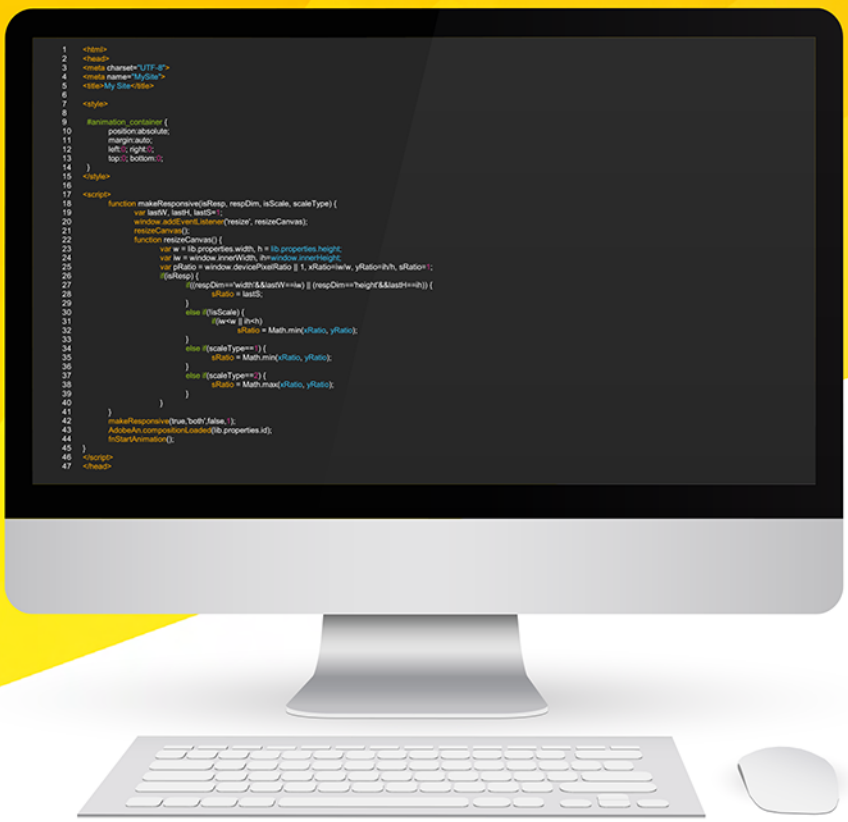
Tomasz Sochacki

JavaScript

Interaktywne aplikacje webowe



Helion

Redaktor prowadzący: Grzegorz Krzystek
Projekt okładki: Studio Gravite

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jasdom
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-5638-2

Printed in Poland.

Spis treści

Rozdział 2.

Podstawy HTML i DOM

W tym rozdziale zajmiemy się omówieniem podstawowych pojęć, takich jak **DOM** i **HTML**. W świecie programistów JavaScript bardzo często pojawia się określenie DOM, np. *pobierz referencję do elementu DOM*, *zmodyfikuj DOM* itp. Zastanowimy się, co to właściwie oznacza, bo w praktyce nie zawsze jest to oczywiste, szczególnie dla osób, które dopiero zaczynają naukę programowania webowego.

Omówimy również podstawowe zagadnienia związane ze strukturą strony internetowej, wyjaśnimy, kiedy i w jaki sposób dodawać do naszej strony skrypty JavaScript, a na koniec omówimy najważniejsze tematy dotyczące tzw. narzędzi deweloperskich w przeglądarce.

Czym są HTML i DOM?

Pracując przy aplikacjach internetowych, musimy dokładnie zrozumieć, jaka jest różnica między HTML a DOM. Otóż HTML to tzw. hipertekstowy język znaczników (*HyperText Markup Language*), który pozwala opisać strukturę strony internetowej. Nie jest to jednak język programowania, ponieważ nie ma np. zmiennych, pętli, instrukcji warunkowych itp. Nie jest też w żaden sposób kompilowany.

HTML to zestaw znaczników, którymi opisujemy, jaka ma być struktura strony internetowej. Na przykład chcąc wyświetlić jakiś tekst, możemy zastosować tzw. znacznik HTML określający paragraf tekstowy, co zapisujemy jako:

```
<p>Jakiś tekst</p>
```

Znaczniki takie najczęściej zapisuje się w plikach z rozszerzeniem *.html*. Nie określają one jednak, jak przeglądarka internetowa ma wyświetlić dany element i jakiego rodzaju interakcje mają być dla niego dostępne. Widząc w strukturze HTML zapis przedstawiony powyżej, przeglądarka rozumie, że chcemy, aby w tym miejscu na stronie internetowej pojawił się jakiś napis. Każda przeglądarka może inaczej zaimplementować sposób wyświetlania znaczników `<p>`, lecz najwięksi gracze na rynku przeglądarek starają się trzymać jednolitych standardów. Pozwala to mieć większą pewność, że nasza struktura opisana w plikach *.html* zostanie tak samo wyświetlona w różnych środowiskach.

Znaczniki HTML nie określają bezpośrednio, jak ma wyglądać dany element, np. czy ma być pogrubiony, wyświetlony czcionką o wielkości 16px, czy w inny sposób. Takie rzeczy definiowane są w tzw. **stylach CSS** (**kaskadowe arkusze stylów**). Style CSS mogą być zapisane w osobnych plikach z rozszerzeniem *.css* lub definiowane bezpośrednio w znacznikach HTML (choć jest to technika niezalecana i zaliczana do złych praktyk). W tej książce nie będziemy jednak omawiać zagadnień związanych ze stylowaniem elementów (poza kilkoma przykładami w rozdziale, gdzie będziemy dynamicznie z poziomu JavaScriptu modyfikować style elementu). Przeglądarki przypisują znacznikom tzw. style domyślne, np. paragrafy tekstowe wykorzystujące znacznik `<p>` mają najczęściej domyślny margines górny i dolny o wartości 16px.

Język HTML nie opisuje również w sposób bezpośredni, jakiego rodzaju interakcje chcemy udostępnić i jak je obsługiwać dla danego znacznika. Na przykład jeśli chcemy umieścić na stronie internetowej przycisk, to najczęściej korzystamy ze znacznika:

```
<button>Kliknij mnie</button>
```

Widząc taki zapis, przeglądarka wyrenderuje przycisk, który wizualnie najczęściej domyślnie będzie szarym elementem, zmieniającym nieco swój wygląd po kliknięciu. Aby jednak wykonać jakąś konkretną akcję w momencie kliknięcia przycisku, potrzebujemy języka JavaScript, który pozwoli nam takie interakcje obsługiwać.

Dochodzimy tu do ważnego punktu: jak wspomnieliśmy, HTML to tylko język znaczników, którymi opisujemy strukturę naszej strony. Czym jest zatem wspomniany na początku tego rozdziału **DOM**? Jest to tzw. obiektowy model dokumentu (*Document Object Model*). Powstaje on na bazie znaczników, które dostarczamy przeglądarce w formie tekstowej (np. plik *.html*). Czytając taki plik, przeglądarka tworzy model dokumentu, który jest w niej przedstawiany użytkownikowi.

Gdy przeglądarka trafi na znacznik

```
<p>Jakiś tekst</p>
```

stworzy nowy obiekt, mający szereg właściwości, np. pole o nazwie `textContent`, w którym przechowywana jest zawartość tzw. węzła tekstowego elementu, czyli nasz ciąg znakowy *Jakiś tekst*. Pozwala nam to np. z poziomu JavaScriptu odnieść się do pola `textContent` i w ten sposób pobrać aktualną wartość ciągu znakowego znajdującego się w tym obiekcie.

Obiekty tworzone przez przeglądarkę nazywane są często **węzłami**. Reprezentują one poszczególne elementy na stronie. DOM jest więc łącznikiem pomiędzy strukturą HTML a językami programowania (np. JavaScriptem), który umożliwia wprowadzenie interakcji i odnoszenie się do wyrenderowanych elementów.

Wcześniej wspomnieliśmy, że z poziomu JavaScriptu możemy odnieść się do aktualnej wartości różnych pól w obiekcie reprezentującym elementy na stronie. Pierwsze wyrenderowanie drzewa DOM, czyli zestawu wzajemnie zagnieżdżonych obiektów, odbywa się przy pierwszej analizie znaczników HTML przez przeglądarkę. DOM jest jednak bytem zmiennym i przy

użyciu JavaScriptu mamy możliwość jego dynamicznej aktualizacji i modyfikacji. Jednocześnie nie zmieniamy pierwotnej struktury opisanej w znacznikach HTML. To jest właśnie często wspominana **interaktywność stron internetowych**. Interaktywnością może być nie tylko obsługa kliknięcia przycisku czy wyświetlenie animacji, ale również dynamiczna zmiana jakiejś części drzewa DOM, np. wyświetlenie dodatkowej informacji czy reklamy po upływie określonego czasu, jaki użytkownik spędzi na stronie.

Do tematu drzewa DOM jeszcze wrócimy przy okazji pobierania tzw. referencji DOM, czyli podczas odwoływania się do właściwości obiektów reprezentujących poszczególne elementy na stronie internetowej. Pamiętaj zatem, że język HTML jest tylko językiem znaczników, które pozwalają określić początkową strukturę strony internetowej. Na podstawie tych znaczników przeglądarka tworzy kolejne obiekty składające się na całą strukturę wyrenderowanej strony, czyli tzw. drzewo obiektów DOM. Z poziomu języka JavaScript odnosimy się już nie do struktury w pliku *.html*, lecz do wyrenderowanych obiektów DOM.

Podstawowa struktura strony internetowej

Skoro już wiemy, że język znaczników HTML służy do opisywania podstawowej struktury strony, musimy się zastanowić, jaka ona powinna być. Każda aplikacja internetowa powinna składać się z dwóch podstawowych elementów: części `<head>` oraz `<body>`. Obie są równie ważne i definiują różne informacje o stronie. Elementy te są zawarte w nadrzędnym znaczniku `<HTML>`, określającym zawartość całego dokumentu. Najprostsza przykładowa struktura strony internetowej wygląda następująco:

```
<!DOCTYPE HTML>
<HTML lang="pl">
  <head>
    Metadane strony, np. znaczniki <meta>, skrypty CSS itp.
  </head>
  <body>
    Treści prezentowane na stronie.
  </body>
</HTML>
```

W sekcji `HEAD` umieszczamy informacje niezwiązane bezpośrednio z treściami, jakie widzi użytkownik, lecz tzw. metadane. Można do nich zaliczyć na przykład wczytanie stylów CSS z zewnętrznych plików, ustawienie odpowiednich znaczników `META`, informacje o tytule strony widocznym na pasku zakładek przeglądarki itp. Możliwe jest również wczytywanie w tym miejscu skryptów JavaScript, jednak ten temat omówimy dokładniej w jednym z kolejnych rozdziałów.

Najczęściej ustawiane są przynajmniej podstawowe znaczniki `META`, np.:

```
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Tytuł strony</title>
```

Ważne jest ustawienie odpowiedniego kodowania znaków. Obecnie najczęściej jest to kodowanie UTF-8, co pozwala dobrze interpretować m.in. polskie znaki diakrytyczne i wiele innych znaków z ogólnej tablicy Unicode. Drugi znacznik jest przydatny do prawidłowego wyświetlania strony na urządzeniach mobilnych o małych szerokościach ekranu. Jest to książka głównie o JavaScripcie, dlatego nie będziemy szczegółowo omawiać wszystkich używanych znaczników `META`, lecz warto wspomnieć jeszcze o kilku:

```
<meta name="Description" content="Krótki opis strony" />
<meta name="Keywords" content="wyraz pierwszy, wyraz drugi" />
<meta name="author" content="Autor strony">
```

Dawniej te znaczniki były wykorzystywane przez wyszukiwarkę Google, m.in. do pozycjonowania stron w wynikach wyszukiwania. Obecnie jednak w kontekście tzw. SEO nie mają one raczej większego znaczenia, choć czasami mogą być przydatne np. do pokazania użytkownikowi krótkiego opisu strony w wynikach wyszukiwania. Toczy się wiele dyskusji nad sensem stosowania takich znaczników, prawda jest jednak taka, że nigdy nie będziemy do końca znać wewnętrznych mechanizmów, jakimi kieruje się silnik wyszukiwarki Google, dlatego nie zaszkodzi dodać tych kilku znaczników zawierających podstawowe informacje o stronie.

Warto także dodać znacznik pozwalający określić ikonę, jaka ma być widoczna z prawej strony zakładki w przeglądarce:

```
<link rel="Shortcut icon" href="adres ikony" />
```

Jest to przydatne dla użytkowników, szczególnie gdy ktoś otwiera wiele różnych zakładek. Przy ich większej liczbie opisy czasem znikają i pozostają właśnie same ikony.

W sekcji `HEAD` często umieszcza się również linki do zewnętrznych arkuszy stylów CSS:

```
<link rel="stylesheet" href="ścieżka_do_pliku_ze_stylami.css">
```

Drugim obowiązkowym elementem strony internetowej jest znacznik `BODY`, w którym zawarte są znaczniki HTML opisujące strukturę strony. W tej sekcji najczęściej znajdują się również informacje o skryptach JavaScript, które należy wczytać, by zapewnić poprawne działanie aplikacji.

Zaczynamy pracę ze znacznikami HTML

Znaczniki HTML służą do określania struktury strony internetowej, która następnie zostanie odpowiednio przetworzona przez przeglądarkę. W tej książce najczęściej będziemy stosować podstawowe znaczniki, jak paragrafy, elementy blokowe i liniowe, obrazki, listy czy elementy formularzy.

Przykładowa struktura strony może wyglądać następująco:

```
<body>
  <header>
```

```
    <h1>Nagłówek strony</h1>
    <p>Jakiś fragment tekstu</p>
  </header>
  <section>
    <p>Jakaś pierwsza sekcja</p>
    <img src="..." />
    <!-- Jakiś komentarz w kodzie -->
  </section>
</body>
```

Oczywiście jest to tylko przykład. Na każdej stronie struktura będzie inna, w zależności od potrzeb. Znaczniki mogą posiadać także tzw. atrybuty. Powyżej takim atrybutem jest `src` dla znacznika reprezentującego obrazek `<img>`.

Najczęściej spotkamy się z następującymi atrybutami:

```
<div id="identyfikator-elementu">
<div class="nazwa-klasy-css druga-klasa-css">
<input name="nazwa-elementu" />
<img src="ścieżka-do-pliku-graficznego.jpg" />
<a href="adres-docelowy-dla-linka">
<div data-name="wartość-atrybutu">
```

Atrybut `id` najczęściej nadawany jest elementom, do których będziemy chcieli się dostać z poziomu JavaScriptu, choć nie jest to obowiązkowe — JavaScript pozwala na bardzo elastyczne przeszukiwanie drzewa DOM. Identyfikator powinien być unikalny dla całej strony. Przeglądarka nie zgłosi błędu, jeśli do dwóch elementów przypiszemy tę samą wartość parametru `id`, ale jest to uznawane za złą praktykę i może utrudniać pracę z takimi elementami z poziomu JavaScriptu.

Chyba najczęściej stosowanym atrybutem jest `class`, pozwalający przypisywać do elementu określone klasy CSS, które zazwyczaj znajdują się w oddzielnych plikach. Gdy chcemy przypisać do elementu kilka klas, oddzielamy je spacją (nie przecinkiem!). Istnieje również możliwość definiowania stylów bezpośrednio w elemencie (tzw. *inline styles*), jednak nie jest to zalecany sposób:

```
<div style="margin: 0; color: red">
```

W takim wypadku konkretne style rozdzielamy średnikami i stosujemy atrybut `style`. Style takie są jednak trudne do nadpisania ze względu na tzw. siłę różnych sposobów stylowania. Nie będziemy w tej książce omawiać stylowania elementów, zachęcam Cię jednak do zgłębienia zagadnień związanych z wadami stylowania *inline*.

Atrybut `name` najczęściej jest stosowany w elementach formularzy internetowych, choć może być użyty praktycznie z każdym znacznikiem HTML. W przeciwieństwie do atrybutu `id` atrybut `name` nie musi być unikalny — ta sama wartość może być przypisana do różnych elementów. Warto jednak zawsze rozważyć zasadność użycia wielokrotnie tej samej wartości tego typu atrybutów.

Istnieją także atrybuty specyficzne dla konkretnych elementów, na przykład każdy element `<img>` powinien posiadać atrybut `src`, w którym wskażemy źródło dla obrazka, jaki ma zostać wczytany w tym miejscu. Podobnie w przypadku znacznika określającego link `<a>` obowiązkowym atrybutem jest `href`, który wskazuje na docelowy adres, gdzie chcemy przenieść użytkownika po kliknięciu.

Ciekawym atrybutem jest `data-name`. W tym przypadku `name` może być dowolną nazwą, można więc stosować różne warianty tego atrybutu, np.:

```
data-address="street"
data-address="city"
data-buy-button
```

Znacznik ten nie musi posiadać wartości (jak w `data-buy-button`). Jest on przydatny, gdy np. chcemy wyróżnić wiele elementów przy użyciu tego samego identyfikatora, tylko zamiast atrybutu `id` stosujemy właśnie `data-`. Z poziomu JavaScriptu również mamy bardzo prosty dostęp do takich elementów.

W kodzie HTML można też umieszczać komentarze, które oznacza się w następujący sposób:

```
<!-- Jakiś komentarz w kodzie -->
```

Warto jednak robić to rozsądnie, gdyż komentarze te będą widoczne w kodzie strony wysłanym do przeglądarki użytkownika. Nie zobaczy on ich co prawda bezpośrednio na stronie, lecz może mieć do nich wgląd, gdy użyje narzędzi deweloperskich. Nie umieszczajmy więc w komentarzach informacji, które powinny zostać niejawne.

Wczytywanie skryptów JavaScript

Zanim zaczniemy pracować z elementami DOM, musimy dokładnie przeanalizować sposoby, w jakie można wczytywać nasze skrypty JavaScript. Mogą one być osadzane bezpośrednio na stronie w pliku HTML wewnątrz znaczników `<script>` lub dołączane z zewnętrznych plików.

Pierwsze rozwiązanie wygląda następująco:

```
<body>
  <p>Jakiś znacznik tekstowy</p>

  <script>
    console.log('skrypt JavaScript');
  </script>

  <p>Inny znacznik tekstowy</p>
</body>
```

W powyższym przykładzie wykorzystaliśmy znacznik `<script>`, aby w jego wnętrzu podać instrukcje JavaScript. Sposób ten nie jest jednak dobrą praktyką. Znacznie lepszym rozwiązaniem jest wyraźne rozdzielenie aplikacji na trzy niezależne elementy, czyli pliki odpowiedzialne za

strukturę strony (*.html*), pliki zawierające style CSS (*.css*) oraz skrypty JavaScript, odpowiedzialne za różne interakcje (*.js*). Taki projekt jest łatwiejszy w utrzymaniu i podczas wprowadzania nowych funkcjonalności.

Skrypty powinniśmy wczytywać z zewnętrznych plików, podając wtedy atrybut `src`, który określa ścieżkę do danego skryptu:

```
<body>
  <p>Jakiś znacznik tekstowy</p>
  <p>Inny znacznik tekstowy</p>
  <script src='script.js'></script>
</body>
```

Zwróć także uwagę na miejsce, w którym wczytujemy skrypt. Tym razem odbywa się to na samym końcu znacznika `<body>`. Czasami skrypty JavaScript definiowane są również w sekcji `<head>`, jednak rozwiązanie to nie zawsze jest efektywne.

Na początek przeanalizujmy sposób, w jaki przeglądarka analizuje zawartość pliku HTML. Gdy w przeglądarce wpisujemy adres strony internetowej, to wysyłamy do odpowiedniego serwera tzw. żądanie (*request*) i zwrotnie otrzymujemy najczęściej dane w formacie tekstowym. Jest to właśnie nasz plik HTML, w którym zawarte są podstawowe informacje o stronie (sekcja `<meta>`) oraz informacje o jej strukturze (sekcja `<body>`).

Przeglądarka analizuje te dane i trafiając na różne znaczniki, podejmuje odpowiednie akcje. Jeśli np. trafi na znacznik `<meta>` określający kodowanie znaków, ustawia je na wskazaną wartość, najczęściej UTF-8. Później np. ustawia tytuł strony, wczytuje style CSS itp.

Gdy przeglądarka trafi na znacznik `<script>`, zawierający w sobie kod JavaScript, to od razu przechodzi do jego wykonywania, a tym samym wstrzymuje dalszą analizę kodu HTML. W przypadku znaczników `<script>` ze wskazaną wartością atrybutu `src` następuje najpierw ściągnięcie tego skryptu z naszego serwera (lub innego miejsca wskazanego w atrybucie) i po tym wykonywanie kodu. Przeglądarki mają mechanizmy zapamiętywania ściągniętych już wcześniej danych (*cache*). W tym przypadku może to przyspieszyć całą operację, ale kroki pozostają te same (skrypt może zostać pobrany z *cache* zamiast z serwera).

Gdy przeglądarka trafia na kolejne znaczniki HTML, tworzy tzw. DOM, czyli model obiektowy całego dokumentu, gdzie poszczególne elementy strony są reprezentowane przez odpowiednie obiekty. Bardzo ważne jest, aby pilnować dokładnie miejsca, w którym tworzymy znaczniki `<script>`. Jeśli umieścimy skrypty w sekcji `<head>`, może się okazać, że elementy, do których próbujemy się odnieść z poziomu JavaScriptu, nie zostały jeszcze wyrenderowane, a tym samym są niedostępne, co skutkuje błędami wykonywania kodu JavaScript.

Istnieje kilka metod radzenia sobie z tym problemem. Najczęściej polecanym sposobem jest wczytywanie wszystkich skryptów JavaScript dopiero na końcu, tuż przed zamknięciem znacznika `</body>`. Daje nam to pewność, że całe drzewo DOM jest już wygenerowane, a tym samym mamy dostęp do wszystkich oczekiwanych przez nas elementów strony.

Niektóre poradniki podają też rozwiązanie bazujące na tzw. evencie *DomContentLoaded*, uruchamianym, gdy zostaną wygenerowane wszystkie elementy drzewa DOM. Metoda ta była szczególnie popularna wśród użytkowników biblioteki jQuery. W większości przypadków nie ma jednak sensu korzystać z tego eventu i w zupełności wystarczy po prostu przeniesienie skryptów na koniec struktury dokumentu.

Istnieje jeszcze inny sposób, wykorzystujący dodatkowe atrybuty, jakie może przyjmować znacznik `<script>` — `defer` oraz `async`. Oba są wspierane praktycznie przez wszystkie popularne przeglądarki, łącznie nawet z Internet Explorer 11. Atrybuty te pozwalają pobierać skrypty JavaScript równolegle z analizą kodu HTML (tzw. parsowaniem kodu). Różnica występuje przy wykonywaniu pobranego kodu.

W przypadku atrybutu `async` kod JavaScript jest wykonywany od razu po ściągnięciu pliku, natomiast atrybut `defer` zapewnia, że skrypty zostaną pobrane równolegle z parsowaniem HTML, lecz zostaną wykonane dopiero po zakończeniu analizy kodu HTML.

Bezpieczniejszy jest jednak atrybut `defer`, szczególnie gdy mamy do wczytania wiele skryptów JavaScript. W przypadku atrybutu `async` skrypty będą pobierane równolegle, ale nie mamy pewności, w jakiej kolejności zostaną wykonane. Jeśli zatem kolejność wykonywania jest istotna, np. ze względu na wzajemne zależności, to unikaj stosowania `async`. Atrybut `defer` również pozwala wczytywać skrypty równolegle z parsowaniem HTML, jednak w tym przypadku zostaną one wykonane w takiej kolejności, w jakiej zostały zadeklarowane w strukturze HTML.

Reasumując, szczególnie na początku nauki polecam dwa rozwiązania: stosowanie atrybutu `defer` lub ewentualnie deklarowanie znaczników `<script>` na końcu struktury dokumentu (przed zamknięciem `</body>`). W pierwszym przypadku skrypty mogą zostać zadeklarowane w sekcji `<head>`, gdyż i tak nie spowodują zablokowania parsowania HTML. Zalecane metody wczytywania skryptów wyglądają więc następująco:

Atrybut `defer`:

```
<head>
  <script src='script.js' defer></script>
</head>
```

Wczytywanie na końcu dokumentu:

```
<body>
  ...
  <script src='script.js'></script>
</body>
```

Rozwiązanie pierwsze jest lepsze, ponieważ skrypty będą wczytywane równolegle, ale mimo to nadal mamy pewność, że zostaną wykonane w odpowiedniej kolejności (w przeciwieństwie do atrybutu `async`). Należy tylko pamiętać, aby w tym wypadku dla wszystkich skryptów dodawać atrybut `defer`, szczególnie jeśli są to skrypty, które do działania potrzebują wyrenderowanego drzewa DOM.

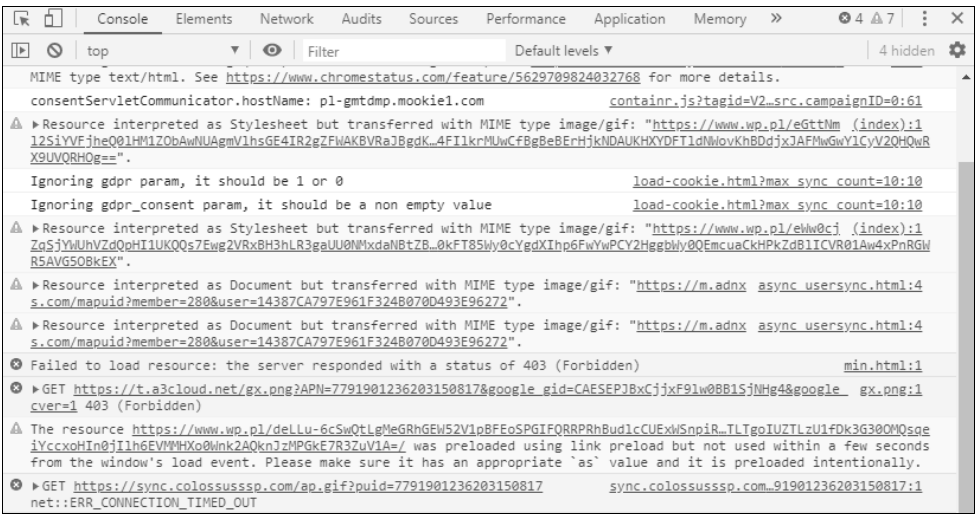
Narzędzia deweloperskie w przeglądarce internetowej

Książka ta przeznaczona jest dla osób, które mają opanowane podstawowe zagadnienia dotyczące języka JavaScript, więc prawdopodobnie konsola przeglądarki nie jest Ci obca. W tym rozdziale omówimy jednak wiele ciekawych jej elementów, przydatnych w codziennej pracy.

W tej książce będziemy omawiać narzędzia deweloperskie dostępne w przeglądarce Chrome. W innych przeglądarkach wszystko wygląda bardzo podobnie. Aby uruchomić konsolę, w systemie Windows należy użyć klawisza *F12*.

Najczęściej chyba będziesz korzystać z zakładki *Console*. Jest w niej widocznych wiele informacji, podzielonych na trzy grupy: *Info*, *Warnings*, *Errors*. Informacje neutralne oraz kod, który uruchamiamy w konsoli, są wyświetlane jako poziom *Info*. Informacje o poziomie *Warnings*, prezentowane na żółtym tle, to komunikaty, które co prawda nie powodują błędów w aplikacji, jednak warto je również analizować. Czasami znajdują się tam np. komunikaty o stosowaniu różnych funkcji, które nie są już zalecane. W nowych przeglądarkach można spotkać komunikat informujący o wykorzystaniu metody `document.write`, której powinno się unikać.

Obowiązkowo musimy natomiast analizować wszystkie komunikaty (w języku potocznym mówimy o tzw. logach) na poziomie *Errors*. Są to informacje wskazujące na błędy w naszej aplikacji. Rysunek 2.1 przedstawia przykładowy wygląd konsoli po wczytaniu strony internetowej. Widzimy trzy ważne dla nas komunikaty na poziomie *Errors*. Dwa pierwsze mówią o braku dostępu do pobrania wskazanych zasobów z serwera, a trzeci informuje nas, że dla próby pobrania danych ze wskazanego endpointu GET otrzymaliśmy odpowiedź negatywną ze względu na zbyt długi czas oczekiwania na odpowiedź serwera (*timeout*).



RYSUNEK 2.1. *Zakładka Console*

Podczas prac programistycznych warto mieć konsolę przypiętą do dolnej lub bocznej części ekranu lub otwartą jako osobne okno, aby na bieżąco śledzić wszystkie ważne informacje, jakie się w niej pojawiają.

Konsola umożliwia również filtrowanie informacji według wskazanego ciągu znakowego (pole *Filter*) oraz ustawianie, jakie poziomy informacji chcemy logować (*Default levels*). Z tym ustawieniem trzeba jednak uważać, szczególnie jeśli z jakichś względów chcemy tymczasowo wyłączyć logowanie na poziomie *Errors*. Jeśli modyfikujemy te ustawienia, warto wyrobić sobie nawyk, aby zawsze sprawdzać, jakie ustawienia są aktualnie włączone.

Z lewej strony widnieje ikona przekreślonego koła, która pozwala w dowolnym momencie wyczyścić wszystkie informacje wyświetlone w konsoli.

Z prawej strony znajduje się ikona koła zębatego. Po jej kliknięciu pojawi się kilka opcji konfiguracyjnych dla sekcji *Console*. Czasami przydatne jest włączenie opcji *Preserve log*. Domyślnie wszystkie informacje wyświetlane w konsoli są czyszczone przy każdym przeładowaniu strony internetowej. Aktywowanie opcji *Preserve log* pozwala zachować informacje, nawet gdy strona zostanie przeładowana. Jest to czasami przydatne podczas analizy zachowania aplikacji, szczególnie w przypadku zakładki *Network*, którą omówimy za chwilę.

Konsola daje również zaawansowane możliwości debugowania kodu przy wykorzystaniu narzędzi dostępnych w zakładce *Sources*, gdzie możemy ręcznie określać miejsca, w których wykonywanie kodu JavaScript ma zostać wstrzymane. Jest to jednak zagadnienie dość skomplikowane i nie będziemy go omawiać w tym podręczniku, gdzie skupiamy się na zagadnieniach podstawowych.

Druga zakładka, do której również często będziemy zaglądać jako programiści aplikacji internetowych, to *Elements*. Gdy ją otworzymy, zobaczymy dwie główne sekcje: strukturę dokumentu oraz informacje o stylach CSS. Możliwe są oczywiście inne ustawienia tego widoku, ale z tym spotkamy się najczęściej. Ważne jest, aby dobrze rozumieć, czym różni się struktura opisana w pliku HTML od obiektowego modelu strony, czyli DOM.

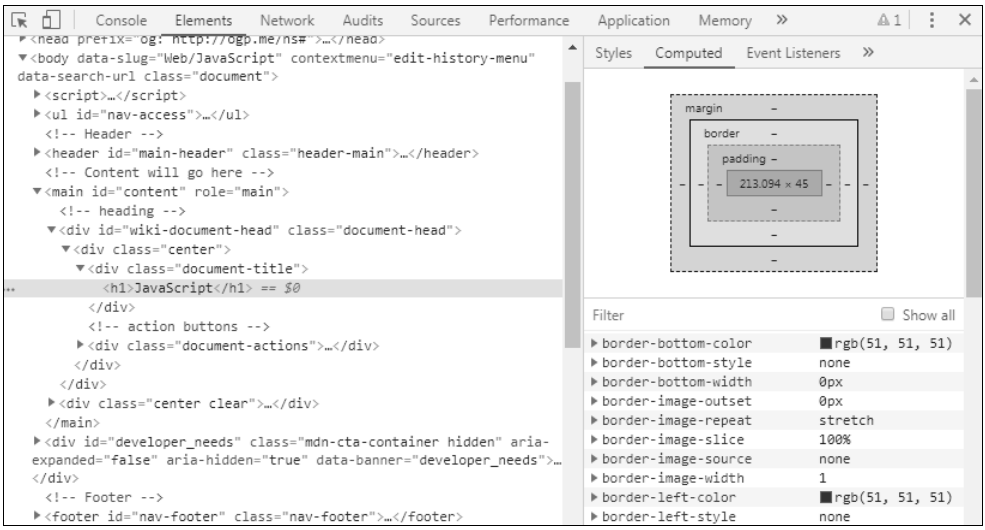
W zakładce *Elements* widzimy elementy strony, które reprezentują wyrenderowany DOM, a nie strukturę, jaką opisaliśmy w pliku HTML. Czasami wizualnie może to wyglądać podobnie, jednak w bardziej rozbudowanych aplikacjach są widoczne różnice. Załóżmy na przykład, że mamy następującą strukturę strony w pliku HTML:

```
<head>
  <script src="script.js" defer></script>
</head>
<body>
  <p>Jakiś napis</p>
</body>
```

W skrypcie JavaScript tworzymy dynamicznie dodatkowy element, np. przycisk:

```
const button = document.createElement('button');
button.textContent = 'Kliknij mnie';
document.querySelector('body').appendChild(button);
```

Co zatem pokaże się w strukturze przedstawionej w zakładce *Elements*? Zobaczymy tam zarówno znacznik tekstowy <p>, jak i nowo dodany przycisk <button>, gdyż jest to struktura przedstawiająca aktualnie wyrenderowane drzewo DOM. Strukturę opisaną w pliku HTML możemy natomiast podejrzeć, wyświetlając tzw. źródło strony. Aby to zrobić, w systemie Windows możemy użyć kombinacji klawiszy *Ctrl+U*. Przykładową zawartość zakładki *Elements* przedstawia rysunek 2.2.



RYSUNEK 2.2. *Zakładka Elements*

Gdy będziemy klikać poszczególne węzły w wyrenderowanym drzewie DOM, będą one od razu podświetlane na stronie. Dodatkowo w każdej chwili możemy kliknąć dowolny element strony prawym przyciskiem myszy i wybrać opcję *Zbadaj* (lub *Zbadaj element*), co skutkuje zaznaczeniem odpowiedniego węzła na liście.

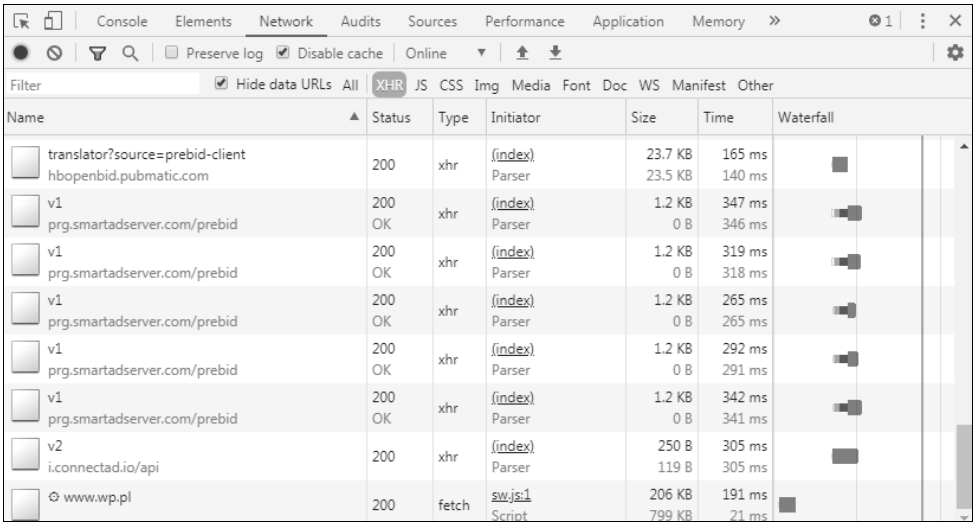
Drugą ważną częścią zakładki *Elements* są informacje o stylach przypisanych do elementu. Mamy tutaj dwie istotne sekcje: *Styles* oraz *Computed*. W sekcji *Styles* znajdują się informacje o stylach przypisywanych przez poszczególne klasy CSS. Warto jednak korzystać także z sekcji *Computed*, gdzie widzimy style ostatecznie przypisane do danego elementu. Czasami zdarza się bowiem, że próbujemy ustawić jakąś właściwość CSS dla elementu, lecz mimo naszych starań otrzymuje on zupełnie inne style. Gdy wybierzemy interesującą nas właściwość CSS, w sekcji *Computed* będziemy mogli podejrzeć ostatecznie przypisaną wartość oraz listę wartości, które zostały zignorowane (dlatego tak ważne jest dobre zrozumienie np. kaskadowości stylów CSS).

Dodatkowo w sekcji tej widzimy w formie graficznej najważniejsze informacje na temat wymiarów elementu w odniesieniu do tzw. modelu pudełkowego (*CSS Box Model*), czyli z uwzględnieniem takich parametrów jak `margins`, `paddings`, `borders`.

Konsola posiada bardzo dużo ciekawych zakładek, a dodatkowo umożliwia instalowanie zewnętrznych dodatków, np. wielu przydatnych narzędzi do pracy z popularnymi frameworkami, np. React. Wrócimy do tego zagadnienia w rozdziale, w którym będziemy mówić o lokalnych magazynach do przechowywania różnych informacji, w tym m.in. o *cookies*, gdzie wykorzystamy popularny dodatek *open source* EditThisCookie.

W systemie Windows w każdej chwili możemy użyć skrótu *Shift+Ctrl+P*, aby otworzyć pole do wyszukiwania różnych funkcjonalności narzędzi deweloperskich. Wpiszmy tam np. frazę „animations", co powinno pokazać opcję *Show Animations*. Po jej kliknięciu otworzy się dodatkowa zakładka *Animations*, pozwalająca precyzyjnie analizować animacje występujące na stronie. Bardzo przydatna jest możliwość regulowania czasu trwania animacji (100%, 25% lub 10%), dzięki której wszystkie animacje na stronie będą się wykonywały np. 10 razy wolniej. Pozwala to bardzo precyzyjnie analizować, jak zachowują się elementy strony podczas animacji, szczególnie gdy są to animacje złożone lub trwające bardzo krótko (rzędu kilkuset milisekund).

Jako ostatnią omówimy zakładkę *Network*, z której będziemy korzystać w rozdziale poświęconym asynchroniczności i komunikacji strony z serwerem. Teraz przedstawimy jej najważniejsze elementy (rysunek 2.3).



RYSUNEK 2.3. *Zakładka Network*

Znajdują się tutaj najważniejsze informacje o tzw. zasobach wczytywanych przez przeglądarkę użytkownika. Możemy wybrać określone typy zasobów, np. pliki JavaScript, CSS, obrazki (oznaczone jako *Img*) itp. Najczęściej jednak będzie nas interesować sekcja *XHR*, czyli zasoby pobierane np. przy wykorzystaniu technologii Ajax (o której będziemy mówić w dalszych rozdziałach).

Listę zasobów możemy dodatkowo filtrować, co jest przydatne, gdy szukamy konkretnych informacji. Dodatkowo dla każdego zasobu widzimy status odpowiedzi HTTP. Zasoby ze statusami 4xx lub 5xx często oznaczone są kolorem czerwonym jako błędy (np. błąd wewnętrzny serwera 500, błąd autoryzacji 401 itp.). Widnieje tu także informacja o czasie potrzebnym przeglądarce do ściągnięcia danego zasobu wraz z informacją o jego wielkości. Sekcja *Waterfall* graficznie przedstawia momenty pobierania poszczególnych zasobów wraz z zaznaczeniem czasu pobierania danego zasobu.

Podobnie jak w zakładce *Console*, tutaj również możemy zachować historię pobierania zasobów po przeładowaniu strony dzięki włączonej opcji *Preserve log*. Ciekawa jest także sekcja *Online*, w której możemy ustawić tzw. *throttling*, czyli zasymulowanie wolniejszego łącza internetowego, np. slow 3G. Pozwala to zasymulować pracę naszej strony na wolnych łączach, co zdarza się głównie na urządzeniach mobilnych.

W prawym górnym rogu mamy ikonę opcji dodatkowych (symbol koła zębatego). Możemy tu ustawić odpowiedni dla nas widok listy pobieranych zasobów (opcja *Use large request rows*) oraz zapisać widoki strony w poszczególnych punktach czasowych (*Capture screenshots*). Druga opcja pozwala dokładnie zobaczyć, jak w różnych momentach wygląda nasza strona, co jest przydatne szczególnie podczas analizy zachowania strony na wolnych łączach internetowych.

Zachęcam Cię do poeksperymentowania z omawianymi tutaj zakładkami narzędzi deweloperskich na różnych stronach internetowych, aby przyswoić sobie te informacje. Naprawdę wiele problemów można bardzo szybko rozwiązać poprzez umiejętne korzystanie z konsoli przeglądarki.

JavaScript. Interaktywne aplikacje webowe

- **Dowiedz się, jak wygląda obsługa przeglądarek WWW w JavaScripcie**
- **Poznaj podstawy tworzenia aplikacji desktopowych w tym języku**
- **Naucz się tworzyć w nim aplikacje mobilne**

JavaScript jest niezwykle uniwersalnym językiem programowania. Idealnie sprawdza się na przykład w projektowaniu wszelkiego rodzaju aplikacji internetowych — od działających pod przeglądarkami, przez te pisane na urządzenia mobilne, po aplikacje, które obsługują smart TV i inne inteligentne urządzenia domowe. W związku z tym wokół JavaScriptu powstał cały ekosystem narzędzi wspomagających i rozszerzających możliwości tego języka. Wystarczy wspomnieć platformę programistyczną Electron, pozwalającą tworzyć aplikacje imitujące oprogramowanie desktopowe, popularny wśród developerów edytor kodu Visual Studio Code, a także rozmaite frameworki i biblioteki ułatwiające codzienną pracę programistom JavaScriptu.

Nim jednak zaczniesz zaprzyjaźniać się z ekosystemem tego języka, warto, byś poświęcił nieco uwagi tzw. czystemu JavaScriptowi. Świetną okazją do tego będzie praca z tą książką, dzięki której nauczysz się tworzyć różnego rodzaju aplikacje internetowe z wykorzystaniem natywnych mechanizmów i możliwości języka JavaScript. Wraz z podręcznikiem *JavaScript. Interaktywne aplikacje webowe*:

- **Poznasz podstawy pracy z HTML i modelami obiektowymi dokumentów (DOM)**
- **Nauczysz się obsługi zdarzeń w JavaScripcie**
- **Opanujesz najważniejsze umiejętności potrzebne do połączenia JavaScriptu z CSS**
- **Dowiesz się, czym jest asynchroniczny JS**
- **Zdobędziesz wiedzę, dzięki której zaimplementujesz swoje projekty w internecie**

Poznaj czysty JavaScript!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-5638-2

Cena: 49,00 zł