie z obrazami JPEG czy plikami tekstowymi), i aby serwer odsyłał żądany plik (zasób) w twojej odpowiedzi.

Zakładamy jednak, że w naszym serwlecie będą wykonywane dodatkowe czynności ZANIM odczytane dane zostaną odesłane w postaci strumienia wyjściowego. Możemy np. umieścić w serwlecie logikę określającą, *który* plik JAR należy odesłać klientowi. Możemy także odsyłać klientowi bajty tworzone „w locie", a więc w trakcie obsługi jego żądania. Wyobraźmy sobie systemy, w którym pobieramy od użytkownika parametry wejściowe, wykorzystujemy je do dynamicznego generowania dźwięku, który następnie odsyłamy do klienta. Dźwięk przekazywany do użytkownika byłby wówczas zasobem, który w ogóle nie istniał przed otrzymaniem komunikatu. Innymi słowy, nie byłoby to zasób mający postać pliku dźwiękowego składowanego gdzieś na serwerze. Byłby to natomiast zasób stworzony specjalnie na żądanie klienta i odesłany do niego w postaci odpowiedzi HTTP.

Być może masz rację, być może nasz przykład z odsyłaniem pliku JAR przechowywanego na serwerze jest nietrafiony… jeśli jednak użyjesz wyobraźni i wzbogacisz tę aplikację o możliwie wyszukane elementy, być może okaże się, że zastosowanie serwletu było *uzasadnione*. Być może wystarczy zrobić coś tak prostego jak dodanie do serwletu kodu, który oprócz odsyłania zawartości pliku JAR zapisze w bazie danych pewne informacje o użytkowniku pobierającym ten plik. Może będziemy musieli sprawdzić, czy dany użytkownik ma dostęp do żądanego kodu JAR, w oparciu o odpowiednie informacje składowane w bazie danych.

Masz do wyboru dwa rodzaje danych wyjściowych: znaki lub bajty

Poniżej zastosowaliśmy tradycyjny, prosty kod `java.io`; jedynym wyjątkiem jest wykorzystany interfejs `ServletResponse`, który daje nam do wyboru *dwa* strumienie: `ServletOutputStream` (dla bajtów) oraz `PrintWriter` (dla danych znakowych).

➤ PrintWriter

Przykład

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

writer.println("jakiś tekst i kod HTML");
```

Zastosowania:

Zapisywanie danych tekstowych w strumieniu znakowym. Chociaż nadal *możemy* zapisywać tego typu dane w strumieniu `OutputStream`, jednak w takich sytuacjach warto korzystać ze strumienia `PrintWriter`, który zaprojektowano specjalnie z myślą o danych znakowych.

➤ OutputStream

Przykład

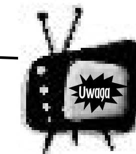
```
ServletOutputStream out = response.getOutputStream();

out.write(obiektKlasyByteArray);
```

Zastosowania:

Zapisywanie *wszelkich innych danych*!

Do Twojej wiadomości: Strumień `PrintWriter` *w rzeczywistości jest „opakowaniem" strumienia* `ServletOutputStream`*. Innymi słowy, klasa* `PrintWriter` *wykorzystuje referencję do klasy* `ServletOutputStream` *i przekazuje jej do przetworzenia swoje wywołania. Istnieje oczywiście tylko JEDEN strumień wyjściowy docierający do klienta, ale klasa* `PrintWriter` *„dekoruje" ten strumień przez dodanie przyjaznych znakowo metod wyższego poziomu.*



MUSISZ zapamiętać te metody

Z wymienionymi obok metodami musisz się dobrze zapoznać przed egzaminem. Ich zapamiętanie nie jest trudne. Zwróć uwagę, że do zapisywania danych w strumieniu `Servlet`**`OutputStream`** używasz metody **`write()`**, ale do zapisywania danych tekstowych w strumieniu **`PrintWriter`** stosuje się... metodę **`println()`**. Można oczywiście przyjąć naturalne założenie, że zapis w strumieniu **`Writer`** wymaga metody **`write`**, ale nie jest to konieczne. Jeśli stosowałeś już pakiet `java.io`, zapamiętanie tych metod nie powinno stanowić większego problemu. Jeśli jednak z tego typu operacjami nie miałeś wcześniej do czynienia, po prostu zapamiętaj:

`println()` dla strumienia **`PrintWriter`**

`write()` dla strumienia **`ServletOutputStream`**

Musisz także zapamiętać, że w nazwach metod zwracających zarówno obiekt strumienia znakowego, jak i obiekt strumienia bajtowego nie występuje pierwsze słowo zwracanego typu:

`ServletOutputStream` out = response.get`OutputStream();`

`PrintWriter` writer = response.get`Writer();`

Musisz także potrafić prawidłowo rozpoznawać NIEPOPRAWNE nazwy metod:

getPrintWriter()
getResponseStream()
getStream()
getOutputWriter()

Te metody NIE istnieją!

Możesz ustawiać nagłówki odpowiedzi, możesz dodawać nagłówki odpowiedzi

Być może zastanawiasz się, jaka jest różnica pomiędzy ustawianiem a dodawaniem nagłówków odpowiedzi. Przemyśl to i spróbuj wykonać poniższe ćwiczenie .

Dopasuj wywołanie metody do jej zachowania

Narysuj strzałkę łączącą wywołanie metody klasy `HttpResponse` z odpowiednim zachowaniem metody. Najbardziej oczywistego wyboru dokonaliśmy już za Ciebie.

```
response.setHeader("foo", "bar");

response.addHeader("foo", "bar");

response.setIntHeader("foo", 32  );
```

Dodaje do odpowiedzi nowy nagłówek wraz z wartością lub dodaje nową wartość do istniejącego nagłówka.

Przydatna metoda, która zastępuje wartość istniejącego nagłówka nową wartością całkowitoliczbową lub dodaje do odpowiedzi nowy nagłówek wraz z całkowitoliczbową wartością.

Jeśli nagłówek z taką nazwą już istnieje w obiekcie odpowiedzi, jego oryginalna wartość jest zastępowana nazwą przekazaną za pośrednictwem drugiego argumentu tej metody. W przeciwnym przypadku metoda ta dodaje do obiektu odpowiedzi nowy nagłówek wraz z wartością.

Kiedy wszystkie te metody wymienimy w jednym miejscu, ich znaczenie wydaje się oczywiste.

Jednak podczas egzaminu powinieneś znać te metody na pamięć — kiedy w środku nocy z następnego wtorku na środę ktoś Cię zapyta: „Która metoda obiektu odpowiedzi umożliwia dodawanie wartości do istniejących nagłówków?", powinieneś bez zastanowienia odpowiedzieć: „Jest to oczywiście metoda `addHeader()`, która pobiera dwa łańcuchy reprezentujące nazwę i wartość nagłówka.". Właśnie tak powinieneś być przygotowany do egzaminu.

Zarówno metoda `setHeader()`, jak i metoda `addHeader()` dodaje do obiektu odpowiedzi nagłówek i wartość, jeśli obiekt ten jeszcze nie zawiera danego nagłówka (pierwszego argumentu metody). Różnica pomiędzy tymi metodami ujawnia się dopiero w sytuacji, gdy obiekt odpowiedzi *zawiera* wskazany nagłówek. W takim przypadku:

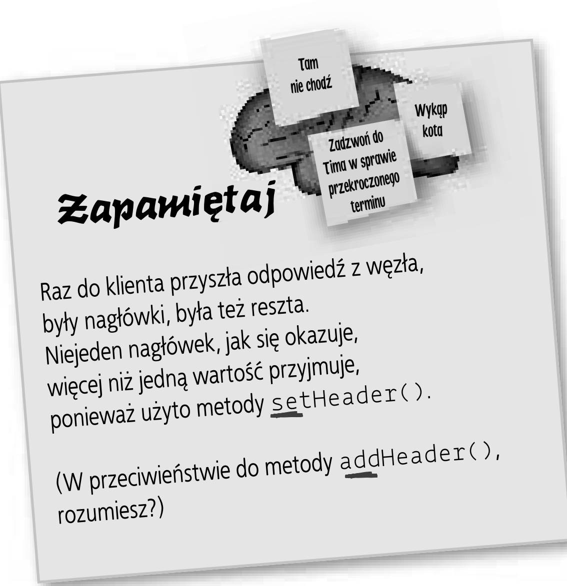
metoda ***set*Header()** nadpisuje istniejącą wartość,

metoda ***add*Header()** dodaje nową wartość.

Kiedy wywołujemy metodę `setContentType("text/html")`, w rzeczywistości ustawiamy odpowiedni nagłówek, a więc wywołanie to nie różni się od wywołania innej metody:

setHeader(„content-type", „text/html");

Jaka jest więc różnica pomiędzy tymi wywołaniami? Żadna, *pod warunkiem, że prawidłowo wpiszesz nazwę nagłówka* `"content type"`. Metoda `setHeader()` nie zasygnalizuje błędu, jeśli popełnisz błąd w pisowni nazwy któregoś z nagłówków — po prostu założy, że dodajesz nowy rodzaj nagłówka. Błąd wyjdzie na jaw nieco później, kiedy okaże się, że nie ustawiłeś prawidłowo rodzaju zawartości odpowiedzi!

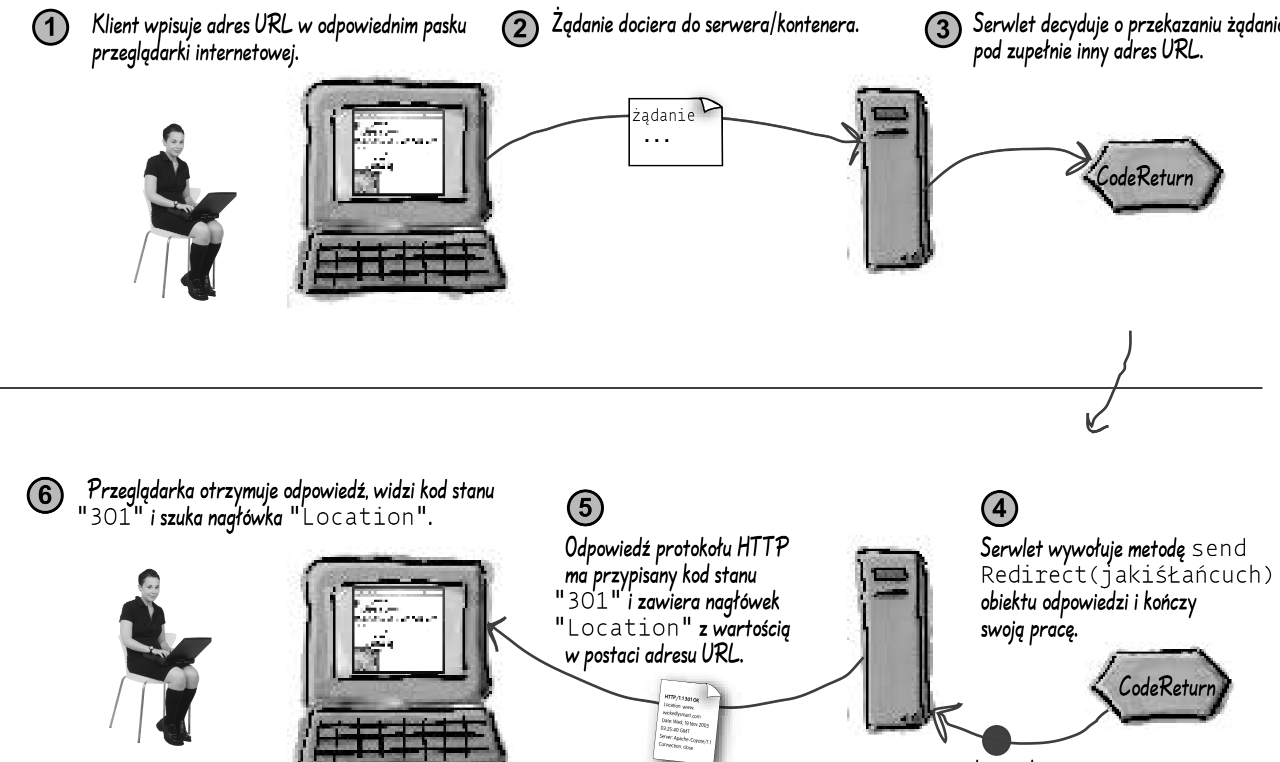


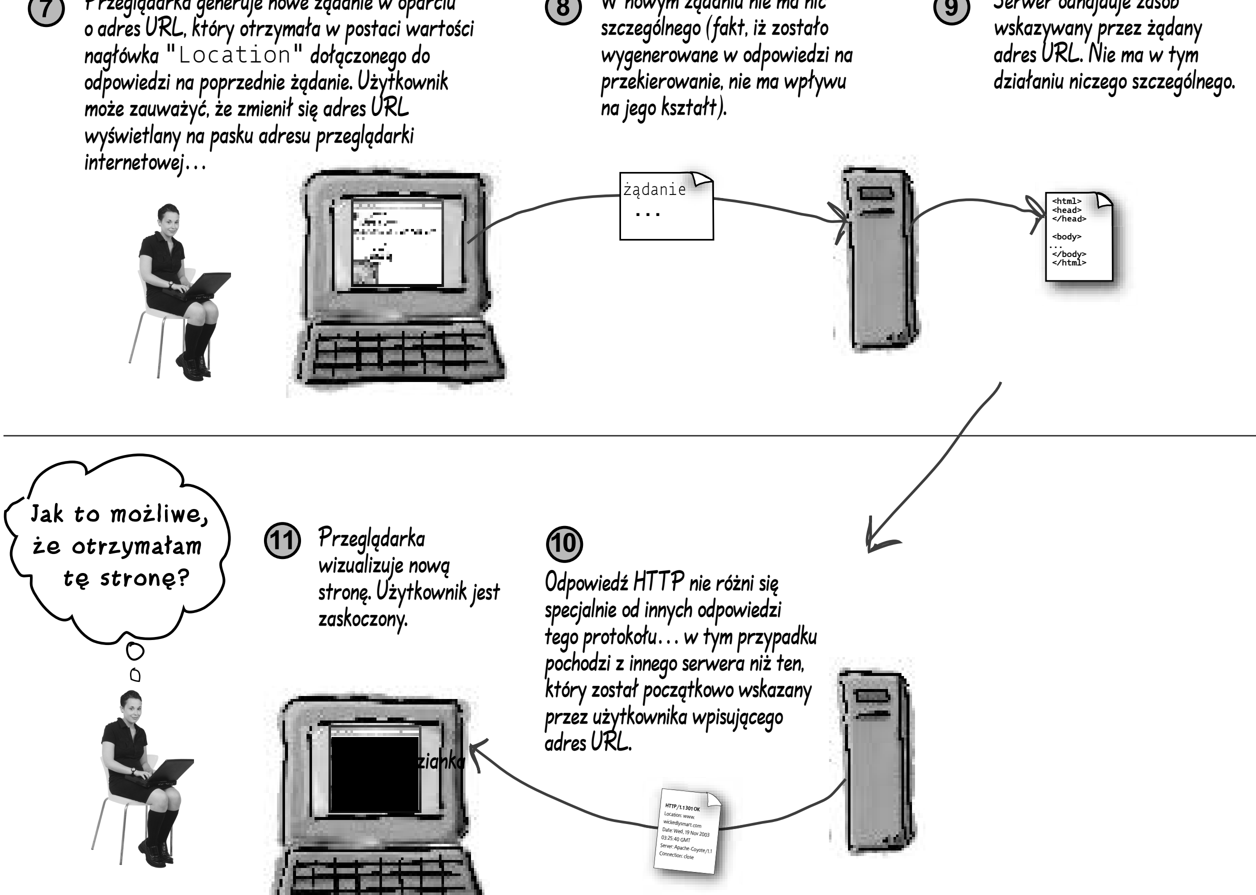
(Pierwsza osoba, która wyśle nam plik MP3 z nagraną recytacją tego wierszyka z prawidłowo rozłożonymi akcentami otrzyma specjalnie przygotowaną koszulkę.)

Niekiedy po prostu nie chcemy samodzielnie operować na obiektach reprezentujących odpowiedzi...

Możesz wybrać opcję przewidującą obsługę odpowiedzi na żądanie przez inny moduł programowy. Możesz albo *przekierować* żądanie pod zupełnie inny adres URL, albo *przydzielić* (ang. dispatch) *żądanie* do innego komponentu swojej aplikacji internetowej (zwykle jest to strona JSP).

Przekierowanie




7
Przeglądarka generuje nowe żądanie w oparciu o adres URL, który otrzymała w postaci wartości nagłówka "Location" dołączonego do odpowiedzi na poprzednie żądanie. Użytkownik może zauważyć, że zmienił się adres URL wyświetlany na pasku adresu przeglądarki internetowej…
8
W nowym żądaniu nie ma nic szczególnego (fakt, iż zostało wygenerowane w odpowiedzi na przekierowanie, nie ma wpływu na jego kształt).
9
Serwer odnajduje zasób wskazywany przez żądany adres URL. Nie ma w tym działaniu niczego szczególnego.
żądanie
...
<html>
<head>
</head>
<body>
...
</body>
</html>
Jak to możliwe, że otrzymałam tę stronę?
11
Przeglądarka wizualizuje nową stronę. Użytkownik jest zaskoczony.
10
Odpowiedź HTTP nie różni się specjalnie od innych odpowiedzi tego protokołu… w tym przypadku pochodzi z innego serwera niż ten, który został początkowo wskazany przez użytkownika wpisującego adres URL.

Przekierowanie z poziomu serwletu zmusza przeglądarkę do dodatkowych działań

Mechanizm przekierowywania żądań umożliwia naszemu serwletowi całkowite pozbywanie się odpowiedzialności na rzecz innych składników tej samej aplikacji lub innych aplikacji. Kiedy nasz serwlet zdecyduje, że danego żądania nie może obsłużyć, może po prostu wywołać metodę `sendRedirect()`:

```
if (mogęPrzetworzyć) {
 // obsługa żądania
} else {
   response.sendRedirect("http://www.helion.pl");
}
```

Adres URL, który ma być użyty dla danego żądania przez przeglądarkę. Witryna reprezentowana przez ten adres zostanie wyświetlona w oknie przeglądarki użytkownika.

Stosowanie względnych adresów URL w wywołaniach metody sendRedirect()

Okazuje się, że w roli argumentu metody `sendRedirect()` możemy używać względnych adresów URL, zamiast określać całe wyrażenia `"http://www..."`. Każdy taki adres URL może mieć jedną z dwóch postaci: rozpoczynającą się od znaku ukośnika lub pozbawioną tego znaku.

Wyobraź sobie, że klient wpisał w polu adresu przeglądarki internetowej następujący adres:

`http://www.wickedlysmart.com/myApp/cool/bar.do`

Kiedy odpowiednie żądanie dotrze do serwletu nazwanego `bar.do`, serwlet ten wywoła metodę `sendRedirect()` ze względnym adresem URL, który nie będzie się rozpoczynał od znaku ukośnika:

`sendRedirect("foo/stuff.html");`

Kontener zna oryginalny adres URL żądania rozpoczynający się od ścieżki myApp/cool, jeśli zatem nie użyjemy znaku ukośnika, ta część ścieżki zostanie dodana na początek adresu foo/stuff.html.

Kontener buduje pełny adres URL (taki adres jest niezbędny dla umieszczonego w odpowiedzi HTTP nagłówka `"Location"`) na bazie adresu URL z oryginalnego żądania:

`http://www.wickedlysmart.com/myApp/cool/foo/stuff.html`

Gdyby jednak argument metody `sendRedirect()` ROZPOCZYNAŁ się od znaku ukośnika:

`sendRedirect("/foo/stuff.html");`

Znak ukośnika na początku adresu przekierowania oznacza, że podano ścieżkę względem katalogu głównego tej aplikacji internetowej (w tym przypadku jest to aplikacja myApp).

Kontener zbudowałby kompletny adres URL względem same aplikacji internetowej, a nie względem adresu URL użytego w oryginalnym żądaniu. Oznacza to, że nowy adres URL miałby postać:

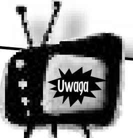
`http://www.wickedlysmart.com/myApp/foo/stuff.html`

Zwróć uwagę, że w tej wersji nie występuje już katalog cool.

Oglądnij to!

Nie możesz wywoływać metody sendRedirect() już po zapisaniu danych w obiekcie odpowiedzi!

Prawdopodobnie zasada ta jest zupełnie oczywista, ale warto ją powtarzać choćby ze względu na jej niemałe ZNACZENIE.
Jeśli przyjrzysz się metodzie `sendRedirect()` w interfejsie API, z pewnością zauważysz, że generuje ona wyjątek `IllegalStateException` w razie podjęcia próby jej wywołania już *po zatwierdzeniu odpowiedzi serwletu.*
Przez *zatwierdzenie* odpowiedzi rozumiemy jej odesłanie do klienta. Zatwierdzenie odpowiedzi oznacza po prostu zapisanie danych wyjściowych w odpowiednim strumieniu.
Z praktycznego punktu widzenia prezentowana reguła określa, że nie możesz zapisywać danych w obiekcie odpowiedzi przed wywołaniem metody **`sendRedirect()`**!
Z pewnością jednak będziesz miał jeszcze do czynienia z przemądrzałym naukowcem, który powie Ci, że z technicznego punktu widzenia możesz zapisywać dane w strumieniu bez ich zatwierdzania — wówczas wywołanie metody `sendRedirect()` nie powinno wywołać wyjątku. Takie działanie byłoby jednak na tyle bezsensowne, że nie będziemy tracić czasu na jego omawianie (oczywiście nie uwzględniając tego, który już poświęciliśmy temu zagadnieniu...).
W naszym serwlecie musimy wreszcie podjąć jakąś sensowną decyzję! Powinniśmy albo obsłużyć otrzymane żądanie, albo wywołać metodę `sendRedirect()`, która przekaże to żądanie komuś INNEMU.
(Nawiasem mówiąc, koncepcja *ograniczonego wyboru po zaakceptowaniu danych w strumieniu wyjściowym* ma zastosowanie także w przypadku ustawianych nagłówków, znaczników kontekstu klienta, kodów stanu, rodzaju zawartości itp.)



Metoda sendRedirect() pobiera zwykły łańcuch znaków, a NIE obiekt reprezentujący adres URL!

Cóż, tak naprawdę metoda ta pobiera łańcuch, który JEST jednocześnie adresem URL. Zasadnicze znaczenie ma jednak fakt, iż metoda `sendRedirect()` NIE pobiera obiektu typu `URL`. Przekazujemy w postaci parametru łańcuch, który jest albo kompletnym adresem URL, albo adresem względnym. Jeśli kontener nie będzie mógł zbudować pełnego adresu na bazie otrzymanego adresu względnego, zostanie wygenerowany wyjątek `IllegalStateException`. Najważniejsze, żebyś zapamiętał, że TAKIE wywołanie jest błędne:

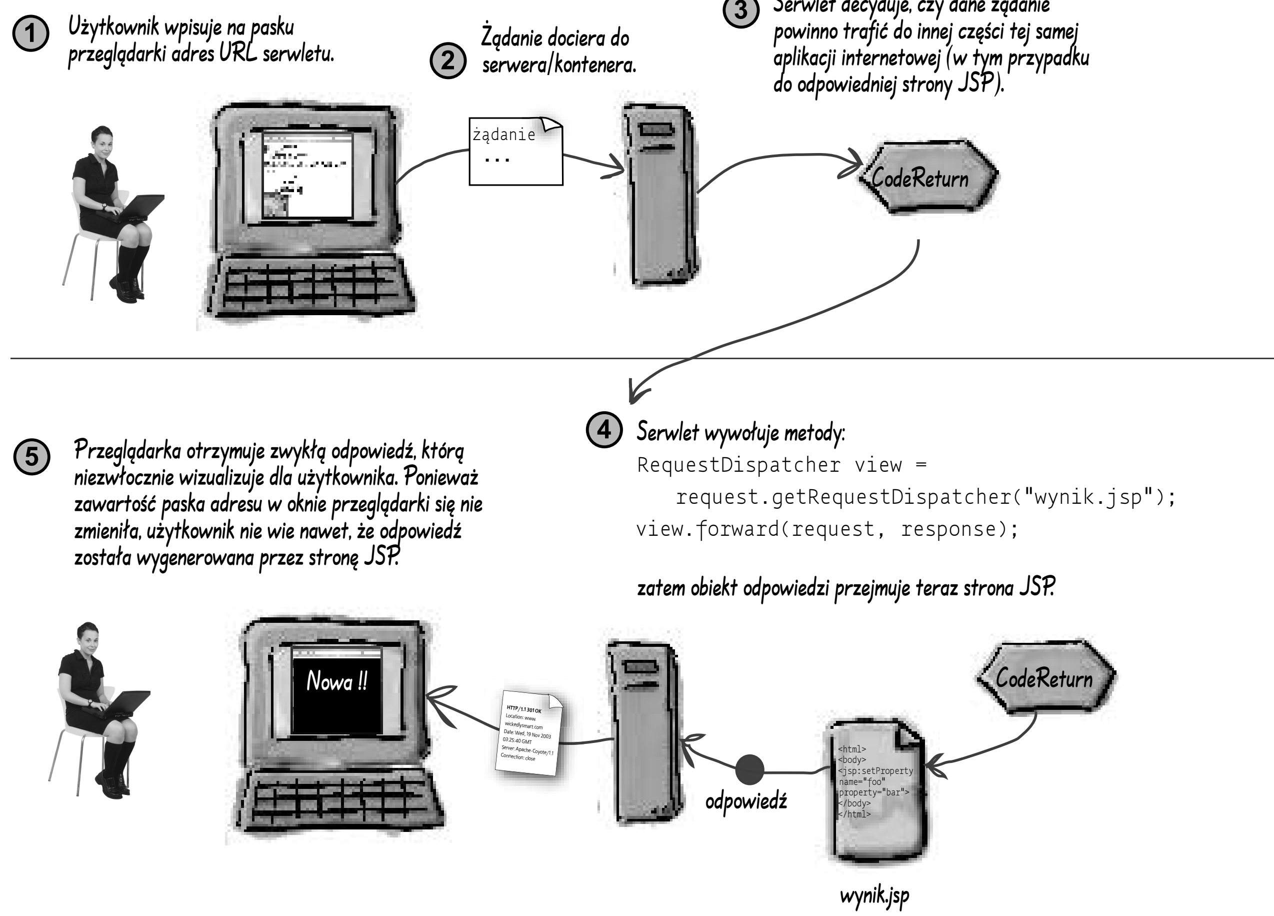
```
sendRedirect(new URL ("http://www.
helion.pl"));
```

Nie! Wygląda dobrze, ale NA PEWNO nie zadziała. Metoda sendRedirect() pobiera łańcuch znaków. I kropka.

Przydzielanie żądań odbywa się wyłącznie po stronie serwera

Na tym właśnie polega zasadnicza różnica pomiędzy przekierowywaniem żądań a ich przydzielaniem — *przekierowanie* zmusza do dodatkowych działań klienta, natomiast *przydział żądana* wymusza realizację pewnych kroków po stronie serwera. Zapamiętaj dwie proste reguły: przekierowanie = *klient*, przydział żądania = *serwer*. Mechanizmowi przekazywania żądań poświęcimy co prawda znacznie więcej uwagi w jednym z rozdziałów w dalszej części tej książki, jednak ta i kolejna strona powinna na tym etapie w zupełności wystarczyć.

Przydział żądania



Przekierowanie kontra przydział żądania

Nie mam na to czasu! Czemu nie zadzwoniłeś do Barneya. Może on ma czas na głupoty!

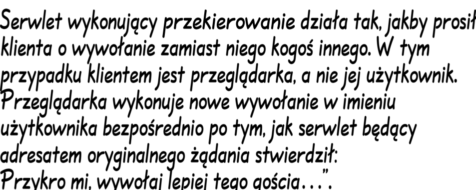
Cześć Kari, mówi Dan... Chciałbym, żebyś mi pomogła z jednym klientem. Przekażę Ci szczegóły na temat kontaktu z nim, ale musisz się tym zająć natychmiast. Tak, WIEM, że masz swoje zajęcia... tak, WIEM, jak ważny jest Widok we wzorcu Modelu-Widoku-Kontrolera... nie, nie mogę znaleźć do tego żądania innej strony JSP... co? odmawiasz... przepraszam — nie słyszę... tracę pakiety...

Przekierowanie

Przydział żądania

Serwlet wykonujący przydział żądania działa tak, jakby prosił współpracownika o przejęcie obsługi danego klienta. Współpracownik serwletu odpowiada klientowi na jego żądanie, ale klient nie wie, że nie komunikuje się ze współpracownikiem oryginalnie wywołanego serwletu.

Użytkownik nigdy nie wie, że ktoś inny przejął jego obsługę, ponieważ adres URL wyświetlany w odpowiednim pasku przeglądarki internetowej nie jest zmieniany.



Serwlet wykonujący przekierowanie działa tak, jakby prosił klienta o wywołanie zamiast niego kogoś innego. W tym przypadku klientem jest przeglądarka, a nie jej użytkownik. Przeglądarka wykonuje nowe wywołanie w imieniu użytkownika bezpośrednio po tym, jak serwlet będący adresatem oryginalnego żądania stwierdził: „Przykro mi, wywołaj lepiej tego gościa…”.

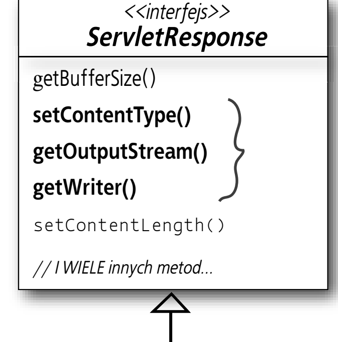
Użytkownik widzi w przeglądarce nowy adres URL.

Przypomnienie. HttpServletResponse

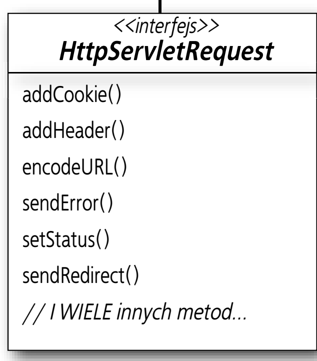
KLUCZOWE ZAGADNIENIA

- Obiektu klasy `Response` używamy do odsyłania danych do klienta.
- Do metod obiektu `HttpServletResponse`, których będziesz wywoływał najczęściej, należą `setContentType()` i `getWriter()`.
- Zachowaj ostrożność — wielu programistów sądzi, że metoda zwracająca obiekt klasy `PrintWriter` nazywa się `getPrintWriter()`, choć jej prawdziwa nazwa to `getWriter()`.
- Metoda `getWriter()` umożliwia wykonywanie znakowych operacji wejścia-wyjścia na strumieniu (operacje te najczęściej dotyczą kodu HTML, ale mogą dotyczyć dowolnych innych danych tekstowych).
- Obiektu odpowiedzi możesz używać także do ustawiania nagłówków, wysyłania informacji o błędach oraz dodawania ciasteczek.
- W rzeczywistych zastosowaniach najprawdopodobniej będziesz używał stron JSP do przesyłania większości odpowiedzi mających postać kodu HTML, ale do odsyłania klientowi danych binarnych (np. plików JAR) nadal możesz korzystać ze strumienia danych wyjściowych.
- Do uzyskiwania obiektu reprezentującego binarny strumień wyjściowy służy metoda `getOutputStream()` obiektu odpowiedzi.
- Metoda `setContentType()` określa, w jaki sposób przeglądarka powinna obsłużyć dane otrzymywane w ramach odpowiedzi HTTP. Typowymi rodzajami zawartości tego typu odpowiedzi są: `"text/html"`, `"application/pdf"` oraz `"image/jpeg"`.
- Nie musisz uczyć się na pamięć dostępnych rodzajów zawartości (znanych także jako typy MIME).
- Nagłówki odpowiedzi możesz ustawiać za pomocą metod `addHeader()` i (lub) `setHeader()`. Różnica pomiędzy tymi metodami zależy od tego, czy dany nagłówek jest już częścią konstruowanej odpowiedzi. Jeśli tak, metoda `setHeader()` *zastąpi* wartość tego nagłówka, natomiast metoda `addHeader()` *doda* do istniejącej odpowiedzi *nową wartość*. Jeśli jednak danego nagłówka nie zdefiniowano jeszcze w obiekcie odpowiedzi, działanie metod `addHeader()` i `setHeader()` będzie identyczne.
- Jeśli nie chcesz odpowiadać na otrzymane żądanie, możesz je przekierować na inny adres URL. Odpowiedzialność za wysłanie nowego żądania na wskazany przez Ciebie adres spadnie na przeglądarkę internetową.
- Aby przekierować żądanie, wywołaj dostępną w obiekcie odpowiedzi metodę `sendRedirect(łańcuchURL)`.
- Metody `sendRedirect(łańcuchURL)` nie można wywoływać po zatwierdzeniu odpowiedzi! Innymi słowy, jeśli zapisałeś już coś w strumieniu wyjściowym, na przekierowanie żądania klienta jest już za późno.
- *Przekierowanie* żądania nie jest równoznaczne z *przydziałem* żądania. Przydział żądania (szczegółowo omówiony w innym rozdziale tej książki) jest realizowany w ramach *serwera*, natomiast *przekierowanie* wymaga udziału *klienta*. Przydział żądania w praktyce sprowadza się do jego przekazania do innego komponentu tego samego serwera, zwykle do komponentu należącego do tej samej aplikacji internetowej. *Przekierowanie* żądania polega natomiast na przekazaniu przeglądarce zalecenia wysłania żądania na inny adres URL.

Interfejs ServletResponse
(javax.servlet.ServletResponse)



Interfejs HttpServletResponse
(javax.servlet.http.HttpServletResponse)



Egzamin próbny

1. W jaki sposób kod serwletu zawarty w metodzie obsługującej żądania (np. **doPost()**) może uzyskać wartość nagłówka "User-Agent" z obiektu żądania? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `String userAgent =`
  `request.getParameter("User-Agent");`
- ❑ B. `String userAgent = request.getHeader("User-Agent");`
- ❑ C. `String userAgent =`
  `request.getRequestHeader("Mozilla");`
- ❑ D. `String userAgent =`
  `getServletContext().getInitParameter("User-Agent");`

2. Które metody protokołu HTTP są wykorzystywane do sygnalizowania klientowi tego, co dociera do serwera WWW? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. GET
- ❑ B. PUT
- ❑ C. TRACE
- ❑ D. RETURN
- ❑ E. OPTIONS

3. Którą metodę interfejsu **HttpServletResponse** wykorzystuje się do przekierowywania żądań protokołu HTTP pod inne adresy URL?

- ❑ A. **sendURL()**
- ❑ B. **redirectURL()**
- ❑ C. **redirectHttp()**
- ❑ D. **sendRedirect()**
- ❑ E. **getRequestDispatcher()**

4. Które metody protokołu HTTP NIE są uważane za idempotentne (powtarzalne)? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `GET`
- ❑ B. `POST`
- ❑ C. `HEAD`
- ❑ D. `PUT`

5. Wiedząc, że **`req`** jest obiektem klasy **`HttpServletRequest`**, wskaż wywołania metod, które zwrócą binarny strumień wyjściowy. (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `BinaryInputStream s = req.getInputStream();`
- ❑ B. `ServletInputStream s = req.getInputStream();`
- ❑ C. `BinaryInputStream s = req.getBinaryStream();`
- ❑ D. `ServletInputStream s = req.getBinaryStream ();`

6. Jak należy ustawiać nagłówek nazwany "`CONTENT-LENGTH`" w obiekcie **`HttpServletResponse`**? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `response.setHeader(CONTENT-LENGTH, "numBytes");`
- ❑ B. `response.setHeader("CONTENT-LENGTH", "numBytes");`
- ❑ C. `response.setStatus(1024);`
- ❑ D. `response.setHeader("CONTENT-LENGTH", 1024 );`

7. Wskaż fragment kodu serwletu, który zwraca strumień binarny umożliwiający zapisywanie w obiekcie **`HttpServletResponse`** obrazów lub danych binarnych innego rodzaju.

- ❑ A. `java.io.PrintWriter out = response.getWriter();`
- ❑ B. `ServletOutputStream out = response.getOutputStream();`
- ❑ C.
```
java.io.PrintWriter out =
        new PrintWriter(response.getWriter());
```
- ❑ D. `ServletOutputStream out = response.getBinaryStream();`

8. Które metody są wykorzystywane w serwlecie do obsługi nadesłanych przez klienta danych z formularza? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `HttpServlet.doHead()`
- ❑ B. `HttpServlet.doPost()`
- ❑ C. `HttpServlet.doForm()`
- ❑ D. `ServletRequest.doGet()`
- ❑ E. `ServletRequest.doPost()`
- ❑ F. `ServletRequest.doForm ()`

9. Które z wymienionych poniżej metod są deklarowane w klasie `HttpServletRequest` zamiast w klasie `ServletRequest`? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. `getMethod()`
- ❑ B. `getHeader()`
- ❑ C. `getCookies()`
- ❑ D. `getInputStream()`
- ❑ E. `getParameterNames ()`

10. Jak programiści serwletów powinni obsługiwać metodę `service()` klasy `HttpServlet` podczas jej rozszerzania we własnych klasach serwletów? (Zaznacz wszystkie prawidłowe odpowiedzi.)

- ❑ A. W większości przypadków powinni przykrywać metodę `service()`.
- ❑ B. Powinni wywoływać metodę `service()` z poziomu metody `doGet()` lub `doPost()`.
- ❑ C. Powinni wywoływać metodę `service()` z poziomu metody `init()`.
- ❑ D. Powinni przykrywać przynajmniej jedną z metod `doXXX()` (np. metodę `doPost()`).

Egzamin próbny — odpowiedzi

1. W jaki sposób kod serwletu zawarty w metodzie obsługującej żądania (np. **doPost()**) może uzyskać wartość nagłówka "User-Agent" z obiektu żądania? (Zaznacz wszystkie prawidłowe odpowiedzi.)

(API)

- ☐ A. `String userAgent = request.getParameter("User-Agent");`
- ☑ B. `String userAgent = request.getHeader("User-Agent");`
- ☐ C. `String userAgent = request.getRequestHeader("Mozilla");`
- ☐ D. `String userAgent = getServletContext().getInitParameter("User-Agent");`

— Opcja B zawiera prawidłowe wywołanie metody z przekazaniem nazwy nagłówka w postaci parametru typu String.

2. Które metody protokołu HTTP są wykorzystywane do sygnalizowania klientowi tego, co dociera do serwera WWW? (Zaznacz wszystkie prawidłowe odpowiedzi.)

(Rozdział 4., metody HTTP)

- ☐ A. GET
- ☐ B. PUT
- ☑ C. TRACE
- ☐ D. RETURN
- ☐ E. OPTIONS

— Ta metoda jest zwykle wykorzystywana do rozwiązywania problemów, a nie do zwykłego przetwarzania żądań.

3. Którą metodę interfejsu **HttpServletResponse** wykorzystuje się do przekierowywania żądań protokołu HTTP pod inne adresy URL?

(API)

- ☐ A. `sendURL()`
- ☐ B. `redirectURL()`
- ☐ C. `redirectHttp()`
- ☑ D. `sendRedirect()`
- ☐ E. `getRequestDispatcher()`

— Opcja D jest w tym przypadku prawidłowa, ponieważ metoda sendRedirect() jako jedyna z wymienionych należy do interfejsu HttpServletResponse.

4. Które metody protokołu HTTP NIE są uważane za idempotentne (powtarzalne)? (Zaznacz wszystkie prawidłowe odpowiedzi.)

(Rozdział 4., żądania idempotentne)

- [ ] A. GET
- [x] B. POST
- [ ] C. HEAD
- [ ] D. PUT

— Zgodnie ze specyfikacją protokołu HTTP, metoda POST ma w założeniu służyć przesyłaniu żądań aktualizujących stan serwera. W ogólności ta sama operacja aktualizująca nie powinna być stosowana więcej niż raz.

5. Wiedząc, że `req` jest obiektem klasy `HttpServletRequest`, wskaż wywołania metod, które zwrócą binarny strumień wyjściowy. (Zaznacz wszystkie prawidłowe odpowiedzi.)

(API)

- [ ] A. `BinaryInputStream s = req.getInputStream();`
- [x] B. `ServletInputStream s = req.getInputStream();`
- [ ] C. `BinaryInputStream s = req.getBinaryStream();`
- [ ] D. `ServletInputStream s = req.getBinaryStream ();`

— Opcja B zawiera zarówno wywołanie właściwej metody, jak i prawidłowy typ zwracanego obiektu.

6. Jak należy ustawiać nagłówek nazwany "`CONTENT-LENGTH`" w obiekcie `HttpServletResponse`? (Zaznacz wszystkie prawidłowe odpowiedzi.)

(API)

- [ ] A. `response.setHeader(CONTENT-LENGTH, "numBytes");`
- [x] B. `response.setHeader("CONTENT-LENGTH", "numBytes");`
- [ ] C. `response.setStatus(1024);`
- [ ] D. `response.setHeader("CONTENT-LENGTH", 1024 );`

— Opcja B demonstruje właściwy sposób ustawiania nagłówka protokołu HTTP z dwoma parametrami łańcuchowymi: jednym reprezentującym nazwę nagłówka i drugim reprezentującym wartość.

7. Wskaż fragment kodu serwletu, który zwraca strumień binarny umożliwiający zapisywanie w obiekcie `HttpServletResponse` obrazów lub danych binarnych innego rodzaju.

(API)

- [ ] A. `java.io.PrintWriter out = response.getWriter();`
- [x] B. `ServletOutputStream out = response.getOutputStream();`
- [ ] C. `java.io.PrintWriter out = new PrintWriter(response.getWriter());`
- [ ] D. `ServletOutputStream out = response.getBinaryStream();`

— Opcja A nie jest prawidłową odpowiedzią, ponieważ wykorzystuje obiekt klasy `PrintWriter` stworzony z myślą o danych tekstowych.

8. Które metody są wykorzystywane w serwlecie do obsługi nadesłanych przez klienta danych z formularza? (Zaznacz wszystkie prawidłowe odpowiedzi.) (API)

- [ ] A. `HttpServlet.doHead()`
- [x] B. `HttpServlet.doPost()`
- [ ] C. `HttpServlet.doForm()`
- [ ] D. `ServletRequest.doGet()`
- [ ] E. `ServletRequest.doPost()`
- [ ] F. `ServletRequest.doForm()`

— Opcje C – F są błędne, ponieważ wymienione w nich metody w ogóle nie istnieją.

9. Które z wymienionych poniżej metod są deklarowane w klasie `HttpServletRequest` zamiast w klasie `ServletRequest`? (Zaznacz wszystkie prawidłowe odpowiedzi.) (API)

- [x] A. `getMethod()`
- [x] B. `getHeader()`
- [x] C. `getCookies()`
- [ ] D. `getInputStream()`
- [ ] E. `getParameterNames()`

— Opcje A, B i C mają związek ze składnikami żądań protokołu HTTP.

10. Jak programiści serwletów powinni obsługiwać metodę `service()` klasy `HttpServlet` podczas jej rozszerzania we własnych klasach serwletów? (Zaznacz wszystkie prawidłowe odpowiedzi.) (API)

- [ ] A. W większości przypadków powinni przykrywać metodę `service()`.
- [ ] B. Powinni wywoływać metodę `service()` z poziomu metody `doGet()` lub `doPost()`.
- [ ] C. Powinni wywoływać metodę `service()` z poziomu metody `init()`.
- [x] D. Powinni przykrywać przynajmniej jedną z metod `doXXX()` (np. metodę `doPost()`).

— Opcja D jest w tym przypadku prawidłowa, ponieważ programiści zazwyczaj skupiają się metodach `doGet()` i `doPost()`.