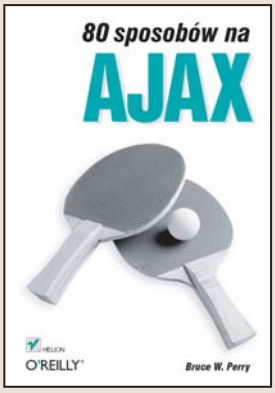
80 sposobów na Ajax



Autor: Bruce Perry
ISBN: 83-246-0557-6
Tytuł oryginału: Ajax Hacks: Tips & Tools for Creating Responsive Web Sites
Format: B5, stron: 424

Techniki tworzenia nowoczesnych aplikacji internetowych

- Przygotowywanie wygodnych w obsłudze formularzy
- Integracja witryny z Google Maps
- Zarządzanie połączeniami sieciowymi

Ajax to nazwa technologii powstałej w wyniku połączenia języka JavaScript, XML oraz kaskadowych arkuszy stylów. Jej wdrożenie pozwala wyeliminować ze stron WWW jedną z ich najbardziej irytujących cech, czyli konieczność przeładowywania zawartości po każdej zmianie. Umiejętne wykorzystanie możliwości Ajaksa sprawia, że aplikacje internetowe przypominają „zwykłe" programy dla systemów Windows lub Mac OS. Dzięki zastosowaniu Ajaksa aplikacja internetowa działa zdecydowanie szybciej, a połączenia z serwerem nie przeszkadzają użytkownikowi w pracy. Łatwiejsza jest także dynamiczna zmiana elementów w różnych częściach strony. Rozwiązania oparte na Ajaksie wprowadzono w wielu dziedzinach, takich jak blogi, narzędzia służące do nauki, newslettery oraz małe portale internetowe.

W książce „80 sposobów na Ajax" znajdziesz przykłady zastosowania tej techniki w tworzeniu interesujących i nieszablonowych witryn WWW. Czytając ją, dowiesz się, jak sprawić, aby witryny WWW były bardziej interaktywne, a aplikacje WWW działały dokładnie tak jak aplikacje desktopowe. Nauczysz się korzystać z interfejsów programistycznych witryn Google Maps, Yahoo! Maps i Geo URL oraz obsługiwać sesje i cookies z poziomu Ajaksa. Poznasz również metody weryfikowania poprawności danych wprowadzanych do formularzy oraz techniki łączenia Ajaksa z innymi nowoczesnymi narzędziami, takimi jak Ruby on Rails.

- Tworzenie interaktywnych formularzy
- Połączenia z Google Maps i Yahoo! Maps
- Korzystanie z usługi Geo URL
- Obsługa plików cookies
- Przeglądanie kanałów RSS
- Integracja z aplikacjami sieciowymi napisanymi w Javie
- Korzystanie z bibliotek Prototype i Rico
- Połączenie Ajaksa z Ruby on Rails
- Wykorzystanie biblioteki script.aculo.us do tworzenia efektów wizualnych

Twórz szybkie, wydajne i wygodne w obsłudze aplikacje sieciowe

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

Spis treści

ROZDZIAŁ PIERWSZY

Podstawy technologii Ajax

Sposoby 1. – 11.

Czy pamiętamy jeszcze czasy, w których użytkownicy określali internet mianem „world wide wait"? Wracamy do neolitycznej epoki Sieci? W przypadku niektórych aplikacji aspekty dotyczące Sieci nie uległy zasadniczej zmianie: wypełniamy formularz, klikamy przycisk, strona internetowa znika, czekamy, następuje odświeżenie strony, poprawiamy popełnione błędy, klikamy, czekamy, czekamy… Byliśmy już wcześniej zawieszeni w tego typu próżni.

Jednakże pewna ilość ostatnio powstałych witryn internetowych, na przykład doskonałe aplikacje kartograficzne, które ostatnio zostały rozwinięte, wymagają znacznie większej reakcji w trakcie współdziałania z użytkownikiem. Stary, konwencjonalny sposób współpracy z użytkownikiem zakładał, że cała strona „znika" po każdym kliknięciu, a nowa pokazuje się w przeglądarce jedynie wtedy, gdy odpowiedź z serwera jest kompletna. Oczywiście, niektóre z nowych aplikacji wymagają momentalnych zmian fragmentów strony internetowej, ale bez konieczności przeładowania całej strony.

Przykładowo, jeżeli Czytelnik kiedykolwiek używał Google Maps, to sposób, w jaki można przeciągnąć myszą regiony mapy w oknie, wywołuje wrażenie, że mapy są przechowywane lokalnie na komputerze. Możemy sobie wyobrazić, jak bardzo ta aplikacja stałaby się niepopularna, gdyby przy każdej próbie „przeciągnięcia" aktualnego widoku strona znikała na kilka (długich) chwil, w trakcie których przeglądarka czeka na odpowiedź z serwera. Aplikacja byłaby powolna i nikt nie chciałby z niej korzystać. Jakie więc czary powodują, że taka aplikacja działa?

To nie jest pasta do podłogi

Mieszanka doskonale znanych technologii oraz eleganckie narzędzie JavaScript stanowią podstawę wystrzałowego i jeszcze potężniejszego modelu aplikacji dla Sieci. Jeżeli Czytelnikowi nasuwają się obawy związane z przeciążonym akronimem, to nie warto się przejmować — jest łatwy. Ten akronim nosi nazwę *Ajax*, co oznacza Asynchronous JavaScript and XML (asynchroniczny JavaScript i XML).

Ajax nie jest ani pastą do podłogi ani polewą deserową (ani nawet produktem do czyszczenia!). Jest to natomiast mieszanka kilku standardowych technologii, które są już znane programistom i projektantom:

- JavaScript, język programowania, który dodaje do stron internetowych możliwość dynamicznego wykonywania skryptów. Kod JavaScript może zostać osadzony na stronie internetowej, co pozwala stronie na implementację nowych, doskonałych możliwości za pomocą techniki o nazwie *wykonywanie skryptów po stronie klienta*. Ta technika jest niemal tak stara jak Sieć.
- `XMLHttpRequest`, obiekt JavaScript wraz z API (ang. *Application Programming Interface* — interfejs programowy aplikacji), który może połączyć się z serwerem za pomocą protokołu HyperText Transfer Protocol (HTTP). Spora ilość magii związanej z Ajaksem ma swoje źródło w tym fragmencie kodu, który jest obsługiwany przez większość głównych przeglądarek internetowych (takich jak Mozilla Firefox, Internet Explorer 6, Safari 1.3 i 2.0 oraz Opera 7.6). Asynchroniczna część Ajaksu wywodzi się z charakterystyki tego obiektu[1].
- Extensible Markup Language (XML), język zaprojektowany do definiowania innych języków. Obiekt `XMLHttpRequest` może obsługiwać odpowiedź serwera w standardowym formacie XML, jak również w postaci zwykłego tekstu.
- HTML i kaskadowe arkusze stylów (CSS), które kontrolują wizualny aspekt strony internetowej. Programiści sieciowi mogą używać JavaScriptu do przeprowadzania dynamicznych zmian interfejsu za pomocą programowania elementów HTML oraz arkuszy CSS.
- Document Object Model (DOM), model przedstawiający plik XML lub stronę internetową jako zespół połączonych obiektów, które mogą być dynamicznie przetwarzane, nawet wtedy, gdy użytkownik pobierze stronę. Widok strony ma postać struktury jako hierarchii drzewa wraz z węzłem nadrzędnym (rodzicem) oraz jego różnymi *gałęziami* (elementami potomnymi). Każdy element HTML jest przedstawiony za pomocą węzła lub gałęzi, do których dostęp jest możliwy dzięki językowi JavaScript. W omawianych sposobach przedstawimy dużo (naprawdę dużo!) programowania DOM.
- Extensible Stylesheet Language and Transformation (XSLT), technologia szablonu umożliwiająca przekształcanie wyświetlania informacji XML przez otrzymującego ją klienta.

Ajax jest całkiem nowy, natomiast wymienione powyżej technologie są względnie stare. Firma Microsoft wypuściła pierwszą implementację obiektu JavaScript, który przeprowadza żądania HTTP (często nazywanego obiektem XMLHTTP) w wersji 5 przeglądarki Internet Explorer. W trakcie pisania tej książki przeglądarka Internet Explorer jest w wersji 6, natomiast wersja 7 znajduje się w fazie beta.

[1] Obiekt `XMLHttpRequest` może przeprowadzać asynchroniczne żądanie do serwera, co oznacza, że po zainicjowaniu żądania pozostała część kodu JavaScript nie musi czekać z wykonaniem na nadejście odpowiedzi. Obiekt `XMLHttpRequest` może również przeprowadzać żądania synchroniczne.

Jednakże obfitość nowych aplikacji, które używają Ajaksu, sugeruje, że przedstawiona grupa technologii została przekształcona do nowego modelu sieciowego. „Web 2.0" jest formą wzbogaconych aplikacji internetowych (są tak nazwane, ponieważ duża część funkcji aplikacji może znajdować się w przeglądarce internetowej klienta) następnej generacji, obejmujących Ajax. Przykładami tego typu aplikacji są: Google Maps, Gmail, pakiet o nazwie Zimbra, interesujące osobiste narzędzie wyszukiwania o nazwie Rollyo (*http://www.rollyo.com*) oraz jedna z pierwszych interaktywnych map — Szwajcarii (*http://maps.search.ch/index_en.html*). Liczba aplikacji wykorzystujących technologię Ajax gwałtownie się zwiększa. Krótką listę można znaleźć w Wikipedii, pod adresem *http://en.wikipedia.org/wiki/List_of_websites_using_Ajax*.

Zachowanie ostrożności

Oczywiście, Ajax nie jest przeznaczony dla każdego (w szczególności dla fanów polewy deserowej)! Ponieważ technologia Ajax może powodować dynamiczną zmianę strony internetowej, która została już pobrana z Sieci, istnieje niebezpieczeństwo kolizji z pewnymi funkcjami, bliższymi lub dalszymi dla wielu użytkowników, takimi jak tworzenie zakładek w przeglądarce. Przykładowo, w przypadku braku podobnych rozwiązań w zakresie wykonywania skryptów, przeprowadzane na istniejącej stronie internetowej dynamiczne zmiany za pomocą modelu DOM nie mogłyby zostać dołączone do adresu URL, który później mógłby zostać wysłany do przyjaciół lub zapisany. (Zarówno podrozdział „Naprawa przycisku Wstecz przeglądarki internetowej w aplikacjach Ajax" **[Sposób 68.]**, jak i „Obsługa zakładek i przycisków Wstecz za pomocą RSH" **[Sposób 69.]** powinny pomóc, rzucając światło na wspomniane kwestie, oraz dostarczyć odpowiednich rozwiązań).

Przedstawione w tej książce świetne podpowiedzi dotyczące technologii Ajax zmieniają zachowanie wielu znanych widgetów sieciowych, takich jak listy `select`, elementy `textarea`, pola tekstowe i przyciski opcji, które wysyłają swoje własne dane i w tle komunikują się z serwerem. Należy jednak pamiętać o tym, że widgety Ajax przede wszystkim powinny być *użyteczne* i zawsze trzeba unikać mylenia oraz irytowania użytkowników sieciowych.

Obiekt XMLHttpRequest

Punktem centralnym wielu sposobów opisanych w tej książce jest obiekt `XMLHttpRequest`, który pozwala kodowi JavaScript na pobranie z serwera pewnych danych, podczas gdy w tym samym czasie użytkownik korzysta z pozostałej części aplikacji. Wymieniony obiekt posiada własne API, które zostanie przedstawione w tym wprowadzeniu.

Podrozdział „Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania" **[Sposób 1.]** prezentuje ustalenie obiektu żądania w języku JavaScript. Kiedy obiekt zostanie już zainicjalizowany, wtedy będzie posiadał kilka metod i właściwości, które będzie można wykorzystać we własnych sposobach.

Praktyką często spotykaną w programowaniu jest wywołanie funkcji, które są przypisane do poszczególnych „metod" obiektów JavaScript. Metody obiektu `XMLHttpRequest` obejmują `open()`, `send()` i `abort()`.

Przedstawiona poniżej lista stanowi zbiór właściwości obsługiwanych przez obiekty żądań, zdefiniowane w większości głównych przeglądarek internetowych, takich jak Internet Explorer 5.0 i nowsze, Safari 1.3 i 2.0, Netscape 7 oraz najnowsze wydania Opery (na przykład 8.5). Obiekty żądań przeglądarki Mozilla Firefox posiadają dodatkowe właściwości i metody, które nie są współdzielone przez obiekty żądań innych głównych przeglądarek[2]. Obsługują one również następujące właściwości:

`onreadystatechange`
: Funkcja wywołania zwrotnego. Funkcja przypisana do tej właściwości jest wywoływana za każdym razem, gdy zmianie ulegnie właściwość `readyState`.

`readyState`
: Liczba. Wartość `0` oznacza *niezainicjalizowana*, funkcja `open()` nie została jeszcze wywołana. Wartość `1` oznacza *wczytywanie*, funkcja `send()` nie została jeszcze wywołana. Wartość `2` oznacza *wczytano*, funkcja `send()` została wywołana, dostępne są nagłówki/informacje o stanie. Wartość `3` oznacza *nieaktywna*, właściwość `responseText` przechowuje częściowe dane. Wartość `4` oznacza *ukończono*.

`responseText`
: Ciąg tekstowy, odpowiedź w formacie zwykłego tekstu.

`responseXML`
: Obiekt DOM `Document`, zwracana wartość w postaci XML.

`status`
: Zwraca kod stanu, na przykład `200` (wszystko w porządku) lub `404` (nie znaleziono).

`statusText`
: Ciąg tekstowy, tekst powiązany ze stanem odpowiedzi HTTP.

Obsługiwane metody obejmują:

`abort()`
: `void`. Przerywa żądanie HTTP.

`getAllResponseHeaders()`
: Ciąg tekstowy, zwraca wszystkie nagłówki odpowiedzi w preformatowanym ciągu tekstowym (warto zapoznać się z podrozdziałem „Zagłębienie się w odpowiedź HTTP" **[Sposób 9.]**).

[2] Obiekt `XMLHttpRequest` przeglądarki Mozilla Firefox posiada właściwości `onload`, `onprogress` i `onerror`, które są typu `Event.Listener`. Firefox dodatkowo definiuje metody `addEventListener()`, `dispatchEvent()`, `overrideMimeType()` i `removeEventListener()`. Więcej informacji dotyczących elementów obiektów żądań przeglądarki Firefox jest dostępnych pod adresem *http://www.xulplanet.com/references/objref/XMLHttpRequest.html*.

`getResponseHeader (string nagłówek)`
Ciąg tekstowy, zwraca wartość określonego nagłówka.

`open (string adres_URL, string asynch)`
`void`. Przygotowuje żądanie HTTP i określa, czy będzie, czy nie będzie asynchroniczne.

`send(string)`
`void`. Wysyła żądanie HTTP.

`setHeader(string nagłówek, string wartość)`
`void`. Wysyła żądanie nagłówka, ale w pierwszej kolejności musi zostać wywołana funkcja `open()`!

SPOSÓB 1.

Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania

Użycie języka JavaScript do ustawienia różnych obiektów żądania zarówno w przeglądarkach firmy Microsoft, jak i na bazie Mozilli

Zgodność przeglądarek jest bardzo istotną kwestią. Należy się upewnić o prawidłowej konstrukcji „silnika" obsługującego uzgodnienia Ajaksu z serwerem, chociaż nigdy nie można przewidzieć, jakie upodobania co do przeglądarek ma użytkownik aplikacji.

Narzędziem programistycznym, umożliwiającym aplikacjom Ajax przeprowadzanie żądań HTTP do serwera, jest obiekt, którego możemy użyć z poziomu kodu JavaScript. W świecie przeglądarek Firefox i Netscape (jak również Safari i Opera) ten obiekt nosi nazwę `XMLHttpRequest`. Jednak, kontynuując tradycję zapoczątkowaną przez IE 5.0, ostatnie wydania przeglądarki Internet Explorer implementują oprogramowania jako obiekt `ActiveX` o nazwie `Microsoft.XMLHTTP` lub `Msxml2.XMLHTTP`.

`Microsoft.XMLHTTP` i `Msxml2.XMLHTTP` odnoszą się do różnych wersji komponentów oprogramowania, które są częścią Microsoft XML Core Services (MSXML). Oto wyjaśnienie przedstawione przez naszego współautora, eksperta w dziedzinie IE:

„Jeżeli użyjemy `Microsoft.XMLHTTP`, wtedy `ActiveXObject` spróbuje zainicjalizować ostatnią dobrze znaną wersję obiektu, która posiada ten identyfikator ID programu. Teoretycznie, takim obiektem mógłby być MSXML 1.0, ale obecnie mało kto posiada tę wersję, ponieważ została zaktualizowana przez Windows Update, IE 6 lub w innych okolicznościach. MSXML w wersji 1.0 był obecny bardzo krótko. Jeśli jest używany obiekt `MSXML2.XMLHTTP`, wówczas oznacza to, że opakowanie używa co najmniej bibliotek MSXML 2.0. Większość programistów nie musi posługiwać się określoną wersją MSXML, na przykład `MSXML2.XMLHTTP.4.0` lub `MSXML2.XMLHTTP.5.0`".

Chociaż firma Microsoft i inżynierowie skupieni wokół projektu Mozilla wybrali inny sposób implementacji tego obiektu, to w niniejszej książce do obiektów `ActiveX` i `XMLHttpRequest` odnosimy się po prostu jako „obiektów żądania", ponieważ posiadają one bardzo podobne funkcje.

Pierwszym krokiem w użyciu technologii Ajax jest konieczność sprawdzenia, czy przeglądarka internetowa użytkownika obsługuje obiekty żądania na podstawie Mozilli lub powiązane z `ActiveX`, a następnie prawidłowe zainicjalizowanie obiektu.

Użycie funkcji do sprawdzania zgodności

Sprawdzanie zgodności opakowujemy funkcją JavaScript, a następnie wywołujemy tę funkcję przed przeprowadzeniem jakichkolwiek żądań HTTP za pomocą obiektu. Na przykład w przeglądarkach internetowych na bazie Mozilli, takich jak Netscape 7.1 i Firefox 1.5 (jak również w Safari 1.3 i 2.0 oraz Operze 8.5), obiekt żądania jest dostępny jako właściwość nadrzędnego obiektu `window`. Odniesieniem do tego obiektu w kodzie JavaScript jest `window.XMLHttpRequest`. Sprawdzenie zgodności dla tego typu przeglądarek może przedstawiać się następująco:

```
if(window.XMLHttpRequest){
   request = new XMLHttpRequest();
   request.onreadystatechange=handleResponse;
   request.open("GET",theURL,true);
   request.send(null);
}
```

Zmienna JavaScript `request` jest zmienną najwyższego poziomu, która będzie odnosiła się do obiektu żądania.

Alternatywny sposób — biblioteka *Prototype* typu open source używa zorientowanego obiektowo kodu JavaScript do opakowania obiektu żądania we własny obiekt, jako obiekt `Ajax.Request` (więcej informacji na ten temat znajduje się w rozdziale 6.).

Jeżeli przeglądarka obsługuje obiekt `XMLHttpRequest`, wówczas:

1. Wykonanie pętli `if(window.XMLHttpRequest)` zwraca wartość `true`, ponieważ obiekt `XMLHttpRequest` nie jest typu `null` lub `undefined`.
2. Za pomocą słowa kluczowego `new` zostanie ustanowiony egzemplarz obiektu.
3. Obserwator zdarzeń `onreadystatechange` obiektu (przedstawiony we wcześniejszym podrozdziale „Obiekt XMLHttpRequest") zostanie zdefiniowany jako funkcja o nazwie `handleResponse()`.
4. Kod wywołuje metody `open()` i `send()` obiektu żądania.

A co z użytkownikami przeglądarki Internet Explorer?

W trakcie publikowania tej książki blogi powiązane z Microsoft Internet Explorer informowały, że przeglądarka IE7 będzie obsługiwała obiekt `XMLHttpRequest`.

W takim przypadku, w modelu obiektowym przeglądarki obiekt `window.XMLHttpRequest` nie będzie istniał. Dlatego też w kodzie staje się niezbędny inny fragment pętli `if`:

```
else if (window.ActiveXObject){
   request=new ActiveXObject("Microsoft.XMLHTTP");
   if (! request){
      request=new ActiveXObject("Msxml2.XMLHTTP");
   }
   if(request){
      request.onreadystatechange=handleResponse;
      request.open(reqType,url,true);
      request.send(null);
   }
}
```

Przedstawiony fragment kodu sprawdza obecność najwyższego poziomu obiektu `window ActiveX`, co będzie wskazywało na użycie przeglądarki Internet Explorer. Następnie, za pomocą dwóch możliwych `ID` programu `ActiveX` (tutaj `Microsoft.XMLHTTP` i `Msxml2.XMLHTTP`), kod inicjalizuje żądanie.

Istnieje nawet możliwość jeszcze bardziej drobiazgowo sprawdzania różnych wersji obiektu żądania przeglądarki IE, na przykład `Msxml2.XMLHTTP4.0`. Jednakże w większości przypadków nie zachodzi potrzeba tworzenia aplikacji bazujących na różnych wersjach bibliotek MSXML, tak więc przedstawiony powyżej kod jest w zupełności wystarczający.

Kod wykonuje jedno końcowe sprawdzenie, aby przekonać się, czy obiekt żądania został prawidłowo skonstruowany (`if(request){...}`).

Dajemy trzy szanse, po których jeśli zmienna `request` wciąż jest typu `null` lub `undefined`, oznacza to, że używana przeglądarka nie radzi sobie z obsługą obiektu żądania Ajax!

Poniżej został przedstawiony kompletny kod sprawdzania zgodności:

```
/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
reqType: typ żądania HTTP, na przykład GET lub POST.
url: adres URL programu serwerowego.
asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */

function httpRequest(reqType,url,asynch){
   // Przeglądarki na bazie Mozilli.
   if(window.XMLHttpRequest){
      request = new XMLHttpRequest();
   } else if (window.ActiveXObject){
      request=new ActiveXObject("Msxml2.XMLHTTP");
      if (! request){
         request=new ActiveXObject("Microsoft.XMLHTTP");
      }
   }
   // Jeżeli nie powiodła się nawet inicjalizacja ActiveXObject,
   // wówczas żądanie wciąż może być typu null.
   if(request){
      initReq(reqType,url,asynch);
   } else {
      alert("Używana przeglądarka nie pozwala na wykorzystanie "+
         "wszystkich funkcji tej aplikacji!");
   }
}
```

```
/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
function initReq(reqType,url,bool){
   /* Określamy funkcję, która będzie obsługiwała odpowiedź HTTP. */
   request.onreadystatechange=handleResponse;
   request.open(reqType,url,bool);
   request.send(null);
}
```

Podrozdział „Użycie obiektu żądania do przekazania danych POST do serwera” **[Sposób 2.]** zaprezentuje, w jaki sposób należy zaimplementować żądania `POST` za pomocą obiektu `XMLHttpRequest`.

SPOSÓB 2.

Użycie obiektu żądania do przekazania danych POST do serwera

Krok wykraczający poza tradycyjny mechanizm przekazywania wartości z formularzy użytkownika

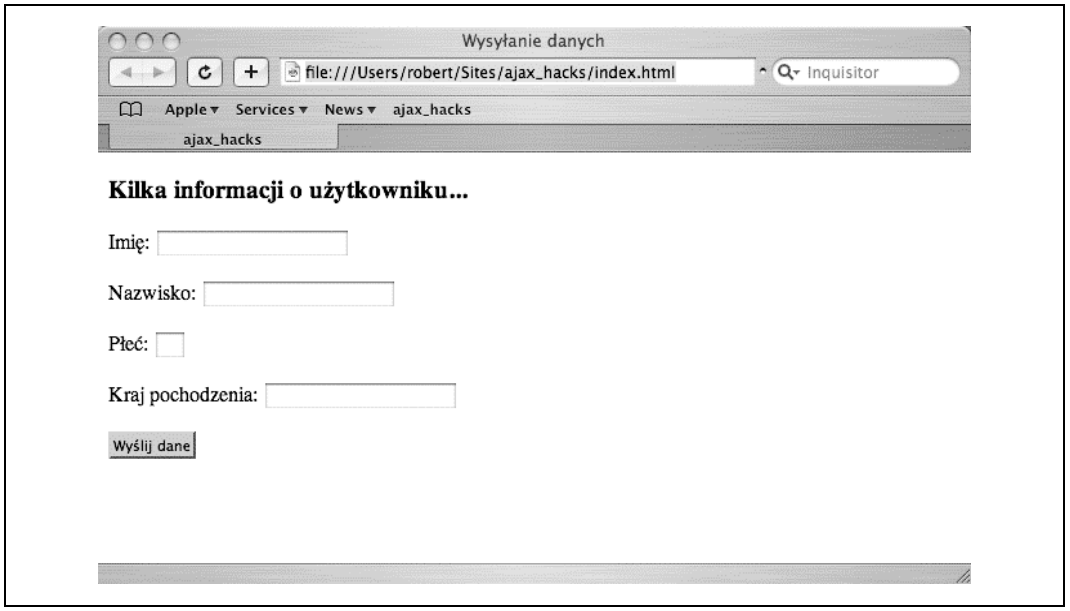
Przedstawiony w tym podrozdziale sposób używa do wysyłania danych i komunikacji z serwerem metody żądania HTTP `POST`, bez przeszkadzania użytkownikowi w korzystaniu z aplikacji. Następnie użytkownikowi zostaje przedstawiona odpowiedź otrzymana z serwera. Różnica między przedstawioną w tym sposobie a typową metodą wysyłania formularza polega na tym, że dzięki technologii Ajax strona nie jest zmieniana lub odświeżana, gdy aplikacja łączy się z serwerem w celu przekazania danych metodą `POST`. Dlatego też użytkownik może kontynuować pracę z aplikacją bez konieczności oczekiwania na przebudowę interfejsu w przeglądarce internetowej.

Możemy sobie wyobrazić, że posiadamy portal sieciowy, w którym kilka fragmentów strony oferuje użytkownikowi różne usługi. Jeżeli jeden z tych fragmentów zostaje zaangażowany w przekazywanie danych, wówczas cała aplikacja może żywiej reagować na działania użytkownika dzięki wykorzystaniu w tle żądań `POST`. W ten sposób cała strona (lub jej fragmenty) nie muszą zostać odświeżone w przeglądarce.

Przykładowa strona internetowa wykorzystująca przedstawiony sposób jest bardzo prosta. Na stronie został zawarty formularz, w którym użytkownik podaje imię i nazwisko, płeć oraz kraj pochodzenia, a następnie naciska przycisk, wysyłając tym samym dane za pomocą metody `POST`. Na rysunku 1.1 został pokazany wygląd omawianej strony w oknie przeglądarki internetowej.

Poniżej znajduje się kod HTML przedstawionej strony:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="/parkerriver/js/hack2.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Wysyłanie danych</title>
</head>
<body>
<h3>Kilka informacji o użytkowniku...</h3>
<form action="javascript:void%200" onsubmit="sendData();return false">
```



Rysunek 1.1. Przekazywanie informacji metodą POST

```
    <p>Imię: <input type="text" name="firstname" size="20"></p>
    <p>Nazwisko: <input type="text" name="lastname" size="20"> </p>
    <p>Płeć: <input type="text" name="gender" size="2"> </p>
    <p>Kraj pochodzenia: <input type="text" name="country" size="20"> </p>
    <p><button type="submit">Wyślij dane</button></p>
</form>
</body>
</html>
```

Przyglądając się powyższemu kodowi, może powstać pytanie dotyczące dziwnego wyglądu fragmentu `form action="javascript:void%200"`. Ponieważ w trakcie wysyłania formularza zostaje wywołana funkcja JavaScript, to atrybutowi `action` nie można podać nic więcej poza adresem URL, który nie posiada zwracanej wartości. Używamy więc zapisu, na przykład `"javascript:void 0"`. Powinniśmy zakodować spację między elementami `void` i `0`, a kod spacji to `%20`. Jeżeli używanie JavaScriptu jest zablokowane w przeglądarce internetowej użytkownika, wtedy kliknięcie przycisku wysyłającego dane formularza nie spowoduje żadnego efektu, ponieważ atrybut `action` nie wskazuje poprawnego adresu URL. Oprócz tego, jeśli użyjemy zapisu `action=""`, to pewne analizatory poprawności HTML wyświetlą komunikat ostrzeżenia. Innym sposobem napisania tego kodu jest włączenie funkcji wywołującej jako części obsługi zdarzenia `window.onload` w pliku JavaScript *.js*. Takie podejście zostało przyjęte w większości sposobów przedstawionych w niniejszej książce.

Pierwszym interesującym fragmentem kodu jest znacznik `script`, który importuje kod JavaScript (zawarty w pliku o nazwie *hack2.js*). Atrybut `onsubmit` znacznika `form` określa funkcję o nazwie `sendData()`, która z kolei formatuje dane dla żądania `POST` (wywołując

inną funkcję `setQueryString()`) i wysyłając dane do serwera. Ze względu na zachowanie zwięzłości pominęliśmy opis sprawdzania, czy występują puste pola (zagadnienie zostanie przedstawione w podrozdziale „Sprawdzanie poprawności pól tekstowych i elementów textarea pod kątem wystąpienia pustych pól" **[Sposób 22.]**), ale aplikacje sieciowe powinny wykonać ten krok przed wysłaniem danych do serwera.

Plik *hack2.js* definiuje niezbędny kod JavaScript. Poniżej została przedstawiona funkcja `setQueryString()`:

```
function setQueryString(){
   queryString="";
   var frm = document.forms[0];
   var numberElements = frm.elements.length;
   for(var i = 0; i < numberElements; i++) {
      if(i < numberElements-1) {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value)+"&";
      } else {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value);
      }
   }
}
```

Przedstawiona funkcja formatuje ciąg tekstowy stylu POST, zawierający wszystkie elementy `input` formularza. Wszystkie pary nazwa/wartość zostają oddzielone znakiem `&`, z wyjątkiem pary przedstawiającej ostatni element `input` formularza. Cały ciąg tekstowy może więc mieć postać:

```
firstname=Bruce&lastname=Perry&gender=M&country=USA
```

W tym momencie posiadamy ciąg tekstowy, który możemy użyć w żądaniu HTTP `POST`. Przyjrzyjmy się więc kodowi JavaScript, który odpowiada za wysłanie żądania. Wszystko rozpoczyna się od funkcji `sendData()`. Kod wywołuje tę funkcję w atrybucie `onsubmit` znacznika HTML `form`:

```
var request;
var queryString;    // Zmienna będzie przechowywała dane wysłane metodą POST.

function sendData(){
   setQueryString();
   var url="http://www.parkerriver.com/s/sender";
   httpRequest("POST",url,true);
}

/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
Parametry:
   reqType: typ żądania HTTP, na przykład GET lub POST.
   url: adres URL programu serwerowego.
   isAsynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */
function initReq(reqType,url,isAsynch){
   /* Określamy funkcję, która będzie obsługiwała odpowiedź HTTP. */
   request.onreadystatechange=handleResponse;
   request.open(reqType,url,isAsynch);
   /* Ustawiamy nagłówek Content-Type dla żądania POST */
```

```
    request.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded; charset=iso-8859-2");
    request.send(queryString);
}

/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
    reqType: typ żądania HTTP, na przykład GET lub POST.
    url: adres URL programu serwerowego.
    asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */

function httpRequest(reqType,url,asynch){
    // Przeglądarki na bazie Mozilli.
    if(window.XMLHttpRequest){
        request = new XMLHttpRequest();
    } else if (window.ActiveXObject){
        request=new ActiveXObject("Msxml2.XMLHTTP");
            if (! request){
                request=new ActiveXObject("Microsoft.XMLHTTP");
            }
    }
    // Jeżeli nie powiodła się nawet inicjalizacja ActiveXObject,
    // wówczas żądanie wciąż może być typu null.
    if(request){
        initReq(reqType,url,asynch);
    } else {
        alert("Używana przeglądarka nie pozwala na wykorzystanie "+
            "wszystkich funkcji tej aplikacji!");
    }
}
```

Celem funkcji `httpRequest()` jest sprawdzenie, czy obiekt żądania przeglądarki internetowej użytkownika został dołączony (warto sobie przypomnieć podrozdział „Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania" **[Sposób 1.]**). Następnie kod wywołuje funkcję `initReq()`, której parametry zostały opisane w komentarzu nad definicją funkcji.

Wiersz `request.onreadystatechange=handleResponse;` odpowiada za określenie funkcji obsługi zdarzeń, która będzie współpracowała z odpowiedzią. Tej funkcji przyjrzymy się nieco dokładniej w dalszej części podrozdziału. W kolejnym kroku kod wywołuje metodę `open()` obiektu żądania, która przygotowuje obiekt do wysłania żądania.

Ustawienie nagłówków

Po wywołaniu metody `open()` kod może ustawić wszystkie nagłówki żądania. W omawianym tutaj przypadku utworzyliśmy nagłówek `Content-Type`, przeznaczony dla żądania `POST`.

Przeglądarka Firefox wymaga dodatkowego nagłówka `Content-Type`, natomiast Safari 1.3 — już nie. (W trakcie tworzenia tego sposobu wykorzystywana była przeglądarka Firefox w wersji 1.02). Dodanie prawidłowych nagłówków jest dobrym pomysłem, ponieważ w większości przypadków serwer po prostu tego oczekuje od żądań typu `POST` serwer.

Oto kod dodający nagłówki oraz wysyłający żądanie typu `POST`:

```
request.setRequestHeader("Content-Type",
   "application/x-www-form-urlencoded; charset=iso-8859-2");
request.send(queryString);
```

Jeżeli jako parametr podamy pierwotną wartość `queryString`, wówczas wywołanie metody będzie się przedstawiało następująco:

```
send("firstname=Robert&lastname=Górczyński&gender=M&country=Polska ");
```

Przyglądanie się wynikowi

Kiedy aplikacja wyśle już dane typu `POST`, wtedy będziemy chcieli wyświetlić użytkownikom wyniki. To zadanie należy do funkcji `handleResponse()`. Warto przypomnieć sobie kod w funkcji `initReq()`:

```
request.onreadystatechange=handleResponse;
```

Gdy właściwość `readyState` obiektu żądania posiada wartość `4`, oznacza to, że operacje na obiekcie zostały zakończone, a kod stanu odpowiedzi HTTP posiada wartość `200`. Ta wartość wskazuje, że realizacja żądania HTTP zakończyła się powodzeniem. Następnie w oknie `alert` zostaje wyświetlony komunikat `responseText`. Jest on w pewnym stopniu rozczarowujący, ale naszym celem było zachowanie rozsądnej prostoty prezentowanego sposobu, ponieważ wiele innych przedstawionych sposobów będzie wykonywało bardziej złożone zadania za jego pomocą!

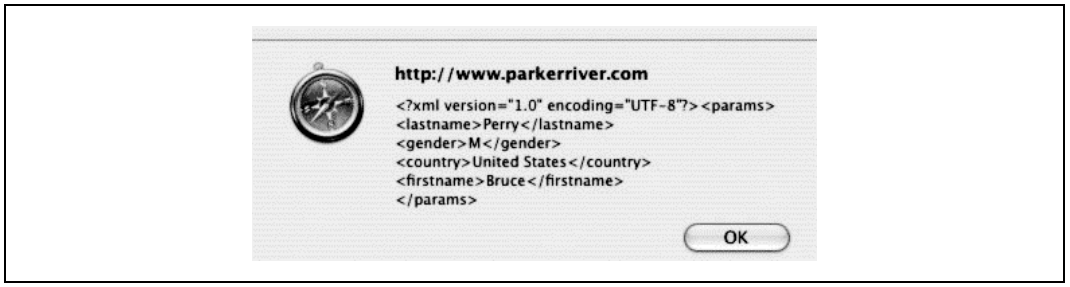
Poniżej znajduje się kod odpowiedzialny za komunikat:

```
// Obsługa zdarzeń dla XMLHttpRequest.
function handleResponse(){
   if(request.readyState == 4){
      if(request.status == 200){
         alert(request.responseText);
      } else {
         alert("Wystąpił problem z komunikacją między obiektem XMLHttpRequest,
a programem serwera.");
      }
   } // Koniec zewnętrznej pętli if.
}
```

Na rysunku 1.2 został pokazany wygląd okna `alert` po otrzymaniu odpowiedzi.

Komponent serwera zwraca dane przekazane metodą `POST` w postaci dokumentu XML. Każda nazwa parametru staje się nazwą elementu, podczas gdy wartość parametru zostaje zawartością elementu. Dane przekazane za pomocą metody `POST` są zagnieżdżone wewnątrz znaczników `params`. Komponent jest serwletem[3] Javy. Serwlet nie jest głównym tematem prezentowanego sposobu, ale mimo wszystko dla Czytelników, którzy są zainteresowani tym, co się dzieje na serwerze, przedstawiamy fragment kodu:

[3] Serwlet — aplet wykonywany na serwerze — *przyp. tłum.*



Rysunek 1.2. Uwaga! Serwer przemówił…

```
protected void doPost(HttpServletRequest httpServletRequest,
   HttpServletResponse httpServletResponse) throws
   ServletException, IOException {
   Map reqMap = httpServletRequest.getParameterMap();
   String val=null;
   String tag = null;
   StringBuffer body = new StringBuffer("<params>\n");
   boolean wellFormed = true;
   Map.Entry me = null;
   for(Iterator iter= reqMap.entrySet().iterator();iter.hasNext();) {
      me=(Map.Entry) iter.next();
      val= ((String[]me.getValue())[0];
      tag = (String) me.getKey();
      if (! XMLUtils.isWellFormedXMLName(tag)){
         wellFormed=false; break;
      }
      body.append("<").append(tag).append(">").
      append(XMLUtils.escapeBodyValue(val)).
      append("</").append(tag).append(">\n");
   }
   if(wellFormed) {
      body.append("</params>");
      sendXML(httpServletResponse,body.toString());
   } else {
      sendXML(httpServletResponse,"<notWellFormedParams />");
   }
}
```

Powyższy kod używa `XMLUtils`, klasy Javy z pakietu typu open source Jakarta Commons Betwixt, w celu sprawdzenia, czy nazwy parametrów są prawidłowo zbudowane. Klasa sprawdza również, czy wartości parametrów zawierają nieprawidłową z punktu widzenia XML zawartość. Jeżeli tak jest, wtedy dane wartości otrzymują odpowiednie sekwencje sterujące. Jeśli z jakiegokolwiek powodu przekazywany metodą `POST` komponent nie zawiera prawidłowo zbudowanych nazw parametrów (na przykład `na<>me` zamiast `name`), wówczas serwlet zwróci pusty element XML wskazujący taką okoliczność.

SPOSÓB 3.

Użycie własnej biblioteki z XMLHttpRequest

Wywołanie kodu, który inicjalizuje obiekt żądania i wysyłanie żądań do własnego pliku JavaScript

W celu łatwego rozstania się z obawami, dotyczącymi dużych aplikacji w technologii Ajax, utworzymy oddzielny plik zarządzający obiektem `XMLHttpRequest`, a następnie zaimportujemy ten plik do każdej strony internetowej, która będzie go wymagała. Na

samym końcu upewnimy się, że jakiekolwiek zmiany, niezależnie od tego, w jaki sposób kod ustawia obiekt żądania, będą przeprowadzane jedynie na tym pliku. Takie rozwiązanie jest przeciwieństwem sytuacji, w której każdy plik JavaScript używa żądań stylu Ajax.

Omawiany sposób przechowuje cały kod powiązany z obiektem w pliku o nazwie *http_request.js*. Następnie każda strona internetowa, która używa obiektu `XMLHttpRequest`, może zaimportować ten plik w następujący sposób:

```
<script type="text/javascript" src="js/http_request.js"></script>
```

Poniżej został przedstawiony kod wspomnianego pliku, łącznie z wszystkimi komentarzami:

```
var request = null;
/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
    reqType: typ żądania HTTP, na przykład GET lub POST.
    url: adres URL programu serwerowego.
    asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie.
    respHandle: nazwa funkcji, która będzie obsługiwała odpowiedź.
    Każde pięć parametrów przedstawione przez arguments[4] stanowią dane
    żądania POST przeznaczonego do wysłania. */
function httpRequest(reqType,url,asynch,respHandle){
    // Przeglądarki na bazie Mozilli.
    if(window.XMLHttpRequest){
        request = new XMLHttpRequest();
    } else if (window.ActiveXObject){
        request=new ActiveXObject("Msxml2.XMLHTTP");
            if (! request){
                request=new ActiveXObject("Microsoft.XMLHTTP");
            }
    }
    // Bardzo mało prawdopodobne, ale sprawdzamy, czy występują żądania null,
    // jeśli także obiekt ActiveXObject nie został zainicjalizowany.
    if(request) {
        // Jeżeli parametr reqType jest typu POST, wówczas
        // piąty argument funkcji stanowią dane przesyłane metodą POST.
        if(reqType.toLowerCase() != "post") {
            initReq(reqType,url,asynch,respHandle);
        } else {
            // Dane przekazywane metodą POST.
            var args = arguments[4];
            if(args != null && args.length > 0){
                initReq(reqType,url,asynch,respHandle,args);
            }
        }
    } else {
        alert("Używana przeglądarka nie pozwala na wykorzystanie "+
            "wszystkich funkcji tej aplikacji!");
    }
}

/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
function initReq(reqType,url,bool,respHandle){
    try{
        /* Określamy funkcję, która będzie obsługiwała odpowiedź HTTP */
        request.onreadystatechange=respHandle;
        request.open(reqType,url,bool);
        // Jeżeli parametr reqType jest typu POST, wówczas
        // piąty argument funkcji stanowią dane przesyłane metodą POST.
```

```
        if(reqType.toLowerCase() == "post") {
           request.setRequestHeader("Content-Type",
              "application/x-www-form-urlencoded; charset=iso-8859-2");
           request.send(arguments[4]);
        } else {
           request.send(null);
        }
    } catch (errv) {
       alert(
          "W tym momemcie aplikacja "+
          "nie może połączyć się z serwerem. "+
          "Proszę spróbować ponownie w ciągu kilku sekund.\n"+
          "Szczegółowe informacje o błędzie: "+errv.message);
       }
    }
```

Aplikacja używająca tego kodu wywoła funkcję `httpRequest()` z czterema lub pięcioma (w przypadku żądań `POST`) parametrami. W innych sposobach przedstawionych w tej książce zobaczymy wiele przykładów wywoływania funkcji `httpRequest()`. Poniżej znajduje się kolejny:

```
var _url = "http://www.parkerriver.com/s/sender";
var _data="first=Bruce&last=Perry&middle=D";
httpRequest("POST",_url,true,handleResponse,_data);
```

Komentarze w kodzie opisują znaczenie każdego z tych parametrów. Ostatni parametr przedstawia dane, które towarzyszą żądaniom `POST`.

Żądania HTTP `POST` umieszczają przekazywane dane za informacjami o nagłówkach żądania. Z drugiej strony, żądania `GET` dołączają do adresu URL nazwy parametrów i ich wartości.

Jeżeli kod nie używa metody `POST`, wtedy kod klienta będzie wykorzystywał jedynie cztery parametry. Czwartym parametrem może być albo nazwa funkcji, która jest zadeklarowana w kodzie klienta (na przykład funkcja obsługująca odpowiedź, umieszczona poza plikiem *http_request.js*), albo dosłowna funkcja. Ta druga możliwość obejmuje zdefiniowanie funkcji wewnątrz wywołania funkcji, co często jest kłopotliwe i trudne do odczytu. Jednakże takie rozwiązanie jest sensowne w sytuacjach, w których obsługa odpowiedzi HTTP jest zwięzła i prosta, na przykład:

```
var _url = "http://www.parkerriver.com/s/sender";
// Ustawienie debugowania.
httpRequest("POST",_url,true,function(){alert(request.responseText);});
```

Funkcja `httpRequest()` inicjuje ten sam proces sprawdzania i ustawiania obiektu `XMLHttpRequest` dla przeglądarki Internet Explorer i przeglądarek spoza firmy Microsoft, który został opisany w podrozdziale „Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania” **[Sposób 1.]**. Natomiast funkcja `initReq()` obsługuje drugi krok ustawienia obiektu żądania: określenie obsługi zdarzeń `onreadystatechange` oraz wywołanie metod `open()` i `send()` przeprowadzających żądanie HTTP. Za pomocą bloku instrukcji `try/catch` kod przechwytuje wszystkie błędy lub wyjątki zgłoszone przez te

metody żądania. Przykładowo, jeżeli kod wywoła metodę `open()` wraz z adresem URL wskazującym na inny serwer niż użyty do pobrania w załączonej stronie internetowej, wówczas blok `try/catch` przechwyci błąd i wyświetli okno `alert`.

Na koniec warto dodać, że dopóki strona internetowa będzie importowała plik *http_request.js*, dopóty zmienna `request` będzie dostępna w zewnętrznym kodzie względem zaimportowanego pliku. W rezultacie zmienna `request` staje się zmienną globalną.

W ten sposób, `request` staje nazwą zastrzeżoną jako nazwa zmiennej, ponieważ zmienne lokalne, które wykorzystują słowo kluczowe `var`, będą unieważniać (z niezamierzonymi konsekwencjami) globalnie użytą nazwę `request`, jak w poniższym przykładzie:

```
function handleResponse(){
   // Unieważnienie zaimportowanej zmiennej request.
   var request = null;
   try{
      if(request.readyState == 4){
         if(request.status == 200){...
```

SPOSÓB 4.

Otrzymywanie danych w postaci XML

Ajax i programy serwerowe dostarczają obiekt DOM Document, który jest gotowy do użycia

W chwili obecnej wiele technologii do wymiany danych wykorzystuje format Extensible Markup Language, głównie dlatego, że XML jest standaryzowanym i rozszerzalnym formatem, powszechnie obsługiwanym w świecie oprogramowania. Z tego powodu różne firmy używają istniejących, dobrze znanych technologii do generowania, wysyłania i otrzymywania danych XML. Odbywa się to bez konieczności przystosowania narzędzi programowych używanych przez różne firmy, z którymi następuje wymiana danych XML.

Przykładem może być urządzenie Global Positioning System (GPS), które może dzielić posiadane dane, przykładowo, z pieszą lub rowerową wycieczką wyposażoną w aplikację sieciową mającą możliwość określenia położenia. Wystarczy jedynie podłączyć do komputera przewód USB dostarczony razem z urządzeniem GPS i uruchomić oprogramowanie, które wysyła i otrzymuje dane z Sieci, i to wszystko. Zwykle formatem danych jest język XML, który został już zdefiniowany w oprogramowaniu GPS. Aplikacja sieciowa i urządzenie GPS „porozumiewają się więc w tym samym języku".

Chociaż w tej książce nie ma miejsca na obszerne wprowadzenie do XML, to prawdopodobnie Czytelnikowi udało się zobaczyć pliki tekstowe tego formatu w jednym z formularzy. XML jest używany jako język „meta", który opisuje i kategoryzuje określone typy informacji. Dane XML rozpoczynają się od opcjonalnej deklaracji XML (na przykład `<?xml version="1.0" encoding="iso-8859-2"?>`), po której następuje element nadrzędny oraz zero bądź więcej elementów potomnych. Przykładem może być następujący kod:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<gps>
<gpsMaker>Garmin</gpsMaker>
```

```
<gpsDevice>
Forerunner 301
</gpsDevice>
</gps>
```

W powyższym kodzie `gps` jest elementem nadrzędnym, natomiast `gpsMaker` i `gpsDevice` są elementami potomnymi.

Ajax i obiekt żądania mogą otrzymywać dane w postaci XML, który jest bardzo użyteczny w trakcie obsługi odpowiedzi usług sieciowych wykorzystujących XML. Kiedy żądanie HTTP zostanie ukończone, wówczas obiekt żądania będzie posiadał prawidłowo nazwany obiekt `responseXML`. Ten obiekt jest obiektem DOM `Document`, który może zostać użyty przez aplikację Ajax. Oto przykład:

```
function handleResponse(){
   if(request.readyState == 4){
      if(request.status == 200){
         var doc = request.responseXML;
...
}
```

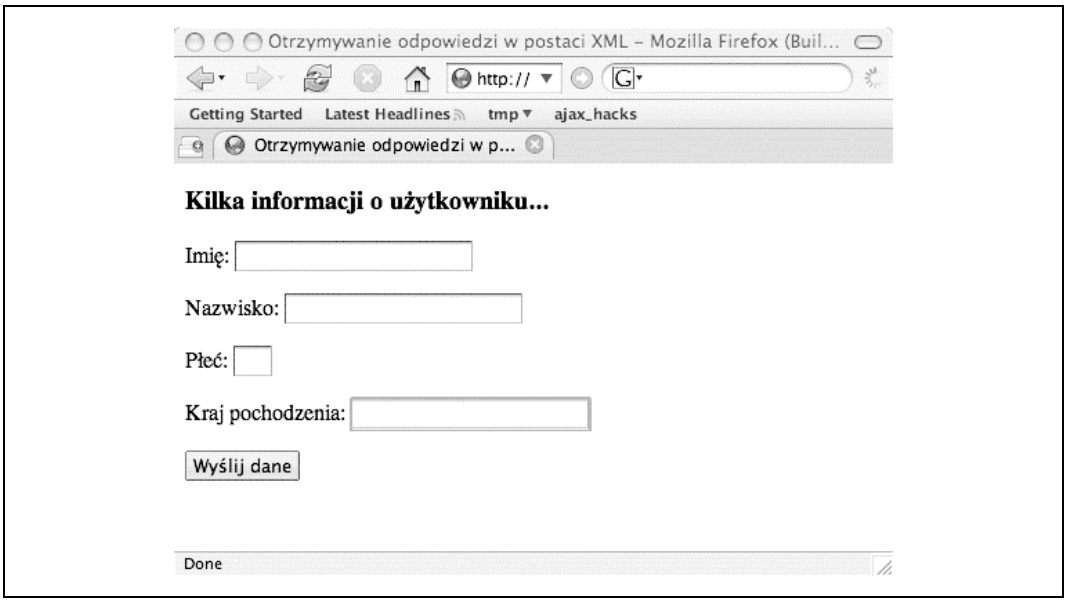
W poprzednim fragmencie kodu zmienna `doc` jest obiektem DOM `Document` i oferuje podobne API do okna przeglądarki wyświetlającej stronę. Omawiany sposób otrzymuje XML z serwera, a następnie angażuje odrobinę programowania DOM z obiektem `Document` w celu wydobycia z XML pewnych informacji.

Jeżeli zachodzi potrzeba uzyskania czystych danych tekstowych XML, wówczas należy użyć właściwości `request.responseText`.

Plik HTML wykorzystywany w tym sposobie jest zasadniczo taki sam, jak plik użyty w podrozdziale „Użycie obiektu żądania do przekazania danych POST do serwera" **[Sposób 2.]**. Na końcu pliku został jednak dodany element `div`, w którym kod wyświetla informacje dotyczące zwróconego XML. Poniżej znajduje się kompletny kod HTML strony:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack3.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Otrzymywanie odpowiedzi w postaci XML</title>
</head>
<body>
<h3>Kilka informacji o użytkowniku...</h3>
<form action="javascript:void%200" onsubmit="sendData();return false">
   <p>Imię: <input type="text" name="firstname" size="20"></p>
   <p>Nazwisko: <input type="text" name="lastname" size="20"> </p>
   <p>Płeć: <input type="text" name="gender" size="2"> </p>
   <p>Kraj pochodzenia: <input type="text" name="country" size="20"> </p>
   <p><button type="submit">Wyślij dane</button></p>
   <div id="docDisplay"></div>
</form>
</body>
</html>
```

Na rysunku 1.3 został pokazany wygląd omawianej strony przed podaniem jakichkolwiek informacji przez użytkownika.



Rysunek 1.3. Wszystko zostało przygotowane do otrzymania danych XML

Kod JavaScript w pliku *hack3.js* przekazuje za pomocą metody `POST` swoje dane do aplikacji serwera, która z kolei odsyła odpowiedź w formacie XML. Krok polegający na sprawdzaniu poprawności pól **[Sposób 22.]** został pominięty ze względu na chęć utrzymania zwięzłości kodu, ale aplikacje sieciowe używające formularzy zawsze powinny implementować tego typu zadanie.

Podobnie jak w przypadku innych przykładów przedstawionych w tym rozdziale, program serwerowy odsyła z powrotem do klienta nazwy parametrów i ich wartości jako `<params><firstname>Bruce</firstname></params>`. W podrozdziale „Użycie obiektu żądania do przekazania danych POST do serwera" **[Sposób 2.]** został zaprezentowany pewien fragment kodu komponentu serwera, który umieszcza razem zwracane wartości. Wspomniana technika doskonale odpowiada naszemu celowi pokazania prostego przykładu programowania XML w aplikacji Ajax:

```
var request;
var queryString; // Zmienna będzie przechowywała dane wysłane metodą POST.

function sendData(){
   setQueryString();
   var url="http://parkerriver/s/sender";
   httpRequest("POST",url,true);
}

// Obsługa zdarzeń dla obiektu XMLHttpRequest.
function handleResponse(){
   if(request.readyState == 4){
```

```
      if(request.status == 200){
         var doc = request.responseXML;
         var info = getDocInfo(doc);
         stylizeDiv(info,document.getElementById(""docDisplay""));
      } else {
         alert(""Wystąpił problem z komunikacją między obiektem
         XMLHttpRequest, ""+
         ""a programem serwera."");
      }
   }// Koniec zewnętrznej pętli if.
}

/* Inicjalizacja obiektu żądania, który został już skonstruowany */
function initReq(reqType,url,bool){
   /* Określamy funkcję, która będzie obsługiwała odpowiedź HTTP */
   request.onreadystatechange=handleResponse;
   request.open(reqType,url,bool);
   request.setRequestHeader(""Content-Type"",
      ""application/x-www-form-urlencoded; charset=iso-8859-2"");
   /* Funkcjonuje jedynie w przeglądarkach na bazie Mozilli. */
   //request.overrideMimeType(""text/XML"");
   request.send(queryString);
}

/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
   reqType: typ żądania HTTP, na przykład GET lub POST.
   url: adres URL programu serwerowego.
   asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */
function httpRequest(reqType,url,asynch){
   // Skrócone... Zobacz [Sposób 01].
}
function setQueryString(){
   queryString="";
   var frm = document.forms[0];
   var numberElements = frm.elements.length;
   for(var i = 0; i < numberElements; i++) {
      if(i < numberElements-1) {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value)+"&";
      } else {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value);
      }
   }
}
/* Dynamiczne dostarczenie zawartości elementów div. Jeżeli chcemy wzbogacić
 elementy div, to do tej funkcji możemy dołączyć informacje o stylu. */
function stylizeDiv(bdyTxt,div){
   // Pozostała zawartość elementu DIV.
   div.innerHTML="";
   div.style.backgroundColor="yellow";
   div.innerHTML=bdyTxt;
}

/* Pobranie informacji o dokumencie XML za pomocą obiektu DOM Document. */
function getDocInfo(doc){
   var root = doc.documentElement;
   var info = "<h3>Nazwa elementu nadrzędnego dokumentu: </h3>"+
root.nodeName;
   var nds;
```

```
    if(root.hasChildNodes()) {
       nds=root.childNodes;
       info+= "<h4>Nazwa/wartość węzła potomnego elementu nadrzędnego:</h4>";
       for (var i = 0; i < nds.length; i++){
          info+= nds[i].nodeName;
          if(nds[i].hasChildNodes()){
             info+= " : \""+nds[i].firstChild.nodeValue+"\"<br />";
          } else {
             info+= " : Pusty<br />";
          }
       }
    }
    return info;
}
```

Mozilla Firefox może używać funkcji `request.overrideType()` do wymuszenia interpretacji odpowiedzi jako potoku określonego typu `mime`, na przykład `request.overrideType("text/xml")`. Obiekt żądania przeglądarki Internet Explorer nie posiada takiej funkcji. Wywołanie tej funkcji nie działa również w przeglądarce Safari 1.3.

Po wysłaniu przez kod danych za pomocą metody `POST` i otrzymaniu odpowiedzi następuje wywołanie metody o nazwie `getDocInfo()`, tworzącej łańcuch tekstowy, który wyświetli pewne informacje dotyczące dokumentu XML oraz jego podelementów lub elementów potomnych:

```
var doc = request.responseXML;
var info = getDocInfo(doc);
```

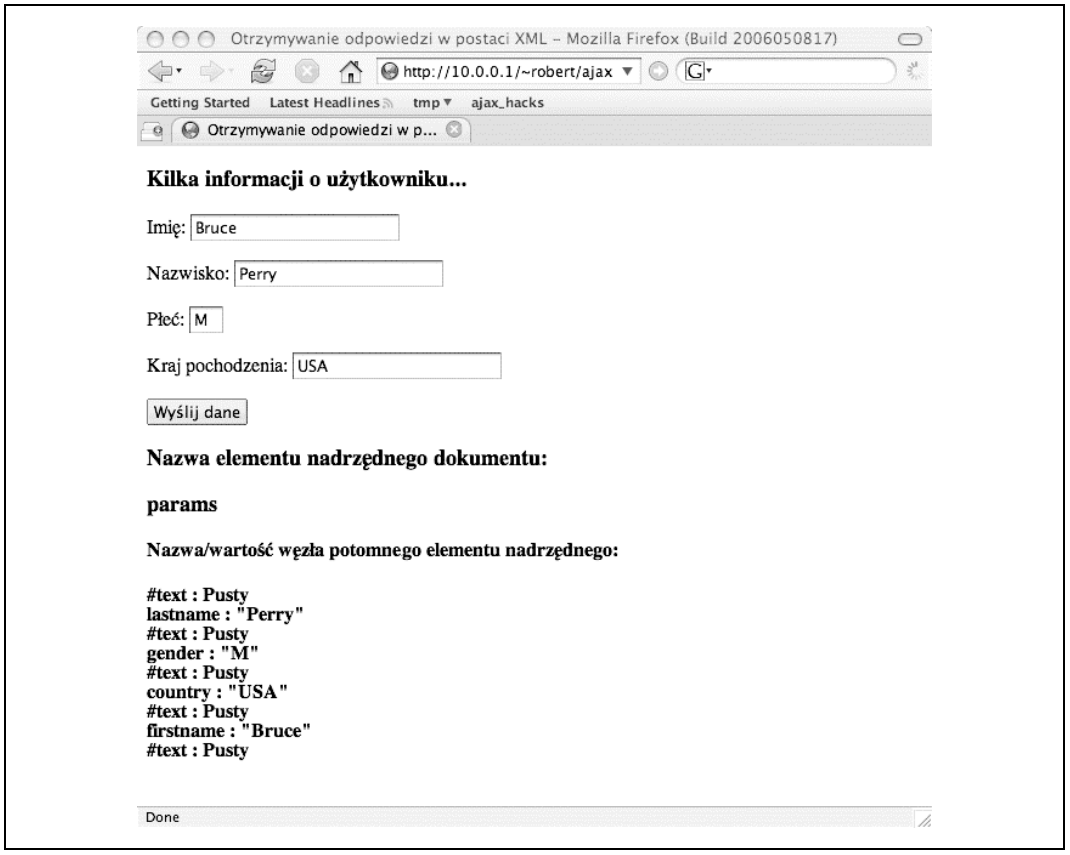
Funkcja `getDocInfo()` pobiera odniesienie do elementu nadrzędnego XML (`var root = doc.documentElement;`), a następnie tworzy łańcuch tekstowy zawierający nazwę elementu nadrzędnego oraz informacje o jego wszystkich węzłach lub elementach potomnych, takie jak nazwę węzła potomnego i jego wartość. Kolejnym krokiem jest przekazanie metodzie `stylizeDiv()` wymienionych wcześniej informacji. Metoda `stylizeDiv()` w celu dynamicznego wyświetlenia zebranych danych używa elementu `div` na końcu strony HTML:

```
function stylizeDiv(bdyTxt,div){
   // Pozostała zawartość elementu DIV.
   div.innerHTML="";
   div.style.backgroundColor="yellow";
   div.innerHTML=bdyTxt;
}
```

Na rysunku 1.4 został pokazany wygląd strony internetowej po tym, gdy aplikacja otrzyma odpowiedź XML.

Kolejne węzły pokazywane przez aplikację są to znaki nowego wiersza w zwróconej odpowiedzi XML.

Sedno API DOM, oferowane przez implementację JavaScript w przeglądarce, dostarcza programistom narzędzia o dużych możliwościach, służącego do programowania złożonych wartości zwrotnych XML.



Rysunek 1.4. Zagłębianie się w wartości zwrotne XML

SPOSÓB 5.

Pobieranie zwykłych starych ciągów tekstowych

Zarządzanie odczytami prognozy pogody, kursami akcji, informacjami wydobywanymi ze strony internetowej lub podobnymi danymi niebędącymi danymi XML w formacie zwykłych starych ciągów tekstowych

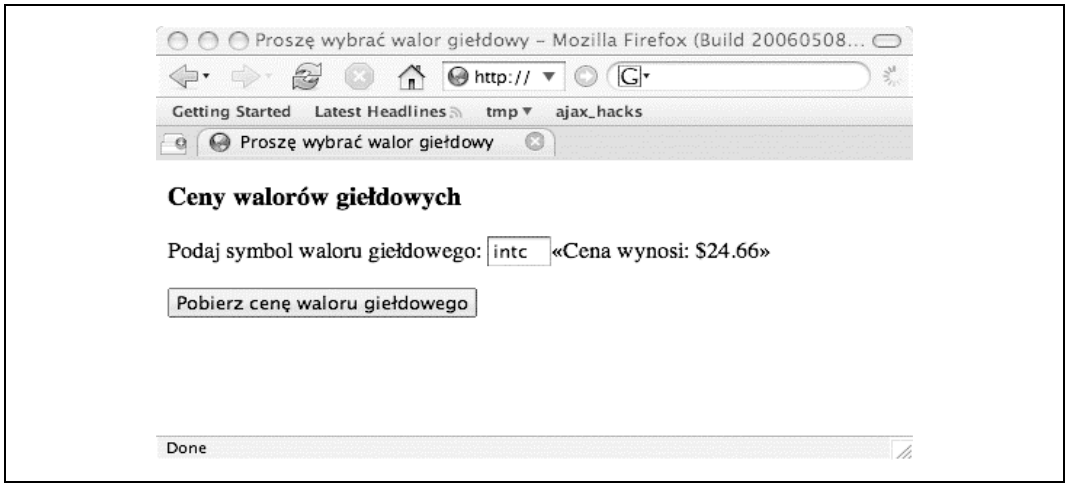
Obiekt żądania posiada doskonałą właściwość dla aplikacji sieciowych, która nie musi obsługiwać wartości zwrotnych serwera jako XML: `request.responseText`. Przedstawiony sposób prosi użytkownika o wybór symbolu giełdowego, a następnie serwer zwraca w celu wyświetlenia cenę wybranego waloru giełdowego. Kod obsługuje zwracaną wartość jako ciąg tekstowy (`string`).

Odmiana tego programu, która zostanie przedstawiona w kolejnym sposobie, wymaga, aby ceny walorów giełdowych były obsługiwane jako liczby. Są to stare ceny, które komponent serwera przechowuje dla określonych symbolów giełdowych, a nie notowanie bieżące pobierane z komercyjnej usługi sieciowej lub wydobywane ze strony internetowej. Przykład takiego mechanizmu został przedstawiony w podrozdziale „Użycie XMLHttpRequest do wydobycia cen energii ze strony internetowej” **[Sposób 39.]**.

W pierwszej kolejności przedstawiamy kod HTML omawianej strony internetowej. Kod JavaScript jest importowany z pliku o nazwie *hack9.js*:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack9.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Proszę wybrać walor giełdowy</title>
</head>
<body>
<h3>Ceny walorów giełdowych</h3>
<form action="javascript:void%200" onsubmit=
   "getStockPrice(this.stSymbol.value);return false">
   <p>Podaj symbol waloru giełdowego: <input type="text" name=
      "stSymbol" size="4"><span id="stPrice"></span></p>
   <p><button type="submit">Pobierz cenę waloru giełdowego</button></p>
</form>
</body>
</html>
```

Wyświetlona w przeglądarce Mozilla Firefox strona internetowa została pokazana na rysunku 1.5. Użytkownik podaje symbol, na przykład „GRMN" (wielkość liter nie ma znaczenia), a następnie klika przycisk *Pobierz cenę waloru giełdowego*. Po naciśnięciu przycisku kod JavaScript powoduje pobranie odpowiedniej ceny waloru giełdowego i wyświetlenie jej po prawej stronie pola tekstowego, wewnątrz elementu `span`.



Rysunek 1.5. Natychmiastowe wyświetlenie ceny waloru giełdowego

Funkcją zajmującą się przetworzeniem żądania jest `getStockPrice()`. Pobiera ona wartość pola tekstowego o nazwie `stSymbol`, a zwraca odpowiednią cenę waloru giełdowego (używa obiektu żądania do komunikacji z komponentem serwera, który pobiera właściwą cenę waloru giełdowego). Poniżej znajduje się kod JavaScript:

```
var request;
var symbol; // Zmienna będzie przechowywać symbol waloru giełdowego.

function getStockPrice(sym){
   symbol=sym;
   if(sym) {
      var url="http://localhost:8080/parkerriver/s/stocks?symbol="+sym;
      httpRequest("GET",url,true);
   }
}

// Obsługa zdarzeń dla obiektu XMLHttpRequest.
function handleResponse(){
   if(request.readyState == 4){
      if(request.status == 200){
         /* Przechwyć wyniki jako ciąg tekstowy */
         var stockPrice = request.responseText;
         var info = "&#171;Cena wynosi: $"+stockPrice+"&#187;";
         document.getElementById("stPrice").style.fontSize="0.9em";
         document.getElementById("stPrice").style.backgroundColor="yellow";
         document.getElementById("stPrice").innerHTML=info;

      } else {
         alert("Wystąpił problem z komunikacją między obiektem XMLHttpRequest,
"+
            "a programem serwera.");
      }
   }// Koniec zewnętrznej pętli if.
}

/* Wróć do sposobu 01, aby zobaczyć kod httpRequest(),
który został tutaj pominięty, aby zapewnić zwięzłość kodu */
```

Funkcja `getStockPrice()` opakowuje wywołanie funkcji `httpRequest()` odpowiedzialnej za ustawienie obiektu żądania. Jeżeli Czytelnik zapoznał się już z innymi sposobami przedstawionymi w tym rozdziale, to rozpozna funkcję `handleResponse()` załączającą znacznie bardziej interesujące działania.

W podrozdziałach „Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania" **[Sposób 1.]** i „Użycie własnej biblioteki z XMLHttpRequest" **[Sposób 3.]** funkcja `httpRequest()` została bardziej szczegółowo omówiona.

Jeżeli żądanie jest kompletne (na przykład `request.readyState` posiada wartość `4`) i stan odpowiedzi HTTP wynosi `200` (co oznacza, że wykonanie żądania zakończyło się powodzeniem), wówczas kod przechwytuje odpowiedź serwera jako wartość właściwości `request.responseText`. Następnie kod wykonuje pracę za pomocą skryptów DOM, wyświetlając cenę waloru giełdowego wraz z kilkoma atrybutami powiązanymi z CSS:

```
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").style.backgroundColor="yellow";
document.getElementById("stPrice").innerHTML=info;
```

Atrybuty `style` powodują, że czcionka staje się odrobinę mniejsza od czcionki preferowanej przez przeglądarkę internetową użytkownika, a kolor tła dla wyświetlanego tekstu zostaje ustalony jako żółty. Właściwość `innerHTML` elementu `span` jest ustawiona dla ceny waloru giełdowego za pomocą podwójnego nawiasu ostrego.

SPOSÓB 6.

Otrzymywanie danych w postaci liczb

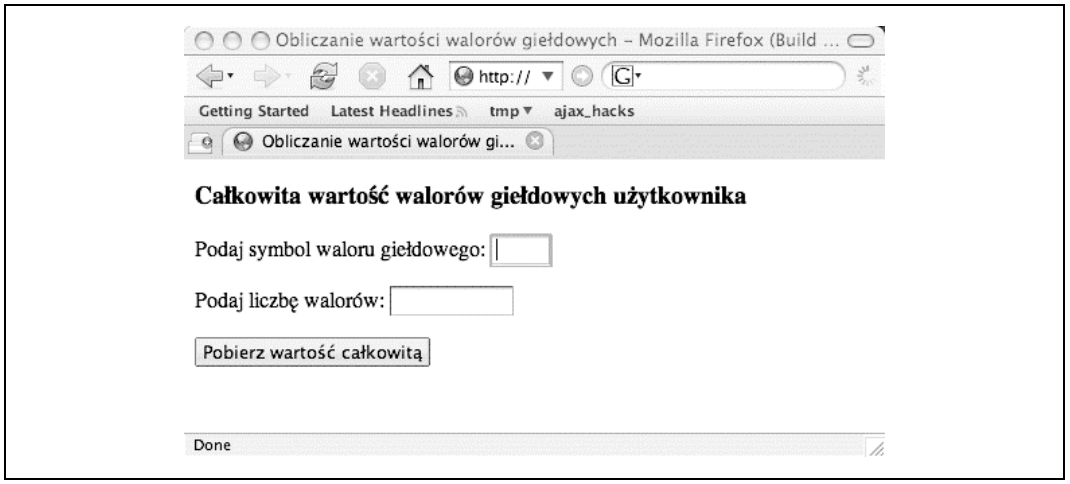
Wykonywanie operacji, które polegają na wartościach zwrotnych obiektu żądania w postaci liczb

W tym sposobie notowanie waloru giełdowego będzie otrzymywane w postaci liczby, a następnie zostanie dynamicznie wyświetlona całkowita wartość posiadanych przez użytkownika akcji na podstawie podanej liczby tych akcji. Jeżeli serwer nie wyśle poprawnej liczby, wtedy aplikacja wyświetli użytkownikowi odpowiedni komunikat błędu.

Dużą zaletą technologii Ajax jest otrzymywanie pojedynczej wartości z serwera zamiast całej strony internetowej. Czasami te pojedyncze informacje są liczbami zamiast ciągami tekstowymi (które zostały omówione w poprzednim sposobie) lub pewnymi innymi obiektami. Zazwyczaj język JavaScript jest wystarczająco sprytny, aby dokonać konwersji wartości na typ liczbowy bez konieczności interwencji programisty. Jednak wciąż pozostaje otwarta kwestia, że nie chcemy, aby aplikacja sieciowa pomnożyła posiadaną przez użytkownika liczbę walorów giełdowych przez *niezdefiniowane* lub inne dane zwrócone przez serwer!

Przedstawiony tutaj sposób upewnia się, że użytkownik podał prawidłową liczbę dla wartości pola „liczba walorów giełdowych". Kod sprawdza również, czy zwrócona przez serwer wartość jest poprawna liczbowo. Następnie w przeglądarce internetowej użytkownika zostaje dynamicznie wyświetlona cena waloru giełdowego oraz całkowita wartość posiadanych przez użytkownika udziałów.

Wygląd formularza sieciowego został pokazany na rysunku 1.6.



Rysunek 1.6. Odkrycie całkowitej wartości posiadanych walorów giełdowych

Przedstawiony poniżej kod stanowi kod HTML strony internetowej:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="/parkerriver/js/hack4.js">
```

```
   </script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Obliczanie wartości walorów giełdowych</title>
</head>
<body>
<h3>Całkowita wartość walorów giełdowych użytkownika</h3>
<form action="javascript:void%200" onsubmit=
   "getStockPrice(this.stSymbol.value,this.numShares.value);return false">
<p>Podaj symbol waloru giełdowego: <input type="text" name="stSymbol"
size="4">
   <span id="stPrice"></span></p>
<p>Podaj liczbę walorów: <input type="text" name="numShares" size="10"></p>
<p><button type="submit">Pobierz wartość całkowitą</button></p>
<div id="msgDisplay"></div>
</form>
</body>
</html>
```

Kiedy użytkownik kliknie przycisk *Pobierz wartość całkowitą*, wtedy ten krok spowoduje wywołanie zdarzenia `onsubmit` elementu `form`. Obsługą zdarzeń dla tego zdarzenia jest funkcja `getStockPrice()`, która jako swoje dwa argumenty pobiera symbol waloru giełdowego oraz liczbę walorów. Fragment kodu `return false` obsługi zdarzeń powoduje *anulowanie* typowego wysyłania wartości formularza przez przeglądarkę do adresu URL wskazanego przez atrybut `action` znacznika `form`.

Obliczenia liczbowe

Przyjrzyjmy się kodowi JavaScript, który w HTML-u jest importowany jako część pliku *hack4.js*:

```
var request;
var symbol;    // Zmienna będzie przechowywać symbol waloru giełdowego.
var numberOfShares;

function getStockPrice(sym,shs){
   if(sym && shs){
      symbol=sym;
      numberOfShares=shs;
      var url="http://localhost:8080/parkerriver/s/stocks?symbol="+sym;
      httpRequest("GET",url,true);
   }
}

// Obsługa zdarzeń dla obiektu XMLHttpRequest.
function handleResponse(){
   if(request.readyState == 4){
      alert(request.status);
      if(request.status == 200){
         /* Sprawdzenie, czy wartość zwrotna rzeczywiście jest liczbą.
      Jeżeli tak jest, to następuje pomnożenie przez liczbę walorów giełdowych
      i wyświetlenie wyniku. */
         var stockPrice = request.responseText;
         try{
            if(isNaN(stockPrice)) { throw new Error(
               "Zwrócona cena nie jest poprawną liczbą.");}
            if(isNaN(numberOfShares)) { throw new Error(
               "Liczba walorów nie jest poprawną liczbą.");}
            var info = "Całkowita wartość walorów: "+ calcTotal(stockPrice);
```

```
                displayMsg(document.getElementById("msgDisplay"),info,"black");
                document.getElementById("stPrice").style.fontSize="0.9em";
                document.getElementById("stPrice").innerHTML ="cena: "+stockPrice;
             } catch (err) {
                displayMsg(document.getElementById("msgDisplay"),
                   "Wystąpił błąd: "+
                   err.message,"red");
             }
          } else {
             alert(
                "Wystąpił problem z komunikacją między obiektem XMLHttpRequest, "+
                "a programem serwera.");
          }
       }// Koniec zewnętrznej pętli if.
}

/* Zobacz sposoby 01 i 02, aby przypomnieć sobie przykłady kodu powiązanego
z funkcją initReq(). Zostały one tutaj pominięte w celu zachowania zwięzłości kodu. */

function calcTotal(price){
   return stripExtraNumbers(numberOfShares * price);
}

/* Usuwamy wszystkie znaki wykraczające poza zakres czterech znaków po przecinku,
jak ma to miejsce na przykład w 12,3454678. */
function stripExtraNumbers(num) {
   // Sprawdzenie, czy liczby są poprawne,
   // zakładamy, że cała liczba jest poprawna.
   var n2 = num.toString();
   if(n2.indexOf(".") == -1)  { return num; }
   // Jeżeli liczba posiada cyfry po przecinku dziesiętnym,
   // wówczas zmniejszamy liczbę tych cyfr do czterech.
   // Używamy parseFloat, jeżeli metodzie zostały przekazane ciągi tekstowe.
   if(typeof num == "string") {
      num = parseFloat(num).toFixed(4);
   } else {
      num = num.toFixed(4);
   }
   // Usuwamy wszystkie dodatkowe zera.
   return parseFloat(num.toString().replace(/0*$/,""));
}

function displayMsg(div,bdyText,txtColor){
   // Sprowadzamy do stanu wyjściowego zawartość DIV.
   div.innerHTML="";
   div.style.backgroundColor="yellow";
   div.style.color=txtColor
   div.innerHTML=bdyText;
}
```

Wszystkie obliczenia liczbowe rozpoczynają się w wywołaniu funkcji `handleResponse()`. W pierwszej kolejności kod otrzymuje odpowiedź jako ciąg tekstowy w wierszu `var stockPrice = request.responseText`. Następnie kod sprawdza poprawność zmiennej `stockPrice` używając metody, która jest częścią podstawowego API JavaScript: `isNaN()`. Jest to najlepszy sposób sprawdzenia, czy wartość ciągu tekstowego w JavaScript może przedstawiać poprawną liczbę. Na przykład `isNaN("dowidzenia")` zwraca wartość `true`, ponieważ „dowidzenia" nie może zostać przekonwertowane na liczbę. Za pomocą tej funkcji kod dokonuje również sprawdzenia wartości liczby walorów giełdowych.

Jeżeli którakolwiek z metod zwróci wartość `true`, wskazując tym samym na niepoprawną wartość liczbową, wówczas kod zgłosi wyjątek. Będzie to inny sposób deklaracji: „Nie możemy użyć tych wartości, proszę je stąd zabrać!". W takim przypadku strona internetowa zwróci użytkownikowi komunikat błędu.

Obsługa wyjątków w Ajaksie zostanie przedstawiona w podrozdziale „Obsługa błędów obiektu żądania" **[Sposób 8.]**.

Jednakże przeprowadzanie obliczeń nie zostało jeszcze zakończone. Kolejnym etapem jest pomnożenie przez funkcję `calcTotal()` całkowitej liczby walorów przez cenę waloru i wyświetlenie użytkownikowi całkowitej wartości walorów giełdowych.

Aby upewnić się, że wyświetlana wartość liczbowa jest wystarczająco przyjemna dla oka (przynajmniej z punktu widzenia notowań giełdy amerykańskiej), funkcja `stripExtraNumbers()` pozwala na pozostawienie co najwyżej czterech cyfr po przecinku dziesiętnym.

Nawet zapis $10,9876 może wyglądać odrobinę dziwacznie (ceny walorów giełdowych są wyświetlane czasami z zachowaniem czterech lub więcej miejsc po przecinku dziesiętnym), ale zdecydowaliśmy o pozostawieniu tego formatu w trakcie wyświetlania całkowitej wartości walorów giełdowych.

Wykonanie skryptów DOM

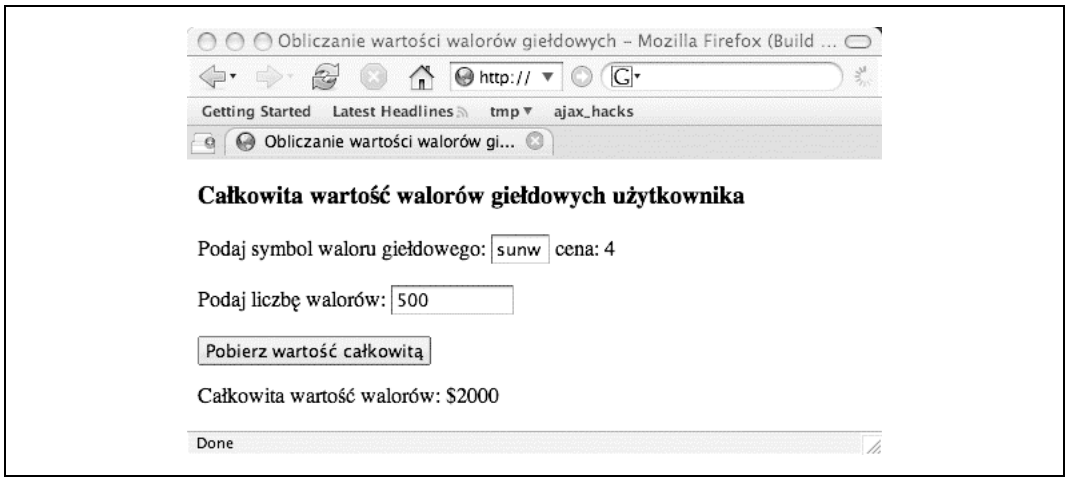
W celu dynamicznego wyświetlania na stronie nowego tekstu i wartości przedstawiony kod wykorzystuje programowanie Document Object Model. Odbywa się to bez konieczności przeprowadzania kolejnego zapytania do serwera i odświeżenia całej strony. Omawiany fragment kodu, wewnątrz funkcji `handleResponse()`, wywołuje funkcję `displayMsg()`, która wyświetla użytkownikowi całkowitą wartość walorów giełdowych. Kod powoduje również dynamiczne osadzenie ceny waloru po prawej stronie pola tekstowego, w którym użytkownik podał jego symbol giełdowy. Cały przedstawiony poniżej kod pobiera odniesienie do elementu `div` o identyfikatorze `stPrice`, powodując że właściwość nadzorująca wielkość czcionki ustawia jej rozmiar na odrobinę mniejszy od ustawienia czcionki przez użytkownika. Kolejnym krokiem jest ustawienie właściwości `innerHTML`:

```
displayMsg(document.getElementById("msgDisplay"),info,"black");
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").innerHTML ="cena: "+stockPrice;
```

Funkcja `displayMsg()` również jest prosta. Posiada parametr przedstawiający kolor czcionki, który pozwala kodowi na ustawienie czerwonego koloru czcionki w przypadku komunikatów błędu:

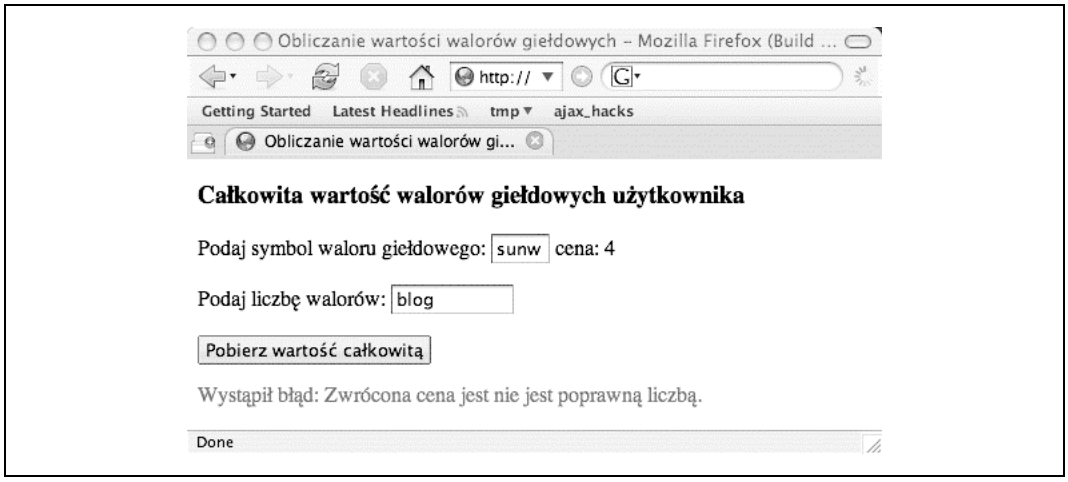
```
function displayMsg(div,bdyText,txtColor){
   // Sprowadzamy do stanu wyjściowego zawartość DIV.
   div.innerHTML="";
   div.style.backgroundColor="yellow";
   div.style.color=txtColor
   div.innerHTML=bdyText;
}
```

Na rysunku 1.7 został pokazany wygląd strony internetowej po tym, gdy użytkownik pobrał wartość giełdową posiadanych walorów.



Rysunek 1.7. Obliczanie wartości inwestycji giełdowej

Przykładowy komunikat błędu został pokazany na rysunku 1.8. Komunikat pojawia się w sytuacji, gdy użytkownik poda wartości, które nie mogą zostać użyte jako liczby, bądź serwer zwróci niepoprawne wartości.



Rysunek 1.8. Mamy dzisiaj zły dzień dla liczb

SPOSÓB 7.

Otrzymywanie danych w formacie JSON

Ajax może otrzymywać dane w efektywnym i posiadającym duże możliwości formacie JavaScript Object Notation

W jaki sposób można użyć technologii Ajax do otrzymania danych z serwera w postaci zwykłych starych obiektów JavaScript? No cóż, można w tym celu użyć formatu o nazwie JavaScript Object Notation (JSON). Przedstawiony w tym rozdziale sposób pobiera informacje podane przez użytkownika, a następnie inicjuje przesłanie danych do serwera, który z kolei zwróci je w składni JSON do użycia na stronie internetowej.

JSON jest prostym i jasnym formatem, prawdopodobnie dlatego lubi go wielu programistów. Dane w formacie JSON są odpowiednie dla prostych obiektów będących zestawem właściwości i wartości. Przykładem jest program serwerowy, który wyciąga informacje z bazy danych lub bufora, a następnie zwraca je w formacie JSON stronie internetowej. Dane w formacie JSON są przedstawione przez:

- Otwierający nawias klamrowy (`{`).
- Jedną lub więcej nazw właściwości, oddzielnych dwukropkiem od ich wartości. Pary właściwość/wartość są rozdzielone przecinkami.
- Zamykający nawias klamrowy (`}`).

Wartościami każdej właściwości w obiekcie mogą być:

- Proste ciągi tekstowe, na przykład `"witaj"`.
- Tablice, takie jak `[1,2,3,4]`.
- Liczby.
- Wartości `true`, `false` lub `null`.
- Inne obiekty, takie jak kompozycja albo inny obiekt zawierający jeden lub więcej obiektów.

Więcej informacji na ten temat znajduje się na witrynie *http://www.json.org/*.

Dokładnie jest to format dosłownego obiektu (`Object`) w JavaScript. Poniżej, jako przykład, w podrozdziale „Użycie obiektu żądania do przekazania danych POST do serwera" **[Sposób 2.]** został przedstawiony wygląd informacji pobieranych od użytkownika, zapisanych w formacie JSON:

```
{
firstname:"Bruce",
lastname:"Perry",
gender:"M",
country:"USA"
}
```

Magia formatu JSON

W tym podrozdziale użyjemy strony HTML, podobnej do wykorzystanej w podrozdziale „Użycie obiektu żądania do przekazania danych POST do serwera" **[Sposób 2.]**, i zapytamy użytkownika o te same informacje. Jednakże w omawianym sposobie zostanie użyty kod JavaScript i technologia Ajax do obsługi zwracanych przez serwer wartości w formacie JSON. Dwa elementy `div` w dolnej części strony HTML pokazują wartość zwrotną serwera, a następnie wyświetlają właściwości i wartości obiektów w bardziej przyjazny sposób.

Poniżej znajduje się kod HTML omawianej strony internetowej:

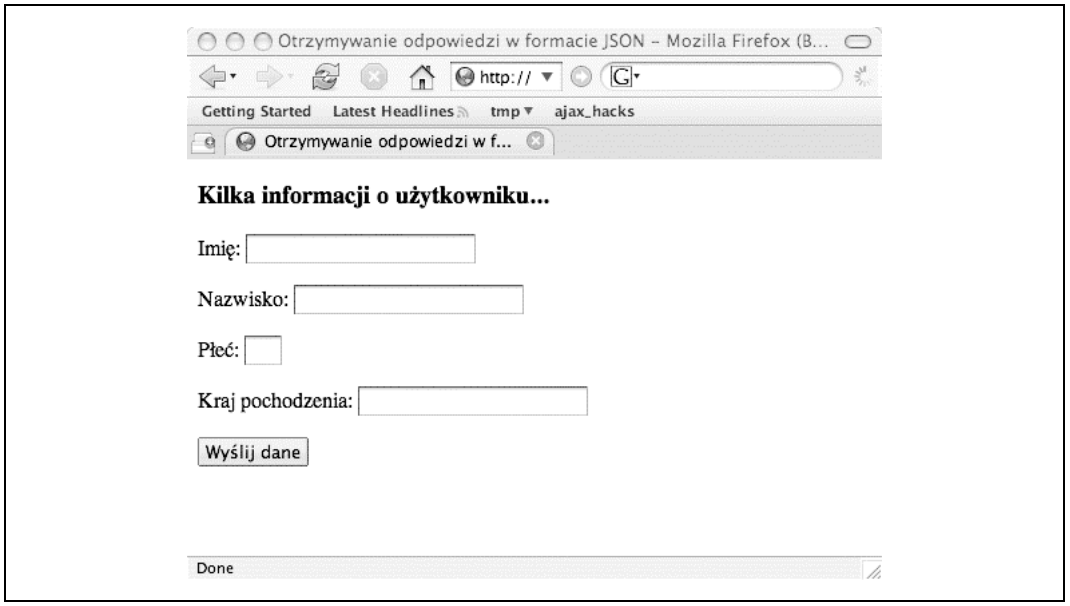
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack5.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Otrzymywanie odpowiedzi w formacie JSON</title>
</head>
<body>
<h3>Kilka informacji o użytkowniku...</h3>
<form action="javascript:void%200" onsubmit="sendData();return false">
   <p>Imię: <input type="text" name="firstname" size="20"></p>
   <p>Nazwisko: <input type="text" name="lastname" size="20"></p>
   <p>Płeć: <input type="text" name="gender" size="2"></p>
   <p>Kraj pochodzenia: <input type="text" name="country" size="20"></p>
   <p><button type="submit">Wyślij dane</button></p>
   <div id="json"></div>
   <div id="props"></div>
</form>
</body>
</html>
```

Wygląd strony internetowej został pokazany na rysunku 1.9.

Kod JavaScript jest importowany przez znacznik `script` z pliku o nazwie *hack5.js*. Podane przez użytkownika wartości są przez JavaScript wysyłane do serwera. Ponieważ ten etap został omówiony w sposobie „Użycie obiektu żądania do przekazania danych POST do serwera" **[Sposób 2.]** oraz innych sposobach, kod zostanie tutaj przedstawiony, ale nie będzie szczegółowo omówiony.

Należy uważać na ataki typu cross-site scripting (XSS) podczas określania w ten sposób jakichkolwiek wartości zwrotnych jako kodu JavaScript. Jest to potencjalne zagrożenie w trakcie używania funkcji `eval()` lub funkcji powiązanych z kodem przedstawionym w tym sposobie.

Przeciwdziałając takiej sytuacji, kod JavaScript po stronie klienta może filtrować i sprawdzać wartości zwrotne (na przykład poprzez przeglądanie właściwości `responseText` obiektu `XMLHttpRequest`) pod kątem wystąpienia oczekiwanych nazw właściwości obiektów, zanim właściwość `responseText` zostanie użyta w funkcji `eval()` (warto zapoznać się ze stroną *http://www.perl.com/pub/a/2002/02/20/css.html*).



Rysunek 1.9. Nadchodzi format JSON

Poniżej znajduje się kod omawianego sposobu. Po przedstawieniu kodu nastąpi omówienie kluczowych jego części, które obsługują zwracane wartości jako obiekty JavaScript.

```
var request;
var queryString; // Zmienna będzie przechowywała dane wysłane metodą POST.

function sendData(){
   setQueryString();
   url="http://localhost:8080/parkerriver/s/json";
   httpRequest("POST",url,true);
}

// Obsługa zdarzeń dla obiektu XMLHttpRequest.
function handleJson(){
   if(request.readyState == 4){
      if(request.status == 200){
         var resp = request.responseText;
         var func = new Function("return "+resp);
         var objt = func();
         var div = document.getElementById("json");
         stylizeDiv(resp,div);
         div = document.getElementById("props");
         div.innerHTML="<h4>W obiekcie formularza...</h4>"+
            "<h5>Właściwości</h5>firstname= "+
            objt.firstname +"<br />lastname="+
            objt.lastname+ "<br />gender="+
            objt.gender+ "<br />country="+
            objt.country;
      } else {
         alert("Wystąpił problem z komunikacją między obiektem XMLHttpRequest, "+
            "a programem serwera.");
      }
   }// Koniec zewnętrznej pętli if.
}
```

```
/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
function initReq(reqType,url,bool){
   /* Określamy funkcję, która będzie obsługiwała odpowiedź HTTP. */
   request.onreadystatechange=handleJson;
   request.open(reqType,url,bool);
   request.setRequestHeader("Content-Type",
      "application/x-www-form-urlencoded; charset=iso-8859-2");
   request.send(queryString);
}

/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
   reqType: typ żądania HTTP, na przykład GET lub POST.
   url: adres URL programu serwerowego.
   asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */

function httpRequest(reqType,url,asynch){
   // Skrócono... Zobacz [Sposób 01] lub [Sposób 02].
}

function setQueryString(){
   queryString="";
   var frm = document.forms[0];
   var numberElements = frm.elements.length;
   for(var i = 0; i < numberElements; i++) {
      if(i < numberElements-1) {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value)+"&";
      } else {
         queryString += frm.elements[i].name+"="+
            encodeURIComponent(frm.elements[i].value);
      }
   }
}

function stylizeDiv(bdyTxt,div){
   // Sprowadzamy do stanu wyjściowego zawartość DIV.
   div.innerHTML=" ";
   div.style.fontSize="1.2em";
   div.style.backgroundColor="yellow";
   div.appendChild(document.createTextNode(bdyTxt));
}
```

Podobnie jak w poprzednich sposobach przedstawionych w tym rozdziale, funkcja `initReq()` inicjalizuje obiekt żądania i wysyła do serwera żądanie HTTP.

Gdy odpowiedź jest gotowa, funkcja obsługi zdarzeń wywołuje funkcję `handleJson()`. Odpowiedzią jest ciąg tekstowy w formacie JSON, w przeciwieństwie do XML lub innego rodzaju tekstu. JavaScript interpretuje ten zwrócony tekst jako obiekt ciągu tekstowego (`string`). Dlatego też kod inicjalizuje otwierający krok, zanim wartości zwrotne serwera zostaną zinterpretowane jako dosłowny obiekt JavaScript. (Przy okazji warto wspomnieć, że w omawianym sposobie serwer pobiera parametry żądania, a następnie przed wysłaniem danych jako odpowiedzi przeformatowuje nazwy parametrów i wartości właściwości do postaci składni JSON).

Dedykowany obsłudze błędów kod nie jest tutaj przedstawiony, ponieważ jego elementy wymagają szerszych wyjaśnień. Odpowiedni kod został przedstawiony w podrozdziale „Obsługa błędów obiektu żądania" **[Sposób 8.]**.

Wewnątrz kodu `handleJson()` (pogrubionego na poprzednim przykładzie) zmienna `resp` odnosi się do tekstu odpowiedzi HTTP, który JavaScript interpretuje jako ciąg tekstowy. Interesujące zjawiska zachodzą w konstruktorze `Function`:

```
var func = new Function("return "+resp);
```

Przedstawiony wiersz kodu tworzy „w locie" nowy obiekt `Function` i przechowuje `Function` w zmiennej o nazwie `func`. Programiści JavaScript mogli zauważyć, że większość funkcji zostało predefiniowanych i zadeklarowanych w kodzie lub utworzonych jako dosłowne funkcje. W przedstawionym przypadku musimy jednak dynamicznie zdefiniować funkcję używając ciągu tekstowego, a konstruktor `Function` dostarcza doskonałego do tego celu narzędzia.

Dziękujemy stronie *http://www.jjbbering.com/2002/4/httprequest.html* za pomoc w opracowaniu kodu.

Inną metoda konwersji ciągów tekstowych JSON, która jest stosowana w Sieci, przedstawia się następująco:

```
var resp = request.responseText;
var obj = eval( "(" + resp + ")" );
```

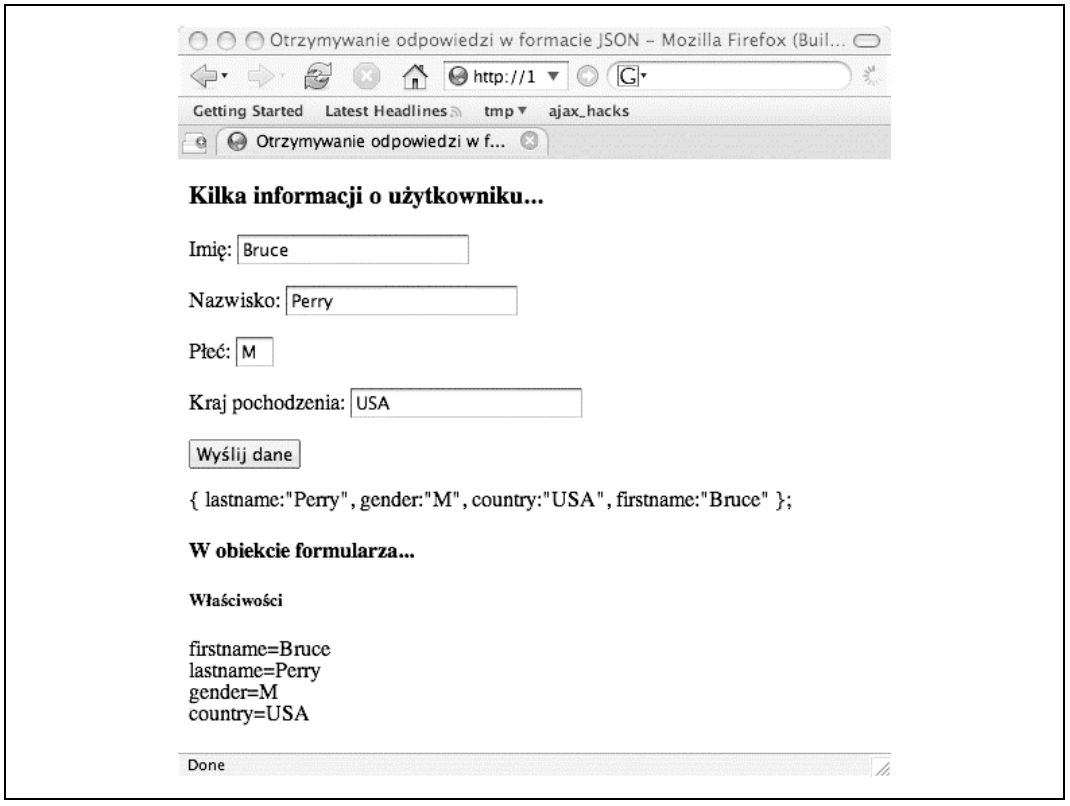
Podczas używania funkcji `eval()` i tablicy nie zachodzi konieczność użycia nawiasów okrągłych, jak to przedstawiono w poniższym fragmencie kodu:

```
var resp = request.responseText;
// Zmienna resp zawiera dane w rodzaju: "[1,2,3,4]"
var arrObject = eval(resp);
```

Kolejne wiersze tworzą funkcję zwracającą dosłowny obiekt, który przedstawia zwracane wartości z serwera. Następnie wywołujemy funkcję i używany zwróconego obiektu w celu dynamicznego wyświetlenia na stronie internetowej wartości serwera za pomocą programowania DOM (bez złożonej serializacji obiektu lub odświeżenia strony!):

```
var objt = func();
   var div = document.getElementById("json");
   stylizeDiv(resp,div);
   div = document.getElementById("props");
   div.innerHTML="<h4>W obiekcie formularza...</h4>"+
      "<h5>Właściwości</h5>firstname= "+
      objt.firstname +"<br />lastname="+
      objt.lastname+ "<br />gender="+
      objt.gender+ "<br />country="+
      objt.country;
```

Zmienna o nazwie `objt` przechowuje dosłowny obiekt. Wartości z obiektu są wydobywane za pomocą składni `objt.firstname`. Na rysunku 1.10 został pokazany wygląd strony internetowej po otrzymaniu przez nią odpowiedzi.



Rysunek 1.10. Wizualizacja właściwości JavaScript jest urocza!

Po stronie serwera

W omawianym sposobie żądanie jest obsługiwane przez serwlet Javy. Dla Czytelników zainteresowanych aktywnością serwera została poniżej przedstawiona metoda `doPost()` dla omawianego kodu:

```
protected void doPost(HttpServletRequest httpServletRequest,
      HttpServletResponse httpServletResponse) throws
      ServletException, IOException {
   Map valMap = httpServletRequest.getParameterMap();
   StringBuffer body = new StringBuffer("{\n");

   if(valMap != null) {
      String val=null;
      String key = null;
      Map.Entry me = null;
      Set entries = valMap.entrySet();

      int size = entries.size();
      int counter=0;
      for(Iterator iter= entries.iterator();iter.hasNext();) {
         counter++;
         me=(Map.Entry) iter.next();
         val= ((String[]me.getValue())[0];
```

```
            key = (String) me.getKey();
            if (counter < size){
               body.append(key).append(":\"").append(val).append("\",\n");
            } else {
               // Usunięcie przecinka z ostatniego wpisu.
               body.append(key).append(":\"").append(val).append("\"\n");
            }
         }
      }
      body.append("}");
      AjaxUtil.sendText(httpServletResponse,body.toString());
   }
```

Klasa `AjaxUtil` wysyła odpowiedź HTTP wraz z nagłówkiem `Content-Type` z wartością `text/plain; charset=iso-8859-2`. Twórcy niektórych witryn internetowych rozważali użycie nagłówka `Content-Type` typu `application/x-json` dla formatu JSON, ale w trakcie pisania tej książki programiści oraz instytucje standaryzujące nie ustanowiły związanego z tym standardu.

Klasa `AjaxUtil` ustawia również nagłówkowi odpowiedzi HTTP `Cache-Control` wartość `no-cache`, co oznacza, że przeglądarka internetowa i agent użytkownika nie buforują odpowiedzi:

```
response.setHeader("Cache-Control", "no-cache");
```

SPOSÓB 8.

Obsługa błędów obiektu żądania

Projektowanie aplikacji Ajax w taki sposób, aby wykrywała błędy serwera i dostarczała przejrzystego komunikatu o błędzie

Wiele ikry kryjącej się za technologią Ajax jest związane z faktem, że JavaScript łączy się z programem serwera bez konieczności interwencji ze strony użytkownika. Jednakże programiści JavaScript często nie mają kontroli nad samym komponentem serwera (którym może być usługa sieciowa lub inne oprogramowanie zaprojektowane poza organizacją programisty). Nawet jeśli dana aplikacja wykorzystuje komponent serwera utworzony przez tę samą firmę, to nie można zakładać, że serwer zawsze będzie zachowywał się normalnie lub użytkownicy będą znajdować się online w chwili wywołania obiektu żądania. Należy się więc upewnić, że aplikacja posiada możliwość odtworzenia zdarzenia, gdy wywoływany program był niedostępny.

Przedstawiony tutaj sposób przechwytuje błędy i wyświetla czytelny komunikat błędu w przypadku, gdy aplikacja Ajax utraci połączenie z serwerem.

Problemy, problemy…

Omówiony w tym podrozdziale sposób dotyczy następujących wyjątkowych zdarzeń i zawiera zalecenia dla aplikacji w kwestii ich odzyskania:

- Aplikacja sieciowa lub komponent serwera, z którym istnieje połączenie, jest chwilowo niedostępny.

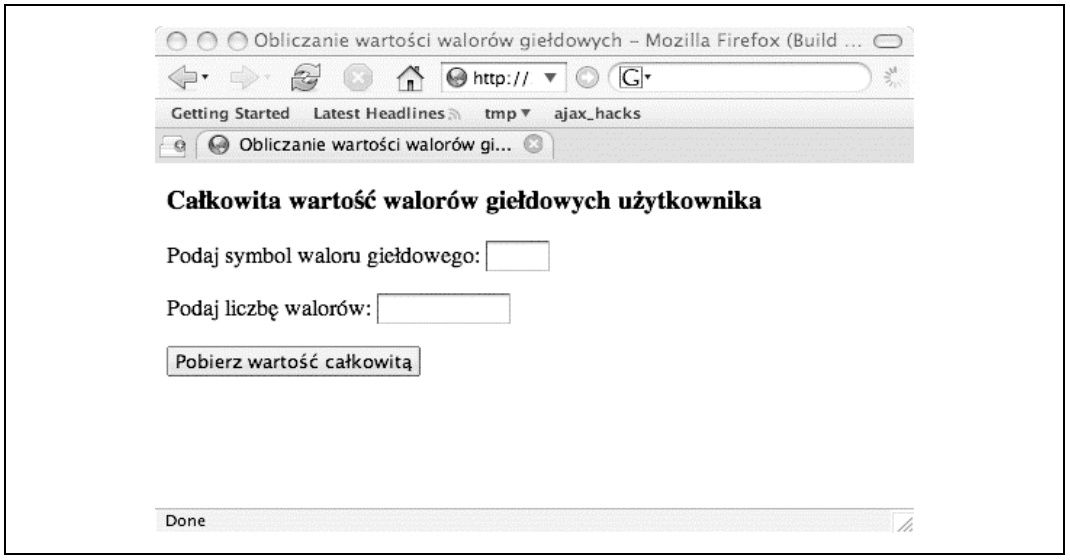
- Serwer, z którym jest połączona aplikacja, został zamknięty lub jego adres URL został bez wiedzy programisty zmieniony.
- Komponent serwera, z którym następuje połączenie, zawiera jeden lub więcej błędów i podczas połączenia następuje jego awaria (o tak!).
- W trakcie wywołania metody `open()` obiektu żądania wywołujący ją kod używa innego adresu węzła niż adres, z którego użytkownik pobrał stronę internetową. W takim przypadku obiekt żądania zgłasza wyjątek, gdy następuje próba wywołania metody `open()`.

Przedstawionego w tym miejscu sposobu można użyć do obsługi wyjątków w każdej aplikacji. W sposobie jest wykorzystany przedstawiony w podrozdziale „Otrzymywanie danych w postaci liczb" **[Sposób 6.]** kod obliczeń walorów giełdowych. Poniżej przyjrzymy się kodowi, który inicjalizuje obiekt żądania, oraz mechanizmowi obsługi wyjątków, ale w pierwszej kolejności poniżej przedstawiamy kod HTML, który importuje kod JavaScript z pliku *hack6.js*:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack6.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Obliczanie wartości walorów giełdowych</title>
</head>
<body>
<h3>Całkowita wartość walorów giełdowych użytkownika</h3>
<form action="javascript:void%200" onsubmit=
      "getStockPrice(this.stSymbol.value,this.numShares.value);return false">
   <p>Podaj symbol waloru giełdowego: <input type="text" name="stSymbol"
size="4">
      <span id="stPrice"></span></p>
   <p>Podaj liczbę walorów: <input type="text" name="numShares" size="10"></p>
   <p><button type="submit">Pobierz wartość całkowitą</button></p>
   <div id="msgDisplay"></div>
</form>
</body>
</html>
```

Kiedy użytkownicy wczytają powyższy plik do swoich przeglądarek internetowych, wówczas zobaczą stronę pokazaną na rysunku 1.11.

Kod, którym jesteśmy szczególnie zainteresowani, może przechwytywać wyjątki angażujące niedostępne aplikacje, niedziałające serwery, błędy w oprogramowaniu serwerów oraz niepoprawne adresy URL. Funkcja `handleResponse()` stanowi obsługę zdarzeń zarządzającą odpowiedziami serwera, jak w wierszu `request.onreadystatechange =handleResponse`. Przedstawiony poniżej kod używa zagnieżdżonych poleceń `try/catch/finally` zajmujących się nieprawidłowymi liczbami obsługiwanymi przez aplikację, jak to zostało omówione w podrozdziale „Otrzymywanie danych w postaci liczb" **[Sposób 6.]**.



Rysunek 1.11. Żądanie podania ceny waloru giełdowego

```
function handleResponse(){
   var statusMsg="";
   try{
      if(request.readyState == 4){
         if(request.status == 200){
            /* Sprawdzenie, czy wartość zwrotna rzeczywiście jest liczbą.
            Jeżeli tak jest, to następuje pomnożenie przez liczbę walorów giełdowych
            i wyświetlenie wyniku. */
            var stockPrice = request.responseText;

            try{
               if(isNaN(stockPrice)) { throw new Error(
                  "Zwrócona cena jest nie jest poprawną liczbą.");}
               if(isNaN(numberOfShares)) { throw new Error(
                  "Liczba walorów nie jest poprawną liczbą.");}
               var info = "Całkowita wartość walorów: $"+
               calcTotal(stockPrice);
               displayMsg(document.getElementById("msgDisplay"),info,"black");
               document.getElementById("stPrice").style.fontSize="0.9em";
               document.getElementById("stPrice").innerHTML ="cena: "+
                  stockPrice;
            } catch (err) {
               displayMsg(document.getElementById("msgDisplay"),
                  "Wystąpił błąd: "+
                  +err.message,"red");
            }
         } else {
         // Jeżeli aplikacja nie jest dostępna, wtedy stan żądania wynosi 503,
         // natomiast w przypadku błędu w aplikacji stan żądania wynosi 500.
         alert(
            " Wystąpił problem z komunikacją między obiektem XMLHttpRequest, "+
            "a programem serwera. "+
            "Proszę wkrótce spróbować ponownie.");
         }
      } // Koniec zewnętrznej pętli if.
   } catch (err) {
      alert("Serwer nie jest dostępny "+
```

```
            "dla tej aplikacji. Proszę wkrótce spróbować"+
            " ponownie. \nBłąd: "+err.message);

    }
}
```

Przyjrzyjmy się, w jaki sposób przedstawiony kod obsługuje różne rodzaje wymienionych wyżej wyjątków.

Zamknięty serwer

Blok `try` przechwytuje każdy wyjątek zgłoszony wewnątrz jego nawiasów klamrowych (`{}`). Jeżeli kod zgłasza wyjątek, wówczas następuje wykonanie kodu umieszczonego w bloku `catch`. Wewnętrzny blok `try`, który został zaprojektowany do zarządzania wyjątkami zgłoszonymi w przypadku wystąpienia nieprawidłowych wartości liczbowych, został omówiony w podrozdziale „Otrzymywanie danych w postaci liczb" **[Sposób 6.]**.

Co więc się zdarzy, jeśli serwer zostanie całkowicie wyłączony, nawet gdy adres URL używany przez aplikację jest w innym przypadku prawidłowy? W takich sytuacjach przeprowadzane przez kod próby uzyskania dostępu do właściwości `request.status` powodują zgłoszenie wyjątku, ponieważ obiekt żądania nigdy nie otrzyma od serwera oczekiwanego nagłówka odpowiedzi, a właściwość `status` nie jest powiązana z żadnymi danymi.

W wyniku opisanej powyżej sytuacji kod wyświetli okno `alert`, które zostało zdefiniowane w zewnętrznym bloku `catch`. Na rysunku 1.12 został pokazany wygląd okna `alert` w przypadku tego typu błędów.

Rysunek 1.12. Ojej, serwer nie działa

Kod wyświetla użytkownikowi komunikat, jak również znacznie bardziej techniczny komunikat błędu powiązany z wyjątkiem. Techniczną część komunikatu można pominąć; staje się ona użyteczna głównie w celach debugowania aplikacji.

Zmienna `err` w przedstawionym kodzie jest odniesieniem do obiektu `Error` JavaScript. Właściwość `message` tego obiektu (jak `err.message`) jest rzeczywistym komunikatem błędu, ciągiem tekstowym wygenerowanym przez silnik JavaScript.

Jeżeli w kodzie nie zostanie umieszczony mechanizm `try/catch/finally`, wówczas użytkownik zobaczy po prostu okno `alert` zawierające wygenerowany przez JavaScript, nie nadający się do odczytania komunikat błędu. Po usunięciu tego okna (lub odejściu w stanie frustracji od komputera) użytkownik wciąż nie będzie miał pojęcia, w jakim stanie znajduje się aplikacja.

Niemożliwość uruchomienia aplikacji serwera

Czasami aplikacja serwera lub sam węzeł działają prawidłowo, ale komponent serwera, z którym aplikacja próbuje nawiązać połączenie, jest wyłączony. W takim przypadku wartość właściwości `request.status` wynosi `503` („Usługa niedostępna"). Ponieważ właściwość status przechowuje wartość inną niż `200`, przedstawiony w tym sposobie kod uruchamia wyrażenie umieszczone wewnątrz bloku `else`:

```
} else {
// Jeżeli aplikacja nie jest dostępna, wtedy stan żądania wynosi 503,
// natomiast w przypadku błędu w aplikacji stan żądania wynosi 500.
alert(
   " Wystąpił problem z komunikacją między obiektem XMLHttpRequest, "+
   "a programem serwera. "+
   "Proszę wkrótce spróbować ponownie.");
}
```

Innymi słowy, użytkownik zobaczy okno `alert` wyjaśniające aktualny stan aplikacji. Wspomniane okno `alert` pojawi się również, jeśli komponent serwera zawiera błąd i ulegnie awarii. Takie zdarzenie zwykle (na przykład w przypadku serwletu Tomcat) powoduje powstanie kodu stanu odpowiedzi o wartości `500` („Wewnętrzny błąd serwera"). Dlatego też właściwość `response.status` posiada wartość `500` zamiast `200` („Wszystko w porządku"). Oprócz tego, każdy kod stanu odpowiedzi o wartości `404`, angażujący statyczny lub dynamiczny komponent, który nie może zostać odnaleziony przez serwer pod podanym adresem URL, zostaje przechwycony przez przedstawione polecenie `try`.

Blok `try/catch/finally` jest dostępny jedynie w przypadku kodu JavaScript w wersji 1.4 lub nowszej. Opcjonalny blok `finally` znajduje się po bloku `catch`. Kod umieszczony w bloku `finally{...}` jest wykonywany niezależnie od tego, czy wyjątek został zgłoszony.

Ups, błędny adres URL

Co się stanie, jeżeli adres URL używany przez aplikację Ajax w metodzie `request.open()` jest nieprawidłowy lub ulegnie zmianie? W takim przypadku wywołanie metody `request.open()` spowoduje zgłoszenie wyjątku, a więc jest to dobre miejsce na umieszczenie poleceń `try/catch/finally`. Kod przedstawiony w początkowej części kolejnego przykładu tworzy obiekt żądania **[Sposób 1.]**. Pojawiająca się później definicja funkcji `initReq()` przechwytuje opisany powyżej wyjątek:

```
function httpRequest(reqType,url,asynch){
   // Przeglądarki na bazie Mozilli.
   if(window.XMLHttpRequest){
      request = new XMLHttpRequest();
   } else if (window.ActiveXObject){
      request=new ActiveXObject("Msxml2.XMLHTTP");
      if (! request){
         request=new ActiveXObject("Microsoft.XMLHTTP");
      }
   }
   // Jeżeli nie powiodła się nawet inicjalizacja ActiveXObject,
   // wówczas żądanie wciąż może być typu null.
   if(request){
      initReq(reqType,url,asynch);
   } else {
      alert("Używana przeglądarka nie pozwala na wykorzystanie "+
         "wszystkich funkcji tej aplikacji!");}
   }
}
/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
function initReq(reqType,url,bool){
   try{
      /* Określenie funkcji, która będzie obsługiwała odpowiedź HTTP. */
      request.onreadystatechange=handleResponse;
      request.open(reqType,url,bool);
      request.send(null);
   } catch (err) {

      alert(
         "Aplikacja nie może w tej chwili nawiązać połączenia z serwerem."+
         " Proszę wkrótce spróbować ponownie.");
   }
}
```

Z inną odmianą takiego błędu mamy do czynienia wtedy, gdy adres URL używany w metodzie `request.open()` zawiera inną nazwę węzła niż węzeł, z którego użytkownik pobrał stronę internetową. Przykładowo, użytkownik pobrał stronę internetową z *http://www.pewna_organizacja.com/app*, ale adres URL używany w `open()` to *http://www.inna_organizacja.com*. Tego typu błąd jest również przechwytywany przez blok `try/catch/finally`.

Istnieje również opcjonalna możliwość anulowania lub przerwania żądania w bloku `catch` za pomocą metody `request.abort()`. Więcej informacji na ten temat znajdzie się w podrozdziale „Ustawienie ograniczenia czasu dla żądania HTTP" **[Sposób 70.]**. We wspomnianym sposobie znajdzie się również analiza ustawienia limitu czasu dla żądania oraz jego przerwanie w sytuacji, gdy żądanie nie zostanie ukończone w podanym okresie.

SPOSÓB 9.

Zagłębienie się w odpowiedź HTTP

Wyświetlenie wartości różnych nagłówków odpowiedzi HTTP w dodatku lub zamiast typowych wartości zwrotnych serwera

Tak zwany *nagłówek odpowiedzi* HTTP jest umieszczoną przez protokół HTTP 1.1 informacją opisową, którą serwery sieciowe wysyłają w ramach odpowiedzi wraz z rzeczywistą stroną internetową lub danymi. Jeżeli Czytelnik posiada doświadczenie w programowaniu obiektu `XMLHttpRequest` (został przeanalizowany na początku bieżącego

rozdziału), wówczas wie, że właściwość `request.status` jest przyrównywana do kodu stanu odpowiedzi HTTP wysłanego z serwera. Jest to ważna wartość do sprawdzenia, zanim strona wykona jakiekolwiek inne zadania związane z odpowiedzią HTTP.

Wartości stanu mogą wynosić `200` (żądanie zostało zakończone prawidłowo), `404` (żądany plik lub ścieżka URL nie zostały odnalezione) lub `500` (wewnętrzny błąd serwera).

Jednakże może zaistnieć konieczność przejrzenia pewnych nagłówków odpowiedzi powiązanych z żądaniem, zawierających informacje takie jak typ oprogramowania serwera sieciowego obsługującego żądanie (nagłówek odpowiedzi `Server`) lub typ zawartości odpowiedzi (nagłówek `Content-Type`). W omawianym sposobie użytkownik jest proszony o podanie w polu tekstowym adresu URL. Kiedy użytkownik opuści to pole lub kliknie myszą poza obszarem tego pola, wówczas przeglądarka wyświetli różne nagłówki odpowiedzi HTTP. Jak zwykle w przypadku technologii Ajax, będzie to miało miejsce bez konieczności odświeżenia strony.

Metoda obiektu żądania zwraca jedynie podzbiór dostępnych nagłówków odpowiedzi, włączając w nie `Content-Type`, `Date`, `Server` i `Content-Length`.

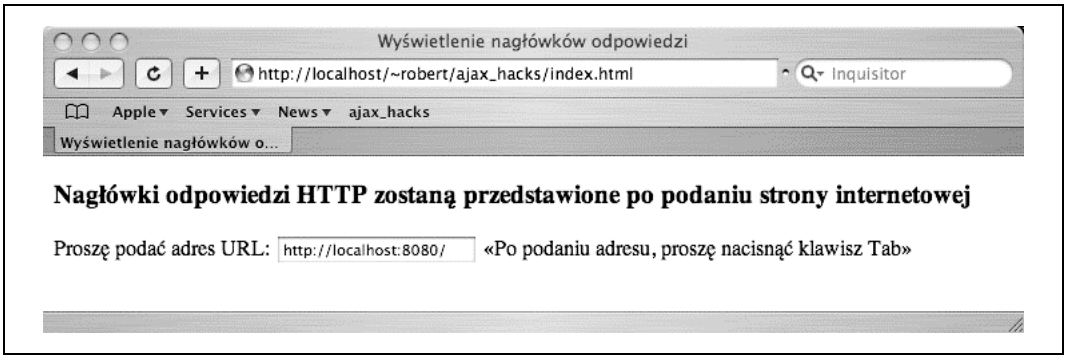
Poniżej został przedstawiony kod HTML strony:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack7.js"></script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Wyświetlenie nagłówków odpowiedzi</title>
   <link rel="stylesheet" type="text/css" href="/parkerriver/css/hacks.css"/>
</head>
<body onload="document.forms[0].url.value=urlFragment">
<h3>Nagłówki odpowiedzi HTTP zostaną przedstawione po podaniu strony
internetowej</h3>

<form action="javascript:void%200">
   <p>Proszę podać adres URL:
      <input type="text" name="url" size="20"
onblur="getAllHeaders(this.value)">
      <span class="message">&#171;Po podaniu adresu, proszę nacisnąć klawisz
      Tab&#187;</span></p>
      <div id="msgDisplay"></div>
</form>
</body>
</html>
```

Na rysunku 1.13 został przedstawiony wygląd tej strony w przeglądarce Safari.

Aplikacja wstępnie wypełnia pole tekstowe częściowym adresem URL (na przykład *http://localhost:8080/*), a użytkownik podaje pozostałą część. Jest to spowodowane faktem, że obiekt żądania nie może wysłać żądania do innego węzła niż ten, który przekazał użytkownikowi stronę internetową. Innymi słowy, częściowo uzupełniony adres URL stanowi podpowiedź dla użytkownika, że aplikacja może wysłać żądania tylko do podanego węzła.



Rysunek 1.13. Możliwości w zakresie nagłówków

Kiedy użytkownik uzupełni adres URL, a następnie naciśnie klawisz *Tab* lub kliknie myszą obszar poza polem tekstowym, wówczas zostanie wywołana obsługa zdarzenia `onblur` pola tekstowego. Obsługa zdarzenia jest zdefiniowana jako funkcja `getAllHeaders()`, która przekazuje obiektowi żądania podany przez użytkownika adres URL. Następnie obiekt żądania wysyła żądanie pod podany adres URL i zwraca stronie internetowej dostępne nagłówki odpowiedzi.

Przedstawiony poniżej kod pochodzi z importowanego przez stronę pliku *hack7.js*. Po zaprezentowaniu kodu nastąpi wyjaśnienie jego fragmentów odpowiedzialnych za wyświetlenie nagłówków odpowiedzi. Podrozdział „Określenie zgodności przeglądarki internetowej za pomocą obiektu żądania" **[Sposób 1.]** wyjaśnił, w jaki sposób zainicjalizować i otworzyć połączenie HTTP z obiektem żądania, znanym również jako `XMLHttpRequest`. Natomiast w podrozdziale „Obsługa błędów obiektu żądania" **[Sposób 8.]** omówione zostały kwestie dotyczące przechwytywania wszystkich błędów za pomocą poleceń JavaScript `try/catch/finally`.

```
var request;
var urlFragment="http://localhost:8080/";

function getAllHeaders(url){
   httpRequest("GET",url,true);
}

// Funkcja dla obsługi zdarzenia onreadystatechange obiektu XMLHttpRequest.
function handleResponse(){
   try{
      if(request.readyState == 4){
         if(request.status == 200){
            /* Wszystkie nagłówki są otrzymane jako pojedynczy ciąg tekstowy. */
            var headers = request.getAllResponseHeaders();
            var div = document.getElementById("msgDisplay");
            div.className="header";
            div.innerHTML="<pre>"+headers+"</pre>";
         } else {
            // Jeżeli aplikacja nie jest dostępna, wtedy stan żądania wynosi 503,
            // natomiast w przypadku błędu w aplikacji stan żądania wynosi 500.
            alert(request.status);
            alert("Wystąpił problem z komunikacją między obiektem
            XMLHttpRequest, "+
               "a programem serwera.");
```

```
            }
        }// Koniec zewnętrznej pętli if.
    } catch (err) {
        alert("Serwer nie jest dostępny "+
            "dla tej aplikacji. Proszę wkrótce spróbować"+
            " ponownie. \nBłąd: "+err.message);
    }
}

/* Inicjalizacja obiektu żądania, który został już skonstruowany. */
function initReq(reqType,url,bool){
    try{
        /* Określenie funkcji, która będzie obsługiwała odpowiedź HTTP. */
        request.onreadystatechange=handleResponse;
        request.open(reqType,url,bool);
        request.send(null);
    } catch (errv) {
        alert(
            "Aplikacja nie może w tej chwili nawiązać połączenia z serwerem. "+
            "Proszę wkrótce spróbować ponownie." );
    }
}

/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
    reqType: typ żądania HTTP, na przykład GET lub POST.
    url: adres URL programu serwerowego.
    asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */
function httpRequest(reqType,url,asynch){
    // Przeglądarki na bazie Mozilli.
    if(window.XMLHttpRequest){
        request = new XMLHttpRequest();
    } else if (window.ActiveXObject){
        request=new ActiveXObject("Msxml2.XMLHTTP");
        if (! request){
            request=new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
    // Jeżeli nie powiodła się nawet inicjalizacja ActiveXObject,
    // wówczas żądanie wciąż może być typu null.
    if(request){
        initReq(reqType,url,asynch);
    } else {
        alert(
            "Używana przeglądarka nie pozwala na wykorzystanie "+
            "wszystkich funkcji tej aplikacji!");
    }
}
```

Interesujące zjawiska zachodzą w funkcji `handleResponse()`. Ta funkcja wywołuje metodę `getAllResponseHeaders()` obiektu żądania i zwraca (raczej niefortunnie) wszystkie dostępne nagłówki odpowiedzi, umieszczone w pojedynczym ciągu tekstowym. Programista prawdopodobnie wolałby otrzymać te wartości w formacie JSON jako tablicę asocjacyjną zamiast monolitycznego ciągu tekstowego, który w celu wydobycia informacji o nagłówku wymaga dodatkowej ilości kodu.

Aby pobrać tylko jeden nagłówek, można użyć funkcji `request.getResponseHeader()`. Przykładem może być `request.getResponseHeader("Content-Type");`.

Następnie kod przechowuje element `div`, w którym zostaną wyświetlone wartości nagłówków:

```
if(request.status == 200){
   /* Wszystkie nagłówki są otrzymane jako pojedynczy ciąg tekstowy. */
   var headers = request.getAllResponseHeaders();
   var div = document.getElementById("msgDisplay");
   div.className="header";
   div.innerHTML="<pre>"+headers+"</pre>";
}...
```

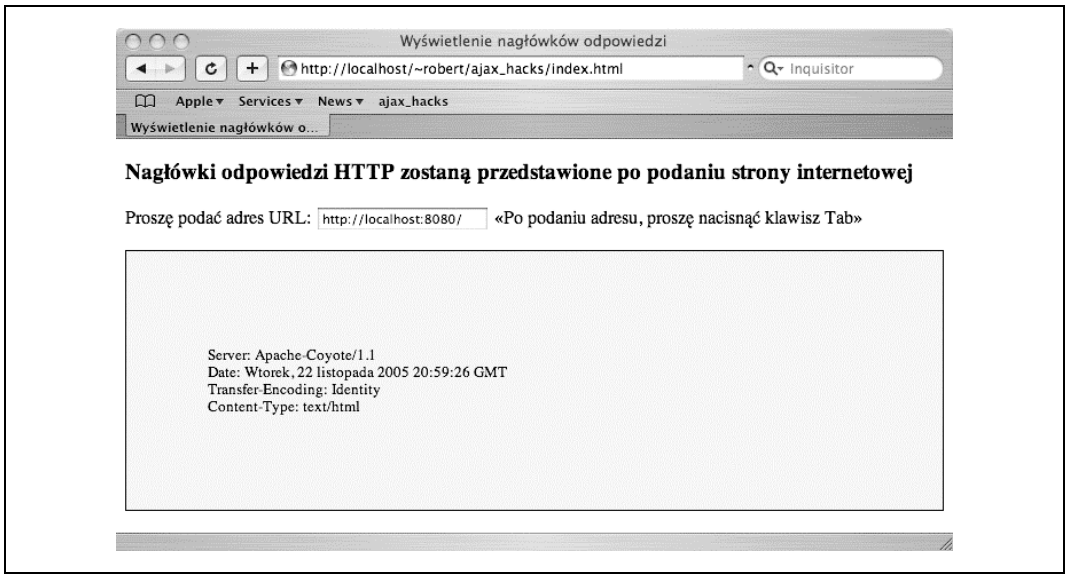
W celu zapewnienia stylu CSS dla wyświetlanych informacji, kod powoduje ustawienie właściwości `className` elementu `div` klasy, która została zdefiniowana w arkuszu stylów. Poniżej znajduje się arkusz stylów, który został dołączony do tej strony internetowej:

```
div.header{ border: thin solid black; padding: 10%;
   font-size: 0.9em; background-color: yellow}
span.message { font-size: 0.8em; }
```

W taki sposób kod dynamicznie łączy element `div` z określoną klasą CSS, która jest zdefiniowana w oddzielnym pliku arkusza stylów. Taka strategia pozwala na oddzielenie programowania DOM od warstwy prezentacyjnej. Na końcu, właściwości `innerHTML` elementu `div` zostają ustawione zwracane wartości nagłówka. Znacznik `pre` może zostać wykorzystany w celu zachowania istniejącego formatowania.

Alternatywnie, istnieje możliwość operowania w zupełnie inny sposób zwracanym ciągiem tekstowym i formatem nagłówka — za pomocą własnej funkcji.

Na rysunku 1.14 zostało pokazane okno przeglądarki internetowej po tym, gdy użytkownik wyśle adres URL.



Rysunek 1.14. Oddzielenie nagłówków od pozostałych informacji

SPOSÓB 10.

Generowanie stylizowanej wiadomości wykorzystującej plik arkusza stylów

Pozwolenie użytkownikom na wybór przygotowanych stylów dla czytanych przez nich komunikatów

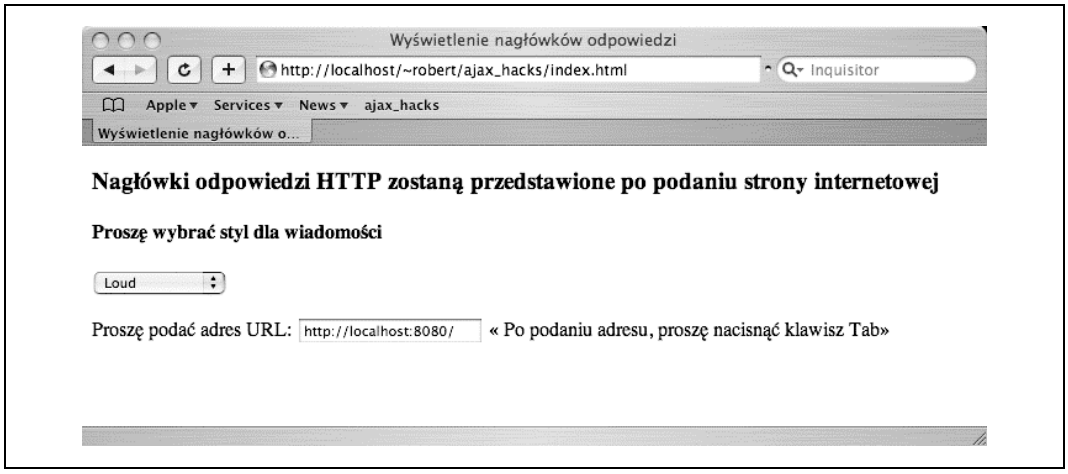
Przedstawiony w tym podrozdziale sposób wysyła żądanie do serwera, który zwraca wiadomość tekstową. Dokonane przez użytkownika wybory określają rzeczywistą zawartość i wygląd wiadomości. Kod HTML przestawionej strony zawiera znacznik `select` wyświetlający listę stylów, które użytkownik może zastosować do wyświetlenia wyników. Na stronie znajduje się również pole tekstowe z częściowym adresem URL, który może zostać uzupełniony i wysłany do serwera.

Informacje zwrotne są powiązane z nagłówkami odpowiedzi zwracanymi przez serwer **[Sposób 9.]**. Jednakże najbardziej interesujące w przedstawionym sposobie jest dynamiczne generowanie wiadomości i przypisanie jej stylu. Poniżej znajduje się kod HTML strony:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="js/hack8.js"></script>
   <script type="text/javascript">
   function setSpan(){
      document.getElementById("instr").onmouseover=function(){
         this.style.backgroundColor='yellow';};
      document.getElementById("instr").onmouseout=function(){
         this.style.backgroundColor='white';};
   }
   </script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Wyświetlenie nagłówków odpowiedzi</title>
   <link rel="stylesheet" type="text/css" href="/parkerriver/css/hacks.css"/>
</head>
<body onload="document.forms[0].url.value=urlFragment;setSpan()">
<h3>Nagłówki odpowiedzi HTTP zostaną przedstawione po podaniu strony
internetowej</h3>
<h4>Proszę wybrać styl dla wiadomości</h4>
<form action="javascript:void%200">
   <p>
      <select name="_style">
         <option label="Loud" value="loud" selected>Krzykliwy</option>
         <option label="Fancy" value="fancy">Fantazyjny</option>
         <option label="Cosmopolitan" value="cosmo">Kosmopolityczny</option>
         <option label="Plain" value="plain">Zwykły</option>
      </select>
   </p>
   <p>Proszę podać adres URL: <input type="text" name="url" size="20" onblur=
      "getAllHeaders(this.value,this.form._style.value)"> <span id=
      "instr" class="message">&#171; Po podaniu adresu, proszę nacisnąć
       klawisz Tab&#187;</span>
   </p>
   <div id="msgDisplay"></div>
</form>
</body>
</html>
```

Celem funkcji `setSpan()` zdefiniowanej wewnątrz znaczników `script` strony internetowej jest podanie pewnych instrukcji („Po podaniu adresu proszę nacisnąć klawisz Tab") na żółtym tle, gdy użytkownik umieści nad nim kursor myszy.

Zanim zostaną omówione niektóre z elementów kodu, Czytelnik może być zainteresowany wyglądem powyższej strony internetowej w przeglądarce. Na rysunku 1.15 zostało pokazane okno przeglądarki z wczytaną stroną.



Rysunek 1.15. Wybór stylu wiadomości

Używane przez tę stronę internetową style CSS pochodzą z pliku arkusza stylów o nazwie *hacks.css*. Kiedy użytkownik dokona wyboru stylu (na przykład „Kosmopolityczny") z listy `select`, poda wartość w polu tekstowym, a następnie naciśnie klawisz *Tab* lub kliknie myszą poza obszarem pola, wówczas wybrany przez użytkownika styl zostanie dynamicznie przypisany pojemnikowi, który przechowuje wiadomość (element `div` z identyfikatorem `id msgDisplay`).

Poniżej został przedstawiony plik arkusza stylów *hacks.css*:

```
div.header{ border: thin solid black; padding: 10%;
   font-size: 0.9em; background-color: yellow; max-width: 80%}

span.message { font-size: 0.8em; }
div { max-width: 80% }

.plain { border: thin solid black; padding: 10%;
   font: Arial, serif font-size: 0.9em; background-color: yellow; }
.fancy { border: thin solid black; padding: 5%;
   font-family: Herculanum, Verdana, serif;
   font-size: 1.2em; text-shadow: 0.2em 0.2em grey; font-style: oblique;
   color: rgb(21,49,110); background-color: rgb(234,197,49)}
.loud { border: thin solid black; padding: 5%; font-family: Impact, serif;
   font-size: 1.4em; text-shadow: 0 0 2.0em black; color: black;
background-color: rgb(181,77,79)}
.cosmo { border: thin solid black; padding: 1%;
   font-family: Papyrus, serif;
   font-size: 0.9em; text-shadow: 0 0 0.5em black; color: aqua;
   background-color: teal}
```

Zaprezentowany powyżej arkusz stylów zawiera definicje kilku klas (`plain`, `fancy`, `loud` i `cosmo`). *Klasa* w arkuszu stylów rozpoczyna się od znaku kropki (na przykład `.fancy`) i definiuje różne właściwości stylu, takie jak rodzina czcionki lub kolor tła. Używając tej techniki, eksperci CSS mogą w jednym miejscu zdefiniować rzeczywiste style, które będą wykorzystywane na wielu stronach internetowych. Oczywiście, doświadczony projektant mógłby zastosować tutaj nieco inne atrybuty stylów, ale trzeba okazać wyrozumiałość!

Powiązany z Ajaksem kod JavaScript na podstawie wyborów użytkownika może przypisać elementom strony przygotowane style. Dlatego też warstwa prezentacyjna aplikacji sieciowej jest oddzielona od warstwy logiki aplikacji lub domeny.

Obsługa zdarzenia `onblur` pola tekstowego wysyła do funkcji o nazwie `getAllHeaders()` wartość w postaci adresu URL oraz nazwy stylu:

```
onblur="getAllHeaders(this.value,this.form._style.value)"
```

Odniesienie `this.form._style.value` jest kodem JavaScript, który przedstawia wartość opcji wybranej z listy `select` (nazwa stylu). Odniesienie `this.value` jest tekstem wprowadzonym przez użytkownika w polu tekstowym.

Przedstawiony poniżej kod JavaScript pochodzi z importowanego przez stronę pliku *hack8.js*. Fragment kodu odpowiedzialny za dynamiczne przypisanie stylu do wyświetlanej wiadomości został pogrubiony:

```
var request;
var urlFragment="http://localhost:8080/";
var st;

function getAllHeaders(url,styl){
   if(url){
      st=styl;
   httpRequest("GET",url,true);
   }
}

// Obsługa zdarzeń dla obiektu XMLHttpRequest.
function handleResponse(){
   try{
      if(request.readyState == 4){
         if(request.status == 200){
            /* Wszystkie nagłówki są otrzymane jako pojedynczy ciąg tekstowy. */
            var headers = request.getAllResponseHeaders();
            var div = document.getElementById("msgDisplay");
            div.className= st == "" ? "header" : st;
            div.innerHTML="<pre>"+headers+"</pre>";
         } else {
            // Jeżeli aplikacja nie jest dostępna, wtedy stan żądania wynosi 503,
            // natomiast w przypadku błędu w aplikacji stan żądania wynosi 500.
            alert(request.status);
            alert("Wystąpił problem z komunikacją między obiektem
            XMLHttpRequest, "+
               "a programem serwera.");
         }
```

```
        }// Koniec zewnętrznej pętli if.
    } catch (err) {
        alert("Serwer nie jest dostępny "+
            "dla tej aplikacji. Proszę wkrótce spróbować"+
            " ponownie. \nBłąd: "+err.message);
    }
}
/* Należy wrócić do sposobów 01, 02 i innych, aby zobaczyć kod funkcji httpRequest()
i initReq(), który został tutaj pominięty w celu zapewnienia zwięzłości kodu. */
```

Wyjątkowo proste

Funkcja `getAllHeaders()` ustawia zmiennej najwyższego poziomu `st` nazwę klasy stylu CSS (`plain`, `fancy`, `loud` lub `cosmo`). Następnie kod w wyjątkowo prosty sposób ustawia właściwość `className` elementu `div` przechowującego wiadomość, co powoduje zmianę stylu przypisanego do tej wiadomości:

```
if(request.status == 200){
    /* Wszystkie nagłówki są otrzymane jako pojedynczy ciąg tekstowy. */
    var headers = request.getAllResponseHeaders();
    var div = document.getElementById("msgDisplay");
    div.className= st == "" ? "header" : st;
    div.innerHTML="<pre>"+headers+"</pre>";
}
```

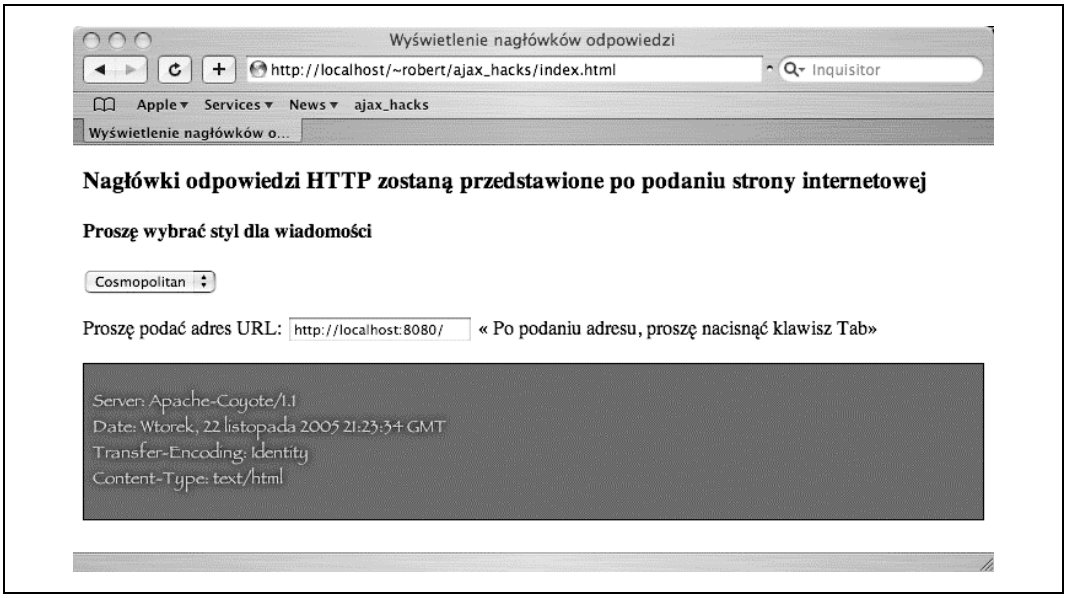
Jeżeli z pewnych powodów wybrana nazwa klasy pochodzącej od klienta sieciowego jest pustym ciągiem tekstowym (nie powinno się tutaj zdarzyć, ponieważ znacznik `select` zawiera jedynie kompletne ciągi tekstowe), wówczas elementowi `div` zostanie przypisana domyślna nazwa klasy stylu — `header`.

Kod JavaScript potencjalnie może zostać zaimportowany do *innej* strony internetowej klienta, tak więc powinny zostać umieszczone pewne mechanizmy sprawdzania wartości danych wejściowych.

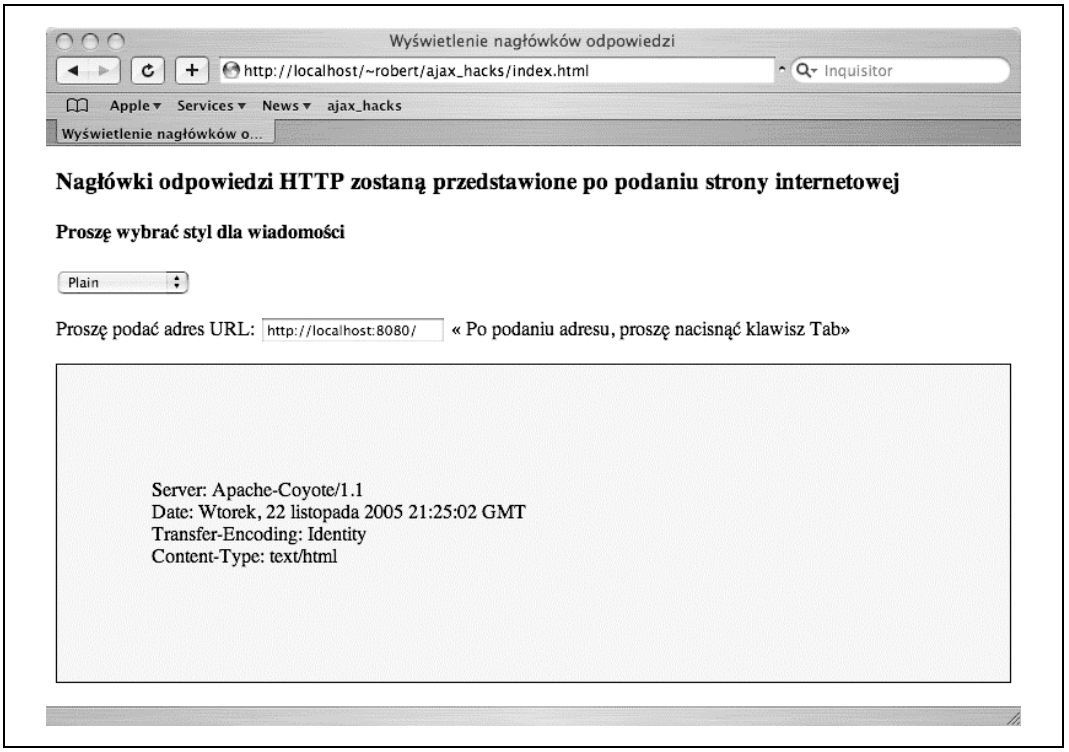
Klasa `header` została również zdefiniowana w pliku *hacks.css*.

Na poniższych rysunkach zostały pokazane przykłady tej samej wiadomości, której użytkownik przypisał różne style. Rysunek 1.16 pokazuje stronę, gdy użytkownik wybierze styl „Kosmopolityczny”.

Natomiast na rysunku 1.17 została pokazana ta sama wiadomość, tym razem wybrany został styl „zwykły”.



Rysunek 1.16. Wiadomość z wybranym stylem kosmopolitycznym



Rysunek 1.17. Niestety, wybrano styl zwykły dla wiadomości

SPOSÓB 11. Generowanie wiadomości stylizowanej „w locie”

Dynamiczne definiowanie i przypisywanie stylów CSS do zawartości strony internetowej

Programowanie z użyciem JavaScript i DOM pozwala na definiowanie atrybutów stylów CSS i zastosowanie ich od podstaw do elementów strony internetowej. Przykładem sytuacji, w której może zaistnieć potrzeba zaimplementowania tych metod, jest strona Wiki, pozwalająca użytkownikom na programowanie własnych projektów stron i stylów.

W większości przypadków oddzielenie definicji stylów od kodu JavaScript jest pożądanym działaniem. Oddzielenie od siebie warstw aplikacji pozwala na niezależne rozwijanie poszczególnych elementów, co powoduje, że programowanie sieciowe staje się mniej złożone, a bardziej efektywne.

W omawianym sposobie, podobnie jak w poprzednich, dynamiczne wyświetlanie informacji z serwera następuje na podstawie wybranego przez użytkownika stylu. Jednak w przeciwieństwie do poprzedniego sposobu, tutaj style zostają umieszczone w kodzie, a następnie wybrany styl jest stosowany w elemencie HTML. Poniżej znajduje się kod strony, na którym elementy dotyczące stylu zostały pogrubione:

```
var request;
var urlFragment="http://localhost:8080/";
var st;

function getAllHeaders(url,styl){
   if(url){
      st=styl;
      httpRequest("GET",url,true);
   }
}

/* Ustawienie jednego lub więcej atrybutów stylu CSS na elemencie DOM
 CSS2Properties Object.
Parametry:
  stType oznacza nazwę stylu, na przykład 'plain,''fancy,''loud,' lub 'cosmo'.
  stylObj jest właściwością stylu elementu HTML, na przykład div.style. */

function setStyle(stType,stylObj){
   switch(stType){
      case 'plain' :
         stylObj.maxWidth="80%";
         stylObj.border="thin solid black";
         stylObj.padding="5%"
         stylObj.textShadow="none";
         stylObj.fontFamily="Arial, serif";
         stylObj.fontSize="0.9em";
         stylObj.backgroundColor="yellow"; break;
      case 'loud' :
         stylObj.maxWidth="80%";
         stylObj.border="thin solid black";
         stylObj.padding="5%"
         stylObj.fontFamily="Impact, serif";
         stylObj.fontSize="1.4em";
         stylObj.textShadow="0 0 2.0em black";
         stylObj.backgroundColor="rgb(181,77,79)"; break;
      case 'fancy' :
         stylObj.maxWidth="80%";
```

```
            stylObj.border="thin solid black";
            stylObj.padding="5%"
            stylObj.fontFamily="Herculanum, Verdana, serif";
            stylObj.fontSize="1.2em";
            stylObj.fontStyle="oblique";
            stylObj.textShadow="0.2em 0.2em grey";
            stylObj.color="rgb(21,49,110)";
            stylObj.backgroundColor="rgb(234,197,49)"; break;
         case 'cosmo' :
            stylObj.maxWidth="80%";
            stylObj.border="thin solid black";
            stylObj.padding="1%"
            stylObj.fontFamily="Papyrus, serif";
            stylObj.fontSize="0.9em";
            stylObj.textShadow="0 0 0.5em black";
            stylObj.color="aqua";
            stylObj.backgroundColor="teal"; break;
         default :
            alert('default');
      }
   }

   // Obsługa zdarzeń dla obiektu XMLHttpRequest.
   function handleResponse(){
      try{
         if(request.readyState == 4){
            if(request.status == 200){
               /* Wszystkie nagłówki są otrzymane jako pojedynczy ciąg tekstowy. */
               var headers = request.getAllResponseHeaders();
               var div = document.getElementById("msgDisplay");
               if(st){
                  setStyle(st,div.style);
               } else {
                  setStyle("plain",div.style);
               }
               div.innerHTML="<pre>"+headers+"</pre>";
            } else {
               // Jeżeli aplikacja nie jest dostępna, wtedy stan żądania wynosi 503,
               // natomiast w przypadku błędu w aplikacji stan żądania wynosi 500.
               alert(request.status);
               alert("Wystąpił problem z komunikacją między obiektem
               XMLHttpRequest, "+
                  "a programem serwera.");
            }
         }// Koniec zewnętrznej pętli if.
      } catch (err) {
         alert("Serwer nie jest dostępny "+
            "dla tej aplikacji. Proszę wkrótce spróbować"+
            " ponownie. \nBłąd: "+err.message);
      }
   }

   /* Inicjalizacja obiektu żądania, który został już skonstruowany. */
   function initReq(reqType,url,bool){
      try{
         /* Określenie funkcji, która będzie obsługiwała odpowiedź HTTP. */
         request.onreadystatechange=handleResponse;
         request.open(reqType,url,bool);
         request.send(null);
      } catch (errv) {
         alert(
            "Aplikacja nie może w tej chwili nawiązać połączenia z serwerem."+
```

```
            " Proszę wkrótce spróbować ponownie.");
    }
}

/* Funkcja opakowująca do skonstruowania obiektu żądania.
Parametry:
    reqType: typ żądania HTTP, na przykład GET lub POST.
    url: adres URL programu serwerowego.
    asynch: czy żądanie będzie wysłane asynchronicznie, czy też nie. */
function httpRequest(reqType,url,asynch){
    // Przeglądarki na bazie Mozilli.
    if(window.XMLHttpRequest){
       request = new XMLHttpRequest();
    } else if (window.ActiveXObject){
       request=new ActiveXObject("Msxml2.XMLHTTP");
       if (! request){
          request=new ActiveXObject("Microsoft.XMLHTTP");
       }
    }

    // Jeżeli nie powiodła się nawet inicjalizacja ActiveXObject,
    // wówczas żądanie wciąż może być typu null.
    if(request){
       initReq(reqType,url,asynch);
    } else {
       alert("Używana przeglądarka nie pozwala na wykorzystanie "+
          "wszystkich funkcji tej aplikacji!");
    }
}
```

Odłożenie na bok arkusza stylów

Każdy element na przedstawionej stronie internetowej posiada właściwość `style`, jeżeli przeglądarka internetowa obsługuje arkusze stylów CSS. Przykładowo, element `div` posiada właściwość o nazwie `div.style`, która pozwala, aby kod JavaScript ustawił jej atrybuty stylu dla tego elementu `div` (na przykład `div.style.fontFamily="Arial"`). Jest to sposób działania funkcji `setStyle()` z wcześniejszego przykładu. Dwoma parametrami funkcji są nazwa stylu, na przykład „fancy” (wybrana z przygotowanej listy), oraz właściwość `style` określonego elementu `div`. Następnie funkcja ustawia wygląd elementu HTML `div` na stronie internetowej.

Pojawiające się na stronie informacje (zestaw nagłówków odpowiedzi) pochodzi z serwera dzięki obiektowi żądania. Podobnie jak miało to miejsce w poprzednim sposobie, użytkownik uzupełnia adres URL, a następnie klika myszą poza obszarem pola tekstowego lub naciska klawisz *Tab*. Ten krok powoduje wywołanie obsługi zdarzenia `onblur`, które w trakcie działania ustawia obiekt żądania oraz style CSS. Kod HTML omawianej strony nie różni się zbytnio od kodu strony przedstawionej w podrozdziale „Generowanie stylizowanej wiadomości wykorzystującej plik arkusza stylów” **[Sposób 10.]**, ale nie zawiera łącza do arkusza stylów. Wszystkie style w bieżącym sposobie są zdefiniowane przez zaimportowanie pliku JavaScript o nazwie *hack10.js*. Poniżej został przedstawiony kod strony:

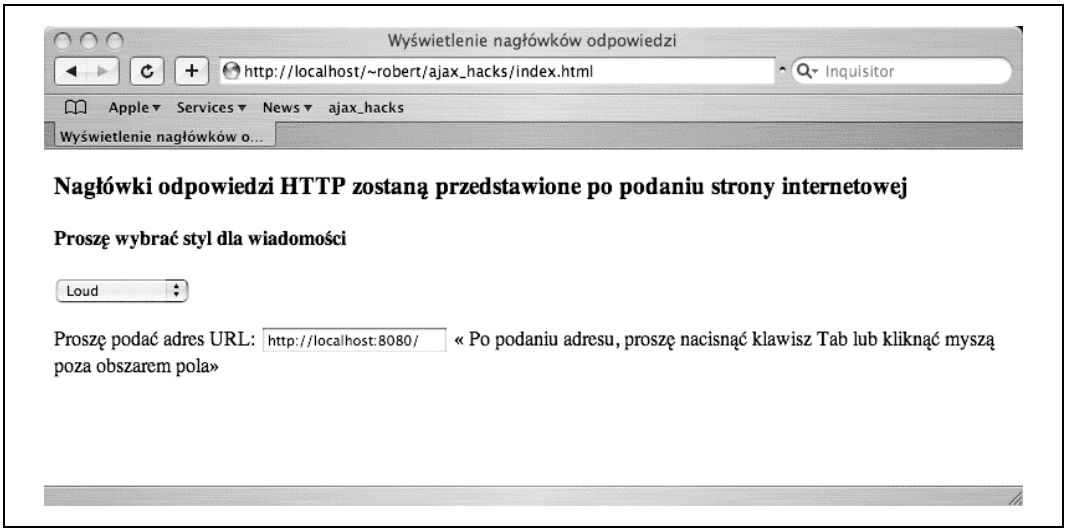
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
   <script type="text/javascript" src="/parkerriver/js/hack10.js"></script>
   <script type="text/javascript">
   function setSpan(){
      document.getElementById("instr").onmouseover=function(){
         this.style.backgroundColor='yellow';};
      document.getElementById("instr").onmouseout=function(){
         this.style.backgroundColor='white';};
   }
   </script>
   <meta http-equiv="content-type" content="text/html; charset=iso-8859-2">
   <title>Wyświetlenie nagłówków odpowiedzi</title>
</head>
<body onload="document.forms[0].url.value=urlFragment;setSpan()">
<h3>Nagłówki odpowiedzi HTTP zostaną przedstawione po podaniu strony
internetowej</h3>
<h4>Proszę wybrać styl dla wiadomości</h4>
<form action="javascript:void%200">
   <p>
      <select name="_style">
         <option label="Loud" value="loud" selected>Krzykliwy</option>
         <option label="Fancy" value="fancy">Fantazyjny</option>
         <option label="Cosmopolitan" value="cosmo">Kosmopolityczny</option>
         <option label="Plain" value="plain">Zwykły</option>
      </select>
   </p>
   <p>Proszę podać adres URL: <input type="text" name="url" size="20" onblur=
      "getAllHeaders(this.value,this.form._style.value)">
      <span id="instr" class="message">&#171; Po podaniu adresu
      proszę nacisnąć klawisz Tab lub kliknąć myszą poza obszarem pola&#187;
      </span>
   </p>
   <div id="msgDisplay"></div>
</form>
</body>
</html>
```

Obsługa zdarzenia `onblur` — funkcja `getAllHeaders()` przekazuje aplikacji nazwę stylu wybranego przez użytkownika z listy `select` (na przykład „cosmo”), jak również adres URL komponentu serwera. Jedynym celem komponentu serwera jest dostarczenie wartości przeznaczonej do wyświetlenia. Nas głównie interesuje dynamiczne generowanie stylów dla każdego typu informacji serwera, które aplikacja może otrzymać za pomocą Ajaksu i obiektu żądania.

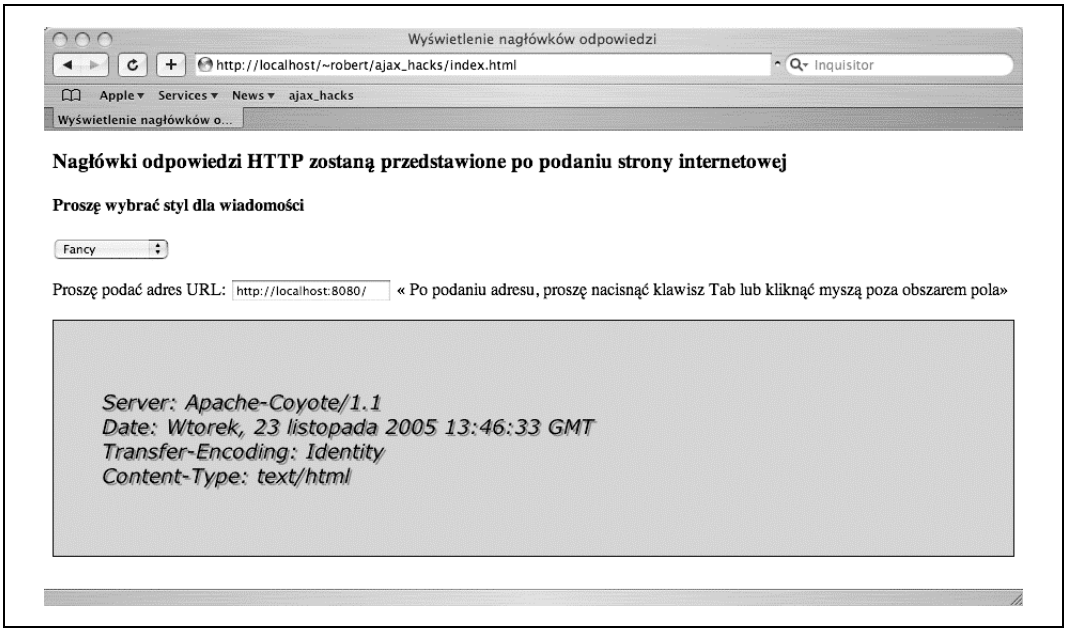
Celem funkcji `setSpan()` zdefiniowanej wewnątrz znaczników `script` strony internetowej jest podanie pewnych instrukcji („Po podaniu adresu proszę nacisnąć klawisz Tab lub kliknąć myszą poza obszarem pola”) na żółtym tle, gdy użytkownik umieści nad nim kursor myszy.

Na rysunku 1.18 został pokazany wygląd strony w przeglądarce internetowej, przed wysłaniem żądania HTTP.

Natomiast wygląd strony, gdy użytkownik opcjonalnie wybierze nazwę stylu, uzupełni adres URL i naciśnie klawisz *Tab*, został pokazany na rysunku 1.19.



Rysunek 1.18. Wybór stylu, który będzie dynamicznie wygenerowany



Rysunek 1.19. Dane serwera z zastosowanym na nich stylem

Żadna z przedstawionych powyżej stron internetowych nie wymaga oczekiwania na przesłanie nowej wersji strony z serwera. Dane z serwera są przynoszone w tle przez obiekt żądania, a kod JavaScript po stronie klienta powoduje zastosowanie stylów na wyświetlanych informacjach. W tym kryje się potęga technologii Ajax!