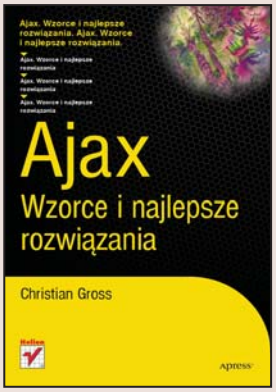
Ajax. Wzorce i najlepsze rozwiązania



Autor: Christian Gross
Tłumaczenie: Maciej Jezierski
ISBN: 83-246-0554-1
Tytuł oryginału: Ajax Patterns and Best Practices
Format: B5, stron: 392

Ajax to nowoczesna technologia umożliwiająca budowanie witryn internetowych nowej generacji. Oddzielenie klienta od serwera i zastosowanie usług internetowych pozwala na tworzenie aplikacji łączących w sobie atrakcyjny i szybki interfejs, znany ze standardowych programów, z zaletami korzystania ze stron WWW. Dzięki temu możemy obniżyć koszty produkcji, zapewnić błyskawiczny dostęp do nowych danych i aktualizacji czy ułatwić używanie aplikacji z dowolnego komputera na świecie mającego dostęp do sieci WWW. Ponadto Ajax bazuje na standardowych technologiach, zatem można stosować go na wszystkich platformach.

Książka „Ajax. Wzorce i najlepsze rozwiązania" rozpoczyna się od wprowadzenia do tego podejścia. Tam też znajdziesz opis związanych z nim mechanizmów, takich jak architektura REST czy obiekty XMLHttpRequest, co pozwoli Ci szybko zrozumieć funkcjonowanie i zalety tej technologii. Jednak główną część książki stanowią praktyczne wzorce. Dzięki nim dowiesz się, jak usprawnić wczytywanie aplikacji poprzez stopniowe pobieranie kodu HTML, jak przyspieszyć działanie witryny za pomocą pamięci podręcznej, a także jak dynamicznie modyfikować zawartość stron. Nauczysz się też zwiększać komfort pracy użytkowników poprzez tworzenie wygodnego i niezawodnego systemu nawigacyjnego oraz sprawne pobieranie danych.

W książce omówiono:

- Funkcjonowanie technologii Ajax
- Architektura REST
- Obiekty XMLHttpRequest
- Stopniowe wczytywanie stron
- Obsługa pamięci podręcznej
- Przetwarzanie i reprezentacja danych
- Obsługa nawigacji
- Dynamiczne modyfikowanie stron
- Stała komunikacja między serwerem a klientem

Stosuj sprawdzone wzorce i najlepsze praktyki budowania witryn internetowych

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

Spis treści

Rozdział 3.

Wzorzec Fragmentacja Zawartości

Cel

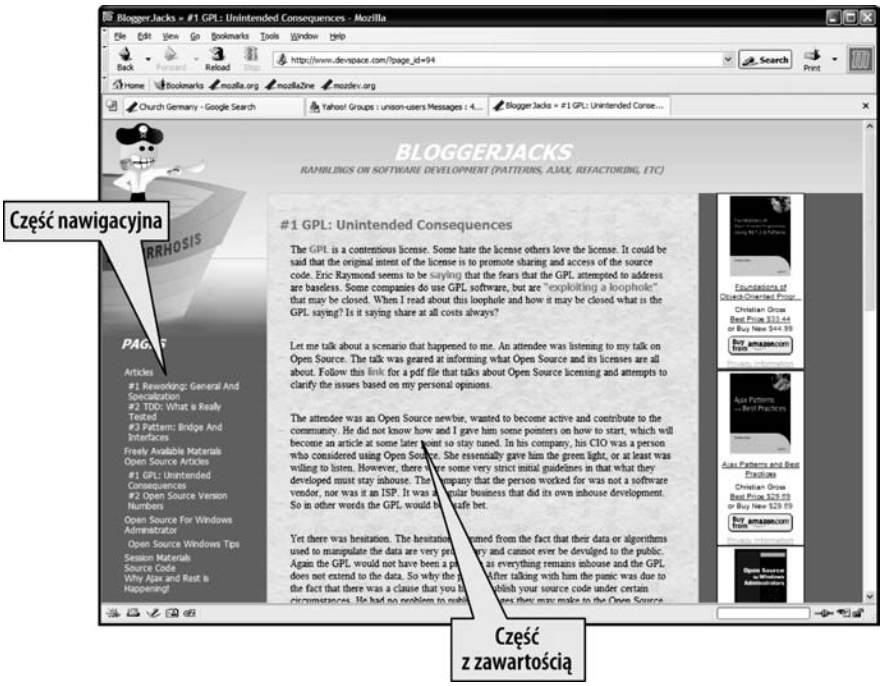
Wzorzec Fragmentacja Zawartości umożliwia stopniowe budowanie strony HTML. Dzięki temu logika pojedynczej strony HTML może być rozproszona, a użytkownik może decydować o czasie i fragmencie logiki, który ma być pobrany.

Przyczyny wykorzystania wzorca

W czasie, kiedy strony internetowe dopiero zaczynały powstawać, projektanci zawartości HTML tworzyli dokumenty, które były niekompletne. Niekompletne strony były uzupełniane za pomocą łączy do innych dokumentów. Cały dokument był sumą stron w drzewie dokumentów.

Pomyśl o tym następująco: zamiast tworzyć książkę, której zawartość czytasz strona po stronie, w przypadku witryny internetowej zebrałbyś materiały razem jak artykuły w gazecie. Ale w przeciwieństwie do gazety, która wymaga przewracania stron, witryna internetowa umożliwia kliknięcie i przeskoczenie do innej zawartości. W miarę upływu czasu witryny internetowe odchodziły od struktury rozproszonej do ściśle samodzielnej struktury hierarchicznej.

Przykład strony o ściśle samodzielnej strukturze hierarchicznej został pokazany na rysunku 3.1.



Rysunek 3.1. *Ścisła struktura hierarchiczna witryny internetowej*

Na rysunku 3.1 strona internetowa została podzielona na dwie części — nawigacyjną z niebieskim tłem i zawartość z brązowym tłem. Kiedy użytkownik kliknie łącze nawigacyjne, zmienia się zawartość. Problem leży w tym, że przeładowywana jest cała strona, nawet jeśli użytkownik jest zainteresowany tylko zawartością z brązowym tłem. Jednym ze sposobów na obejście tego problemu byłoby użycie ramek HTML, tak żeby część nawigacyjna znajdowała się w jednej ramce, a zawartość w drugiej. Po kliknięciu łącza nawigacyjnego zmieniałaby się tylko ramka z zawartością. Jednak, jak pokazał czas, mimo że ramki rozwiązują problem osobnego pobierania zawartości, są problematyczne z perspektywy nawigacji i interfejsu użytkownika. Dlatego też w witrynach internetowych ramki są używane w coraz mniejszym stopniu.

Dla twórcy witryny internetowej najlepszym rozwiązaniem byłaby możliwość zmieniania tylko tej części zawartości, która powinna zostać zmieniona, i pozostawienia reszty bez zmian. W końcu zawartość, której nie musimy zmieniać, działa.

Zastosowanie

Używaj wzorca Fragmentacja Zawartości w następujących sytuacjach:

- Rodzaj witryny powoduje, że nie wiadomo, jak powinna wyglądać strona HTML. Na rysunku 3.1 znajduje się część nawigacyjna z niebieskim tłem i część z zawartością na brązowym tle. Nie jest znana zawartość żadnej z części, ale wiadomo, gdzie znajduje się część z zawartością.

- Zawartość do pobrania jest zbyt duża i mogłaby spowodować, że użytkownik długo czekałby na jej wyświetlenie. Na przykład wyszukiwanie i oczekiwanie na zebranie wszystkich wyników nie jest dobrym wyjściem, ponieważ użytkownik mógłby zbyt długo czekać. Lepszym podejściem byłoby wykonywanie wyszukiwania i wyświetlanie wyników w miarę ich znajdowania.
- Wyświetlana zawartość nie jest powiązana. Yahoo!, MSN i Excite są aplikacjami portalowymi wyświetlającymi jedną zawartość obok innej, zupełnie z nią niezwiązanej. Jeśli zawartość miałaby być generowana z pojedynczej strony HTML, logika po stronie serwera musiałaby zawierać olbrzymi blok decyzyjny, żeby wiedzieć, która zawartość została pobrana, a która nie. Lepszym podejściem jest uznanie każdego bloku za odrębną część, która jest oddzielnie pobierana.

Powiązane wzorce

Fragmentacja Zawartości jest podstawowym wzorcem każdej aplikacji ajaksowej, Możesz nawet założyć, że zastosowanie tego wzorca wynika wprost z zastosowania Ajaksa. Tak czy inaczej, wciąż trzeba zidentyfikować i określić kontekst wzorca Fragmentacja Zawartości. Jego unikalną cechą jest to, że korzystanie z niego polega na wykonywaniu zawsze tych samych kroków, do których należą: wygenerowanie zdarzenia, żądanie, odpowiedź i wstawienie części zawartości. Inne wzorce omówione w tej książce są podobne, mają jednak różne odchylenia, takie jak wykonanie żądania i niepobieranie zawartości od razu (na przykład wzorzec Trwała Komunikacja).

Architektura

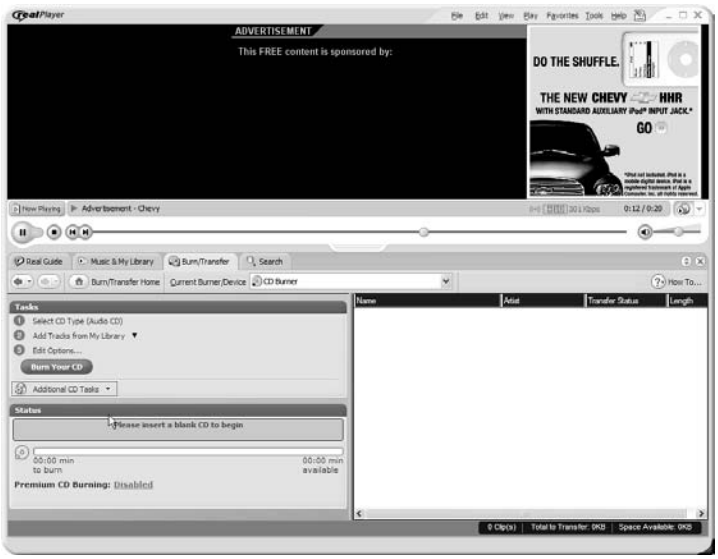
Architektura wzorca Fragmentacja Zawartości jest stosunkowo prosta. Klient wywołuje URL. Serwer wysyła odpowiedź z jakąś zawartością, która jest następnie odbierana i przetwarzana przez klienta. Implementacja wzorca Fragmentacja Zawartości zawsze wykonuje następujące kroki:

1. Generowane jest zdarzenie, które może być wynikiem naciśnięcia przycisku albo pobrania strony HTML.
2. Zdarzenie wywołuje funkcję, która jest odpowiedzialna za stworzenie adresu URL używanego do wysłania żądania do klienta.
3. Serwer odbiera żądanie i na jego podstawie tworzy jakąś zawartość. Zawartość jest wysyłana jako odpowiedź do klienta.
4. Klient otrzymuje odpowiedź i umieszcza ją w odpowiednim miejscu strony HTML.

Implementowanie porządku w aplikacji internetowej

Po ponownym spojrzeniu na rysunek 3.1 można stwierdzić, że ściśle hierarchiczna struktura strony internetowej nie jest złą rzeczą. Użycie HTML powoduje, że w wyniku zastosowania takiej struktury trzeba generować zawartość w jednym kroku. Właśnie to całościowe tworzenie zawartości powoduje problemy. Tradycyjne aplikacje nie działają w ten sposób, co widać na rysunku 3.2.

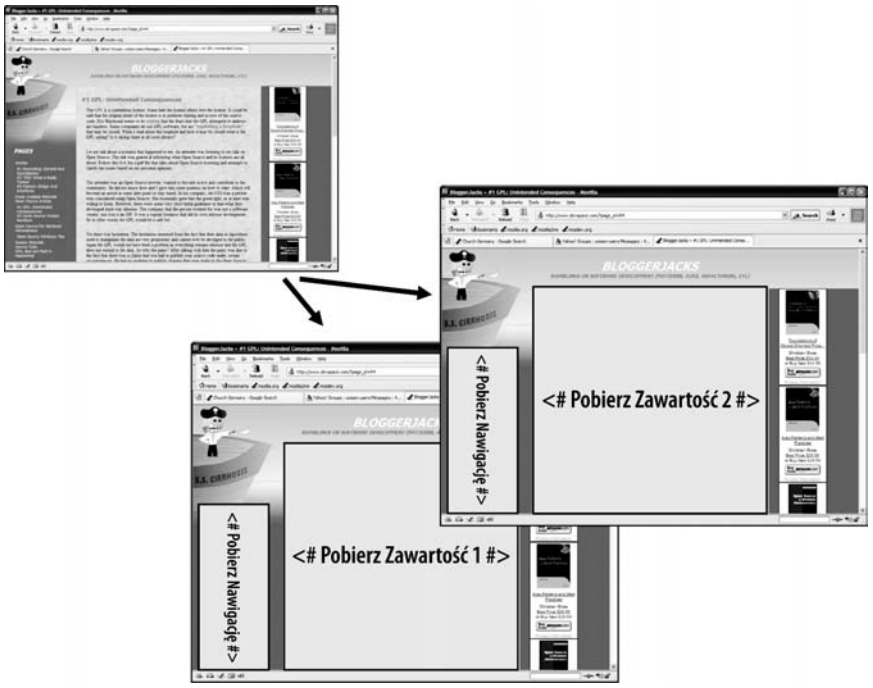
Rysunek 3.2. *Tradycyjna aplikacja kliencka*



Na rysunku 3.2 znajduje się RealPlayer, który jest przykładem tradycyjnej aplikacji klienckiej mieszającej nowsze technologie oparte na HTML z tradycyjnymi elementami interfejsu użytkownika. Kliknięcie przycisku *Burn Your CD* spowoduje, że RealPlayer nagra Twoją płytę CD, ale nie spowoduje zmian w reklamie znajdującej się na górnej połowie aplikacji. Logika związana z reklamą i ta związana z nagrywaniem płyt to dwa oddzielne fragmenty logiki, które akurat są wykorzystywane w tym samym oknie.

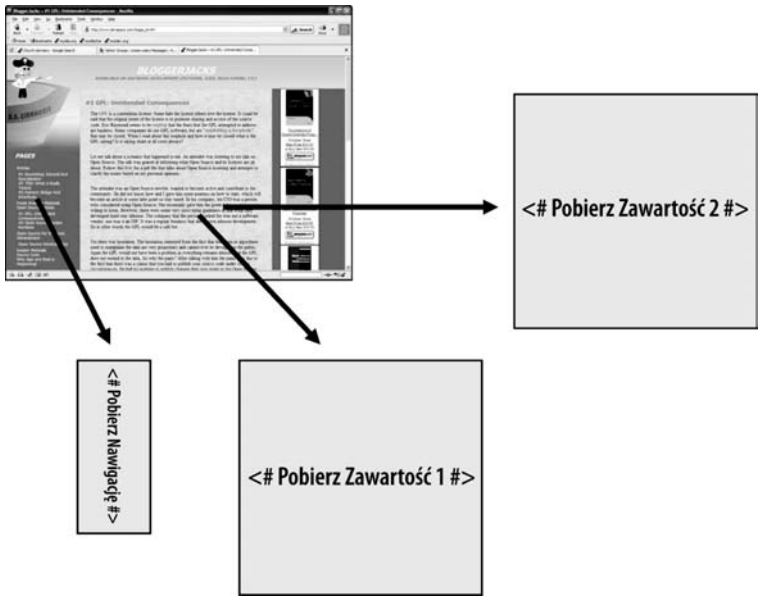
Na rysunku 3.3 z aplikacji internetowej z rysunku 3.1 wydzielono odrębne fragmenty logiki.

Oryginalna strona HTML na rysunku 3.3 zawiera łącza do dwóch innych stron, które reprezentują przykładowy blog i zawartość artykułu. Przykładowa zawartość ma dwa bloki wykonawcze — `Pobierz Nawigację` i `Pobierz Zawartość (1,2)`. Logika użyta do wygenerowania `Pobierz Zawartość 1` jest inna niż ta użyta do wygenerowania `Pobierz Zawartość 2`. W kontekście generowania strony HTML, kiedy wykonywana jest logika `Pobierz Zawartość 1` czy `Pobierz Zawartość 2`, wykonywana jest także logika `Pobierz Nawigację`. Oznacza to, że logika `Pobierz Zawartość` jest wykonywana wielokrotnie w celu wygenerowania tej samej zawartości. Niektórzy czytelnicy mogą się kłócić, że `Pobierz Nawigację` generuje różne dane (na przykład różne foldery są otwarte), ale w rzeczywistości są to te same dane tylko inaczej sformatowane. Krótko mówiąc, występuje tu nadmiarowe generowanie danych, które powinno być unikane.



Rysunek 3.3. *Architektura witryny internetowej*

Rozwiązaniem jest rozproszenie logiki, dzięki czemu strona HTML będzie generowana przy użyciu architektury podobnej do przedstawionej na rysunku 3.4.



Rysunek 3.4. *Ulepszona architektura strony internetowej*

Strona na rysunku 3.4 jest złożeniem wielu fragmentów logiki umieszczonej po stronie serwera. Po pobraniu szkieletu strony HTML obiekt `XMLHttpRequest` pobiera zawartość bloków `Pobierz Nawigację`, `Pobierz Zawartość 1` i `Pobierz Zawartość 2`. To, kiedy i jak pobierane są poszczególne bloki zawartości, zależy od tworzonych przez nie zdarzeń i łączy. Każdy blok zawartości jest oddzielnym żądaniem, które musi zostać wywołane przez typ `XMLHttpRequest`.

Proponowana architektura ma następujące zalety:

- Klient pobiera tylko to, co konieczne, i tylko wtedy, kiedy to konieczne. Nie ma potrzeby ponownego pobierania bloku zawartości, chyba że jest to niezbędne.
- Architektura jest rozdzielona na różne bloki kodu, które mogą być dynamicznie łączone w różnych kontekstach.
- Architektura podobna do tej w przypadku tradycyjnej aplikacji klienckiej w tym, że modyfikuje się tylko te elementy, których dotyczą zdarzenia.
- Całościowy wygląd aplikacji nie zmienia się, ponieważ wygenerowane bloki kodu składają się na wygląd strony HTML, która je pobiera.

Na rysunku 3.4 widać, skąd wziął swoją nazwę wzorzec Fragmentacja Zawartości: zawartość pojedynczej strony jest sumą fragmentów, na które została ona podzielona. Każdy z tych bloków jest obsługiwany i pobierany osobno.

Określanie zawartości wewnątrz jej fragmentu

Fragmenty zawartości, do których odwołuje się obiekt `XMLHttpRequest`, mogą mieć dowolny format zrozumiały zarówno dla serwera, jak i klienta. Cokolwiek wyśle serwer, musi być to zrozumiałe dla klienta. Na rysunku 3.4 fragmenty zawartości będą w HTML, ponieważ zostaną umieszczone bezpośrednio w stronie HTML. Jednak HTML nie jest jedynym formatem, który może być wysłany przez serwer.

W tym rozdziale omówione będą następujące formaty:

- **HTML**. Serwer może wysłać HTML bezpośrednio do klienta. Otrzymany HTML nie jest przetwarzany, lecz bezpośrednio umieszczany w stronie HTML. Jest to podejście ślepego przetwarzania, ponieważ klient nie wie, co wykonuje pobrany kod HTML. Bezpośrednie umieszczanie kodu HTML jest bardzo prostym i „idiotoodpornym" sposobem budowania zawartości. Klient nie musi niczego przetwarzać, wystarczy tylko, że zna miejsce, w którym ma zostać umieszczony kod HTML. Jeśli przetwarzanie będzie konieczne, pobrana zawartość (o ile jest zgodna z XML-em) będzie również dostępna w postaci instancji modelu obiektowego. Instancja modelu obiektowego pozwala na ręczne zmienianie zawartości HTML. Zaleca się, żeby HTML wysyłany do klienta był zgodny z XHTML-em (HTML-em, który jest implementacją odpowiedniego schematu XML) albo przynajmniej z XML-em.
- **Obrazki**. Nie ma możliwości bezpośredniego wysyłania obrazków, ponieważ mają one postać binarną, a obiekt `XMLHttpRequest` nie może

przetwarzać takich danych. Zazwyczaj odnośniki do obrazków są wysyłane w postaci znaczników HTML. Te są następnie umieszczane w dokumencie HTML, co powoduje pobranie zdalnych obrazków. Można pobierać binarne dane i korzystać z tych, które zostały zakodowane, przesłane, a następnie zdekodowane przy użyciu kodowania Base64. Jednakże operacje na danych binarnych nie są zalecane, ponieważ powodują więcej problemów niż korzyści.

- **JavaScript**. Serwer może wysłać do klienta JavaScript, który może być wykonany za pomocą instrukcji `eval`, a klient może wysyłać istniejące obiekty JavaScriptu do serwera w celu dalszego przetwarzania. Na pierwszy rzut oka może wydawać się, że wykonywanie pobranych JavaScriptów jest potencjalnie niebezpieczne. Zazwyczaj jednak nie jest to problemem, ponieważ implementacje JavaScriptu we wszystkich przeglądarkach używają polityki tego samego źródła i piaskownicy. Wysyłanie zewnętrznego JavaScriptu do wykonania może być problemem bezpieczeństwa, jeśli w implementacji języka JavaScript znajduje się błąd. Wysyłanie JavaScriptu jest pożądane, jeśli chcesz dynamicznie wykonywać i dodawać logikę, która nie została pobrana wraz z początkową stroną HTML. Ta metoda daje bardzo duże możliwości rozbudowy funkcjonalności klienta, który nie musi nawet o tym wiedzieć. Na przykład element formularza HTML wymaga sprawdzenia poprawności. Ponieważ dla różnych użytkowników inaczej sprawdzana jest poprawność, wysyłanie wszystkich implementacji sprawdzania poprawności byłoby niepożądane. Rozwiązaniem mogłoby być umożliwienie określenia, które elementy formularza będą prezentowane, i następnie dynamiczne ładowanie odpowiedniej implementacji sprawdzania poprawności jako fragmentu zawartości. Pamiętaj jednak o tym, że przesyłanie fragmentów JavaScriptu może narazić Twoją aplikację na atak hakerów. Dlatego też dobrze przemyśl użycie tej techniki.

- **XML**. Preferowanym podejściem jest wysyłanie i odbieranie danych w postaci XML. XML może być przetworzony albo sparsowany po stronie klienta za pomocą obiektowego modelu XML lub poprzez użycie biblioteki XSLT (ang. *Extensible Stylesheet Language Transformations* — transformacje rozszerzalnego języka arkuszy stylów), która może przekształcić go na inny model obiektowy, taki jak HTML. XML jest preferowany, ponieważ stanowi dobrze poznaną technologię, a narzędzia do wykonywania na nim operacji są sprawdzone, działające i stabilne. XML jest technologią o uznanej pozycji, dzięki czemu stworzone w niej dokumenty możesz przeszukiwać, rozdzielać i tworzyć, a także sprawdzać ich poprawność bez potrzeby pisania dodatkowego kodu. Niektórzy uważają, że XML ma duże rozmiary z powodu dużej liczby nawiasów ostrych i innych znaczników XML. Jednak zaletą tego jest to, że XML wygenerowany przez aplikację po stronie serwera może być przetworzony zarówno przez klienta opartego na przeglądarce internetowej, jak i klienta bez graficznego interfejsu użytkownika. Wybór sposobu parsowania XML-a i informacji, które należy przetwarzać, w całości zależy od klienta, o ile potrafi on parsować XML. XML jest elastyczny i powinno się z niego korzystać. Będzie on bardzo często wykorzystywany w tej książce i traktowany jako podstawowy format wymiany danych.

Są inne metody wymiany danych, takie jak JSON (ang. *JavaScript Object Notation* — zapis obiektów JavaScript)[1]. Jednakże radzę dobrze przemyśleć konsekwencje wynikające z wyboru innych formatów. Nie chodzi o to, że uważam je za źle zaprojektowane czy nieodpowiednie. Tym, co mnie do nich nie przekonuje, jest to, że nie udostępniają one tak rozbudowanego środowiska do przetwarzania, wyszukiwania, sprawdzania poprawności i tworzenia jak XML. Na przykład przy użyciu XPath mogę wyszukać konkretne elementy w XML-u bez parsowania całego dokumentu XML. Oczywiście, w niektórych przypadkach XML może nie zapewniać takiej wydajności jak na przykład JSON. Dla czytelników, którzy są pewni, że nigdy nie będą potrzebowali rozbudowanych funkcji XML-a, JSON może być odpowiedni. W tej książce nie będę jednak omawiał innych technologii takich jak JSON.

Teraz, kiedy już poznałeś architekturę, możemy przejść do przedstawienia implementacji demonstrujących sposób jej realizacji.

Implementacja

Podczas implementacji wzorca Fragmentacja Zawartości należy postępować według wcześniej wymienionych kroków (zdarzenie, żądanie, odpowiedź, umieszczenie). Logikę można łatwo zaimplementować, używając obiektu `Asynchronous`, ponieważ istnieje możliwość wywołania go przez zdarzenie i zawiera on bezpośrednią implementację metody do pobierania odpowiedzi z serwera. Poniżej przedstawię przykładową implementację, która zilustruje, jak generować zdarzenia w HTML-u, wywoływać funkcje, tworzyć żądania za pomocą `XMLHttpRequest` oraz przetwarzać odpowiedzi, używając dynamicznego HTML-a i JavaScriptu.

Implementacja szkieletu strony HTML

Implementacja wzorca Fragmentacja Zawartości wymaga stworzenia strony HTML służącej jako szkielet. Zastosowanie szkieletu pozwoli udostępnić strukturę, w której umieszczane będą fragmenty zawartości. Strona zawierająca szkielet jest kontrolerem i ma minimalną zawartość.

Poniższy kod HTML jest przykładem szkieletowej strony HTML, w której będą dynamicznie umieszczane fragmenty zawartości:

```
<html>
<head>
<title>Fragment dokumentu HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
   document.getElementById("insertplace").innerHTML = responseText;
```

[1] *http://www.crockford.com/JSON/index.html*

```
}
</script>
</head>
<body onload="asynchronous.call('/chap03/chunkhtml01.html')">
<table>
   <tr><td id="insertplace">Brak zawartości</td></tr>
</table>
</body>
</html>
```

W powyższym kodzie HTML tworzona jest instancja klasy `Asynchronous` i przypisywana funkcja zwrotna do właściwości `asynchronous.complete`. Sposób działania klasy `Asynchronous` oraz właściwości, do których należy przypisać odpowiednie wartości, zostały opisane w rozdziale 2. Instancja `asynchronous` jest tworzona po wczytaniu strony HTML. Po wczytaniu całej strony wykonywane jest zdarzenie `onload`. Jest to pierwszy krok implementacji wzorca: wygenerowanie zdarzenia. Zdarzenie `onload` wywołuje metodę `asynchronous.call`, która wykonuje żądanie `XMLHttpRequest` w celu pobrania fragmentu HTML. Jest to drugi krok implementacji wzorca: wykonanie żądania.

Po zakończeniu żądania serwer generuje odpowiedź, która jest odbierana po stronie klienta poprzez wywołanie metody `asynchronous.complete`. Pobranie odpowiedzi jest kolejnym krokiem implementacji wzorca. W tym przykładzie do właściwości `asynchronous.complete` jest przypisana anonimowa funkcja JavaScriptu. Żeby umieścić wyniki żądania `XMLHttpRequest` w elemencie HTML, w jej implementacji wywoływana jest metoda `getElementById`. Element HTML, w tym przypadku znacznik HTML `td`, jest odnajdywany poprzez identyfikator `insertplace`. Umieszczenie wyników w kodzie HTML polegające na odwołaniu się do elementu HTML i przypisaniu do jego własności `innerHTML` wyników żądania jest ostatnim krokiem implementacji wzorca.

Niespotykaną dotąd rzeczą jest w tym przykładzie to, że po wczytaniu i przetworzeniu strony HTML na stronie, która jest uważana za kompletną, wywoływany jest inny fragment logiki. Używa się go do pobrania reszty zawartości w postaci fragmentu zawartości. Kod po stronie serwera mógłby przecież wygenerować całą stronę. Jednak w ten sposób można pokazać, jak prosta może być implementacja wzorca Fragmentacja Zawartości. Przykład ilustruje reakcję na zdarzenie `onload`, ale można użyć dowolnego zdarzenia. Przykłady w rozdziale 2. używały zdarzenia wywoływanego naciśnięciem przycisku. Skrypt mógłby nawet emulować zdarzenia poprzez użycie metody `Click()`.

Powyższy przykład ilustruje oddzielenie wyglądu strony HTML od jej logiki. W części strony, w której umieszczana jest zawartość, projektant HTML musiałby tylko zamarkować miejsce, na przykład wpisując w nie jakiś tekst. Programista aplikacji po stronie serwera tworzyłby zawartość, która następnie zastępowałaby tekst w zamarkowanym miejscu. Projektant HTML nie musi przejmować się jakąkolwiek technologią programowania po stronie serwera, ponieważ szkielet strony HTML zawiera tylko instrukcje wykonywane po stronie klienta. Programista aplikacji po stronie serwera nie musi przejmować się wyglądem strony HTML, ponieważ tworzona przez niego zawartość nie zawiera żadnych informacji, które dotyczą wyglądu. W czasie testów programista aplikacji internetowej skupia się na logice, podczas gdy projektant HTML skupia się na jej wyglądzie i zachowaniu.

Umieszczanie zawartości przy użyciu dynamicznego HTML-a

Magia tego przykładu tkwi w zdolności dynamicznego HTML-a do dynamicznego umieszczania zawartości w określonym miejscu. Zanim pojawił się dynamiczny HTML, do połączenia wielu strumieni danych trzeba było używać ramek albo logiki po stronie serwera. Kilka lat temu dynamiczny HTML został formalnie ustandaryzowany przez World Wide Web Consortium (W3C) jako HTML DOM (ang. *HTML Document Object Model* — obiektowy model dokumentu HTML). DOM określony przez W3C nie jest tak rozbudowany jak modele obiektowe dostępne w Internet Explorerze Microsoftu czy przeglądarkach wywodzących się z Mozilli. Model obiektowy używany w tej książce jest mieszaniną HTML DOM i funkcji dostępnych w większości przeglądarek (na przykład w Internet Explorerze oraz w wywodzących się z Mozilli).

W poprzednim przykładzie atrybut `id` unikalnie identyfikuje element na stronie HTML. Dzięki unikalnie identyfikowanemu elementowi można określić miejsce początkowe, od którego możliwa jest nawigacja i manipulowanie obiektowym modelem HTML. Innym sposobem na znalezienie miejsca początkowego jest bezpośrednie określenie typu znacznika i na jego podstawie znalezienie odpowiedniego elementu HTML. Bez względu na to, które podejście zastosujesz, jedno z nich musi zostać użyte do pobrania miejsca początkowego. Niektórzy czytelnicy mogą mówić, że mógłbyś użyć innych właściwości czy metod, ale nie są one kompatybilne z HTML DOM i dlatego powinno się ich unikać.

Następujący kod HTML demonstruje, jak znaleźć punkt początkowy przy użyciu dwóch podejść:

```
<html>
<head>
<title>Fragment dokumentu HTML</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText,
   responseText, responseXML) {
   document.getElementsByTagName("table")[ 0].rows[ 0].cells[ 0].innerHTML
       = responseText;
   document.getElementById("insertplace").innerHTML = responseText;
}
</script>
</head>
<body onload="asynchronous.call('/chap03/chunkhtml01.html')">
<table>
   <tr><td>Brak zawartości</td></tr>
   <tr><td id="insertplace">Brak zawartości</td></tr>
</table>
</body>
</html>
```

W implementacji anonimowej funkcji dla metody `asynchronous.complete` do umieszczenia zawartości w elemencie dynamicznego HTML-a wykorzystane są dwie metody (`getElementsByTagName`, `getElementById`). Obydwie pobierają element reprezentujący (elementy reprezentujące) punkt początkowy.

Metoda `getElementsByTagName` pobiera wszystkie elementy HTML o typie określonym przez przekazany parametr. W tym przykładzie parametrem jest `table`. Wskazuje on, że z dokumentu mają zostać pobrane wszystkie elementy typu `table`. Zwracana jest instancja kolekcji `HTMLCollection` zawierająca w tym przykładzie wszystkie elementy typu `table`. Klasa `HTMLCollection` ma właściwość `length`, która określa, ile elementów zostało znalezionych. Do znalezionych elementów można się odwoływać za pomocą zapisu służącego do odwoływania się do elementów tablicy w JavaScripcie (nawiasy kwadratowe), przy czym indeksem pierwszego elementu jest 0.

W tym przykładzie zaraz po identyfikatorze metody `getElementsByTagName("table")` znajduje się kilka nawiasów kwadratowych (`[0]`) służących do pobrania pierwszego elementu kolekcji. Indeks zerowy jest tu użyty odgórnie i oznacza odniesienie do pierwszej znalezionej tabeli. Indeks użyty w tym przykładzie jest prawidłowy tylko dlatego, że na stronie HTML znajduje się jedna tabela. Z tego powodu indeks zerowy w tym przykładzie zawsze będzie prawidłowy, co oznacza, że będzie odwoływał się do odpowiedniej tabeli, wiersza i komórki. Wyobraź sobie jednak sytuację, w której na stronie znajduje się wiele tabel. W takim przypadku używanie z góry określonego indeksu może, ale nie musi się wiązać z odwołaniem do odpowiedniej tabeli. Co gorsza, jeśli wzorzec Fragmentacja Zawartości był wielokrotnie wywoływany, kolejność znalezionych elementów może ulec zmianie, co spowoduje, że wystąpią odwołania do niewłaściwych elementów.

Metody `getElementsByTagName` najlepiej używać tylko w przypadku, gdy operacje są wykonywane na wszystkich znalezionych elementach i nie ma potrzeby ich rozróżniania. Do przykładów takich operacji można zaliczyć dodawanie kolumny w tabeli czy zmianę stylu. Do operacji na pojedynczych elementach najlepiej użyć metody `getElementById`.

Za pomocą metody `getElementsByTagName` można pobrać wszystkie elementy w dokumencie HTML. Ilustruje to poniższy przykład:

```
var collection = document.getElementsByTagName("*");
```

Wywołanie metody `getElementsByTagName` z gwiazdką jako parametrem powoduje pobranie wszystkich elementów dokumentu HTML. Niektórzy mogą zauważyć, że można to samo osiągnąć, odczytując właściwość `document.all`. Chociaż jest to prawda, właściwość ta nie jest zgodna z DOM i spowoduje wyświetlenie ostrzeżeń w przeglądarkach opartych na Mozilli.

Przyjrzyjmy się następującemu fragmentowi przykładowego kodu:

```
document.getElementsByTagName("table")[0].rows[0].cells[0].innerHTML
```

Identyfikatory po nawiasach kwadratowych metody `getElementsByTagName` reprezentują wywoływane właściwości i metody. Odnoszą się one bezpośrednio do pobranego obiektu, który w tym przypadku jest tabelą zawierającą wiersze i komórki. Jeśli pobrany element nie byłby tabelą, wywołanie tych właściwości i metod spowodowałoby powstanie błędu.

Powróćmy do przykładowego kodu źródłowego i przyjrzyjmy się temu fragmentowi:

```
document.getElementById("insertplace").innerHTML = responseText;
```

Metoda `getElementById` pobiera element HTML z podanym atrybutem `id`. W atrybucie `id` ważna jest wielkość liter. Wynikiem wywołania tej metody jest pobranie znacznika `td` z atrybutem `id` o wartości `insertplace`. Jeśli w dokumencie znajdowałoby się wiele elementów z takim samym atrybutem `id`, wywołanie metody `getElementById` spowodowałoby pobranie tylko pierwszego znalezionego elementu. Pozostałe elementy nie są zwracane i będą niedostępne, ponieważ metoda `getElementById` zwraca tylko instancję pojedynczego elementu HTML. W przeciwieństwie do metody `getElementsByTagName`, nie ma gwarancji, że zwrócony element będzie miał określony typ, inny niż typ elementu, którego atrybut `id` jest taki sam, jak parametr przekazywany podczas wywoływania metody `getElementById`. Dlatego też model obiektowy, którego używamy po wywołaniu tej metody, może, ale nie musi być odpowiedni dla pobranego elementu. W przypadku właściwości `innerHTML` nie stanowi to problemu, ponieważ mają ją niemal wszystkie widoczne elementy. Problem może się pojawić, jeśli założymy, że pobrany element jest tabelą, podczas gdy w rzeczywistości będzie on komórką tabeli. W takim wypadku zastosowanie do niego modelu obiektowego tabeli spowoduje powstanie wyjątku.

Podczas pisania kodu w JavaScripcie dobrą praktyką jest sprawdzanie pobranego elementu przed wykonaniem na nim jakichkolwiek działań. Musisz założyć, że podczas używania `getElementsByTagName` wiesz, jaki typ będą miały elementy HTML, ale nie wiesz, gdzie się znajdują ani co reprezentują. Z kolei podczas używania `getElementById` wiesz, co reprezentuje i gdzie się znajduje element, ale nie znasz jego typu i, co się z tym wiąże, hierarchii obiektów.

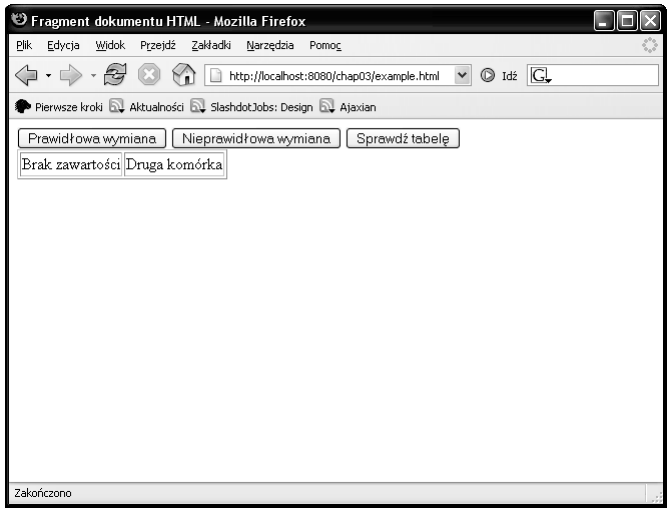
Szczególna natura innerHTML

Właściwość `innerHTML` jest szczególna, ponieważ wydaje się łatwa w użyciu, ale korzystanie z niej może wiązać się z poważnymi konsekwencjami. Żeby pokazać, gdzie leży problem, przyjrzyjmy się następującemu kodowi HTML:

```
<html>
<head>
<title>Fragment dokumentu HTML</title>
<script language="JavaScript" type="text/javascript">
function GoodReplace() {
   document.getElementById("mycell").innerHTML = "witaj";
}
function BadReplace() {
   document.getElementById("mytable").innerHTML = "witaj";
}
function TestTable() {
   window.alert(document.getElementsByTagName(
      "table")[0].rows[0].cells[0].innerHTML);
}
</script>
</head>
<body>
<button onclick="GoodReplace()">Prawidłowa wymiana</button>
```

```
<button onclick="BadReplace()">Nieprawidłowa wymiana</button>
<button onclick="TestTable()">Sprawdź tabelę</button>
<table id="mytable" border="1">
   <tr id="myrow"><td id="mycell">Brak zawartości</td><td>Druga komórka</td></tr>
</table>
</body>
</html>
```

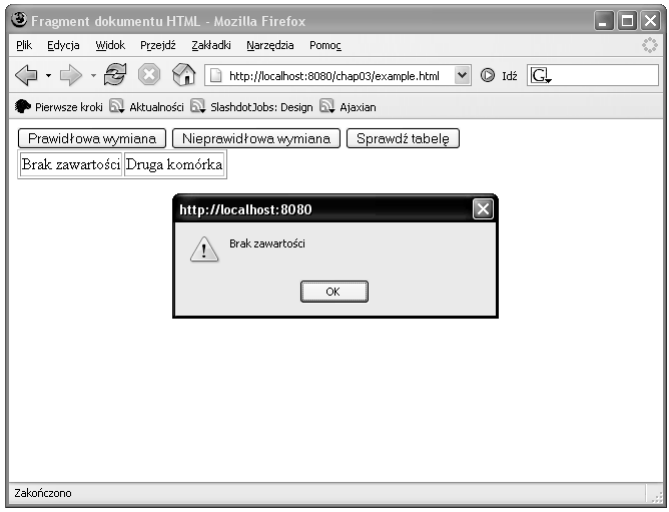
W powyższym przykładzie znajdują się trzy przyciski (*Prawidłowa wymiana*, *Nieprawidłowa wymiana* i *Sprawdź tabelę*), a elementom HTML — tabeli, jej wierszowi i komórce — nadano identyfikatory. Naciśnięcie przycisku *Prawidłowa wymiana* spowoduje prawidłowe umieszczenie kodu HTML. Naciśnięcie przycisku *Nieprawidłowa wymiana* spowoduje nieprawidłowe umieszczenie kodu HTML. Przycisk *Sprawdź tabelę* jest używany do sprawdzenia wyników umieszczenia HTML wykonanego przez naciśnięcie *Prawidłowa wymiana* albo *Nieprawidłowa wymiana*. Efekt pobrania strony HTML do przeglądarki będzie podobny do rysunku 3.5.



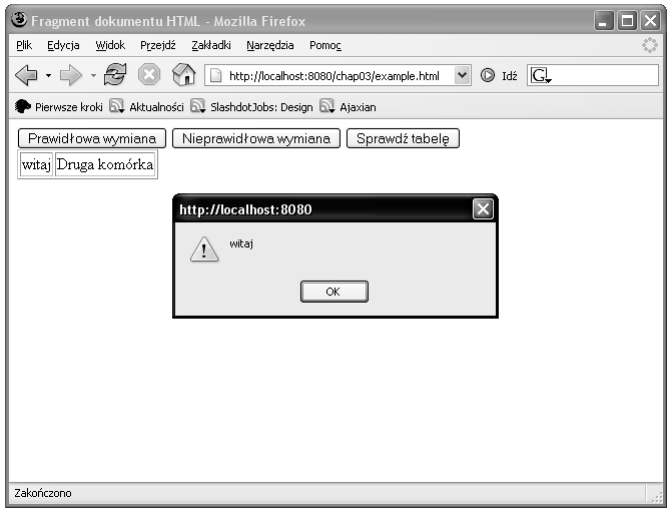
Rysunek 3.5. *Początkowa strona HTML*

Żeby sprawdzić, czy strona HTML jest prawidłowa, należy nacisnąć przycisk *Sprawdź tabelę*. Spowoduje to wywołanie funkcji `TestTable`, która sprawdza, czy istnieje zawartość wewnątrz komórek tabeli, a jeśli tak, to wyświetla ją w okienku dialogowym. Okienko dialogowe powinno wyglądać podobnie do przedstawionego na rysunku 3.6.

Okienko dialogowe na rysunku 3.6 potwierdza, że komórka tabeli zawiera wartość *Brak zawartości*. Oznacza to, że strona HTML jest prawidłowa. Po naciśnięciu przycisku *Prawidłowa wymiana* wywoływana jest funkcja `GoodReplace` zmieniająca zawartość komórki tabeli z *Brak zawartości* na *witaj*. Żeby sprawdzić, czy strona HTML jest nadal prawidłowa, należy nacisnąć przycisk *Sprawdź tabelę*. Jeśli strona HTML jest prawidłowa, powinno się pokazać okienko dialogowe z napisem *witaj* i tak się dzieje, co widać na rysunku 3.7.



Rysunek 3.6. *Wyświetlenie zawartości komórki mycell*

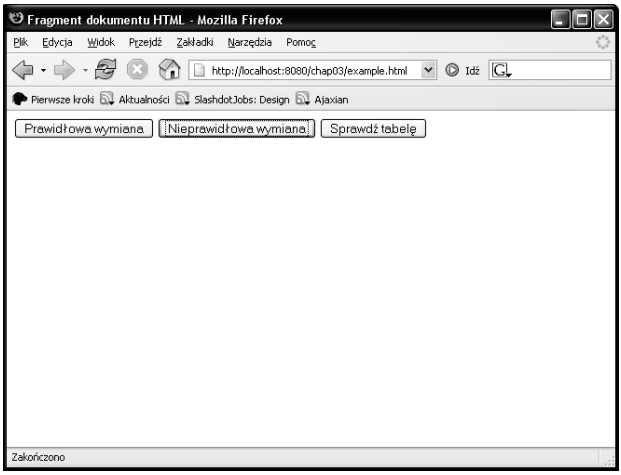


Rysunek 3.7. *Zmodyfikowana zawartość komórki mycell*

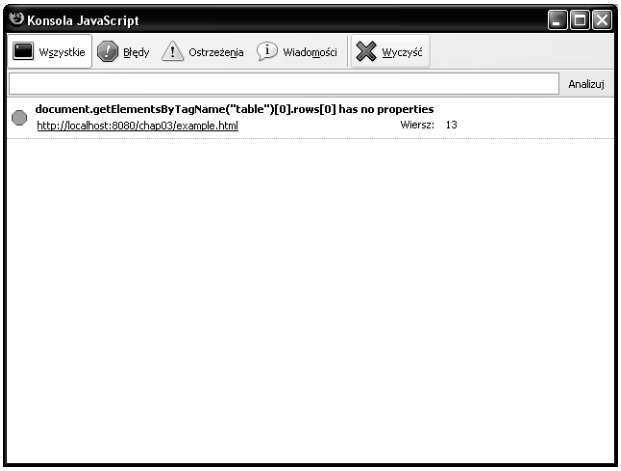
Spróbujmy rzecz nieco skomplikować przez naciśnięcie przycisku *Nieprawidłowa wymiana*. Spowoduje ono wywołanie funkcji `BadReplace`, która przypisuje nowy tekst do właściwości `innerHTML` tabeli HTML. Oznacza to, że zawartość HTML `<table><tr><td>...</table>` zostaje zamieniona na `<table>witaj</table>`. Zmieniony HTML jest nieprawidłowy i zostanie wyświetlony tak, jak pokazano na rysunku 3.8.

Na rysunku 3.8 widać, że wiersze tabeli zostały usunięte. Naciśnięcie przycisku *Sprawdź tabelę* w celu sprawdzenia poprawności strony spowoduje powstanie błędu przedstawionego na rysunku 3.9.

Rysunek 3.8.
Zawartość HTML po podmianie wierszy i komórek tabeli



Rysunek 3.9.
Wyjątek modelu obiektowego



Wygenerowany wyjątek jest istotny i wynika ze sposobu działania właściwości `innerHTML`. Podczas przypisywania zawartości HTML do `innerHTML` używa się danych tekstowych. Podczas pobierania wartości właściwości `innerHTML` elementy potomne zamieniane są na tekst. Przypisywanie wartości do właściwości `innerHTML` oznacza zamianę elementów potomnych na podany tekst HTML. Następnie HTML podany w postaci tekstu jest konwertowany na zestaw elementów HTML, które są pokazywane użytkownikowi. Funkcje `GoodReplace` i `BadReplace` są przykładem przeprowadzania operacji na właściwości `innerHTML`. Jednakże zmiana wartości właściwości `innerHTML` w przypadku, kiedy nie powinna ona być zmieniana, albo zmiana powodująca naruszenie struktury HTML spowoduje powstanie nieprawidłowości. Na przykład w powyższym przypadku nie mogłeś stworzyć tabeli bez wierszy czy komórek.

Innym sposobem operowania na obiektowym modelu dokumentu HTML jest używanie pojedynczych elementów HTML. Można tworzyć ich instancje, przeprowadzać jakieś operacje i usuwać. Dzięki wykorzystaniu obiektowego modelu dokumentu znacznie trudniej jest spowodować nieprawidłowości, ponieważ pozwala on na używanie tylko

niektórych metod i właściwości. Podczas korzystania z obiektowego modelu dokumentu HTML nie można w prosty sposób usunąć wszystkich wierszy tabeli i zamienić ich na tekst. Obiektowy model tabeli nie ma metod umożliwiających stworzenie konstrukcji przedstawionej na rysunku 3.8.

Należy pamiętać, że we wzorcu Fragmentacja Zawartości wymieniane są całe fragmenty HTML. Dlatego mimo tego, że właściwość `innerHTML` jest elastyczna i daje duże możliwości, wymiana nieodpowiedniego fragmentu w nieodpowiednim czasie może dać w efekcie źle sformatowany HTML. Powinieneś pamiętać, że podczas odwoływania się do elementów HTML w kontekście omawianego wzorca należy odwoływać się tylko do tych elementów strony szkieletowej, w których mają być umieszczane poszczególne fragmenty zawartości. Podstawową zasadą powinno być to, że skrypt w szkielecie HTML nie odwołuje się bezpośrednio do elementów w dynamicznie umieszczanych fragmentach, ponieważ tworzy to dynamiczne zależności, które mogą, ale nie muszą działać. Jeśli takie zależności są niezbędne, umieść je w metodach zawartych w umieszczanym fragmencie kodu i następnie odwołuj się tylko do tych metod. JavaScript umożliwia przypisanie określonych funkcji do elementów HTML.

Identyfikowanie elementów

Jak już wspomniałem, podczas wyszukiwania elementów z wykorzystaniem typu znacznika nie jest możliwe określenie identyfikatora, a podczas wyszukiwania z wykorzystaniem identyfikatora nie jest możliwe określenie typu znacznika. Bez względu na to, w jaki sposób zostaną znalezione elementy, będą one stanowiły początkowy punkt, w którym zostaną przeprowadzane operacje. Na podstawie tego punktu skrypt może poruszać się po elementach będących jego rodzicami albo elementami potomnymi przy wykorzystaniu standardowych właściwości i metod.

Owe standardowe właściwości i metody są dostępne niemalże dla wszystkich elementów HTML. Dlatego też programiści skryptów powinni wykorzystywać je podczas poruszania się po hierarchii elementów, zmieniania wyglądu czy przy próbie ich identyfikacji. W tabeli 3.1 przedstawiono właściwości, które mogą być wykorzystane podczas pisania skryptów.

Fragmenty w postaci danych binarnych, URL i obrazków

Tworzenie fragmentów zawartości z danych binarnych czy obrazków w ich pierwotnej postaci przy użyciu obiektu `XMLHttpRequest` jest raczej skomplikowane, ponieważ ten rodzaj danych jest uszkadzany podczas odczytu. Właściwości `responseText` i `responseXML` obiektu `XMLHttpRequest` oczekują odpowiednio tekstu albo XML-a. Użycie danych w innym formacie nie jest możliwe. Oczywiście jest wyjątek: dane binarne zakodowane w formacie Base64 są traktowane jako tekst i można je pobrać, wykorzystując obiekt `XMLHttpRequest`. Innym rozwiązaniem może być używanie odnośników do danych binarnych zamiast samych danych. W przypadku znacznika `img` oznacza to, że do atrybutu `src` należy przypisać URL wskazujący, gdzie znajduje się obrazek.

Tabela 3.1. *Właściwości elementów HTML przydatne podczas pisania skryptów*

Identyfikator właściwości	Opis
`attributes[]`	Zawiera przeznaczoną tylko do odczytu kolekcję atrybutów związanych z elementem HTML. Pojedynczy atrybut może być pobrany przy użyciu metody `getAttribute`. Do przypisania lub napisania atrybutu używana jest metoda `setAttribute`, a do jego usunięcia — `removeAttribute`.
`childNodes[]`	Do instancji `NodeList` odwołuje się najczęściej jak do tablicy, przy czym jest ona tylko do odczytu. Żeby dodać element potomny do bieżącego elementu, należy użyć `appendChild`. Do usuwania elementu potomnego używana jest metoda `removeChild`, a do zamiany `replaceChild`.
`className`	Służy do przypisywania do elementu HTML klasy z arkusza stylów. Typ `class` jest bardzo ważny w dynamicznym HTML-u, ponieważ można dzięki niemu dynamicznie zmieniać wygląd elementów HTML.
`dir`	Wskazuje na kierunek przepływu tekstu: z lewej do prawej (`ltr`) albo z prawej do lewej (`rtl`).
`disabled`	Włącza (wartość `false`) bądź wyłącza (wartość `true`) element. Jest przydatny, jeśli skrypt nie chce, żeby użytkownik nacisnął jakiś przycisk albo inny element interfejsu, zanim nie zostanie zakończona jakaś operacja.
`firstChild`, `lastChild`	Pobiera pierwszy albo ostatni węzeł potomny.
`id`	Identyfikator używany do odnalezienia danego elementu. Właściwość ta jest używana na przykład podczas wywoływania metody `getElementById`.
`nextSibling`, `previousSibling`	Pobiera następny albo poprzedni sąsiedni element. W połączeniu z `firstChild` i `lastChild` może być używany do przechodzenia przez zbiór elementów. To podejście może być używane do przechodzenia listy, w której poszczególne elementy wskazują, jaki powinien być następny element, na przykład podczas implementowania wzorca Dekorator albo podobnej struktury.
`nodeName`	Zawiera nazwę elementu, co w przypadku HTML-a oznacza nazwę znacznika (na przykład `td`, `table` itp.).
`nodeType`	Zawiera typ elementu, ale jest wykorzystywany głównie podczas przetwarzania XML-a. W odniesieniu do HTML-a ta właściwość jest mało przydatna.
`nodeValue`	Zawiera wartość danych w węźle. Ta właściwość także jest bardziej przydatna podczas przetwarzania dokumentów XML. W odniesieniu do HTML-a nie można używać tej właściwości jako zamiennika dla `innerHTML`.
`parentElement`	Pobiera element będący rodzicem bieżącego elementu. Może być używana na przykład do poruszania się po tabeli zawierającej wiersze i komórki.
`style`	Zawiera bieżące właściwości stylu elementu. Jest instancją typu `CSSStyleDeclaration`.
`tabIndex`	Określa kolejność elementu podczas wybierania klawiszem *Tab* w odniesieniu do całego dokumentu HTML.
`tagName`	Zawiera nazwę znacznika bieżącego elementu. Ta właściwość może być użyta podczas próby określenia typu elementu po pobraniu go metodą `getElementById`.

Obrazki pobierane są pośrednio. Żeby zrozumieć sposób, w jaki się to odbywa, spójrzmy na aplikację, która wykorzystuje `XMLHttpRequest` do pobrania dokumentu zawierającego jedną linijkę. Linijka ta zawiera URL do pliku z obrazkiem.

Poniżej znajduje się implementacja przykładowego programu:

```
<html>
<head>
<title>Fragment dokumentu HTML z obrazkiem</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">

var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
   document.getElementById("image").src = responseText;
}

</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkimage01.html')">Pobierz
obrazek</button>
<br>
<img id="image" />
</body>
</html>
```

Do pobrania obrazka używany jest znacznik `img`. W większości obrazków znacznik `img` jest określony przy użyciu atrybutu `src` będącego odnośnikiem do miejsca, w którym się one znajdują. W tym przykładzie nie ma atrybutu `src`, zamiast tego jest `id`. Po wczytaniu i pokazaniu strony HTML wyświetlany jest uszkodzony obrazek, ponieważ ze znacznikiem `img` nie jest związany żaden plik. Żeby znacznik `img` wyświetlał się prawidłowo, należy nacisnąć przycisk *Pobierz obrazek*, który wykonuje żądanie pobierające jednolinijkowy plik zawierający URL obrazka. Po pobraniu jednolinijkowego pliku przez `XMLHttpRequest` wywoływana jest implementacja funkcji właściwości `complete`. Przypisuje ona do atrybutu (właściwości) `src` URL zdalnego obrazka. Następnie przeglądarka aktualizuje dane, pobiera obrazek i wyświetla go.

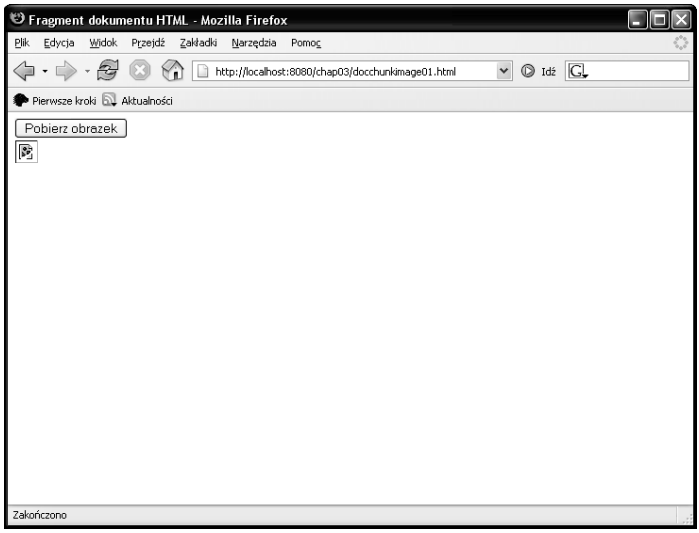
Jednolinijkowy plik znajduje się pod adresem URL */chap03/chunkimage01.html* i ma następującą zawartość:

```
/static/patches01.jpg
```

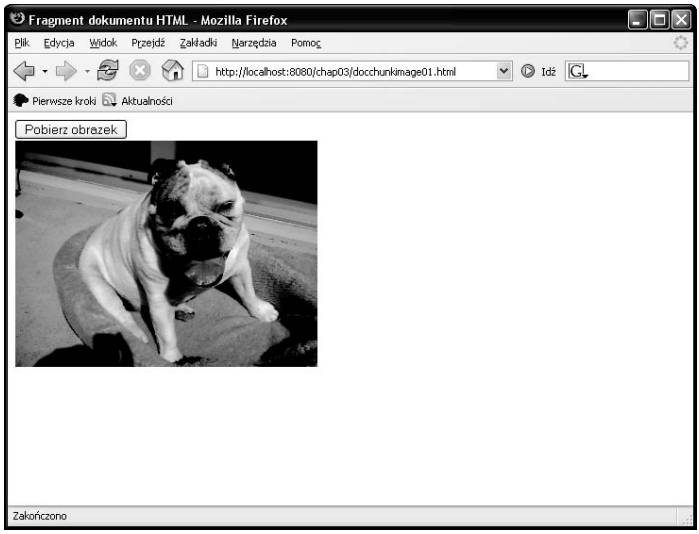
Gdy omawiana strona HTML, w której nie został określony atrybut `src`, zostanie pobrana, będzie ona wyświetlona podobnie jak na rysunku 3.10.

Na rysunku 3.10, poniżej przycisku *Pobierz obrazek* widać mały prostokąt wskazujący nieprawidłowy znacznik `img`, który nie spowodował wczytania żadnego obrazka. Po naciśnięciu przycisku *Pobierz obrazek* pobierane jest łącze do obrazka. Następnie jest ono przypisywane do znacznika `img`, co powoduje wczytanie obrazka. Na rysunku 3.11 widać przetworzoną stronę.

Rysunek 3.10. *Początkowa strona HTML bez obrazka*



Rysunek 3.11. *Strona HTML po wczytaniu obrazka*



Pobieranie i przypisywanie łącza, które jest następnie przetwarzane przez przeglądarkę, wydaje się dziwne. To pośrednie podejście nie ma przedstawiać, jak skomplikowana może być aplikacja internetowa, ale jest potrzebne, ponieważ nie ma możliwości bezpośredniego pobierania danych binarnych. Jednak nie jest to aż tak wielki problem, ponieważ przeglądarka wczytuje obrazek, do którego się odnosimy, i umieszcza go w swojej pamięci podręcznej. Jeśli będziemy się do niego ponownie odnosić, zostanie on pobrany właśnie z pamięci podręcznej przeglądarki. Oczywiście dzieje się tak tylko w przypadku, kiedy serwer HTTP implementuje pamięć podręczną. Jest jeden minus tego rozwiązania. Jeśli wykonujemy żądanie pobierające URL, który jest odnośnikiem do obrazka, wymagane są dwa żądania: jedno do pobrania adresu URL obrazka i drugie do pobrania jego samego. Jeśli oba żądania używają protokołu HTTP 1.1, a tak będzie w większości przypadków, zostaną one wykonane podczas jednego połączenia.

Kolejną odmianą przedstawionej strategii jest pobranie całego kodu HTML tworzącego obrazek, a nie samego adresu URL. Ta strategia nie zachowuje połączenia żądania, ale jest samodzielnym rozwiązaniem, które nie wymaga dodatkowego skryptu. Poniższy fragment przedstawia, w jaki sposób pobierany jest cały znacznik `img`.

```
<img src="/static/patches01.jpg" />
```

Zarówno podczas umieszczania znacznika `img`, jak i odpowiedniego atrybutu `src` przeglądarka dynamicznie pobierze obrazek, jak to pokazano w poprzednim przykładzie. Zaletą umieszczania kodu HTML jest to, że aplikacja po stronie serwera może umieścić wiele obrazków albo inny rodzaj HTML. Ponadto podczas umieszczania całego znacznika `img` nie występuje pierwszy etap, w którym wyświetlany jest uszkodzony obrazek. Jednakże oba podejścia są tak samo dobre, a ich wybór zależy od rodzaju aplikacji. Kiedy na stronie umieszczany jest HTML, może ona migotać, gdy będzie zmieniany jej rozmiar. Kiedy przypisujesz wartość do właściwości `src`, nie występuje migotanie, ale trzeba określić pusty obrazek albo ukryć element obrazka.

Fragmentacja JavaScriptu

Kolejnym rodzajem fragmentacji jest wysyłanie JavaScriptu. Może ono być bardzo efektywne, ponieważ nie musisz przetwarzać danych, a tylko wykonać JavaScript. Z punktu widzenia skryptu po stronie klienta jest to bardzo łatwe do zaimplementowania. Nie powinieneś jednak zakładać, że pobranie JavaScriptu będzie szybsze niż parsowanie i przetworzenie danych XML, a następnie skonwertowanie ich przed wykonaniem. Zaletą podejścia wykorzystującego JavaScript jest prostota i efektywność. Po prostu łatwiej jest wykonać fragment JavaScriptu i odnosić się do udostępnionych właściwości i funkcji.

Wykonywanie JavaScriptu

Rozpatrzmy poniższy kod HTML, który wykonuje określony JavaScript:

```
<html>
<head>
<title>Fragment zawartości HTML z JavaScriptem</title>
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">

var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
   eval(responseText);
}

</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkjs01.html')">Pobierz
skrypt</button>
<table>
```

```
<tr><td id="insertplace">Brak zawartości</td></tr>
</table>
</body>
</html>
```

Po naciśnięciu przycisku *Pobierz skrypt* wykonywane jest żądanie `XMLHttpRequest` pobierające dokument */chap03/chunkjs01.html*. Dokument zawiera fragment JavaScriptu, który jest wykonywany za pomocą funkcji `eval`. Pobierany jest następujący fragment kodu:

```
window.alert("hurra, wywołane dynamicznie");
```

Przykładowy fragment nie jest zbytnio rozbudowany i wyświetla okienko dialogowe. Tym, co niepokoi wielu ludzi podczas wykonywania takiego kodu JavaScript, jest to, że jest on pobrany z zewnątrz. Administrator i użytkownik może zastanawiać się nad konsekwencjami związanymi z bezpieczeństwem, ponieważ pobrany JavaScript może zawierać wirusy. Jednakże tak właściwie nie jest to możliwe, gdyż JavaScript jest wykonywany w piaskownicy i dotyczy go polityka tego samego źródła. Oczywiście jeśli programista ominie politykę tego samego źródła, problemy związane z bezpieczeństwem nabiorą znaczenia.

Najprostszą implementacją pobierania fragmentu kodu do wykonania jest dynamiczne tworzenie po stronie serwera fragmentów JavaScriptu, które wykonują jakieś metody. Fragmenty JavaScriptu powodują, że przeglądarka wykonuje jakieś polecenia. Na przykład fragment JavaScriptu pobierany w poprzednim przykładzie mógłby być wykorzystywany do przypisania jakichś danych do znacznika `span` albo `td`, co przedstawiono poniżej:

```
document.getElementById("mycell").innerHTML = "witaj";
```

Wygenerowany skrypt jest zakodowany na sztywno, ponieważ oczekuje, że na docelowej stronie HTML będą znajdowały się określone elementy.

Generowanie JavaScriptu operującego na obiektowym modelu dokumentu

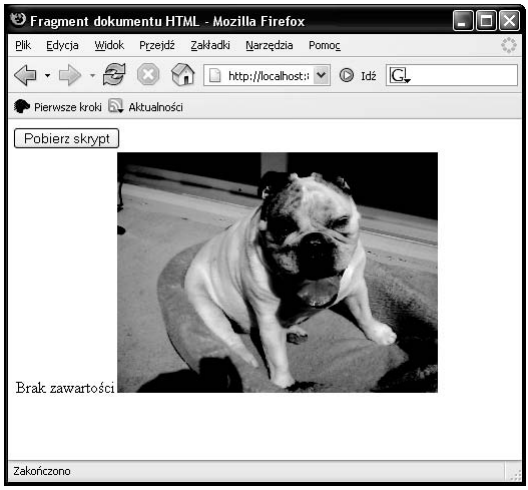
Wcześniej widziałeś rozwiązanie, w którym obrazek początkowo był uszkodzony, a następnie wyświetlony po pobraniu prawidłowego łącza. Obrazek można także pobrać poprzez zmianę obiektowego modelu dynamicznego HTML-a. Model obiektowy zmieniasz poprzez wykorzystanie fragmentu JavaScriptu do wstawienia znacznika `img`. Poniżej znajduje się fragment JavaScriptu tworzący znacznik `img` i wstawiający go do dokumentu HTML.

```
var img = new Image();
img.src = "/static/patches01.jpg";
document.getElementById("insertplace").appendChild(img);
```

W tym przykładzie zmienna `img` jest instancją obiektu `Image`, który odnosi się do znacznika HTML `img`. Do właściwości `src` przypisywany jest adres URL obrazka. Ostatnia linia tego fragmentu kodu używa metody `appendChild`, żeby dodać instancję obiektu `Image` do dokumentu HTML. Jeśli zmienna `img` nie zostanie powiązana z dokumentem

HTML, obrazek zostanie pobrany, ale nie będzie dodany do dokumentu HTML i w związku z tym nie będzie wyświetlony. Strona HTML wygenerowana po wykonaniu powyższego skryptu pokazana jest na rysunku 3.12.

Rysunek 3.12. *Strona HTML wygenerowana po wstawieniu obrazka*



Rysunek 3.12 nie jest zbyt widowiskowy, ponieważ ilustruje tylko sposób, w jaki obrazek może być wstawiony na stronę HTML. Interesujące jest to, że pozostał tekst *Brak zawartości*, który nie został zamieniony tak jak w poprzednich przykładach. Przyczyna tego tkwi w tym, że została wykorzystana metoda `appendChild` (a nie `replaceChild` albo `removeChild` i następnie `appendChild`).

Zaletą podejścia wykorzystującego model obiektowy dynamicznego HTML-a jest to, że umożliwia ono pobieranie obrazków albo innych elementów w tle, a moment ich wyświetlenia może być określony w skrypcie.

Tworzenie instancji obiektu

Innym rodzajem fragmentów JavaScriptu, które mogą być pobierane, są stany obiektów. Przy wykorzystaniu stanu obiektu możesz wprowadzić poziom pośredni umożliwiający dodawanie funkcji podczas wykonywania strony HTML. We wszystkich poprzednich przykładach fragmenty kodu HTML na początkowej stronie HTML musiały mieć wszystkie skrypty i znać adresy URL pobieranych zasobów. Dzięki wykorzystaniu pośredniego poziomu JavaScript po stronie klienta nie musi znać szczegółów adresów URL czy struktur danych. Klient odwołuje się do ogólnego fragmentu kodu, który jest wykonywany. Ogólny fragment kodu jest zarządzany przez serwer i zawiera poszczególne instrukcje wykonujące, które mogą wykonywać operacje niezaprogramowane pierwotnie po stronie klienta. Używanie pośredniego poziomu umożliwia dodawanie do klienta funkcji, których nie miał on w czasie projektowania aplikacji.

Przyjrzyjmy się poniższemu przykładowi strony HTML:

```
<html>
<head>
<title>Fragment zawartości HTML z JavaScriptem</title>
```

```
<script language="JavaScript" src="/lib/factory.js"></script>
<script language="JavaScript" src="/lib/asynchronous.js"></script>
<script language="JavaScript" type="text/javascript">
var asynchronous = new Asynchronous();
asynchronous.complete = function(status, statusText, responseText, responseXML) {
   eval(responseText);
   dynamicFiller.makeCall(document.getElementById("insertplace"));
}
</script>
</head>
<body>
<button onclick="asynchronous.call('/chap03/chunkjs04.js')">Rozpocznij
przetwarzanie</button>
<table>
   <tr><td id="insertplace">Brak zawartości</td></tr>
</table>
</body>
</html>
```

Podobnie jak w poprzednich przykładach tworzona jest instancja `Asynchronous`. Element `button` powoduje wykonanie asynchronicznego żądania z adresem URL */chap03/chunkjs04.js*. Kiedy żądanie pobierze fragment JavaScriptu, jest on wykonywany przez instrukcję `eval`. Po powrocie z instrukcji `eval` wykonywana jest metoda `dynamicFiller.makeCall`, której wywołanie jest ogólnym fragmentem kodu. Implementacja tej metody jest kodem zarządzanym przez serwer, a odniesienie się do niej jest wykonywane przy użyciu niekompletnej zmiennej, ponieważ początkowy skrypt nie zawiera definicji zmiennej `dynamicFiller`. Oczywiście wczytany i przetworzony skrypt nie może odnosić się do niekompletnej zmiennej, ponieważ spowodowałoby to powstanie wyjątku. Ale tym, co może zrobić skrypt, jest wczytanie implementacji, zanim zostanie użyta niekompletna zmienna. I to właśnie jest przedstawione w powyższej stronie HTML. Uprzedzam pytanie: w plikach *factory.js* i *asynchronous.js* nie ma definicji zmiennej `dynamicFiller`. JavaScript umożliwia stosowanie niekompletnych zmiennych, typów i funkcji, co pozwala na wczytywanie i przetwarzanie skryptów bez wywoływania wyjątków.

Poniższy kod źródłowy zawiera implementację niekompletnej zmiennej `dynamicFiller`:

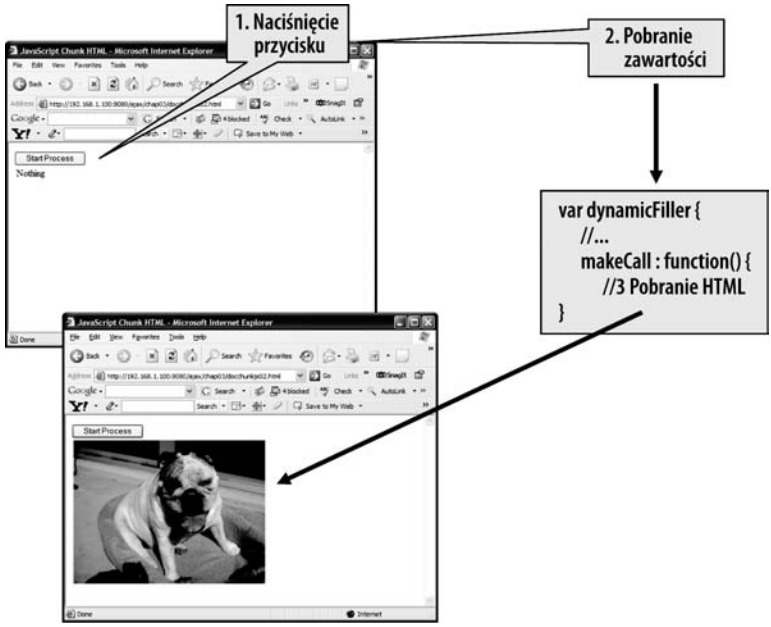
```
var dynamicFiller = {
   generatedAsync : new Asynchronous(),
   reference : null,
   complete : function(status, statusText, responseText, responseXML) {
      dynamicFiller.reference.innerHTML = responseText;
   },
   makeCall : function(destination) {
      dynamicFiller.reference = destination;
      dynamicFiller.generatedAsync.complete = dynamicFiller.complete;
      dynamicFiller.generatedAsync.call('/chap03/chunkjs05.html');
   }
}
```

Przykładowy kod źródłowy JavaScriptu jest napisany przy wykorzystaniu inicjalizatora obiektu. Inicjalizator obiektu jest zapisaną postacią obiektu JavaScript. Nie powinieneś porównywać go do definicji klasy JavaScript. To dwie zupełne różne rzeczy. Podczas

przetwarzania inicjalizatora obiektu tworzona jest instancja obiektu, której identyfikatorem jest zadeklarowana zmienna. W tym przykładzie instancją obiektu jest zmienna `dynamicFiller`.

Zmienna `dynamicFiller` ma dwie właściwości (`generatedAsync` i `reference`) oraz dwie metody (`complete` i `makeCall`). Właściwość `generatedAsync` jest instancją `Asynchronous` wykorzystywaną do wykonywania asynchronicznych żądań do serwera. Właściwość `reference` jest elementem HTML, na którym metoda `complete` będzie przeprowadzać operacje. Metoda `makeCall` jest wykorzystywana do wykonywania żądania `XMLHttpRequest`, a do parametru `destination` przypisana jest właściwość `reference`.

Po połączeniu wszystkich fragmentów kod HTML szkieletu zawiera ogólny kod, który ma odniesienie do niekompletnej zmiennej. Żeby uzyskać pełną zmienną, pobierana i wykonywana jest zawartość w języku JavaScript. Pełna zmienna zawiera kod pobierający zawartość, która jest umieszczana w stronie szkieletowej. Na rysunku 3.13 przedstawiono sekwencję wykonywanych zdarzeń.



Rysunek 3.13. *Sekwencja zdarzeń podczas pobierania i wykonywania JavaScriptu*

Na rysunku 3.13 widać początkową stronę pobieraną po naciśnięciu przycisku. Pobieraną zawartością jest JavaScript, przy czym początkowa strona nie wie, co on wykonuje. Zawartość jest pobierana i następnie wykonywana. W szkieletowej stronie HTML zaprogramowano odniesienie do zmiennej `dynamicFiller` i wywołanie metody `makeCall`. Metoda `makeCall` nie istnieje podczas wykonywania szkieletowej strony HTML i staje się dostępna dopiero po pobraniu i wykonaniu zawartości. Pobrana zawartość jest wykonywana i pobiera kolejny fragment zawartości, który jest umieszczany w stronie HTML. W wyniku tego pobierany jest obrazek wyświetlany w miejscu, gdzie znajdował się tekst *Nothing*.

Rola szkieletowej strony HTML zmieniła się. Teraz stała się ona stroną inicjalizującą, która pobiera inne fragmenty kodu. Pobrane fragmenty kodu są całkowicie dynamiczne i zawierają odniesienia i kod, który nie jest znany w szkieletowej stronie HTML. Zaletą tej implementacji jest to, że dokument może być pobierany stopniowo poprzez pobieranie fragmentów kodu, które są określane dopiero po ich pobraniu. Wzorzec Fragmentacja Zawartości umożliwia dynamiczne ładowanie zawartości, a dodanie wczytywania JavaScriptu umożliwia dynamiczne określanie logiki używanej przez szkieletową stronę HTML.

Najważniejsze elementy wzorca

Poniżej wymienione są najważniejsze elementy wzorca Fragmentacja Zawartości:

- Strona HTML jest połączeniem szkieletowej strony HTML i fragmentów zawartości.
- Szkieletowa strona HTML jest odpowiedzialna za organizowanie, odniesienia i pobieranie odpowiednich fragmentów. Powinna sterować poszczególnymi fragmentami. Szkieletowa strona HTML przenosi przetwarzanie fragmentów zawartości do innych fragmentów kodu.
- Fragmenty zawartości są unikalnie identyfikowane przez URL. Różne fragmenty zawartości nie używają tych samych adresów URL. Fragmenty zawartości są używane do zaimplementowania funkcji określanych przez użytkownika.
- Fragmenty zawartości mogą przyjmować jedną z trzech postaci: XML (preferowany), HTML (preferowany XHTML) albo JavaScript. Są jeszcze inne rodzaje zawartości, ale nie będą one omawiane w tej książce, a korzystanie z nich powinno być dobrze przemyślane.