Spis treści

Wprowadzenie ix

1. Pobieranie i wykonanie 1

Pozycjonowanie skryptu 2
Grupowanie skryptów 4
Skrypty nieblokujące 6
  Skrypty odroczone 6
  Dynamiczne elementy skryptu 8
  Wstrzykiwanie skryptu za pomocą *XMLHttpRequest* *10*
  Zalecany szablon nieblokujący 11
Podsumowanie 16

2. Dostęp do danych 17

Zarządzanie zasięgiem 18
  Łańcuchy zasięgów i określanie identyfikatorów 19
  Wydajność odszukiwania identyfikatorów 21
  Rozszerzenie łańcucha zasięgu 24
  Zasięg dynamiczny 26
  Zamknięcia, zasięgi i pamięć 27
Elementy obiektów 29
  Prototypy 30
  Łańcuchy prototypów 31
  Elementy zagnieżdżone 33
  Przechwytywanie wartości elementów obiektów 34
Podsumowanie 36

3. Skrypty w modelu DOM 37

DOM w świecie przeglądarek 37
  Powolny z natury 38
Dostęp do DOM i jego modyfikacja 38
  innerHTML a metody DOM 40
  Klonowanie węzłów 43
  Kolekcje HTML 44
  Przejście przez DOM 49
Przemalowywanie i ponowne wlewanie 53
  Kiedy ma miejsce ponowne wlewanie? 54
  Kolejkowanie i czyszczenie zmian drzewa renderowania 54
  Minimalizacja przemalowywania i ponownego wlewania 56

Przechwytywanie informacji o układzie ... 59
Wyciąganie elementów z przepływu do wykonania animacji ... 60
IE oraz *:hover* ... *61*
Delegowanie zdarzeń ... 61
Podsumowanie ... 63
4. Algorytmy i sterowanie przepływem ... 65
Pętle ... 65
Typy pętli ... 65
Wykonywanie pętli ... 67
Iteracja oparta na funkcji ... 72
Instrukcje warunkowe ... 73
if-else kontra switch ... 73
Optymalizacja *if-else* ... 75
Tablice wyszukiwania ... 77
Rekurencja ... 79
Limit stosu wywołań ... 79
Wzory rekurencyjne ... 81
Iteracja ... 81
Memoizacja ... 83
Podsumowanie ... 85
5. Łańcuchy i wyrażenia regularne ... 87
Łączenie łańcuchów ... 88
Operatory plus (+) oraz plus-równa się (+=) ... 88
Łączenie tablicy ... 91
String.prototype.concat ... 93
Optymalizacja wyrażeń regularnych ... 93
Jak działa wyrażenie regularne ... 94
Pojęcie nawracania ... 95
Niekontrolowane nawracanie ... 99
Uwaga na temat testów porównawczych ... 103
Więcej metod poprawiania wydajności wyrażenia regularnego ... 104
Kiedy nie korzystać z wyrażeń regularnych ... 107
Przycinanie łańcuchów ... 108
Przycinanie za pomocą wyrażeń regularnych ... 108
Przycinanie bez wyrażeń regularnych ... 111
Rozwiązanie hybrydowe ... 112
Podsumowanie ... 114
6. Reagujące interfejsy ... 115
Wątek interfejsu użytkownika przeglądarki ... 115

Ograniczenia przeglądarek 117
Co oznacza zbyt długo? 119
Działanie według zegarów 120
Podstawowe wiadomości o zegarach 120
Dokładność zegara 123
Przetwarzanie tablic z zegarami 124
Dzielenie zadań 126
Kod z ograniczeniem czasowym 128
Zegary i wydajność 129
Wątki robocze Web 130
Środowisko wątków roboczych 130
Komunikacja wątków roboczych 131
Ładowanie plików zewnętrznych 132
Praktyczne zastosowania 132
Podsumowanie 134

7. Ajax 135
Transmisja danych 135
Żądanie danych 135
Wysyłanie danych 142
Formaty danych 145
XML 146
JSON 150
HTML 153
Formatowanie niestandardowe 155
Wnioski dotyczące formatu danych 157
Wskazówki dotyczące wydajności Ajax 159
Buforowanie danych 159
Poznanie ograniczeń swojej biblioteki Ajax 161
Podsumowanie 163

8. Praktyki programowania 165
Unikanie podwójnego interpretowania 165
Używanie literałów obiektów lub tablic 167
Bez powtarzania pracy 168
Leniwe ładowanie 169
Warunkowe wcześniejsze ładowanie 170
Zastosowanie szybkich części 171
Operatory bitowe 171
Metody wbudowane 174
Podsumowanie 176

9. Tworzenie i wdrażanie wysoko wydajnych aplikacji JavaScript....... 177
Apache Ant 178
Łączenie plików JavaScript 179
Wstępne przetwarzanie plików JavaScript......................180
Minifikacja JavaScript.......................................183
Procesy czasu kompilacji a czas wykonywania..................185
Kompresja JavaScript 185
Buforowanie plików JavaScript................................187
Obejście problemów buforowania 188
Zastosowanie sieci dostarczania zawartości 189
Wdrażanie zasobów JavaScript.................................189
Sprawny proces kompilacji JavaScript 190
Podsumowanie...192

10. Narzędzia.. 193
Profilowanie JavaScript 194
YUI Profiler...195
Funkcje anonimowe 199
Firebug 200
Panel konsoli programu Profiler 200
Konsola API 201
Panel Net..203
Narzędzia programistyczne w Internet Explorer................204
Web Inspector w Safari.......................................206
Panel Profiles 206
Panel Resources..209
Narzędzia deweloperskie w Chrome 210
Blokowanie skryptów..211
Page Speed...212
Fiddler..214
YSlow 216
dynaTrace Ajax Edition 217
Podsumowanie...220

Indeks... 221
O autorze.. 229
Kolofon.. 230

Wprowadzenie

Na początku, gdy w roku 1996 wprowadzono JavaScript jako część Netscape Navigator, wydajność pracy nie była zbyt ważna. Internet był jeszcze w powijakach i sam z siebie powolny. Począwszy od komutowanych połączeń przez linię telefoniczną, po domowe komputery o słabej mocy, wszystkie te cechy sprawiały, że surfowanie w sieci było często lekcją cierpliwości. Użytkownicy wiedzieli, że muszą czekać na załadowanie stron sieciowych, a gdy zakończyło się to sukcesem, stanowiło powód do radości.

Pierwotnym celem języka JavaScript była poprawa komfortu obsługi użytkownika w witrynach. Zamiast ciągłego powracania do serwera przy każdym najprostszym zadaniu, takim jak sprawdzenie formularza, JavaScript pozwalał na zagnieżdżanie tej funkcji bezpośrednio na stronie. Takie postępowanie oszczędzało czas długiej, ponownej wyprawy do serwera. Wyobraźmy sobie frustrację w sytuacji wypełniania długiego formularza, przesłania go i oczekiwania 30–60 sekund, aby otrzymać komunikat o tym, że jedno z pól zostało nieprawidłowo wypełnione. We wczesnym okresie Internetu można przypisać językowi JavaScript bezsprzeczne zasługi w znacznym oszczędzeniu czasu użytkowników.

Ewolucja Internetu

W ciągu następnej dekady nastąpiła dalsza ewolucja komputerów i Internetu. Po pierwsze stały się one szybsze. Gwałtowne przyśpieszenie mikroprocesorów, dostęp do taniej pamięci oraz pojawienie się połączeń światłowodowych wprowadziły Internet do nowej epoki. Przy szerszym niż kiedykolwiek dostępie do połączeń o dużej szybkości, witryny zaczęły stawać się cięższe, obejmując coraz więcej informacji i multimediów. Sieć uległa przemianie od dość płaskiego krajobrazu powiązanych ze sobą dokumentów, po obszar wypełniony różnymi projektami i interfejsami. Zmieniło się wszystko oprócz JavaScript.

Wcześniejsze metody stosowane do zmniejszenia liczby połączeń z serwerem stały się przestarzałe. Tam, gdzie kiedyś były dziesiątki linii kodu JavaScript, teraz były ich setki, a nawet tysiące. Wprowadzenie Internet Explorer 4 oraz dynamicznego HTML (możliwości zmiany aspektów strony bez przeładowywania) sprawiło, że liczba stron JavaScript stale się zwiększa.

Ostatnim znaczącym krokiem w ewolucji przeglądarek stało sie wprowadzenie modelu DOM (Document Object Model), ujednoliconego podejścia do dynamicznego HTML, które zostało przyjęte przez Internet Explorer 5, Netscape 6 i Opera. Wkrótce potem nastąpiła standaryzacja JavaScript do trzeciej edycji ECMA-262. Biorąc pod uwagę, że wszystkie przeglądarki obsługują model DOM oraz mniej więcej tę samą wersję JavaScript, narodziła się platforma aplikacji Web. Pomimo tego znacznego skoku, standardowych API, na których można się opierać pisząc JavaScript, maszyny JavaScript odpowiadające za wykonywanie kodu pozostały w większości niezmienione.

Dlaczego optymalizacja jest konieczna

Maszyny JavaScript, które w 1996 r. obsługiwały witryny z kilkudziesięcioma wierszami kodu JavaScript, są nadal takie same, jednak dziś obsługują tysiące wierszy kodu JavaScript. Pod wieloma względami zarządzanie językiem JavaScript i działania wstępne w przeglądarkach nie nadąża za potrzebami, co pozwoliłoby mu odnieść prawdziwy sukces. Stało się to oczywiste przy wprowadzeniu Internet Explorer 6, który na początku był chwalony za stabilność i szybkość, jednak szybko zaczęto go krytykować jako okropną, powolną i zawierającą błędy platformę.

W rzeczywistości IE 6 wcale nie stał się wolniejszy, musiał po prostu wykonywać więcej zadań niż wcześniej. Wczesne, tworzone w 2001 r. (gdy wprowadzano IE6) aplikacje Web były o wiele lżejsze i w mniejszym stopniu korzystały z JavaScript niż aplikacje tworzone w roku 2005. Różnica ilości kodu JavaScript stała się oczywista, gdy pojawiły się pierwsze problemy związane ze statyczną procedurą oczyszczania pamięci. Aby ustalić, czy należy oczyścić pamięć, maszyna JavaScript sprawdzała, czy liczba obiektów w pamięci przekroczyła ustaloną wartość. Wczesne aplikacje Web rzadko zbliżały się do tego progu, ale więcej kodu JavaScript oznacza więcej obiektów, więc bardziej złożone aplikacje Web zaczęły częściej go osiągać. Problem stał się wyraźny: programiści JavaScript i aplikacje Web szli do przodu, a maszyny JavaScript nie.

Pomimo że inne przeglądarki miały bardziej logiczne i nieco wydajniejsze procedury czyszczenia pamięci, do wykonywania kodu w większości z nich nadal był używany interpreter JavaScript. Interpretacja kodu jest z założenia wolniejsza niż kompilacja, gdyż trzeba uruchamiać proces translacji kodu na instrukcje komputera. Niezależnie od tego, jak sprytne i zoptymalizowane będą interpretery, zawsze będą negatywnie wpływać na wydajność.

Kompilatory są pełne różnego rodzaju optymalizatorów, które pozwalają programistom na dowolne pisanie kodu bez zastanawiania się, czy jest on najlepszy, czy nie. Kompilator może dokonać analizy składniowej i określić, co kod próbuje zrobić. Następnie go zoptymalizuje, tworząc najszybszy kod maszynowy do wykonania zadania. Interpretery mają

niewiele tego typu optymalizacji, co zwykle oznacza, że kod jest wykonywany dokładnie tak, jak został napisany.

W efekcie JavaScript zmusza programistę do wykonywania optymalizacji, którymi w innych językach zajmuje się kompilator.

Maszyny JavaScript nowej generacji

W roku 2008 pojawiło się pierwsze znaczące zwiększenie możliwości maszyn JavaScript. Google wprowadził na rynek swoją nową przeglądarkę o nazwie Chrome. Jest to pierwsza przeglądarka wyposażona w maszynę optymalizującą JavaScript, określaną jako V8. Maszyna V8 w JavaScript jest maszyną kompilacyjną typu JIT (ang. *just-in-time*), która tworzy kod maszynowy na podstawie kodu JavaScript, a potem go wykonuje. Daje to w wyniku niezwykle szybkie wykonanie JavaScript.

Wkrótce tą drogą podążyły inne przeglądarki, oferując własne maszyny optymalizujące JavaScript. Funkcja Safari 4 o nazwie Squirrel Fish Extreme (nazywana także Nitro) to maszyna JIT JavaScript, zaś Firefox 3.5 zawiera maszynę TraceMonkey, która optymalizuje często wykonywane ścieżki kodu.

W przypadku tych nowszych maszyn JavaScript, optymalizacja wykonywana jest na poziomie kompilacji, czyli tam gdzie powinna. Kiedyś programiści zostaną całkiem uwolnieni od zmartwień związanych z optymalizacją swego kodu. Ten dzień jeszcze jednak nie nadszedł.

Wydajność wciąż jest problemem

Pomimo postępów w podstawowym wykonaniu skryptów JavaScript, wciąż istnieją aspekty JavaScript, których nie obsługują nowe maszyny. Opóźnienia powodowane przez sieci oraz działania wpływające na wygląd strony są nadal obszarami, których przeglądarki odpowiednio nie optymalizują. Podczas gdy proste operacje, jak algorytmy wstawiania funkcji, łamania kodu oraz łączenia łańcuchów, można łatwo zoptymalizować w kompilatorach, to dynamiczne i wieloskładnikowe struktury aplikacji sprawiają, że ta optymalizacja rozwiązuje tylko część problemów z wydajnością.

Choć nowsze maszyny JavaScript dały nam pojęcie, jak wygląda o wiele szybszy Internet, dzisiejsze lekcje dotyczące wydajności będą nadal potrzebne i ważne w dającej się przewidzieć przyszłości.

Techniki i podejścia podawane w tej książce dotyczą wielu różnych aspektów JavaScript, które obejmują czas wykonania, pobieranie, interakcje z modelem DOM, cykl życia strony i wiele innych. Tylko niewielki podzbiór tej tematyki, związany z podstawową wydajnością (ECMAScript), będzie mógł stać się nieistotny dzięki postępom maszyn JavaScript, ale ten czas dopiero nadejdzie.

Inne zagadnienia obejmują obszary, w których nie pomogą szybsze maszyny JavaScript: interakcję z DOM, opóźnienia sieci, blokowanie i jednoczesne pobieranie JavaScript oraz wiele więcej. Tematy te nie tylko pozostaną ważne, ale staną się przedmiotem dalszych badań, w miarę jak będzie poprawiał się czas wykonania JavaScript na niższym poziomie.

Jak zbudowana jest ta książka

Kolejne rozdziały książki zorganizowano według normalnego cyklu życia tworzenia JavaScript. Rozdział 1 rozpoczyna od omówienia optymalnych metod ładowania JavaScript na stronę. Rozdziały od 2 do 8 skupiają się na specjalnych technikach programowania, dzięki którym kod JavaScript może działać możliwie szybko. W rozdziale 9 omówiono najlepsze sposoby budowy i wdrażania plików JavaScript w środowisku eksploatacyjnym, zaś rozdział 10 obejmuje narzędzia wydajności, które mogą pomóc w identyfikacji kolejnych problemów już po wdrożeniu kodu. Pięć rozdziałów powstało przy współpracy z innymi autorami:

- Rozdział 3, *Skrypty DOM*, autor Stoyan Stefanov
- Rozdział 5, *Łańcuchy i wyrażenia regularne*, autor Steven Levithan
- Rozdział 7, *Ajax*, autor Ross Harmes
- Rozdział 9, *Budowa i wdrażanie wydajnych aplikacji JavaScript*, autor Julien Lecomte
- Rozdział 10, *Narzędzia*, autor Matt Sweeney

Każdy z tych autorów jest uznanym twórcą aplikacji Web i wniósł istotny wkład w działanie całej społeczności programistów Web. Ich nazwiska znajdują się na pierwszej stronie odpowiednich rozdziałów, aby można było łatwo określić ich autora.

Ładowanie JavaScript

Rozdział 1, *Pobieranie i wykonanie* rozpoczyna się od podstaw JavaScript: ładowania kodu na stronę. Wydajność JavaScript naprawdę zaczyna się od pobierania kodu na stronę w możliwie wydajny sposób. W rozdziale skupiono się na problemach wydajności związanych z pobieraniem kodu JavaScript i pokazano kilka sposobów ich ograniczenia.

Technika kodowania

Istotnym źródłem problemów z wydajnością w JavaScript jest źle napisany kod, który wykorzystuje nieefektywne algorytmy lub narzędzia. Kolejnych siedem rozdziałów skupia się na identyfikacji stwarzającego problem kodu i prezentacji szybszych alternatywnych rozwiązań realizujących to samo zadanie.

Tematem rozdziału 2, *Dostęp do danych*, jest sposób przechowywania i udostępniania danych przez JavaScript. Miejsce przechowywania danych jest tak samo ważne jak to, co przechowujemy. W tym rozdziale wyjaśniono, jak łańcuch zasięgu i łańcuch prototypu mogą wpływać na całkowitą wydajność skryptu.

Stoyan Stefanov, który ma duże doświadczenie w zakresie wewnętrznego działania przeglądarek, napisał rozdział 3, *Skrypty w modelu DOM*. Stoyan tłumaczy, że interakcja modelu DOM jest wolniejsza niż dla innych części JavaScript ze względu na sposób jego implementacji. Opisuje wszystkie aspekty modelu DOM, między innymi, jak przemalowywanie strony i ponowne wlewanie informacji spowalnia wykonanie kodu.

Rozdział 4, *Algorytmy i sterowanie przepływem,* tłumaczy, jak popularne paradygmaty programistyczne, jak pętle i rekurencje, mogą stać na przeszkodzie wydajności wykonania. Omawiane są techniki optymalizacji, takie jak memoizacja, w kontekście ograniczeń wykonywania JavaScript.

Wiele aplikacji Web wykonuje w JavaScript złożone operacje na łańcuchach. Dlatego specjalista od łańcuchów, Steven Levithan, opisuje to zagadnienie w rozdziale 5, *Łańcuchy i wyrażenia regularne*. Programiści sieciowi od lat walczą ze słabą wydajnością obsługi łańcuchów w przeglądarkach, a Steven tłumaczy, dlaczego niektóre działania są wolne i jak obchodzić te problemy.

Rozdział 6, *Reagujące interfejsy*, mocno skupia się na odbiorze przez użytkownika. Uruchomiony Java-Script może spowodować zamrożenie przeglądarki, ogromnie frustrując użytkownika. Ten rozdział omawia kilka technik, które sprawiają, że interfejs użytkownika będzie zawsze reagował. W rozdziale 7, *Ajax*, Ross Harmes omawia najlepsze sposoby osiągania szybkiej komunikacji między klientem a serwerem w JavaScript. Ross pisze, jak różne formaty danych mogą wpłynąć na wydajność Ajax i dlaczego `XMLHttpRequest` nie zawsze stanowi najlepszy wybór.

Rozdział 8, *Praktyki programowania*, jest zbiorem najlepszych praktyk, które są wyjątkowe w programowaniu w JavaScript.

Wdrażanie

Po napisaniu i przetestowaniu kodu JavaScript jest czas na udostępnieniu zmian dla każdego. Nie należy jednak umieszczać surowych plików źródłowych w rozwiązaniach eksploatacyjnych. Julien Lecomte pokazuje w rozdziale 9, *Tworzenie i wdrażanie wysoko wydajnych aplikacji JavaScript*, jak podczas wdrożenia poprawić wydajność programu JavaScript. Julien omawia wykorzystanie systemów budulcowych do automatycznego zmniejszenia plików oraz kompresji HTTP w celu dostarczenia ich do przeglądarki.

Testowanie

Po wdrożeniu całego kodu JavaScript kolejnym krokiem jest testowanie wydajności. W rozdziale 10, *Narzędzia,* Matt Sweeney omawia metodologię i narzędzia testowania. Pokazuje, jak użyć JavaScript do pomiaru wydajności, a także opisuje popularne narzędzia zarówno do oceny wydajności uruchomieniowej JavaScript, jak i do wskazywania problemów wydajnościowych dzięki przechwytywaniu HTTP (ang. HTTP sniffing).

Dla kogo przeznaczona jest ta książka

Książka ta jest skierowana do programistów aplikacji Web, którzy są średnio lub bardziej zaawansowani w programowaniu w JavaScript i chcą poprawić wydajność interfejsu swoich aplikacji Web.

Konwencje stosowane w książce

W książce zastosowano następujące konwencje typograficzne:

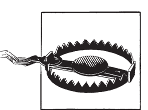
Kursywa
: Wskazuje na nowe terminy, adresy URL, adresy email, nazwy plików oraz rozszerzenia plików.

`Czcionka o stałej szerokości`
: Używana w listingach programów, a także w tekście, do wyróżnienia odwołań do elementów programu, jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe oraz słowa kluczowe.

`Pogrubiona` `czcionka o stałej szerokości`
: Pokazuje polecenia i inne teksty, które powinny być bez zmian wpisane przez użytkownika.

`Czcionka o stałej szerokości` *`pisana kursywą`*
: Pokazuje tekst, który powinien zostać zastąpiony wartościami podanymi przez użytkownika lub określonymi przez dany kontekst.

Ta ikona symbolizuje wskazówkę, sugestię lub uwagę ogólną.



Ta ikona symbolizuje niebezpieczeństwo lub zalecenie zachowania ostrożności.

Korzystanie z przykładowego kodu

Ta książka ma być pomocna w pracy. Kod z tej książki można wykorzystać w swoich programach i dokumentacji. Nie trzeba prosić nas o zgodę, o ile ktoś nie użyje znaczącej części kodu. Na przykład napisanie programu, który wykorzystuje klika wierszy kodu z książki, nie wymaga zgody. Natomiast sprzedaż i dystrybucja CD-ROM z przykładami z książek wydawnictwa O'Reilly wymaga zgody. Odpowiedź na pytanie cytatem z tej książki i przytoczenie przykładowego kodu nie wymaga zgody. Natomiast włączenie znaczącej ilości przykładowego kodu z książki do dokumentacji swojego produktu takiej zgody wymaga.

Doceniamy powoływanie się na nas, ale tego nie wymagamy. Odwołanie zwykle zawiera tytuł, autora, wydawcę oraz ISBN. Na przykład „*High Performance JavaScript*, by Nicholas C. Zakas. Copyright 2010 Yahoo!, Inc, 978-0-596-80279-0".

Każdy, kto zastanawia się, czy jego sposób wykorzystania przykładowego kodu wykracza poza zakres udzielonego pozwolenia, może skontaktować się z nami pod adresem *permissions@oreilly.com*.

Kontakt

Komentarze i pytania dotyczące książki prosimy kierować do wydawcy:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (tel. w USA I Kanadzie)
707-829-0515 (tel. międzynarodowe i lokalne)
707-829-0104 (fax)

Książka ta ma własną witrynę, gdzie umieszczono erratę, przykłady oraz dodatkowe informacje. Dostęp do strony to:

http://www.oreilly.com/catalog/9780596802790

Komentarze i pytania techniczne należy wysyłać na adres email:

bookquestions@oreilly.com

Więcej informacji o książce, konferencjach, ośrodkach zasobów (Resource Centers) oraz sieci wydawnictwa O'Reilly Network, znajduje się na naszej witrynie:

http://www.oreilly.com

Podziękowania

Po pierwsze, chciałbym podziękować wszystkim współautorom. Są to: Matt Sweeney, Stoyan Stefanov, Stephen Levithan, Ross Harmes oraz Julien Lecomte. Ich połączone doświadczenia i wiedza zawarta w części tej książki sprawiły, że rezultat jest bardziej fascynujący i zajmujący.

Dziękuję wszystkim na świecie guru od wydajności, których miałem okazję spotkać i wymienić z nimi opinie. Dotyczy to zwłaszcza Steve'a Soudersa, Tenni Theurer oraz Nicole Sullivan. Ta trójka pomogła mi rozszerzyć horyzonty w obszarze wydajności w sieci, za co jestem im ogromnie wdzięczny.

Duże podziękowania należą się wszystkim, którzy oceniali tę książkę przed publikacją, a są to między innymi Ryan Grove, Oliver Hunt, Matthew Russell, Ted Roden, Remy Sharp i Venkateswaran Udayasankar. Ich uwagi były bezcenne podczas przygotowywania książki do wydania.

Ogromne podziękowania należą się wszystkim z wydawnictwa O'Reilly i firmy Yahoo!, dzięki którym było możliwe wydanie tej książki. Chciałem napisać książkę dla Yahoo! już od chwili, gdy w 2006 r. zacząłem pracować w tej firmie, a dzięki Yahoo! Press udało się to tak świetnie zrealizować.

Rozdział 1

Pobieranie i wykonanie

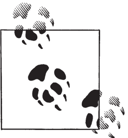
Wydajność działania JavaScript w przeglądarce jest niewątpliwie największym problemem związanym z korzystaniem z tego języka, na jaki napotykają twórcy oprogramowania. Problem jest złożony z powodu blokującej natury języka JavaScript, która sprawia, że podczas wykonywania kodu w JavaScript nie mogą być wykonywane żadne inne operacje. W większości przeglądarek do aktualizacji interfejsu użytkownika i wykonania skryptów JavaScript używany jest jeden proces, więc w danym momencie może mieć miejsce tylko jedno z tych zdarzeń. Im dłużej wykonywany jest JavaScript, tym dłużej trzeba czekać na zwolnienie przeglądarki i reakcję na działania użytkownika.

Na poziomie podstawowym oznacza to, iż sama obecność znacznika `<script>` wystarczy, aby strona czekała na analizę składni i wykonanie skryptu. Nie ma znaczenia, czy właściwy kod JavaScript jest wbudowany, czy też znajduje się w zewnętrznym pliku; pobieranie i renderowanie strony musi zostać zatrzymane i czekać na zakończenie skryptu. Jest to konieczny element cyklu życia strony, gdyż skrypt podczas wykonywania może powodować zmiany na stronie. Typowym tego przykładem jest użycie `document.write()` w środku strony (często wykorzystywane przez reklamy). Na przykład:

```
<html>
<head>
   <title>Script Example</title>
</head>
<body>
<p>
<script type="text/javascript">
   document.write("Data to " + (new Date()).toDateString());
</script> </p>
</body>
</html>
```

Gdy przeglądarka natrafi na znacznik <script>, jak na tej stronie HTML, nie ma możliwości stwierdzenia, czy JavaScript wstawi zawartość do <p>, wprowadzi dodatkowe elementy lub być może nawet zamknie znacznik. Dlatego przeglądarka wstrzymuje przetwarzanie nadchodzącej strony, wykonuje kod JavaScript, a następnie kontynuuje analizę składniową

i renderowanie strony. To samo ma miejsce w przypadku ładowania JavaScript za pomocą atrybutu `src`; przeglądarka musi najpierw pobrać kod z zewnętrznego pliku, co zajmuje czas, a następnie przeanalizować i wykonać kod. Wizualizacja strony i interakcja z użytkownikiem są w tym czasie całkowicie zablokowane.



Dwa podstawowe źródła informacji, dotyczących wpływu JavaScript na szybkość pobierania, to zespół Yahoo! Exceptional Performance (*http://developer.yahoo.com/performance/*) oraz Steve Souders, autor książek *High Performance Web Sites* (*http://oreilly.com/catalog/9780596529307/*) (O'Reilly) oraz *Even Faster Web Sites* (*http://oreilly.com/catalog/9780596522315/*) (O'Reilly). Ich wspólne badania miały znaczny wpływ na treść tego rozdziału.

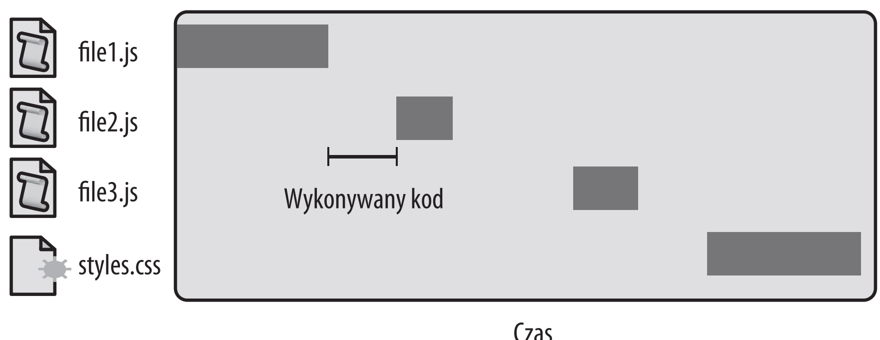
Pozycjonowanie skryptu

Specyfikacja HTML 4 wskazuje, iż znacznik `<script>` można umieścić w dokumencie HTML wewnątrz bloku `<head>` lub `<body>` i może on wystąpić wiele razy w każdym z nich. Tradycyjnie znaczniki `<script>`, używane do ładowania zewnętrznych plików JavaScript, pojawiały się w `<head>`, wraz ze znacznikami `<link>` do pobierania plików CSS oraz innymi metadanymi dotyczącymi strony. Teoria głosiła, że najlepiej będzie utrzymywać razem możliwie wiele stylów i zależności dotyczących zachowania, pobierając je na początku, aby strona wyglądała i zachowywała się prawidłowo. Na przykład:

```
<html>
<head>
   <title>Script Example</title>
   <-- Przykład nieefektywnego umieszczenia skryptu -->
   <script type="text/javascript" src="file1.js"></script>
   <script type="text/javascript" src="file2.js"></script>
   <script type="text/javascript" src="file3.js"></script>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
 <p>Hello world!</p>
</body>
</html>
```

Wprawdzie kod ten wygląda nieszkodliwie, jednak powoduje poważne problemy z wydajnością: w bloku `<head>` są ładowane trzy pliki JavaScript . Ponieważ każdy znacznik `<script>` blokuje dalszą wizualizację strony do momentu pełnego załadowania i wykonania kodu JavaScript, odbierana wydajność będzie dużo gorsza. Trzeba pamiętać, że przeglądarki nie zaczynają wizualizować czegokolwiek na stronie do czasu napotkania znacznika

otwierającego blok <body>. Takie umieszczenie skryptów na górze strony prowadzi zwykle do zauważalnego opóźnienia, zwykle widocznego jako biała pusta strona, zanim użytkownik będzie mógł cokolwiek przeczytać lub wejść w jakąś interakcję ze stroną. Zrozumienie sposobu takiego działania umożliwi schemat kaskadowy pokazujący moment ładowania każdego z zasobów. Rysunek 1-1 pokazuje, kiedy poszczególne skrypty oraz plik arkusza styli są pobierane podczas ładowania strony.



Rysunek 1-1. Wykonanie kodu JavaScript blokuje ściąganie innych plików

Rysunek 1-1 ukazuje interesujący wzorzec. Rozpoczyna się pobieranie pierwszego pliku JavaScript, co blokuje pobieranie w tym czasie jakichkolwiek innych plików. Ponadto występuje opóźnienie pomiędzy momentem całkowitego załadowania pliku *file1.js*, a momentem rozpoczęcia pobierania pliku *file2.js*. Jest to czas potrzebny na to, aby kod zawarty w pliku *file1.js* został w pełni wykonany. Każdy plik musi czekać do chwili, aż poprzedni plik zostanie pobrany i wykonany, zanim będzie mogło rozpocząć się kolejne pobieranie. W czasie pobierania kolejnych plików użytkownik ma przed sobą pusty ekran. Tak właśnie zachowuje się dziś większość głównych przeglądarek.

Internet Explorer 8, Firefox 3.5, Safari 4 oraz Chrome 2 pozwalają na równoległe pobieranie plików JavaScript. To dobra wiadomość, gdyż znaczniki <script> nie blokują innych znaczników <script>, umożliwiając pobieranie zasobów zewnętrznych. Niestety, pobieranie JavaScript nadal blokuje pobieranie innych zasobów, na przykład obrazów. Choć więc pobieranie skryptów nie blokuje pobierania innych skryptów, strona nadal musi czekać na pobranie i wykonanie kodu JavaScript, aby kontynuować działanie. Zatem, mimo że główne przeglądarki poprawiły wydajność dzięki dopuszczeniu równoległego pobierania, problem nie został rozwiązany do końca. Blokowanie przez skrypty nadal sprawia kłopoty.

Ponieważ skrypty blokują pobieranie wszystkich typów zasobów na stronie, zaleca się umieszczanie wszystkich znaczników <script> możliwie blisko końca bloku <body>, aby nie miały one wpływu na pobieranie całej strony. Na przykład:

```
<html>
<head>
```

```
    <title>Script Example</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <p>Hello world!</p>
    <-- Przykład zalecanego położenia skryptu  -->
    <script type="text/javascript" src="file1.js"></script>
    <script type="text/javascript" src="file2.js"></script>
    <script type="text/javascript" src="file3.js"></script>
</body>
</html>
```

Ten kod ukazuje zalecane położenie znaczników `<script>` w pliku HTML. Choć pobieranie skryptów nadal będzie się wzajemnie blokować, to reszta strony została już załadowana i wyświetlona użytkownikowi, więc cała strona nie jest odbierana jako powolna. Jest to pierwsza zasada zespołu Yahoo! Exceptional Performance, dotycząca JavaScript: umieszczaj wszystkie skrypty na dole!

Grupowanie skryptów

Dobrze jest ograniczyć całkowitą liczbę znaczników `<script>` zawartych na stronie, ponieważ podczas początkowego pobierania każdy znacznik `<script>` blokuje renderowanie strony. Dotyczy to zarówno skryptów wbudowanych, jak i tych zapisanych w plikach zewnętrznych. Ilekroć podczas analizy strony HTML zostanie napotkany znacznik `<script>`, występuje przerwa w renderowaniu na czas wykonania kodu; minimalizacja tych opóźnień poprawia całościową wydajność strony.

Steve Souders stwierdził także, że skrypt wewnętrzny umieszczony po znaczniku `<link>`, odwołujący się do zewnętrznego arkusza styli, powoduje blokowanie przeglądarki podczas oczekiwania na pobranie arkusza styli. Dzieje się tak, aby zagwarantować, że wbudowany skrypt będzie działać z najbardziej poprawną informacją o stylach. Z tego właśnie powodu Souders zaleca, aby nigdy nie umieszczać wbudowanych skryptów po znaczniku `<link>`.

Problem ten wygląda nieco inaczej, gdy mamy do czynienia z zewnętrznymi plikami JavaScript. Każde żądanie HTTP pociąga za sobą dodatkowe obciążenie dla wydajności, więc pobieranie jednego pliku o wielkości 100 kB będzie szybsze niż pobieranie czterech plików po 25 kB. Aby to osiągnąć, wystarczy ograniczyć liczbę zewnętrznych plików skryptów, do których odwołuje się strona.

Zazwyczaj duża witryna lub aplikacja internetowa będzie wymagała kilku plików JavaScript. Można zminimalizować ich wpływ na wydajność, łącząc je w jeden plik, aby następnie wywołać ten plik za pomocą jednego znacznika `<script>`. Połączenie plików może zostać wykonane w trybie offline, za pomocą narzędzia omawianego w rozdziale 9, lub w czasie rzeczywistym przy użyciu narzędzia takiego jak obsługa wieloplikowa w Yahoo!.

Yahoo! stworzył obsługę wieloplikową do dystrybucji plików biblioteki Yahoo! User Interface (YUI) za pośrednictwem swojej sieci Content Delivery Network (CDN). Każda witryna może pobrać dowolną liczbę plików YUI, używając odpowiedniego URL z nazwami łączonych plików. Na przykład ten URL obejmuje dwa pliki: *http://yui.yahooapis.com/combo ?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js*.

Ten URL powoduje załadowanie wersji 2.7.0 plików *yahoo-min.js* oraz *event-min.js*. Pliki te znajdują się na serwerze jako oddzielne elementy, ale są łączone w momencie żądania tego URL. Zamiast korzystania z dwóch znaczników `<script>` (po jednym do załadowania każdego z plików), można użyć jednego znacznika `<script>` do załadowania obydwu plików:

```
<html>
<head>
   <title>Script Example</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
   <p>Hello world!</p>
   <-- Przykład zalecanej pozycji skryptu -->
   <script type="text/javascript" src="http://yui.yahooapis.com/combo?2.7.0/build/
      yahoo/yahoo-min.js&2.7.0/build/event/event-min.js "></script>
</body>
</html>
```

Ten kod zawiera na dole strony pojedynczy znacznik `<script>`, który powoduje załadowanie wielu plików Java-Script. Ilustruje to najlepszą praktykę dołączania zewnętrznych plików JavaScript na stronie HTML.

Skrypty nieblokujące

Fakt blokowania przez JavaScript procesów w przeglądarkach, zarówno żądań HTTP, jak i aktualizacji UI, jest podstawowym problemem wydajności, z jakim mają do czynienia twórcy oprogramowania. Ograniczenie wielkości plików JavaScript oraz liczby żądań HTTP to tylko pierwsze etapy tworzenia sprawnie reagujących aplikacji web. Im więcej funkcji wymaga aplikacja, tym większej ilości kodu wymaga JavaScript, więc utrzymywanie małych plików źródłowych nie zawsze jest możliwe. Ograniczanie się do pobierania jednego dużego pliku JavaScript spowoduje zablokowanie przeglądarki na długi czas, mimo że mamy do czynienia z tylko jednym żądaniem http. Aby obejść tę sytuację, trzeba przyrostowo dodawać do strony coraz więcej kodu JavaScript tak, aby nie blokowało to przeglądarki.

Sekret skryptów nieblokujących polega na tym, że plik źródłowy pobieramy po zakończeniu pobierania strony. W kategoriach technicznych oznacza to pobieranie kodu źródłowego JavaScript po uruchomieniu zdarzenia `window`'s `load`. Jest kilka technik, które pozwalają uzyskać ten efekt.

Skrypty odroczone

Standard HTML 4 definiuje dodatkowy atrybut znacznika `<script>` o nazwie `defer`. Atrybut `defer` wskazuje, że skrypt zawarty w elemencie nie będzie modyfikował DOM, a więc wykonanie można bezpiecznie odłożyć na czas późniejszy. Atrybut `defer` jest obsługiwany tylko przez Internet Explorer 4+ oraz Firefox 3.5+ i dlatego nie jest najlepszy jako narzędzie do ogólnego zastosowania w różnych przeglądarkach. W innych przeglądarkach atrybut `defer` jest po prostu ignorowany, a więc znacznik `<script>` jest traktowany w sposób domyślny (blokujący). Rozwiązanie to jest jednak dobre, jeśli obsługują je przeglądarki ustalone jako docelowe. Poniżej pokazano przykładowe użycie:

```
<script type="text/javascript" src="file1.js" defer></script>
```

Znacznik `<script>` z atrybutem `defer` można umieścić w dowolnym miejscu dokumentu. Plik JavaScript zacznie być pobierany w punkcie, w którym znacznik `<script>` jest przetwarzany, ale kod nie będzie wykonywany aż do całkowitego załadowania DOM (do momentu wywołania obsługi zdarzenia `onload`). Pobieranie odroczonego pliku JavaScrip nie blokuje innych procesów przeglądarki, więc pliki te mogą być pobierane równolegle z innymi elementami strony.

Każdy element `<script>` oznaczony za pomocą `defer` nie zostanie wykonany do momentu całkowitego załadowania DOM; jest to prawda zarówno dla skryptów wewnętrznych,

jak i dla zewnętrznych plików skryptów. Pokazana poniżej prosta strona ilustruje, jak atrybut `defer` zmienia zachowanie skryptów:

```
<html>
<head>
   <title>Script Defer Example</title>
</head>
<body>
   <script defer>
      alert("odrocz");
   </script>
   <script>
      alert("skrypt");
   </script>
   <script>
      window.onload = function(){
         alert("pobierz");
      };
    </script>
</body>
</html>
```

Ten kod w trakcie przetwarzania strony wyświetla trzy powiadomienia . W przeglądarkach, które nie obsługują `defer`, kolejność komunikatów jest następująca: *„odrocz”*, *„skrypt”* oraz *„pobierz”*. W przeglądarkach obsługujących `defer` kolejność komunikatów jest taka: *„skrypt”*, *„odrocz”* i *„pobierz”*. Zauważmy, że odroczony element `<script>` nie jest wykonywany od razu, lecz przed wywołaniem obsługi zdarzeń `onload`.

Jeśli wśród docelowych przeglądarek jest jedynie Internet Explorer oraz Firefox 3.5, odraczanie skryptów w opisany sposób będzie przydatne. Jeśli obsługiwana ma być szeroka gama przeglądarek, trzeba wybrać inne rozwiązania, które działają w bardziej spójny sposób.

Zwykle bezpieczniej jest dodawać nowe węzły `<script>` do elementu `<head>` zamiast do `<body>`, zwłaszcza jeśli kod jest wykonywany podczas ładowania strony. W programie Internet Explorer może wystąpić błąd „operation aborted” (przerwano działanie), gdy cała zawartość `<body>` nie została jeszcze załadowana*.

* Artykuł „The dreaded operation aborted error” na stronie *http://www.nczonline.net/blog/2008/03/17/the-dreaded-operation-aborted-error/* zawiera pogłębione omówienie tego problemu.

Dynamiczne elementy skryptu

Model DOM (Document Object Model) pozwala na dynamiczne tworzenie niemal każdej części dokumentu HTML za pomocą JavaScript. U podstaw element `<script>` nie różni się niczym od pozostałych elementów na stronie: odwołania można uzyskiwać przez DOM, można je przenosić, usuwać z dokumentu, a nawet tworzyć. Nowy element `<script>` można bardzo łatwo utworzyć za pomocą standardowych metod DOM.

W sytuacji, gdy plik jest pobierany przy użyciu dynamicznego węzła skryptu, otrzymany kod jest zwykle od razu wykonywany (poza Firefox i Operą, które będą czekać, aż zostaną wykonane wszystkie wcześniejsze dynamiczne węzły skryptów). Działa to poprawnie, gdy skrypt sam się wykonuje, ale może stwarzać problemy, jeśli kod zawiera tylko interfejsy wykorzystywane przez inne skrypty na stronie. W takim przypadku trzeba sprawdzić, czy kod został całkowicie pobrany i czy jest gotowy do działania. Można to osiągnąć za pomocą zdarzeń wyzwalanych przez dynamiczny węzeł `<script>`.

Firefox, Opera, Chrome oraz Safari 3+ wyzwalają zdarzenie `load`, gdy natrafią na atrybut `src` elementu `<script>`. Można więc być powiadomionym o gotowości skryptu, nasłuchując następującego zdarzenia:

```
var script = document.createElement("script")
script.type = "text/javascript";

//Firefox, Opera, Chrome, Safari 3+
script.onload = function(){
    alert("Skrypt ściągnięty!");
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

Internet Explorer obsługuje alternatywną implementację, która wyzwala zdarzenie `readystatechange`. W elemencie `<script>` jest właściwość `readyState`, która podczas pobierania zewnętrznego pliku jest w różnych momentach zmieniana. Istnieje pięć możliwych wartości stanu `ready State`:

`"uninitialized"` Stan domyślny (niezainicjowane)

`"loading"` Pobieranie rozpoczęte (pobieranie)

`"loaded"` Pobieranie zakończone (pobrane)

`"interactive"` Dane zostały całkowicie pobrane, ale nie ma pełnej dostępności (interakcja)

`"complete"` Dane są gotowe do użycia (zakończone)

Dokumentacja firmy Microsoft dotycząca `readyState` oraz każda z możliwych wartości wskazują, że podczas życia elementu `<script>` nie wszystkie stany będą wykorzystywane,

ale nie ma wskazań mówiących, które z nich będą zawsze użyte. W praktyce dwa najbardziej interesujące stany to `"loaded"` oraz `"complete"`. Internet Explorer nie działa spójnie i nie ma jasności, która z tych dwóch wartości stanu `readyState` wskazuje na czas zakończenia. Czasami bowiem element `<script>` osiąga stan `"loaded"`, ale nigdy nie osiąga stanu `"complete"`, a innym razem stan `"complete"` zostaje osiągnięty bez użycia stanu `"loaded"`. Najbezpieczniejszym sposobem wykorzystania zdarzenia zmiany stanu `readystatechange` jest sprawdzenie obu stanów i usunięcie obsługi zdarzenia, gdy którykolwiek z nich zostanie użyty (aby mieć pewność, że zdarzenie nie będzie obsłużone dwukrotnie):

```
var script = document.createElement("script")
script.type = "text/javascript";

//Internet Explorer script.onreadystatechange = function(){
   if (script.readyState == "loaded" || script.readyState == "complete"){
      script.onreadystatechange = null;
alert("Script loaded.");
    }
   };
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

W większości przypadków przy dynamicznym pobieraniu plików JavaScript będziemy korzystać z jednej metody. Poniższa funkcja zawiera w sobie zarówno funkcje standardowe, jak i funkcje specyficzne dla IE:

```
function loadScript(url, callback){
var script = document.createElement("script")
script.type = "text/javascript";

if (script.readyState){
   //IE
   script.onreadystatechange = function(){
      if (script.readyState == "loaded" || script.readyState == "complete"){
         script.onreadystatechange = null;
         callback();
      }
   };
   } else {
   //Inne
      script.onload = function(){
         callback();
      };
   }

   script.src = url;
   document.getElementsByTagName("head")[0].appendChild(script);
}
```

Funkcja ta przyjmuje dwa argumenty: URL pobieranego pliku JavaScript oraz funkcję wywołania, która ma być wykonana po ukończeniu pobierania. Wykrywanie cech służy do stwierdzenia, który element obsługi zdarzenia powinien monitorować postępy skryptu. Ostatnim krokiem jest przypisanie właściwości `src` oraz dodanie do strony elementu `<script>`. Funkcja `loadScript()` jest używana w poniższy sposób:

```
loadScript("file1.js", function(){
    alert("File is loaded!");
});
```

Można dynamicznie pobierać tyle plików JavaScript, ile jest potrzebnych na stronie, ale trzeba upewnić się, że będą one pobierane w odpowiedniej kolejności. Wśród wszystkich podstawowych przeglądarek tylko Firefox i Opera gwarantują, że kolejność wykonywania skryptów pozostanie zgodna z tym, jak została określona. Inne przeglądarki będą pobierać i wykonywać różne pliki kodu w takiej kolejności, w jakiej będą one zwracane przez serwer. Kolejność można zagwarantować, łącząc pobrania w łańcuch, jak na przykład:

```
loadScript("file1.js", function(){
   loadScript("file2.js", function(){
      loadScript("file3.js", function(){
         alert("All files are loaded!");
      });
   });
});
```

Ten kod czeka z rozpoczęciem ładowania pliku *file2.js* do chwili, gdy będzie dostępny *file1.js*, a także czeka z pobieraniem pliku *file3.js* do momentu, gdy *file2.js* będzie dostępny. Choć podejście takie jest możliwe, może być trudne w zarządzaniu, jeśli do pobrania i wykonania jest wiele plików.

Jeśli ważna jest kolejność wielu plików, preferowanym podejściem jest połączenie ich w jeden plik, w którym każda część ustawiona jest we właściwej kolejności. Taki pojedynczy plik można pobrać, aby otrzymać wszystkie kody jednocześnie (ponieważ dzieje się to asynchronicznie, nie płacimy żadnej dodatkowej ceny za stosowanie dużego pliku).

Ze względu na kompatybilność z różnymi przeglądarkami i prostotę użycia, dynamiczne ładowanie plików jest najczęściej używanym szablonem nieblokującego ściągania plików JavaScript.

Wstrzykiwanie skryptu za pomocą *XMLHttpRequest*

Innym podejściem do nieblokujących skryptów jest pobieranie kodu JavaScript za pomocą obiektu XMLHttpRequest (XHR), a następnie wstrzykiwanie skryptu na stronę. Technika ta obejmuje tworzenie obiektu XHR, pobieranie pliku JavaScript, a następnie

wstrzyknięcie na stronę kodu JavaScript przy użyciu dynamicznego elementu `<script>`. Oto prosty przykład:

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function(){
   if (xhr.readyState == 4){
      if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){
         var script = document.createElement("script");
         script.type = "text/javascript";
         script.text = xhr.responseText;
         document.body.appendChild(script);
      }
   }
};
xhr.send(null);
```

Kod ten wysyła żądanie `GET` po plik *file1.js*. Obsługa zdarzenia `onreadystatechange` sprawdza, czy stan `readyState` jest równy 4, a następnie sprawdza, czy kod stanu HTTP jest prawidłowy (każda wartość pomiędzy 200 i 299 oznacza prawidłową odpowiedź, zaś 304 oznacza odpowiedź zbuforowaną). Jeśli otrzymamy prawidłową odpowiedź, tworzony jest nowy element `<script>`, zaś do otrzymanego z serwera `responseText` jest przypisywana jego właściwość `text`. Takie postępowanie powoduje utworzenie elementu `<script>` z wbudowanym kodem. Po dodaniu do dokumentu nowego elementu `<script>` kod jest wykonywany i gotowy do wykorzystania.

Podstawową zaletą tego podejścia jest możliwość pobrania kodu JavaScript bez natychmiastowego wykonywania. Kod zwracany jest na zewnątrz bloku `<script>`, a więc nie jest on automatycznie wykonywany po pobraniu, co pozwala na odłożenie wykonania do czasu osiągnięcia gotowości. Inną zaletą jest fakt, że ten sam kod może działać we wszystkich bez wyjątku współczesnych przeglądarkach.

Głównym ograniczeniem tej metody jest fakt, że plik JavaScript musi znajdować się w tej samej domenie, co żądająca go strona, co sprawia, że nie jest możliwe pobieranie z sieci CDN. Z tego powodu wstrzyknięcie skryptu XHR nie jest stosowane w wielkoskalowych aplikacjach sieciowych.

Zalecany szablon nieblokujący

Zalecane podejście do ładowania znaczącej ilości kodu JavaScript na stronę jest procesem dwustopniowym: najpierw dołączamy kod potrzebny do dynamicznego ładowania JavaScript, a następnie ładujemy resztę kodu JavaScript potrzebną do inicjalizacji skryptu. Pierwsza część kodu jest tak mała, jak to tylko możliwe, często zawiera tylko funkcję `load Script()`, dlatego jest szybko pobierana oraz wykonywana i nie powinna powodować

zakłóceń na stronie. Gdy początkowy kod jest już na swoim miejscu, wykorzystujemy go do załadowania reszty kodu JavaScript. Na przykład:

```
<script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
   loadScript("the-rest.js", function(){
      Application.init();
   });
</script>
```

Umieszczamy kod ładowania tuż przed znacznikiem zamykającym blok `</body>`. Takie postępowanie daje kilka korzyści. Po pierwsze, jak wspomnieliśmy wcześniej, zapewnia, że wykonanie JavaScript nie będzie blokować wyświetlania innych elementów strony. Po drugie, gdy zakończy się pobieranie drugiego pliku JavaScript, cały model DOM niezbędny dla aplikacji jest już utworzony i gotowy do współpracy. Dzięki temu nie trzeba sprawdzać innego zdarzenia (jak `window.onload`), aby stwierdzić, kiedy strona będzie gotowa do inicjalizacji.

Inną możliwością jest osadzenie funkcji `loadScript()` bezpośrednio na stronie i uniknięcie w ten sposób innego żądania HTTP. Na przykład:

```
<script type="text/javascript">
   function loadScript(url, callback){

      var script = document.createElement("script")
      script.type = "text/javascript";

      if (script.readyState){
   //IE
         script.onreadystatechange = function(){
            if (script.readyState == "loaded" ||
                  script.readyState == "complete"){
               script.onreadystatechange = null;
               callback();
            }
         };
      } else {
   //Inne
         script.onload = function(){
            callback();
       };
      }

      script.src = url;
      document.getElementsByTagName("head")[0].appendChild(script);
   }

   loadScript("the-rest.js", function(){
   Application.init();
```

```
    });
</script>
```

Jeśli zdecydujemy się na to ostatnie podejście, zalecana jest minimalizacja wstępnego skryptu za pomocą takiego narzędzia, jak kompresor YUI (patrz rozdział 9), aby wpływ liczby pobieranych bajtów na stronę był minimalny.

Po całkowitym pobraniu kodu inicjowania strony można swobodnie korzystać z `loadScript()`, aby w miarę potrzeb ładować na stronę kolejne funkcje.

Metoda YUI 3

Koncepcja niewielkiej początkowej ilości kodu na stronie, a następnie pobierania dalszych funkcji, stanowi jądro projektu YUI 3. Aby użyć YUI 3 na swojej stronie, trzeba zacząć od włączenia pliku ziarna YUI:

```
<script type="text/javascript"
src="http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js"></script>
```

Plik ziarna ma ok. 10 kB (6 kB po kompresji) i obejmuje dostatecznie szeroką funkcjonalność, aby pobrać dowolne inne komponenty YUI z Yahoo! CDN. Jeśli na przykład chcemy skorzystać z narzędzia DOM, podajemy jego nazwę (`"dom"`) za pomocą metody `use()` zdefiniowanej w YUI, a następnie zapewniamy wywołanie zwrotne, które będzie wykonane, gdy kod będzie gotowy:

```
YUI().use("dom", function(Y){
    Y.DOM.addClass(document.body, "loaded");
});
```

W tym przykładzie tworzona jest nowa instancja obiektu `YUI`, a następnie wywoływana jest metoda `use()`. Plik ziarna zawiera wszystkie informacje dotyczące nazw plików i zależności, więc podanie `"dom"` powoduje faktycznie utworzenie programu obsługi URL wieloplikowego ze wszystkimi poprawnymi plikami zależności i tworzy dynamiczny element skryptu do pobrania i wykonania tych plików. Gdy cały kod jest dostępny, wywoływana jest metoda wywołania zwrotnego, zaś instancja `YUI` jest przekazywana jako argument, co pozwala natychmiast rozpocząć korzystanie z nowo pobranej funkcjonalności.

Biblioteka LazyLoad

Ryan Grove z Yahoo! Search utworzył bibliotekę LazyLoad (dostępną pod adresem *http://github.com/rgrove/lazyload/*), aby dostarczyć narzędzie o szerszym zakresie zastosowań. LazyLoad jest wersją funkcji `loadScript()` o rozszerzonych możliwościach. Po zminimalizowaniu (nie po kompresji) plik LazyLoad ma około 1,5 kB.

Przykładowe zastosowanie:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
   LazyLoad.js("the-rest.js", function(){
      Application.init();
   });
</script>
```

LazyLoad daje także możliwość pobierania wielu plików JavaScript oraz dopilnowania, aby zostały wykonane we właściwej kolejności we wszystkich przeglądarkach. Aby załadować wiele plików JavaScript, trzeba przekazać tablicę adresów URL do metody `LazyLoad.js()`:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
   LazyLoad.js(["first-file.js", "the-rest.js"], function(){
      Application.init();
   });
</script>
```

Chociaż pliki są pobierane w sposób nieblokujący za pomocą dynamicznego ładowania skryptów, zaleca się stosowanie możliwie najmniejszej liczby plików. Każde pobieranie jest nadal oddzielnym żądaniem HTTP, zaś funkcja wywołania zwrotnego nie będzie wykonywana do momentu pobrania i wykonania wszystkich plików.

LazyLoad może także dynamicznie załadować pliki CSS. Jest to zwykle mniejszym problemem, gdyż pobieranie plików CSS jest zawsze wykonywane równolegle i nie blokuje innych działań na stronie.

Biblioteka LABjs

Innym podejściem do nieblokującego ładowania JavaScript jest LABjs (*http://labjs.com/*), biblioteka typu open source, napisana przez Kyle'a Simpsona z wejściem przygotowanym przez Steve'a Soudersa. Biblioteka ta zapewnia dokładniejszą kontrolę nad procesem ładowania i próbuje pobierać możliwie dużo kodu w sposób równoległy. LABjs jest także dość mała, ma 4,5 kB (zminimalizowana, nie skompresowana) i zajmuje minimalną ilość miejsca na stronie. Oto przykład zastosowania:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
   $LAB.script("the-rest.js")
      .wait(function(){
         Application.init();
```

```
        });
</script>
```

Metoda `$LAB.script()` jest używana do definiowania pliku JavaScript, który ma zostać pobrany, podczas gdy `$LAB.wait()` służy do wskazania, że wykonanie powinno zaczekać z uruchomieniem danej funkcji do czasu pobrania i wykonania pliku. LABjs zachęca do tworzenia łańcuchów, więc każda metoda zwraca odwołanie do obiektu `$LAB`. Aby pobrać wiele plików JavaScript, trzeba dołączyć inne wywołanie `$LAB.script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js") .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Tym, co odróżnia LABjs, jest możliwość zarządzania zależnościami. Zwykłe włączenie za pomocą znaczników `<script>` oznacza, że każdy plik jest pobierany (sekwencyjnie lub równolegle, jak wcześniej podano), a następnie jest wykonywany sekwencyjnie. W niektórych przypadkach jest to naprawdę konieczne, ale w innych nie.

LABjs pozwala określić przy użyciu `wait()`, które pliki powinny czekać na pozostałe. W poprzednim przykładzie nie ma gwarancji, że kod w pliku *first-file.js* będzie wykonany przed kodem z *the-rest.js*. Aby to zagwarantować, trzeba dodać wywołanie `wait()` po pierwszym `script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js").wait()
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Teraz kod *first-file.js* będzie na pewno wykonany przed kodem z *the-rest.js*, choć zawartość plików jest pobierana równolegle.