ASP.NET 3.5. Tworzenie portali internetowych w nurcie Web 2.0

Autor: Omar Al Zabir
Tłumaczenie: Marek Pałczyński
ISBN: 978-83-246-1841-5
Tytuł oryginału: Building a Web 2.0 Portal with ASP.NET 3.5
Stron: 320

Poznaj sekrety zaawansowanych technologii budowy portali internetowych Web 2.0

- Jak zaprojektować witrynę dla platformy ASP.NET i ASP.NET AJAX?
- Jak rozbudować serwis zgodnie z zasadami ergonomii?
- Jak zwiększyć wydajność serwera?

Portale sieciowe Web 2.0, opierające się na technologii AJAX, umożliwiają użytkownikom personalizowanie stron, a także agregowanie danych z różnych źródeł. Wszystko to sprawia, że są doskonałymi serwisami korporacyjnymi i należą do najefektywniejszych aplikacji sieciowych. Zastosowanie mechanizmów AJAX pozwala na udostępnienie interaktywnego i rozbudowanego interfejsu, działającego znacznie szybciej i bardziej wydajnie niż w tradycyjnych serwisach. Natomiast wykorzystanie widżetów (komponentów typu plag-and-play) zapewnia przejrzystość architektury portalu i łatwość jego rozbudowy, ponieważ są one opracowywane niezależnie od warstwy rdzeniowej systemu.

Książka „ASP.NET 3.5. Tworzenie portali internetowych w nurcie Web 2.0" zawiera opis najnowszych metod i technologii projektowania oraz budowy portali z wykorzystaniem platformy ASP.NET i środowiska ASP.NET AJAX. W podręczniku przedstawiono także praktyczne rozwiązania problemów związanych z projektowaniem, wdrażaniem, utrzymaniem, a także skalowaniem i usprawnianiem serwisu. Dzięki tej pozycji poznasz poszczególne fazy budowy prototypowego portalu, zaawansowane techniki technologii AJAX oraz sposoby optymalizacji kodu. Nauczysz się m. in. przygotowywać widżety klienckie za pomocą kodu JavaScript, tworzyć własne mechanizmy obsługi wywołań, zwiększać wydajność serwera i skalowalność usług sieciowych. Zdobędziesz zatem całą potrzebną Ci wiedzę i umiejętności, które pozwolą zbudować stabilny, nowoczesny i bezpieczny portal internetowy.

- Wprowadzenie do budowy portali internetowych
- Architektura portali i widżetów
- Projekt warstwy sieciowej w środowisku ASP. NET AJAX
- Projekt warstwy danych i warstwy biznesowej na platformie NET 3.5
- Widżety klienckie
- Optymalizacja pracy środowiska ASP.NET AJAX
- Tworzenie asynchronicznych i transakcyjnych usług sieciowych z uwzględnieniem buforowania danych
- Skalowalność usług sieciowych
- Zwiększenie wydajności serwera i klienckiej części aplikacji
- Zarządzanie witryną

Zaprojektuj bardzo wydajną i supernowoczesną witrynę internetową

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

Spis treści

ROZDZIAŁ 5.

Widżety klienckie

W rozdziale 3. zostały opisane zagadnienia związane z budowaniem widżetów serwerowych — widżetu czytnika RSS (lub Atom) oraz widżetu fotograficznego Flickr. Zaletą stosowania widżetów serwerowych jest to, że można wykorzystywać komfortowe środowisko pracy (oprogramowania Visual Studio) do ich tworzenia i debugowania, używając przy tym swojego ulubionego języka programowania (C# lub VB.NET). Jednak widżety serwerowe opóźniają ładowanie strony i wymagają częstego odsyłania danych. Wszystkie widżety widoczne na stronie muszą zostać załadowane po stronie serwera podczas pierwszego pobierania strony oraz w czasie jej asynchronicznych uaktualnień. Jeśli widżety pobierają dane z zewnętrznych źródeł, czas pobierania strony zależy od łącznego czasu przygotowania każdej z nich. Ponadto widżety serwerowe wymagają odsyłania danych nawet podczas nieskomplikowanych operacji takich jak stronicowanie lub edytowanie pozycji na liście. Przekazywanie danych jest konieczne, ponieważ po stronie serwera jest przechowywany model obiektu, który z kolei jest powiązany z informacjami zapisanymi w bazie danych. Po stronie klienta nie są przechowywane żadne informacje, które mogłyby usprawnić niektóre operacje realizowane w przeglądarce. Zatem mimo że widżety serwerowe znacznie ułatwiają opracowywanie kodu i zarządzanie nim, charakteryzują się niższą wydajnością pracy w porównaniu z widżetami klienckimi.

Widżety klienckie bazują przede wszystkim na technologii JavaScript, dzięki czemu zapewniają znacznie wyższy poziom interaktywności i funkcjonalności w operacjach niewymagających odsyłania danych. Widżety klienckie pobierają dane z zewnętrznych źródeł bezpośrednio z poziomu skryptu JavaScript i przechowują model obiektu oraz informacje o stanie obiektu po stronie klienta. Dzięki temu realizują zadania stronicowania, edycji czy sortowania bezpośrednio w przeglądarce bez odsyłania danych do serwera. Co więcej, widżety klienckie mogą buforować zewnętrzne informacje w pamięci przeglądarki, więc przygotowywanie strony podczas kolejnych wizyt użytkownika może przebiegać znacznie szybciej niż w przypadku widżetów serwerowych. Dane niezbędne do wyświetlenia elementu są bowiem przechowywane w przeglądarce. W tym rozdziale zostanie omówione zagadnienie opóźnienia ładowania widżetów serwerowych w celu przyspieszenia ładowania strony. Rozwiązanie to zostanie przedstawione na przykładzie dwóch klienckich widżetów — czytnika danych RSS oraz komponentu wyświetlającego fotografie serwisu Flickr. Opisana zostanie także procedura budowy pośredniczącej usługi sieciowej, która pozwoli widżetom klienckim na pobieranie danych z zewnętrznych źródeł i buforowanie ich w pamięci przeglądarki.

Opóźnienie ładowania widżetów serwerowych

Podczas interpretowania skryptu strony na serwerze konieczne jest wykonanie kodu wszystkich zawartych na niej widżetów. Powoduje to znaczne opóźnienie w dostarczaniu treści zarówno podczas pierwszej wizyty, jak i w czasie kolejnych wizyt oraz w przypadku przełączania zakładek. Widżety pobierają dane z zewnętrznych źródeł lub serwerowej bazy danych w kodzie zdarzenia `Page_Load`. Wywołanie tego zdarzenia w każdym z widżetów (które funkcjonują w taki sam sposób jak kontrolki sieciowe) okazuje się czasochłonne. Aby zwiększyć odczuwalną szybkość ładowania strony, trzeba dostarczyć do przeglądarki szkielet widżetów wraz z komunikatami informującymi o pobieraniu danych, a następnie stopniowo wypełniać poszczególne kontrolki właściwą treścią.

Rozwiązanie to jest wykorzystywane na przykład w portalu Pageflakes, w którym najpierw ładuje się szablon widżetów, a w obszarze każdej z kontrolek wyświetla się komunikat o trwającym procesie pobierania informacji. Każdy z widżetów wywołuje usługę sieciową i pobiera dane niezbędne do wyświetlenia swojej zawartości. Mimo że całkowity czas ładowania strony jest dość długi (każdemu widżetowi odpowiada co najmniej jedno wywołanie usługi sieciowej), subiektywne odczucie użytkownika okazuje się korzystniejsze, ponieważ może on obserwować cały proces. W rezultacie załadowanie kolejno wszystkich widżetów na stronę jest postrzegane jako znacznie szybszy proces niż w klasycznym mechanizmie generowania dokumentu.

Opóźnione ładowanie oznacza, że widżety nie pobierają danych w kodzie zdarzenia `Page_Load`, ale w procedurze asynchronicznej aktualizacji wyzwalanej przez komponent stopera. Widżet najpierw wyświetla komunikat o postępie ładowania, a następnie wykorzystuje obiekt `Timer` do wyzwolenia asynchronicznej aktualizacji. Z kolei w czasie asynchronicznej aktualizacji komponent pobiera informacje z zewnętrznych źródeł danych i przygotowuje kod wynikowy. Technika ta eliminuje problem wstrzymywania wykonywania procedury `Page_Load` w oczekiwaniu na dostarczenie zewnętrznych danych. Czas ładowania całej strony nie jest wówczas uzależniony od szybkości gromadzenia danych z zewnętrznych źródeł. Jednym z najłatwiejszych sposobów implementacji omawianego rozwiązania jest zastosowanie kontrolki `MultiView`, która będzie zawierała widok z komunikatem o postępie prac oraz widok obejmujący główny interfejs użytkownika widżetu. Kontrolka `Timer` może zainicjować odesłanie danych po pewnym czasie (na przykład 100 ms), które z kolei spowoduje zmianę widoku na właściwy interfejs użytkownika oraz wyłączenie stopera.

Opóźnienie ładowania widżetu RSS (Atom)

Pierwszy etap prac polega na przekształceniu widżetu RSS w taki sposób, aby opóźnił ładowanie treści, oraz na podzieleniu procedury ładowania danych na dwie fazy. Po modyfikacji strona będzie pobierana bardzo szybko. Jednak w miejscu widżetów wyświetli się komunikat informujący o ładowaniu danych. Poszczególne widżety będą ładowane kolejno jeden po drugim. Interfejs użytkownika widżetu trzeba więc podzielić na dwa widoki, obsługiwane przez kontrolkę `MultiView`. Pierwszy z nich obejmuje jedynie komunikat o postępie prac. Natomiast drugi wykorzystuje kontrolkę `DataList` o nazwie `FeedList`. Kod stosownego skryptu został przedstawiony w listingu 5.1.

Listing 5.1. Widżet RSS z opóźnionym ładowaniem podzielony na dwa widoki.

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="RSSWidget.ascx.cs"
Inherits="Widgets_RSSWidget" EnableViewState="false" %>
<asp:Panel ID="SettingsPanel" runat="Server" Visible="False" >
...
</asp:Panel>

<asp:MultiView ID="RSSMultiview" runat="server" ActiveViewIndex="0">

<asp:View runat="server" ID="RSSProgressView">
    <asp:image runat="server" ID="image1" ImageAlign="middle"
        ImageUrl="~/indicator.gif" />
    <asp:Label runat="Server" ID="label1" Text="Loading..." Font-Size="smaller"
        ForeColor="DimGray" />
</asp:View>

<asp:View runat="server" ID="RSSFeedView">

    <asp:DataList ID="FeedList" runat="Server" EnableViewState="False">
    <ItemTemplate>
    <asp:HyperLink ID="FeedLink" runat="server" Target="_blank"
        CssClass="feed_item_link" NavigateUrl='<%# Eval("link") %>'
        ToolTip='<%# Eval("description") %>'>
    <%# Eval("title") %>
    </asp:HyperLink>
    </ItemTemplate>
    </asp:DataList>

</asp:View>

</asp:MultiView>

<asp:Timer ID="RSSWidgetTimer" Interval="1" OnTick="LoadRSSView" runat="server" />
```

Kontrolka `Timer` wywołuje serwerową funkcję `LoadRSSView` w sposób asynchroniczny. Funkcja z kolei zmienia bieżący widok na `RSSFeedView` i wyświetlane informacje RSS zgodnie z kodem zamieszczonym w przykładzie 5.2. Definicja funkcji znajduje się w pliku kodu towarzyszącym kontrolce sieciowej czytnika RSS.

Listing 5.2. Funkcja LoadRSSView widżetu RSS wyzwalana przez kontrolkę Timer.

```
protected void LoadRSSView(object sender, EventArgs e)
{
    this.ShowFeeds( );
    this.RSSMultiview.ActiveViewIndex = 1;
    this.RSSWidgetTimer.Enabled = false;
}
```

Po wprowadzeniu zmian w czasie pierwszego ładowania funkcja `Page_Load` nie wykonuje żadnych operacji. Jej zadanie polega na pobieraniu danych RSS jedynie w czasie asynchronicznych aktualizacji. Ponieważ procedura obsługi zdarzenia `Page_Load` kończy się bezzwłocznie, czas ładowania widżetu nie zależy od czasu pobrania informacji z zewnętrznych źródeł danych. Kod rozwiązania został przedstawiony w listingu 5.3.

Listing 5.3. Podczas pierwszego ładowania zdarzenie Page_Load nie powoduje wykonania jakichkolwiek instrukcji.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!this._Host.IsFirstLoad) this.LoadRSSView(sender, e);
}
```

Procedura obsługi zdarzenia `Page_Load` wywołuje funkcję `LoadRSSView` jedynie w przypadku aktualizacji. Wówczas kod `LoadRSSView` jest wykonywany bardzo szybko, ponieważ dane zostają zapisane w pamięci podręcznej środowiska ASP.NET.

Opóźnienie ładowania widżetu Flickr

Procedura opóźnienia ładowania widżetu Flickr jest analogiczna do procedury opóźnienia ładowania widżetu RSS. Do wyświetlenia komunikatu o postępie prac trzeba zastosować kontrolkę `MultiView`, która jako drugi widok wyświetli załadowane przez stronę zdjęcia. W czasie pierwszego pobierania dokumentu kod procedury `Page_Load` nie powinien wykonywać żadnych czynności, więc nie spowoduje opóźnienia w dostarczaniu strony. Po przekazaniu komunikatu do przeglądarki kontrolka `Timer` powinna wywołać operację asynchronicznej aktualizacji. Wówczas strumień plików zdjęciowych zostanie pobrany z serwisu Flickr i zinterpretowany przez kod interfejsu użytkownika.

Problemy wynikające z opóźnienia ładowania widżetów

Choć czas pobierania strony wydaje się krótszy, w rzeczywistości całkowity czas ładowania danych jest znacznie dłuższy, ponieważ każdy widżet musi wykonać co najmniej jedną asynchroniczną aktualizację. Ponadto technika ta wiąże się ze znacznym obciążeniem skryptu *Default.aspx* ze względu na konieczność asynchronicznego aktualizowania danych podczas pierwszego ładowania. Skrypt *Default.aspx* nie jest przetwarzany raz, ale *n* razy przy *n* widżetach o opóźnionym ładowaniu danych. Asynchroniczne aktualizacje bazują na żądaniach HTTP POST, więc nie ma możliwości buforowania informacji pobranych z zewnętrznych źródeł w pamięci podręcznej przeglądarki. Nawet jeśli informacje w danym kanale RSS nie zmienią się przez tydzień, będą musiały zostać przesłane z serwera podczas każdej asynchronicznej aktualizacji wykonanej w tym okresie. Dane nie są buforowane w przeglądarce, więc czas kolejnego ładowania widżetu nie ulega skróceniu.

Na rysunku 5.1 został przedstawiony zapis asynchronicznych aktualizacji z odwołaniem do strony *Default.aspx* wykonanych przez cztery widżety RSS z zaimplementowaną funkcją opóźnionego ładowania. We wszystkich przypadkach przesyłane są żądania HTTP POST.

Podczas kolejnych wizyt wykonanie tych samych czterech asynchronicznych aktualizacji kończy się zwróceniem identycznych wyników, ponieważ informacje w kanale RSS nie zmieniają się zbyt często. Nie ma jednak możliwości zbuforowania odpowiedzi i wyeliminowania w ten sposób niepotrzebnych odwołań. Zgodnie z wcześniejszym stwierdzeniem asynchroniczne aktualizacje, jako żądania HTTP POST, nie podlegają rejestracji w pamięci podręcznej.

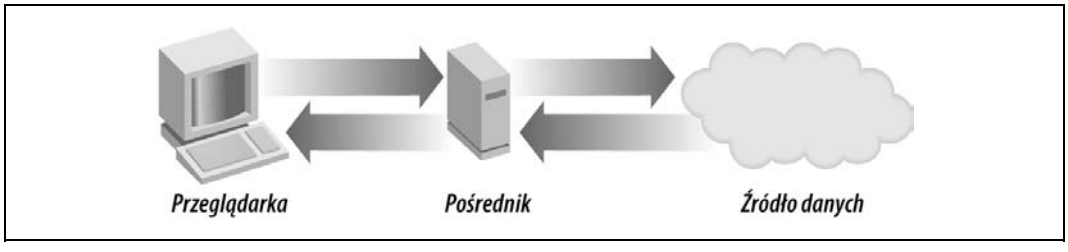
Aby skrócić czas ładowania widżetów w czasie kolejnych wizyt, należy pobrać dane z przeglądarki z wykorzystaniem żądania HTTP GET. Niezbędne jest w tym przypadku wygenerowanie takich nagłówków odpowiedzi, które wskażą dane w pamięci podręcznej przeglądarki. Do pobrania informacji z pierwotnego źródła danych trzeba wykorzystać skrypt JavaScript. Musi on również odpowiednio zinterpretować pobrane dane i przygotować właściwy kod HTML. A ponieważ nie można użyć serwerowych mechanizmów generowania kodu HTML, trzeba zaprojektować skrypty klienckie w taki sposób, aby wykorzystywały dane z wcześniejszych wizyt na stronie.

Rysunek 5.1. Odpowiedź na żądanie asynchronicznego uaktualnienia po uwzględnieniu opóźnienia w ładowaniu widżetu.

Jednak przeglądarki nie pozwalają na międzydomenowe odwołania do usług sieciowych. Nie można więc zastosować techniki XML HTTP do bezpośredniego pobierania danych z zewnętrznych domen. Niedopuszczalne jest na przykład załadowanie dokumentu XML wprost spod adresu *http://msdn.microsoft.com/rss.xml*. Możliwe jest natomiast wywołanie jednej z własnych usług sieciowych, która będzie pełniła rolę pośrednika (proxy) w dostępie do oryginalnego źródła danych. W następnym podrozdziale zostanie przedstawiony sposób przygotowania wspomnianego pośrednika, który będzie pobierał informacje spod zewnętrznych adresów URL i realizował zadania związane z inteligentnym buforowaniem danych.

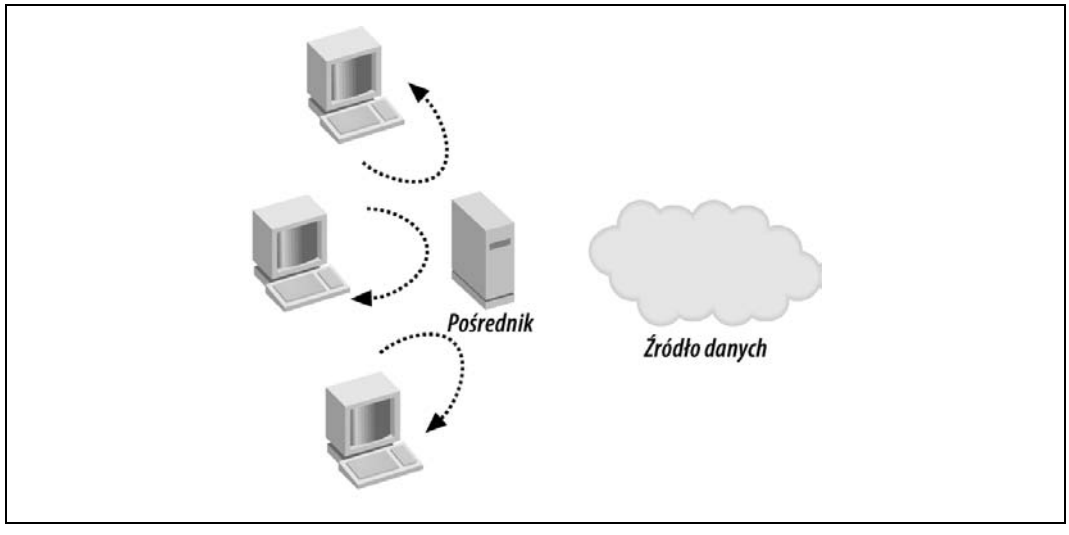
Pośrednik w dostępie do danych

Pośrednik w dostępie do danych jest usługą sieciową, pracującą na jednym z serwerów serwisu, która może pobierać informacje spod zewnętrznych adresów URL i przekazywać je do przeglądarki (rysunek 5.2).



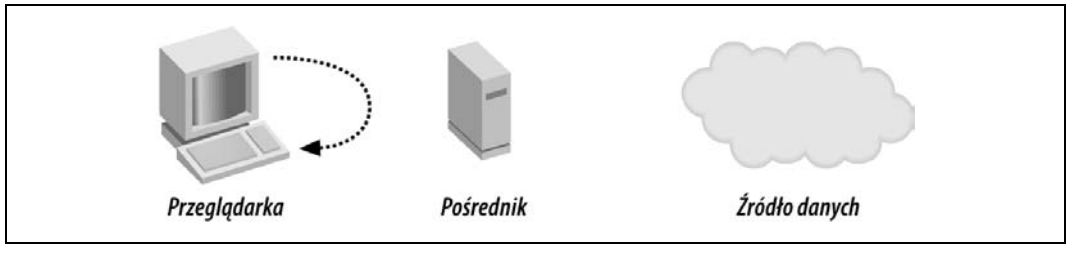
Rysunek 5.2. Przeglądarka wysyła żądanie do usługi pośrednika, a ta pobiera informacje ze źródła danych.

Usługa pośrednika może zbuforować dane na serwerze i na pewien czas wyeliminować konieczność ponawiania wywołań pod ten sam adres URL. Na przykład jeśli stu użytkowników korzysta z tego samego kanału RSS, a informacje w kanale nie zmieniają się przez wiele dni, usługa pośrednika może zbuforować dane pochodzące ze pierwotnego źródła na jeden dzień i obsługiwać setki lub tysiące zapytań bezpośrednio z wykorzystaniem danych zgromadzonych w pamięci serwera. Opisany mechanizm został zilustrowany na rysunku 5.3.



Rysunek 5.3. Usługa pośrednika buforuje dane w pamięci serwera i uniemożliwia wywoływanie zewnętrznej usługi przez wielu użytkowników.

Buforowanie danych po stronie serwera znacznie zwiększa szybkość pobierania informacji, ponieważ ogranicza transmisję sieciową do pojedynczego odwołania. Serwer nie musi komunikować się z zewnętrznym źródłem danych. Pośrednik może dodatkowo wygenerować nagłówki, które poinformują przeglądarkę o obowiązku zbuforowania odpowiedzi na określony czas. Przed upływem tego czasu ewentualne odwołania do pośrednika będą zastępowane pobraniem danych z pamięci podręcznej, co jest wyjątkowo szybkie. Zatem kolejne odwołania do serwera proxy w celu pobrania tych samych danych nie będą wcale wymagały przesyłania żądań przez sieć, tak jak to zostało pokazane na rysunku 5.4.



Rysunek 5.4. Jeśli odpowiedź jest zbuforowana w pamięci przeglądarki, żądanie nie zostaje dostarczone do pośrednika, przez co nie powoduje wygenerowania jakiegokolwiek ruchu sieciowego.

Oznacza to, że jeśli usługa pośrednika pobierze dane RSS i wymusi zbuforowanie ich w przeglądarce na godzinę, użytkownik będzie mógł przejść do innego serwisu, a po powrocie widżet RSS natychmiast wyświetli swoją treść bez odwoływania się do serwera. Jeżeli więc dana

strona byłaby złożona z samych widżetów RSS, cała jej treść zostałaby załadowana po jednym odwołaniu do skryptu *Default.aspx*. Wszystkie pozostałe informacje byłyby zarejestrowane w pamięci podręcznej przeglądarki. Zasada wykorzystania mechanizmu buforowania danych po stronie klienta do zwiększenia szybkości pobierania danych RSS została opisana w rozdziale 9.

Usługa sieciowa pośrednika w dostępie do danych

Usługa pośrednika w dostępie do danych została zdefiniowana w pliku *Proxy.asmx* i obejmuje trzy metody:

`GetString(url, cacheDuration)`
: Metoda ta zwraca dane spod określonego adresu URL w formie ciągu tekstowego i buforuje je po stronie przeglądarki na określony czas (`cacheDuration`).

`GetXml(url, cacheDuration)`
: Metoda ta zwraca dokument XML pobrany spod określonego adresu URL i buforuje go po stronie przeglądarki na określony czas (`cacheDuration`).

`GetRss(url, count, cacheDuration)`
: Metoda ta pobiera spod określonego adresu URL dane kanału RSS przekształcone w projekcję LINQ (opisaną w rozdziale 3.). Informacje są przechowywane po stronie serwera przez 15 minut, a po stronie klienta zgodnie z wartością `cacheDuration`.

Działanie metod `GetString` i `GetXml` nie jest szczególnie skomplikowane. Sprowadza się do użycia obiektu `WebClient` w celu pozyskania danych spod podanego adresu URL i zbuforowania odpowiedzi na określony czas w pamięci podręcznej. Kod obydwu metod został przedstawiony w listingu 5.4.

Listing 5.4. Metody GetString i GetXml usługi sieciowej pośrednika.

```
[WebMethod]
[ScriptMethod(UseHttpGet=true)]
public string GetString(string url, int cacheDuration)
{
    using( WebClient client = new WebClient( ) )
    {
        string response = client.DownloadString(url);
        this.CacheResponse(cacheDuration);
        return response;
    }
}

[WebMethod]
[ScriptMethod(UseHttpGet = true, ResponseFormat=ResponseFormat.Xml)]
public string GetXml(string url, int cacheDuration)
{
    return GetString(url, cacheDuration);
}
```

Różnica między metodami `GetString` i `GetXml` sprowadza się do tego, że metoda `GetString` zwraca ciąg treści zgodnie z formatem JSON, natomiast metoda `GetXml` zwraca ciąg tekstowy dokumentu XML. Atrybut `ResponseFormat` metody `GetXml` stanowi informację dla środowiska ASP.NET AJAX o obowiązku wygenerowania dokumentu XML w formie zwykłego tekstu zamiast przekształcania go w format JSON.

Metoda `GetRss` jest jednak nieco bardziej skomplikowana. Jej zadanie polega na pobraniu danych kanału RSS i przechowaniu ich przez 15 minut w pamięci podręcznej platformy ASP.NET. Dzięki temu kolejne odwołania do tego samego serwera są realizowane z wykorzystaniem pamięci podręcznej środowiska ASP.NET. Metoda `GetRss` przygotowuje dodatkowo specjalny nagłówek odpowiedzi, który zapewnia zbuforowanie danych po stronie przeglądarki na określony czas. Widżet przetwarzający informacje RSS może więc kontrolować czas przechowywania odpowiedzi w przeglądarce.

W listingu 5.5 został zawarty ten sam kod ładowania i interpretacji danych RSS, który został zaprezentowany w rozdziale 3. w części dotyczącej widżetu RSS.

Listing 5.5. Metoda GetRss w usłudze sieciowej pośrednika.

```
[WebMethod]
[ScriptMethod(UseHttpGet = true)]
public object GetRss(string url, int count, int cacheDuration)
{
    var feed = Context.Cache[url] as XElement;
    if( feed == null )
    {
        if( Context.Cache[url] == string.Empty ) return null;
        try
        {
            HttpWebRequest request = WebRequest.Create(url) as HttpWebRequest;

            request.Timeout = 15000;
            using( WebResponse response = request.GetResponse( ) )
            {
                using( XmlTextReader reader = new XmlTextReader( response.
                GetResponseStream( ) ) )
                {
                    feed = XElement.Load(reader);
                }
            }

            if( feed == null ) return null;
            Context.Cache.Insert(url, feed, null, DateTime.MaxValue, TimeSpan.
                FromMinutes(15));
        }
        catch
        {
            Context.Cache[url] = string.Empty;
            return null;
        }
    }

    XNamespace ns = "http://www.w3.org/2005/Atom";

    // Sprawdzenie, jakie dane są przetwarzane: RSS czy Atom.

    try
    {
        // RSS.
        if( feed.Element("channel" ) != null )
            return (from item in feed.Element("channel").Elements("item")
                select new
                {
                    title = item.Element("title").Value,
                    link = item.Element("link").Value,
                    description = item.Element("description").Value
                }).Take(count);
```

```
            // Atom.
            else if( feed.Element(ns + "entry") != null )
                return (from item in feed.Elements(ns + "entry")
                    select new
                    {
                        title = item.Element(ns + "title").Value,
                        link = item.Element(ns + "link").
                        Attribute("href").Value,
                        description = item.Element(ns + "content").Value
                    }).Take(count);

            // Błędny format.
            else
                return null;
        }
        finally
        {
        this.CacheResponse(cacheDuration);
        }
    }
```

Trudności w projektowaniu usługi sieciowej pośrednika

W aplikacjach bazujących na widżetach klienckich usługa sieciowa pośrednika jest najczęściej wykorzystywaną usługą sieciową witryny. Za każdym razem, kiedy skrypt JavaScript musi pobrać dane z zewnętrznej domeny, musi też wywołać jedną z metod usługi. W związku z tym podczas projektowania usługi sieciowej pośrednika trzeba wziąć pod uwagę wiele zagadnień związanych ze skalowalnością rozwiązania. Połączenie generowane przez tysiące widżetów, które z kolei wymuszają ustanowienie tysięcy połączeń z serwerami spoza domeny, wprowadza istotne obciążenie procesów ASP.NET. Czas odpowiedzi zdalnych serwerów jest nieprzewidywalny i zmienny. Mocno obciążony serwis zewnętrzny może dostarczyć odpowiedź po 20 lub nawet 30 sekundach, co oznacza, że odwołanie do usługi pośrednika zostanie na ten czas zawieszone. Jeśli taki problem się powtórzy dla 100 żądań przychodzących, wszystkie dostępne wątki robocze środowiska ASP.NET zostaną wykorzystane. Aplikacja sieciowa nie będzie mogła obsługiwać żadnych żądań, dopóki żądania przesłane do zdalnych usług nie zostaną zrealizowane lub przerwane (ponieważ dopuszczalny czas realizacji upłynie) i nie zwolnią wątku roboczego ASP.NET. Użytkownicy będą mieli wrażenie powolnego działania serwisu lub nawet zaobserwują brak reakcji na żądania. Zagadnienia związane ze skalowalnością aplikacji sieciowych, które w dużej mierze zależą od działania usług sieciowych, oraz z pobieraniem informacji z zewnętrznych źródeł danych zostały opisane w rozdziale 6.

Po zaimplementowaniu opisanych rozwiązań dysponujemy wszystkimi metodami, które są potrzebne do pobierania informacji z zewnętrznych źródeł danych bezpośrednio do przeglądarki z wykorzystaniem serwera proxy.

Budowa klienckiego widżetu RSS

Utwórzmy kliencką wersję widżetu RSS. Informacje z kanału RSS można buforować po stronie klienta, ponieważ nie zmieniają się szczególnie często. Nic nie stoi na przeszkodzie, aby były przechowywane w pamięci podręcznej na przykład przez godzinę. Pamiętajmy, że popularne serwisy RSS mają wielu odbiorców, a buforowanie informacji po stronie serwera

i dostarczanie ich do setek lub tysięcy użytkowników eliminuje konieczność wielokrotnego pobierania tych samych danych bezpośrednio ze źródła informacji.

Oto wykaz kilku najważniejszych różnic między klienckimi i serwerowymi widżetami RSS:

- Widżet kliencki nie pozyskuje danych RSS za pomocą kodu serwerowego, więc nie obejmuje kodu LINQ to XML, który pobierałby odpowiedni dokument XML. Kod LINQ to XML jest zawarty w usłudze pośrednika.
- Widżet kliencki nie jest wyposażony w kontrolkę `MultiView`. Komunikat o postępie prac jest wyświetlany przez skrypt JavaScript.
- Widżet kliencki nie zawiera kontrolki `DataList`, ponieważ odpowiedni kod HTML związany z kanałami RSS jest generowany przez skrypt JavaScript.
- Załadowanie danych RSS do przeglądarki należy do zadań klasy JavaScript o nazwie `FastRssWidget`, zapisanej w pliku *FastRssWidget.js*. Klasa ta odpowiada za wywołanie usługi sieciowej pośrednika i zinterpretowanie pozyskanych danych.
- W rozwiązaniu bazującym na widżetach klienckich obiekt klasy `FastRssWidget` jest powoływany przez kontrolkę serwerową. Ta sama kontrolka przekazuje odpowiedni skrypt startowy z adresami URL kanałów i ładuje cały kod do przeglądarki klienckiej.

Analizując listing 5.6, można zauważyć, że poza obszarem ustawień i pustym panelem widżet nie zawiera żadnych elementów interfejsu użytkownika.

Listing 5.6. W nowej wersji rozwiązania skrypt FastRssWidget.ascx nie zawiera prawie żadnych elementów interfejsu użytkownika.

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="FastRssWidget.ascx.cs"
Inherits="Widgets_FastRssWidget" EnableViewState="false" %>
<asp:Panel ID="SettingsPanel" runat="Server" Visible="False" >
...
</asp:Panel>

<asp:Panel ID="RssContainer" runat="server"></asp:Panel>
```

W serwerowym kodzie widżetu zdarzenie `Page_Load` rejestruje znacznik włączenia skryptu *FastRssWidget.js*, zgodnie z przykładem zamieszczonym w listingu 5.7.

Listing 5.7. Zdarzenie Page_Load kontrolki FastRssWidget dodaje znacznik skryptu odpowiedzialny za załadowanie pliku FastRssWidget.js, a następnie inicjalizuje klasę, wyświetlającą dane po stronie klienta.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (this._Host.IsFirstLoad)
    {
        ScriptManager.RegisterClientScriptInclude(this,
            typeof(Widgets_FastRssWidget),
            "FastRssWidget",
            this.ResolveClientUrl(
                this.AppRelativeTemplateSourceDirectory
                + "FastRssWidget.js"));

        ScriptManager.RegisterStartupScript(this,
            typeof(Widgets_FastRssWidget),
            "LoadRSS",
            string.Format("
                var rssLoader{0} =
                    new FastRssWidget( '{1}', '{2}', {3} );
```

```
                    rssLoader{0}.load( );",
                    this.UniqueID,
                    this.Url,
                    this.RssContainer.ClientID,
                    this.Count),
                true);
    }
}
```

Następnie kod zdarzenia wprowadza instrukcję JavaScript, która powołuje obiekt klasy i przekazuje do niej adres URL, identyfikator zewnętrznego panelu klienckiego oraz liczbę pozycji kanału, które należy wyświetlić. Operacja ta jest realizowana tylko raz, podczas pierwszego ładowania widżetu. Klasa kliencka wykorzystuje te trzy parametry, odwołując się do usługi pośrednika. W odpowiedzi na żądanie otrzymuje dane RSS, które przekształca w odsyłacze wyświetlane w zewnętrznym panelu. Identyfikator panelu jest przekazywany do klasy klienckiej z serwera. Służy on do wyznaczenia elementu `DIV`, który powinien zostać wykorzystany do wyświetlenia odsyłaczy.

Jednak podczas odsyłania danych po stronie klienta trzeba ponownie wygenerować kod treści kontrolki. A to dlatego, że operacja asynchronicznej aktualizacji uwzględnia odesłanie do przeglądarki pustego panelu kontenera bez odsyłaczy wygenerowanych wcześniej przez skrypt JavaScript. W praktyce w czasie asynchronicznych uaktualnień usuwane są wszystkie elementy interfejsu użytkownika, które zostały utworzone w wyniku działania kodu klienckiego. Odpowiedni skrypt JavaScript musi je więc odtworzyć po każdorazowym odesłaniu danych. Zadanie to jest realizowane przez procedurę obsługi zdarzenia `OnPreRender`. Przesyła ona do jednostki klienckiej blok skryptu, który przywraca początkową wartość adresu URL oraz wartość parametru `Count`, a następnie wywołuje funkcję `load` wcześniej utworzonego obiektu klasy `FastRssWidget`. Cała operacja jest zbliżona do procedury pierwszego ładowania kodu, podczas której tworzony jest obiekt klasy z odpowiednimi parametrami i wywoływana jest funkcja `load`. Jedyna różnica polega na tym, że za drugim razem nie jest tworzony nowy obiekt klasy — realizacja kodu bazuje na założeniu, że obiekt już istnieje. Rozwiązanie to zostało przedstawione w listingu 5.8.

Listing 5.8. W czasie zdarzenia OnPreRender do klienta jest wysyłany blok skryptu, który odświeża interfejs użytkownika.

```
protected override void OnPreRender(EventArgs e)
{

    base.OnPreRender(e);

    if (!this._Host.IsFirstLoad)
        ScriptManager.RegisterStartupScript(this,
            typeof(Widgets_FastRssWidget),
            "LoadRSS",
            string.Format("
                rssLoader{0}.url = '{1}';
                rssLoader{0}.count = {2};
                rssLoader{0}.load( );",
                this.UniqueID,
                this.Url,
                this.Count),
            true);
}
```

To wszystkie zmiany wprowadzane po stronie serwera. Kod klasy klienckiej jest nieco bardziej skomplikowany. Trzeba bowiem przenieść znaczną część skryptu serwerowego do kodu klienckiego. Treść klasy jest zapisana w pliku *FeedRssWidget.js*.

Konstruktor klasy oraz funkcja `load` zostały przestawione w listingu 5.9.

Listing 5.9. Kliencka klasa FeedRssWidget.

```
var FastRssWidget = function(url, container, count)
{
    this.url = url;
    this.container = container;
    this.count = count;
}

FastRssWidget.prototype = {

    load : function( )
    {
        var div = $get( this.container );
        div.innerHTML = "Loading...";
        Proxy.GetRss ( this.url, this.count, 10, Function.createDelegate( this, this.
        onContentLoad ) );
    },
```

Parametry przekazywane do konstruktora to adres URL, identyfikator kontenera oraz liczba odsyłaczy prezentowanych w obszarze treści. Pobranie informacji RSS należy do zdań metody `load`, która z kolei wywołuje funkcję `Proxy.GetRss`, przekazując do niej ciąg URL, informację o liczbie odsyłaczy oraz czas przechowywania danych w pamięci podręcznej (wyrażony w minutach). Odpowiedź jest buforowana na 10 minut, więc powtórne pobranie strony lub ponowne przejście do strony po wizycie w innym serwisie w ciągu 10 minut spowoduje dostarczenie odpowiedzi bezpośrednio z pamięci podręcznej przeglądarki bez ustanawiania połączenia z usługą sieciową pośrednika. Funkcja obsługi odpowiedzi została przedstawiona w listingu 5.10.

Listing 5.10. Metoda Proxy.GetRss wywołuje funkcję onContentLoad jako funkcję zwrotną, generującą odsyłacze do komunikatów kanału.

```
onContentLoad : function( rss )
{
    var div = $get( this.container );
    div.innerHTML = "";

    for( var i = 0; i < rss.length; i ++ )
    {
        var item = rss[i];

        var a = document.createElement("A");
        a.href = item.link;
        a.innerHTML = item.title;
        a.title = item.description;
        a.className = "feed_item_link";
        a.target = "_blank";
        div.appendChild(a);
    }
}
```

Funkcja `onContentLoad` tworzy odsyłacze do poszczególnych komunikatów RSS w kodzie klienckim. Przygotowany w ten sposób widżet kliencki ma kilka zalet w porównaniu z widżetem serwerowym. Oto one:

- Nie są pobierane żadne dane mechanizmu `ViewState`, ponieważ kontrolka sieciowa nie obejmuje prawie żadnych elementów interfejsu użytkownika. Ilość danych przekazywanych podczas pierwszego ładowania i w czasie asynchronicznych uaktualnień jest niewielka.
- Treść kontrolki jest buforowana w pamięci przeglądarki, co eliminuje konieczność wymiany danych przez sieć.
- Treść kontrolki jest dostarczana za pomocą usługi pośrednika, a nie w wyniku asynchronicznej aktualizacji. Rozwiązanie to pozwala na wykorzystanie zalet buforowania serwerowego.

Budowa klienckiego widżetu Flickr

Budowa klienckiego widżetu Flickr przebiega według tych samych zasad, które obowiązują podczas przygotowywania widżetu RSS. Kod serwerowy nie dostarcza gotowego dokumentu HTML. Klasa kliencka pozyskuje stosowny kod za pośrednictwem metody `Proxy.GetXml`. Pobiera cały dokument XML do jednostki klienckiej, co pozwala na uzupełnienie go w przeglądarce o funkcje stronicowania ładowanych zdjęć i nie wymaga asynchronicznych aktualizacji lub wywołań pośrednich. Użytkownik może przeglądać zdjęcia po bardzo krótkim czasie oczekiwania, a zwrócony dokument XML zostaje przechowany w pamięci podręcznej przeglądarki przez 10 minut. W przypadku ewentualnych kolejnych wizyt (w ciągu 10 minut) dokument XML zostanie dostarczony z bufora przeglądarki, więc widżet Flickr załaduje się natychmiast i bez konieczności asynchronicznego aktualizowania treści lub odwołań do usługi pośrednika.

Kliencka klasa `FastFlickrWidget` ma ten sam format, jaki został opisany w przypadku klasy `FastRssWidget`. Treść klasy znajduje się w pliku *Widgets\FastFlickrWidget.js*. Jest również przedstawiona w listingu 5.11.

Listing 5.11. Konstruktor i funkcja load klasy FastFlickrWidget.

```
var FastFlickrWidget = function(url, container, previousId, nextId)
{
    this.url = url;
    this.container = container;
    this.pageIndex = 0;
    this.previousId = previousId;
    this.nextId = nextId;
    this.xml = null;
}

FastFlickrWidget.FLICKR_SERVER_URL="http://static.flickr.com/";
FastFlickrWidget.FLICKR_PHOTO_URL="http://www.flickr.com/photos/";

FastFlickrWidget.prototype = {
    load : function( )
    {
        this.pageIndex = 0;
```

```
        var div = $get( this.container );
        div.innerHTML = "Loading...";

        Proxy.GetXml( this.url, 10, Function.createDelegate(this,
this.onContentLoad));
    },
    onContentLoad : function( xml )
    {
        this.xml = xml;
        this.showPhotos( );
    },
```

Konstruktor klasy pobiera cztery parametry: adres URL kanału Flickr, identyfikator elementu `DIV` będącego kontenerem dla treści oraz identyfikatory odsyłaczy do wcześniejszej i następnej strony. Przekazanie odsyłaczy jest niezbędne, ponieważ kod klasy zmienia sposób ich prezentacji w zależności od indeksu przeglądanej strony.

Funkcja `showPhotos` (przedstawiona w listingu 5.12) wykonuje wszystkie operacje niezbędne do utworzenia tabeli (o wymiarach 3x3 pola) oraz wyświetlenia odsyłaczy i zdjęć.

Listing 5.12. Funkcja showPhotos z pliku FastFlickrWidget.js.

```
showPhotos : function( )
{
    var div = $get( this.container );
    div.innerHTML = "";

    if( null == this.xml )
        return (div.innerHTML = "Error occured while loading Flickr feed");

    var photos = this.xml.documentElement.getElementsByTagName("photo");

    var row = 0, col = 0, count = 0;

    var table = document.createElement("table");
    table.align = "center";
    var tableBody = document.createElement("TBODY");
    table.appendChild( tableBody );
    var tr;

    for( var i = 0; i < 9; i ++ )
    {
        var photo = photos[i + (this.pageIndex * 9)];

        if( photo == null )
        {
            Utility.nodisplay( this.nextId );
            break;
        }

        if( col == 0 )
        {
            tr = document.createElement("TR");
            tableBody.appendChild(tr);
        }

        var td = document.createElement("TD");

        var img = document.createElement("IMG");
        img.src = this.getPhotoUrl(photo, true);
        img.style.width = img.style.height = "75px";
        img.style.border = "none";
```

```
            var a = document.createElement("A");
            a.href = this.getPhotoPageUrl(photo);
            a.target = "_blank";
            a.title = this.getPhotoTitle(photo);

            a.appendChild(img);
            td.appendChild(a);
            tr.appendChild(td);

            if( ++ col == 3 ) { col = 0; row ++ }
        }

        div.appendChild(table);

        if( this.pageIndex == 0 ) Utility.nodisplay(this.previousId);
    },
    previous : function( )
    {
        this.pageIndex --;
        this.showPhotos( );
        Utility.display( this.nextId, true );
        if( this.pageIndex == 0 )
            Utility.nodisplay( this.previousId );
    },

    next : function( )
    {
        this.pageIndex ++;
        this.showPhotos( );
        Utility.display( this.previousId, true );
    }
```

Powyższy kod powstał niemal wprost z przekształcenia kodu C# serwerowego widżetu Flickr w odpowiadający mu skrypt JavaScript. Można w nim zauważyć odwołania do klasy `Utility`, która jest niestandardową klasą, obejmująca kilka użytecznych funkcji, odpowiedzialnych między innymi za wyświetlanie i ukrywanie elementów interfejsu użytkownika oraz przetwarzanie elementów modelu DOM w sposób niezależny od rodzaju przeglądarki. Kod klasy `Utility` jest zawarty w pliku *MyFramework.js* zapisanym w głównym katalogu projektu.

Kontrolka serwerowa obejmuje panel ustawień i podstawowe elementy interfejsu użytkownika — pusty kontener oraz odsyłacze do poprzedniej i następnej strony. Deklaracja kontrolki została przedstawiona w listingu 5.13.

Listing 5.13. Plik FastFlickrWidget.ascx.

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="FastFlickrWidget.ascx.cs"
Inherits="Widgets_FastFlickrWidget" EnableViewState="false" %>
<asp:Panel ID="settingsPanel" runat="server" Visible="False">
...
</asp:Panel>

<asp:Panel ID="FlickrPhotoPanel" runat="server">

</asp:Panel>

<div style="text-align: center; width:100%; white-space:nowrap">
<asp:LinkButton ID="ShowPrevious" runat="server" >< Prev</asp:LinkButton>
 
<asp:LinkButton ID="ShowNext" runat="server" >Next ></asp:LinkButton></center>
</div>
```

Kod klasy JavaScript `FastFlickrWidget` powstaje w procedurze obsługi zdarzenia `Page_Load` i bazuje na parametrach dostarczonych przez mechanizm rejestracji danych o stanie kontrolki (`State`). Skrypt procedury został przedstawiony w listingu 5.14. Do przeglądarki jest również przekazywany blok skryptu odpowiedzialnego za obsługę kliknięć odsyłaczy zmiany strony — dzięki niemu wywoływane są po stronie klienckiej funkcje `previous` i `next`.

Listing 5.14. Zdarzenie Page_Load kontrolki sieciowej FastFlickrWidget.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (this._Host.IsFirstLoad)
    {
        ScriptManager.RegisterClientScriptInclude(this,
        typeof(Widgets_FastFlickrWidget),
        "FastFlickrWidget",
        this.ResolveClientUrl(
            this.AppRelativeTemplateSourceDirectory + "FastFlickrWidget.js"));

        ScriptManager.RegisterStartupScript(this,
            typeof(Widgets_FastFlickrWidget),
            "LoadFlickr",
            string.Format("
                var flickrLoader{0} =
                    new FastFlickrWidget( '{1}', '{2}', '{3}', '{4}' );
                flickrLoader{0}.load( );",
                this.UniqueID,
                this.GetPhotoUrl( ),
                this.FlickrPhotoPanel.ClientID,
                this.ShowPrevious.ClientID,
                this.ShowNext.ClientID),
            true);

        this.ShowPrevious.OnClientClick =
            string.Format("flickrLoader{0}.previous( ); return false;", this.UniqueID);
        this.ShowNext.OnClientClick =
            string.Format("flickrLoader{0}.next( ); return false;", this.UniqueID);
    }
}
```

Procedura obsługi zdarzenia `Page_Load` generuje blok skryptu potrzebny do inicjalizacji klasy `FlckrRssWidget` po stronie klienta i wywołania funkcji `load`. Podobnie jak w przypadku widżetu RSS funkcja `load` klasy klienckiej jest wywoływana podczas procedury obsługi zdarzenia `OnPreRender`. Zatem w przypadku asynchronicznej aktualizacji danych skrypt JavaScript może odświeżyć interfejs użytkownika z wykorzystaniem nowych parametrów, zgodnie z kodem zamieszczonym w listingu 5.15.

Listing 5.15. Zdarzenie OnPreRender kontrolki sieciowej FastFlickrWidget.

```
protected override void OnPreRender(EventArgs e)
{
    base.OnPreRender(e);

    if( !this._Host.IsFirstLoad )
        ScriptManager.RegisterStartupScript(this,
        typeof(Widgets_FastFlickrWidget), "LoadFlickr",
        string.Format("
            flickrLoader{0}.url = '{1}';
            flickrLoader{0}.load( );",
```

```
            this.UniqueID,
            this.GetPhotoUrl( ),
            this.FlickrPhotoPanel.ClientID),
        true);
}
```

Przygotowany w ten sposób widżet kliencki ma kilka zalet w porównaniu z widżetem serwerowym. Oto one:

- Nie są pobierane żadne dane mechanizmu `ViewState`, ponieważ kontrolka sieciowa nie obejmuje prawie żadnych elementów interfejsu użytkownika. Ilość danych przekazywanych podczas pierwszego ładowania i w czasie asynchronicznych uaktualnień jest niewielka.
- Dokument XML serwisu Flickr jest buforowany w pamięci przeglądarki, co pozwala na zmniejszenie liczby odwołań sieciowych.
- Wykorzystywane jest buforowanie danych po stronie serwera Proxy (dzięki czemu tysiąc użytkowników żądających dostarczenia tych samych zdjęć nie prześle tysiąca żądań do serwisu Flickr).
- Stronicowanie zdjęć jest wykonywane bezzwłocznie ponieważ zależy jedynie od skryptu klienckiego, co czyni tę wersję kontrolki znacznie szybszą w działaniu.

Podsumowanie

W tym rozdziale zostały omówione zagadnienia szybszego wyświetlania stron przez opóźnienie ładowania jej komponentów. Zaprezentowane rozwiązania pozwalają na łatwe skrócenie czasu pobierania dokumentów. Jednak jeszcze większą szybkość działania aplikacji i lepsze wykorzystanie pamięci podręcznej przeglądarki można uzyskać przez zastosowanie widżetów klienckich, które udostępniają użytkownikom wiele rozbudowanych funkcji, ale nie wymagają asynchronicznej aktualizacji. W rozdziale tym zostały również omówione zasady przygotowania jednego z najważniejszych komponentów strony startowej — pośrednika w dostępie od danych. W kolejnym rozdziale zostaną opisane niektóre problemy związane ze skalowalnością witryn Ajax, których funkcjonowanie w dużym stopniu zależy od usług sieciowych i wymaga komunikacji z wieloma zewnętrznymi serwisami.