Dynamiczne wczytywanie skryptów

Spis treści

- Dynamiczne wczytywanie skryptów
  - Co to?
  - Sposób na ninja
  - Sposób prymitywny
  - Sposób rozbudowańszy
  - Sposób pro dla browserów
  - Jedyny pro sposób dla serwerów
  - Pro sposób dla wszystkich
  - Przyszłość – ES Modules
  - Rebelia – Chainable Module Definition

Umieszczając JS na swojej stronie, zapewne nieraz słyszałeś, że masz je umieścić na końcu `body` i najlepiej je wsadzić w jeden, góra trzy pliki i zminifikować (https://github.com/mishoo/UglifyJS/). Takie są standardowe zalecenia dotyczące JavaScript na stronie. Ale czy istnieje jakaś alternatywa? Przecież wiadomo, że nawet po wykonaniu tych wszystkich kroków, w tym niby mniejszym pliku JS znajduje się pełno kodu, który nie zostanie wykorzystany na danej podstronie (no przecież nie wszędzie wpychamy slider czy też okienka). Okazuje się, że istnieje! Zwie się "dynamiczne (leniwe) ładowanie skryptów".

Co to?

Jeśli znasz PHP, na pewno słyszałeś o autoładowaniu klas. Jak to działa?

```
function __autoload(string $name) {
    require_once('/tajny/folder/z_klasami/' . $name . '.php');
}
$p = new Klasaktorejwczesniejniktniezaladowalaitaknierzucibledu();
```

Czemu nasza klasa o zbyt długiej nazwie błędu nie rzuci? Bo PHP – dzięki magii `__autoload` – przy próbie utworzenia jej instancji, załączy plik z nią! Na podobnej zasadzie działa to w JS.

Sposób na ninja

Ten sposób stosują "dostawcy usług zewnętrznych", czyli np. widgety Google+, Twittera, Facebooka, a także Google Analytics. Przykładowy kod (z Google+):

```
<script type="text/javascript">
    window.___gcfg = {lang: 'pl'};
    (function() {
        var po = document.createElement('script'); po.type = 'text/
javascript'; po.async = true;
        po.src = 'https://apis.google.com/js/plusone.js';
        var s = document.getElementsByTagName('script')[0]; s.paren
tNode.insertBefore(po, s);
    })();
</script>
```

Jak to działa? `window.___gcfg` robi tutaj za ekwiwalent przestrzeni nazw (które to pojęcie wytłumaczyłem w innym tutorialu (http://comandeer.pl/tutorials/js-beauty.html#namespaces)), w której Google przechowuje sobie ustawienia dla swojego widgetu (jak na razie głównie język). Później pojawia się interesująca nas część. Otóż zostaje utworzony dynamicznie znacznik `script` (żeby nie brudzić globalnego scope, zrobiono to w funkcji natychmiastowego wywołania; też wytłumaczone w tamtym tutku) i wstawiony do dokumentu. Przenalizujmy to sobie

```
var po = document.createElement('script');
```

Ta linijka tworzy właśnie znacznik `script`. w ten sposób można tworzyć dowolne znaczniki (nawet te nieistniejące). Zapisujemy sobie go do zmiennej `po`, żeby móc go później wygodnie używać. Następnie nadajemy mu potrzebne atrybuty: `[type]`, `[async]` i `[src]`. Tak naprawdę jeśli nadalibyśmy sam `[src]`, powyższy kod działałby równie dobrze. Atrybut `[type]` i tak jest zbędny w HTML5. Natomiast dziwić może fakt, że można usunąć `[async]` (jak sama nazwa wskazuje, wymusza asynchroniczne wczytywanie skryptu). Mathias Bynens wyjaśnia:

> Kyle Simpson points out that all dynamically inserted scripts default to .async=true as per the spec. Most browsers have always implemented it this way. The only exceptions are Firefox 3.6 and Opera 10+, who execute scripts in insertion order by default, which may cause an unnecessary delay. By setting .async=true explicitly we make sure ga.js doesn't wait for other previously loaded scripts and doesn't block any subsequently loaded scripts. This line of code only affects Firefox 3.6. (Sadly, it doesn't seem to make a difference in Opera). If you're not using any other scripts, or Firefox 3.6 support is not an issue, you can safely remove it to save even more bytes
>
> *— http://mathiasbynens.be/notes/async-analytics-snippet*

Zatem powyższy kod można spokojnie skrócić, wywalając ustawianie `[type]` i `[async]` – ważne jest jedynie `[src]` (w końcu jakiś skrypt musimy wczytać).

Kolejna linijka to wstawianie skryptu na stronę. Wstawiamy go w rodzicu pierwszego elementu `script` na stronie… Zaraz? Że co? A czemu tak? Otóż skrypt Google+ (czy też Google Analytics) jest skryptem zewnętrznym i nie ma jakiejkolwiek kontroli nad kodem strony, na której jest umieszczany. Równie dobrze taka strona może wyglądać tak:

```
<!DOCTYPE html>
    <meta charset="UTF-8">
    <title>Tytuł</title>
    <p>reszta</p>
```

Taki kod jest w pełni poprawnym HTML5 (a ile można znaleźć niepoprawnych, to też wiadomo) – nie ma tu ani `head`, ani `body`, do których tradycyjnie wstawia się skrypty. Dlatego też nie można sobie pozwolić na założenie, że na stronie, nad którą nie mamy kontroli, te elementy istnieją. Natomiast rodzic pierwszego elementu `script` istnieć musi. Czemu? Na początek znajdźmy ten "pierwszy element `script`". Wiesz co to? Tak, to nasz skrypt wstawiający znacznik `script` (czyli ten, co właśnie go omówiłem, jeśli wciąż masz wątpliwości)! A co jest jego rodzicem? Coś musi być – na pewno znajduje się w `document` (to de facto główny węzeł DOM (https://developer.mozilla.org/en-US/docs/DOM), który istnieje zawsze). Zatem jeśli nie znasz strony, na której pracujesz, a masz tam coś wstawić, wstaw to w rodzica elementu, który na pewno istnieje.

Ten sposób wstawiania skryptów jednak nie daje dużej kontroli nad procesem jego wczytywania – po prostu go wstawia… i tyle. Jednak jest doskonałym punktem wyjścia do własnych rozwiązań.

Sposób prymitywny

Zatem chcesz wczytywać swój skrypcik i dopiero po jego wczytaniu odpalić zadania, za które jest odpowiedzialny? OK, to musisz odkryć kiedy nasz dynamicznie wstawiony skrypt zostanie wczytany! Na szczęście nie jest to zadanie trudne, bo odpowiada za to zdarzenie `load` – to dla niego należy dodać nasze zadania. A jak to zrobić najwygodniej? Za pomocą funkcji!

```
function addScript( uri, callback ) {

}
```

Jak na razie jest pusta, ale to za chwilę się zmieni. Na sam początek warto sprawdzić czy `uri` jest prawdziwym URI (albo przynajmniej czy jest tekstem – w tym przypadku właśnie to sprawdzę, dla uproszczenia sprawy, ale w rzeczywistym projekcie lepiej jednak użyć odpowiedniego wyrażenia regularnego (http://mathiasbynens.be/demo/url-regex)). Jeśli nie, przerywamy funkcję.

```
if ( typeof uri !== 'string' ) {
    return false;
}
```

> Zamiast zwrócić fałsz, można walnąć wyjątkiem.

Kolejny krok to stworzenie naszego znacznika `script`:

```
var s = document.createElement( 'script' );
s.src = uri;
```

Pomijam tutaj zarówno atrybut `[async]`, jak i `[type]`, ale możesz je dodać (spowalniając nieco skrypt (http://mathiasbynens.be/notes/async-analytics-snippet)).

Następnie wstawiamy skrypt na stronę:

```
document.body.appendChild( s );
```

Wstawiam do `body`, bo to moja strona i na mojej stronie `body` jest zawsze. Jeśli nie chcesz stosować `body`, wstawiaj skrypt w podobny sposób, jak robi to Google.

Teraz część z odpalaniem zadań po załadowaniu skryptu. Skorzystamy z częstej taktyki programistów JS – callbacków. Jak widzisz, drugi parametr funkcji `addScript` właśnie tak się nazywa. Cóż to ten callback? To taka "funkcja zwrotna". Przydaje się najczęściej przy skryptach asynchronicznych (czyt. Ajax). Z racji tego, że są wykonywane asynchronicznie, reszta kodu nie musi czekać aż skończą swe działanie. Z tego też względu nie można z działań asynchronicznych zwrócić wartości… I ten problem rozwiązuje callback. Funkcja ta jest wywoływana przez funkcję asynchroniczną w chwili, gdy ta kończy swe działanie. Najczęściej callback otrzymuje jako parametr to, co wygeneruje funkcja asynchroniczna (np. treść strony, którą pobraliśmy Ajaksem). Jeśli kiedyś korzystałeś z Ajaksu w jQuery, callbacki nie są dla Ciebie obce (`success`, `error`…) i wiesz na jakiej zasadzie działają. Jeśli nie wiesz, to za chwilę zobaczysz jeden w akcji.

Oczywiście musimy mieć pewność, że nasz `callback` jest funkcją. Sprawdzimy to przy pomocy operatora `typeof`. Jeśli jest funkcją, należy przypisać go do zdarzenia `load` naszego skryptu:

```
if (typeof callback === 'function' ) {
    s.onload = callback;
}
```

Voila! Czemu nie zastosowałem tutaj `addEventListener`? Ze względu na IE 8. Zresztą – tak jest ciut szybciej.

Całość naszej funkcji wygląda następująco:

```
function addScript( uri, callback ) {
    if ( typeof uri !== 'string' ) {
        return false;
    }

    var s = document.createElement( 'script' );
    s.src = uri;

    document.body.appendChild( s );

    if ( typeof callback === 'function' ) {
        s.onload = callback;
    }
}
```

Przykład zastosowania:

```
<!DOCTYPE html>
    <html lang="pl" dir="ltr">
        <head>
            <meta charset="UTF-8">
            <title>Ehhhh</title>
        </head>
        <body>
            <script>
            function addScript( uri, callback ) {
                if ( typeof uri !== 'string' ) {
                    return false;
                }

                var s = document.createElement( 'script' );
                s.src = uri;

                document.body.appendChild( s );

                if ( typeof callback === 'function' ) {
                    s.onload = callback;
                }
            }

            addScript( 'http://comandeer.pl/js/zoom.js', function()
 {
                console.log( 'Wczytano skrypt' );
            } );
            </script>
        </body>
    </html>
```

Gdy skrypt się wczyta, do konsoli zostanie dodany wpis "Wczytano skrypt".

Sposób rozbudowańszy

"Czemu to rozbudowywać?" – zapytasz. A ja odpowiem: spróbuj załadować jQuery i jakiś skrypt od niego zależny. Na bank okaże się, że często to ten drugi skrypt wczyta się pierwszy. Musimy jakoś upewnić się, że zostanie zachowana narzucona kolejność wczytywania. Na szczęście jest i na to sposób! Wystarczy wrzucić wymagane skrypty do tablicy, a następnie zagnieżdżać obsługę zdarzenia `load`. Szybki szkic rozwiązania:

```
function addScripts( uris, callback ) {
    if ( !uris instanceof Array || uris.length < 1 ) {
        return false;
    }

    function add( i ) {
        var uri = uris[ i ],
            s = document.createElement( 'script' );

        s.src = uri;

        document.body.appendChild( s );

        if ( uris[ ++i ] ) {
            s.onload = function() {
                add( i );
            };
        } else if ( typeof callback === 'function' ) {
            s.onload = callback;
        }
    };

    add( 0 );
}
```

Co robi ten kod? Otóż zamiast przyjmować adres skryptu, przyjmuje tablicę adresów. Oczywiście sprawdzamy, czy na pewno mamy tablicę i czy zawiera przynajmniej jeden element:

```
if ( !uris instanceof Array || uris.length < 1 )
```

Istnieje jeszcze kilka innych sposobów na sprawdzenie, czy zmienna jest tablicą (np. `Array.isArray( uris )`), ale należy pamiętać, żeby nie robić `typeof uris === 'array'`, bo typ dla tablicy to `object`, nie `array`!

Pojawia się także wewnętrzna funkcja `add`, która to właśnie wykonuje brudną robotę i dodaje skrypty. Jako parametr bierze indeks elementu, który ma wstawić, a następnie inicjuje wstawianie kolejnego elementu (funkcja rekurencyjna). Jeśli kolejnego elementu nie ma, do zdarzenia `load` ostatniego skryptu jest dawany nasz callback. Prosty i skuteczny sposób na zapewnienie wczytywania skryptów zgodnie z podaną kolejnością.

Przykład:

```
<!DOCTYPE html>
    <html lang="pl" dir="ltr">
        <head>
            <meta charset="UTF-8">
            <title>Ehhhh</title>
        </head>
        <body>
            <script>
            function addScripts( uris, callback ) {
                if ( !uris instanceof Array || uris.length < 1 ) {
                    return false;
                }

                function add( i ) {
                    var uri = uris[ i ],
                        s = document.createElement( 'script' );

                    s.src = uri;

                    document.body.appendChild( s );

                    if ( uris[ ++i ] ) {
                        s.onload = function() {
                            add( i );
                        };
                    } else if ( typeof callback === 'function' ) {
                        s.onload = callback;
                    }
                };

                add( 0 );
            }

            addScripts( [
                'https://ajax.googleapis.com/ajax/libs/jquery/1.7.
2/jquery.min.js'
                ,'http://comandeer.pl/js/zoom.js'
            ], function() {
                console.log( 'Wczytano skrypt' );
            } );
```

```
            </script>
        </body>
    </html>
```

Sposób pro dla browserów

Ten problem postanowiono rozwiązać w sposób kulturalny i jakoś go ustandaryzować. Tak powstało Asynchrounous Module Definition (https://github.com/amdjs/amdjs-api/wiki/AMD), na które składają się *de facto* dwie funkcje – `define` i `require`.

`define` służy do deklarowania modułów:

```
define( 'opcjonalnanazwamodulu', [ 'tablica/modulow', 'lubinnychpli
kowjs', 'potrzebnychtemumodulowido', 'uruchomienia' ], function( $1
, $2, $3, $4 ) { //poszczególne parametry to wczytane moduły z tabl
icy
    //tutaj kod modułu
    return obiektModulu; //każdy moduł powinien się "zwracać"
} );
```

`require` natomiast wykonuje daną funkcję po wczytaniu koniecznych do tego modułów

```
require( [ 'tablica/modulow', 'lubinnychplikowjs', 'potrzebnychdo',
 'uruchomienia' ], function( $1, $2, $3, $4 ) {
    //tutaj operacje
} );
```

Zwróć uwagę na parametry w obydwóch funkcjach. Znajdują się tam wczytane moduły, na których można operować. Dzięki temu kod staje się hermetyczny i nic nie wypływa do globalnego scope (zostaje w `require`/`define`). Dlatego ważne jest, żeby każdy moduł "zwracał się" przez `return`.

Oczywiście nie wszystkie skrypty są zgodne z AMD (nie stosują `define`), co nie przeszkadza w ich wczytywaniu przy wykorzystaniu tej techniki. jQuery jest ciekawym przykładem, który stoi pośrodku: dodaje się do globalnego scope, ale równocześnie zachowuje jako moduł AMD.

Najpopularniejsza implementacja AMD to Require.js (http://requirejs.org), z którego osobiście korzystam.

Jedyny pro sposób dla serwerów

Czyli jak to zrobić w node.js przy pomocy składni CommonJS (CJS):

```
var modul = require( 'nazwamodulu' ),
    inny = require( 'lub/sciezka/do/pliku.js' );
```

Tyle. Teoretycznie można zastosować pokazane powyżej AMD, ale to nie ma sensu.

Pro sposób dla wszystkich

Oczywiście składni CJS zaczęto używać także dla przeglądarek. Co więcej, coraz częściej mamy do czynienia z izomorficznymi aplikacjami internetowymi (http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/), co wymusza znalezienie sposobu na współdzielenie tego samego kodu na serwerze i w przeglądarce. Tym samym powstało UMD – Universal Module Definition (https://github.com/umdjs/umd), które pozwala korzystać nam równocześnie z AMD, jak i CJS. Jest to obecnie najczęściej polecany sposób rozprowadzania kodu JS.

Przyszłość – ES Modules

Przyszłość przyniesie nam moduły natywnie wbudowane w ECMAScript. Niestety, ich składnia jest niekompatybilna zarówno z AMD, jak i CJS. Niemniej istnieją już odpowiednie narzędzia (https://github.com/systemjs/systemjs) zapewniające choć cień zgodności.

Po wyczerpujący opis modułów ES6 odsyłam do artykułu na 2ality.com (http://www.2ality.com/2014/09/es6-modules-final.html).

> Jeśli chcesz dowiedzieć się praktycznie wszystkiego o systemach modułów w JS i różnicach między nimi, zajrzyj do **wyczerpującego artykułu Addy'iego Osmaniego (http://addyosmani.com/writing-modular-js/)**.

Rebelia – Chainable Module Definition

Czyli przeniesienie łańcuszków z jQuery na poziom modułów. Jedyna implementacja to Melchior.js (http://labs.voronianski.com/melchior.js/). Wygląda to nawet fajnie i składniowo jest ładniejsze od AMD (ale nie CJS!), ale jest całkowicie niekompatybilne. Tym samym raczej nie zyska dużej popularności i pozostanie ciekawostką (zwłaszcza, że za chwilę wejdą do użycia ES Modules).