Poznaj potencjał języka JavaScript!

JavaScript Aplikacje WWW

HELION

O'REILLY®

Alex MacCaw

Tytuł oryginału: JavaScript Web Applications

Tłumaczenie: Daniel Kaczmarek

ISBN: 978-83-246-3887-1

Wydawnictwo HELION
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jascww.zip

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

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Modele i dane

Gdy aplikacja ma utrzymywać stan na kliencie, jednym z wyzwań stojącym przed programistą jest zapewnienie odpowiedniego zarządzania danymi. Standardowo w momencie wywołania strony dane pobiera się bezpośrednio z bazy danych, a wynik ich przetwarzania umieszcza się bezpośrednio na stronie. Jednak w stanowych aplikacjach JavaScript proces zarządzania danymi przebiega zupełnie inaczej. Nie obowiązuje w nich model wywołanie-odpowiedź, nie są także dostępne zmienne na serwerze. Dane są za to pobierane zdalnie i tymczasowo przechowywane po stronie klienta.

Wprawdzie zaimplementowanie odpowiedniego mechanizmu może być dość trudnym zadaniem, jednak uzyska się dzięki temu co najmniej kilka korzyści. Na przykład dostęp do danych znajdujących się po stronie klienta jest praktycznie natychmiastowy, tak jakby dane pobierane były z pamięci. Może to zdecydowanie poprawić działanie interfejsu aplikacji, ponieważ każda czynność użytkownika skutkuje natychmiastowym zwróceniem wyniku. Dzięki temu aplikacja znajduje zdecydowanie większe uznanie u użytkowników.

Aby odpowiednio skonstruować mechanizm przechowywania danych po stronie klienta, trzeba najpierw szczegółowo przeanalizować wszystkie okoliczności. Zadanie najeżone jest różnorodnymi przeszkodami i pułapkami, w które wpadają zwłaszcza początkujący programiści — szczególnie gdy ich aplikacje coraz bardziej się rozrastają. W tym rozdziale zobaczymy, jak w najbardziej efektywny sposób zaimplementować przechowywanie danych po stronie klienta. Wskazane zostaną także najbardziej zalecane wzorce i praktyki.

MVC i przestrzenie nazw

Aby opracować taką architekturę aplikacji, która będzie przejrzysta i łatwa w utrzymaniu, trzeba przede wszystkim zapewnić pełną separację widoków aplikacji, jej stanu i danych. W przypadku wykorzystania wzorca MVC zarządzanie danymi odbywa się w modelu („M" w nazwie MVC). Modele powinny być rozłączne względem widoków i kontrolerów. Natomiast logika wyznaczająca sposób manipulowania danymi oraz odpowiedzialna za zachowanie aplikacji powinna być osadzona w modelu oraz oznaczona odpowiednią przestrzenią nazw.

W JavaScripcie odpowiednią przestrzeń nazw dla funkcji i zmiennych można zapewnić przez uczynienie ich właściwościami obiektu. Na przykład:

```
var User = {
  records: [ /*… */ ]
};
```

Tablica użytkowników ma teraz odpowiednią przestrzeń nazw `User.records`. Funkcje związane z obsługą użytkowników również mogą należeć do przestrzeni nazw wyznaczanej przez model `User`. Można na przykład zaimplementować funkcję `fetchRemote()`, która będzie pobierać z serwera dane użytkownika:

```
var User = {
  records: [],
  fetchRemote: function(){ /*… */ }
};
```

Umieszczenie wszystkich właściwości modelu w przestrzeni nazw zapewni, że nie wystąpią żadne konflikty, a jednocześnie utrzymana zostanie zgodność z paradygmatem MVC. Poza tym zmniejszy się zagrożenie, że tworzony kod źródłowy zamieni się z biegiem czasu w niezrozumiałą spiralę funkcji i wywołań zwrotnych.

Zakres wykorzystania przestrzeni nazw można jeszcze pogłębić i umieścić wszystkie funkcje operujące na instancjach użytkowników w rzeczywistych obiektach użytkowników. Przyjmijmy, że na rekordzie użytkownika zaimplementowana jest funkcja `destroy()`. Odnosi się ona do konkretnego użytkownika, zatem powinna być wywoływana na instancji obiektu `User`:

```
var user = new User;
user.destroy()
```

Aby osiągnąć zamierzony efekt, należy uczynić z `User` klasę, a nie zwykły obiekt:

```
var User = function(atts){
  this.attributes = atts || {};
};

User.prototype.destroy = function(){
  /*… */
};
```

Natomiast wszelkie funkcje i właściwości, które nie dotyczą konkretnego użytkownika, mogą być właściwościami bezpośrednio obiektu `User`:

```
User.fetchRemote = function(){
  /*… */
};
```

Więcej ciekawych informacji na temat przestrzeni nazw można znaleźć na blogu Petera Michaux, który opublikował bardzo ciekawy artykuł na ten właśnie temat (*http://michaux.ca/articles/javascript-namespacing*).

Tworzenie ORM

Biblioteki odwzorowań obiektowo-relacyjnych, czyli ORM, są zwykle wykorzystywane w językach programowania innych niż JavaScript. Stanowią one jednak bardzo przydatne narzędzia do zarządzania danymi, a także znacznie ułatwiają użycie modeli w aplikacji JavaScript. Za pomocą ORM można na przykład powiązać model ze zdalnym serwerem — w wyniku takiego powiązania wszelkie zmiany w instancjach modelu będą wysyłane w tle do serwera w ramach wywołań Ajax. Można także powiązać instancję modelu z elementem HTML, dzięki czemu wszelkie zmiany w instancji modelu spowodują odpowiednią zmianę w widoku. Przykłady te zostaną rozwinięte nieco później, a na razie zobaczmy, jak tworzy się ORM.

Zasadniczo ORM jest po prostu warstwą obiektów opakowującą jakieś dane. Standardowo ORM stanowią warstwę abstrakcji dla baz danych SQL, jednak w naszym przypadku ORM będzie po prostu warstwą abstrakcji dla typów danych JavaScriptu. Zaletą takiej dodatkowej warstwy jest to, że podstawowy zbiór danych można rozszerzyć o dodatkowe rozwiązania przez dodanie własnych funkcji i właściwości. Dzięki temu można dodawać mechanizmy weryfikacji poprawności albo utrzymywania danych, obserwatory czy wywołania zwrotne do serwera, a jednocześnie nadal mieć możliwość wielokrotnego wykorzystywania kodu źródłowego.

Dziedziczenie przez prototypy

Do stworzenia naszej własnej biblioteki ORM wykorzystamy `Object.create()`, co jest podejściem nieco odmiennym od przykładów opisywanych w rozdziale 1, w których korzystaliśmy z klas. Nowe podejście pozwoli nam jednak na skorzystanie z dziedziczenia przez prototypy zamiast używania funkcji konstruktorów i słowa kluczowego `new`.

Funkcja `Object.create()` przyjmuje jeden argument — obiekt prototypu — i zwraca nowy obiekt z przekazanym obiektem prototypu. Inaczej mówiąc, przekazuje się do niej obiekt, a funkcja zwraca nowy obiekt, potomny po obiekcie przekazanym.

Funkcja `Object.create()` została dopiero niedawno dodana do specyfikacji ECMAScript, 5th Edition i dlatego jeszcze nie wszystkie przeglądarki ją obsługują — należy do nich między innymi IE. Nie jest to jednak wielki problem, ponieważ w razie potrzeby obsługę tej funkcji można zaimplementować samemu:

```
if (typeof Object.create !== "function")
    Object.create = function(o) {
      function F() {}
      F.prototype = o;
      return new F();
    };
```

Powyższy przykład pochodzi z artykułu Douglasa Crockforda na temat dziedziczenia przez prototypy (*http://javascript.crockford.com/prototypal.html*). Warto się z nim zapoznać, aby lepiej poznać mechanizmy rządzące prototypami i dziedziczeniem w języku JavaScript.

Utworzymy teraz obiekt `Model`, którego zadaniem będzie tworzenie nowych modeli i instancji:

```
var Model = {
  inherited: function(){},
  created: function(){},

  prototype: {
    init: function(){}
  },

  create: function(){
    var object = Object.create(this);
    object.parent = this;
    object.prototype = object.fn = Object.create(this.prototype);

    object.created();
    this.inherited(object);
    return object;
  },

  init: function(){
    var instance = Object.create(this.prototype);
```

```
    instance.parent = this;
    instance.init.apply(instance, arguments);
    return instance;
  }
};
```

Programistów, którzy nie poznali dotychczas funkcji `Object.create()`, definicja `Model` może nieco zniechęcać, więc rozłożymy ją na czynniki pierwsze. Funkcja `create()` zwraca nowy obiekt, który dziedziczy po obiekcie `Model`. W ten sposób tworzone będą nowe modele. Funkcja `init()` zwraca natomiast nowy obiekt, który dziedziczy po `Model.prototype` — czyli instancję obiektu `Model`:

```
var Asset = Model.create();
var User  = Model.create();

var user = User.init();
```

Dodawanie właściwości ORM

Jeśli teraz do `Model` dodamy właściwości, staną się one dostępne we wszystkich modelach potomnych:

```
// Dodanie właściwości obiektu
jQuery.extend(Model, {
  find: function(){}
});

// Dodanie właściwości instancji
jQuery.extend(Model.prototype, {
  init: function(atts) {
    if (atts) this.load(atts);
  },

  load: function(attributes){
    for(var name in attributes)
      this[name] = attributes[name];
  }
});
```

Metoda `jQuery.extend()` odpowiada wykonaniu pętli `for` i wykonaniu w niej ręcznego kopiowania właściwości. Tak właśnie postępujemy w funkcji `load()`. Teraz właściwości obiektu i instancji są propagowane w dół, do konkretnych modeli:

```
assertEqual( typeof Asset.find, "function" );
```

W praktyce liczba dodawanych właściwości będzie dość znaczna, dlatego najlepiej jest od razu zawrzeć w obiekcie `Model` funkcje `extend()` i `include()`:

```
var Model = {
  /* ... cięcie ... */

  extend: function(o){
    var extended = o.extended;
    jQuery.extend(this, o);
    if (extended) extended(this);
  },

  include: function(o){
    var included = o.included;
    jQuery.extend(this.prototype, o);
    if (included) included(this);
```

```
  }
};

// Dodanie właściwości obiektu
Model.extend({
  find: function(){}
});

// Dodanie właściwości instancji
Model.include({
  init: function(atts) { /*... */ },
  load: function(attributes){ /*... */ }
});
```

Można już zatem tworzyć nowe zasoby i ustawiać ich atrybuty:

```
var asset = Asset.init({name: "foo.png"});
```

Utrzymywanie rekordów

Potrzebny jest również mechanizm utrzymywania rekordów, to znaczy zapamiętywania odwołań do utworzonych instancji, aby móc z nich korzystać w późniejszym czasie. Do tego celu wykorzystany zostanie obiekt `records` ustawiony w modelu `Model`. Obiekt będzie dodawany do modelu w momencie zapisywania instancji modelu i usuwany z obiektu modelu, gdy usuwana będzie instancja modelu:

```
// Obiekt z zapisanymi zasobami
Model.records = {};

Model.include({
  newRecord: true,

  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this;
  },

  destroy: function(){
    delete this.parent.records[this.id];
  }
});
```

A w jaki sposób można zmieniać istniejącą instancję? To proste — wystarczy zmienić odwołanie do obiektu:

```
Model.include({
  update: function(){
    this.parent.records[this.id] = this;
  }
});
```

Warto także zaimplementować wygodną funkcję odpowiedzialną za zapisywanie instancji, tak by nie trzeba było później sprawdzać, czy instancja została zapisana już wcześniej albo czy trzeba tę instancję utworzyć:

```
// Zapisanie obiektu do records i zapamiętanie odwołania do obiektu
Model.include({
  save: function(){
    this.newRecord ? this.create() : this.update();
  }
});
```

Wreszcie dobrze też zaimplementować funkcję `find()`, która będzie wyszukiwać zasoby o wskazanym identyfikatorze:

```
Model.extend({
  // Znalezienie zasobu o danym ID lub rzucenie wyjątku
  find: function(id){
    return this.records[id] || throw("Rekord nieznany");
  }
});
```

ORM w najprostszej postaci jest już gotowy, można go zatem wykorzystać w praktyce:

```
var asset  = Asset.init();
asset.name = "taki sam, taki sam";
asset.id   = 1
asset.save();

var asset2  = Asset.init();
asset2.name = "ale inny";
asset2.id   = 2;
asset2.save();

assertEqual( Asset.find(1).name, "taki sam, taki sam" );

asset2.destroy();
```

Dodawanie obsługi identyfikatorów

Obecnie za każdym razem, gdy zapisywany jest rekord, trzeba ręcznie zdefiniować jego identyfikator ID. To przykry obowiązek, ale na szczęście można to zadanie zautomatyzować. Najpierw potrzebny jest jakiś mechanizm generowania identyfikatorów — może nim być generator identyfikatorów GUID (ang. *Globally Unique Identifier*). Z technicznego punktu widzenia JavaScript nie może generować pełnoprawnych, 128-bitowych identyfikatorów GUID z powodu ograniczeń API, które potrafi generować tylko liczby pseudolosowe. Generowanie prawdziwie losowych identyfikatorów jest zawsze trudnym zadaniem i systemy operacyjne obliczają je na podstawie adresu MAC, pozycji myszy, sum kontrolnych BIOS-u, a nawet na podstawie pomiarów szumu elektrycznego albo produktów rozpadu radioaktywnego; czasami wręcz używa się lamp z zawartością cieczy oraz wosku, które po rozgrzaniu tworzą różne efekty kolorystyczne! Jednak natywna funkcja `Math.random()`, choć pseudolosowa, w zupełności nam wystarczy.

Robert Kieffer napisał prosty i zwięzły generator identyfikatorów GUID, który za pomocą funkcji `Math.random()` generuje GUID pseudolosowe (*http://www.broofa.com/2008/09/javascript-uuid-function/*). Generator jest na tyle prosty, że możemy jego kod źródłowy zaprezentować w całości:

```
Math.guid = function(){
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
    return v.toString(16);
  }).toUpperCase();
};
```

Mamy więc funkcję, która generuje identyfikatory GUID, zatem zintegrowanie jej z ORM nie nastręczy już trudności. Wystarczy w tym celu zmienić kod funkcji `create()`:

```
Model.extend({
  create: function(){
    if ( !this.id ) this.id = Math.guid();
    this.newRecord = false;
    this.parent.records[this.id] = this;
  }
});
```

Od teraz każdy nowo utworzony rekord będzie miał identyfikator w postaci losowego GUID:

```
ar asset = Asset.init();
asset.save();

asset.id //=> "54E52592-313E-4F8B-869B-58D61F00DC74"
```

Adresowanie odwołań

Uważni Czytelnicy zapewne zwrócili uwagę na błąd związany z odwołaniami w naszej bibliotece ORM. Nie klonujemy instancji ani wtedy, gdy zostają zwrócone przez `find()`, ani w momencie ich zapisywania, przez co jeśli zmienimy którąś z ich właściwości, zmiana dotknie oryginalnego zasobu. Jest to poważny problem, ponieważ oczekujemy, że zasoby będą się zmieniać jedynie po wykonaniu funkcji `update()`:

```
var asset = new Asset({name: "foo"});
asset.save();

// Asercja przekazuje prawidłowo
assertEqual( Asset.find(asset.id).name, "foo" );

// Zmiana właściwości bez wywoływanai update()
asset.name = "wem";

// O nie! Asercja kończy się niepowodzeniem, ponieważ nazwą jest teraz "wem"
assertEqual( Asset.find(asset.id).name, "foo" );
```

Aby wyeliminować błąd, w trakcie działania `find()` utworzymy nowy obiekt. Ponadto konieczne będzie zduplikowanie obiektu zawsze wtedy, gdy dojdzie do utworzenia lub zmiany rekordu:

```
Asset.extend({
  find: function(id){
    var record = this.records[id];
    if ( !record ) throw("Rekord nieznany");
    return record.dup();
  }
});

Asset.include({
  create: function(){
    this.newRecord = false;
    this.parent.records[this.id] = this.dup();
  },

  update: function(){
    this.parent.records[this.id] = this.dup();
  },

  dup: function(){
    return jQuery.extend(true, {}, this);
  }
});
```

Jest jeszcze jeden problem — `Model.records` to obiekt, który jest współużytkowany przez wszystkie modele:

```
assertEqual( Asset.records, Person.records );
```

Daje to niepożądany efekt wymieszania ze sobą wszystkich rekordów:

```
var asset = Asset.init();
asset.save();

assert( asset in Person.records );
```

Rozwiązanie polega na tym, by za każdym razem, gdy tworzony jest nowy model, ustawiać też nowy obiekt `records`. `Model.created()` jest wywołaniem zwrotnym do tworzenia nowego obiektu, dlatego można w nim ustawiać dowolne obiekty dotyczące tego właśnie modelu:

```
Model.extend({
  created: function(){
    this.records = {};
  }
});
```

Ładowanie danych

Jeśli aplikacja internetowa działa w jakiejś części poza przeglądarką, konieczne jest umożliwienie ładowania danych zdalnie z serwera. Zazwyczaj w trakcie uruchamiania aplikacji ładowany jest jakiś podzbiór danych, a po wykonaniu określonych czynności przez użytkownika aplikacja ładuje kolejne potrzebne dane. Zależnie od rodzaju aplikacji i ilości danych komplet danych można pozyskiwać od razu w momencie pierwszego ładowania strony. Jest to przypadek idealny, ponieważ wówczas użytkownicy nie będą już musieli czekać na załadowanie dodatkowych danych. Jednak w przypadku większości aplikacji rozwiązanie takie nie wchodzi w grę, ponieważ zbiór potrzebnych danych jest zbyt duży, by bez problemu zmieścił się w pamięci dostępnej dla przeglądarki.

Tylko dzięki załadowaniu danych od razu przy starcie aplikacji będzie można zapewnić użytkownikom komfort pracy z nią, a czas odpowiedzi zredukować do minimum. Istnieje jednak istotna różnica między początkowym ładowaniem danych, które rzeczywiście są potem wykorzystywane, a ładowaniem danych nadmiarowych, które nigdy nie zostaną użyte. Trzeba zatem przewidzieć, których danych użytkownicy będą potrzebować, albo dokonać odpowiednich pomiarów już w trakcie pracy aplikacji.

Na przykład jeśli ładowana jest lista stronicowana, to dlaczego nie załadować od razu zawartości następnej strony, aby przełączanie między stronami następowało od razu? Albo, co dałoby jeszcze lepszy efekt, po prostu wyświetlić długą listę i automatycznie ładować i wstawiać do niej dane, gdy użytkownik będzie tę listę przewijał (służy do tego wzorzec nieskończonego suwaka). Im mniejsze będzie opóźnienie widoczne dla użytkownika, tym lepiej.

Należy zapewnić, że w trakcie pobierania nowych danych interfejs użytkownika nie będzie się blokował. Warto wyświetlić jakiś wskaźnik postępu ładowania danych, ale całość interfejsu powinna przez cały czas być dostępna dla użytkownika. Liczba sytuacji, w której będzie dochodziło do zablokowania interfejsu, powinna być jak najmniejsza, a najlepiej, by w ogóle do tego nie dochodziło.

Dane można wplatać w definicję strony bądź pobierać je w oddzielnych żądaniach HTTP za pomocą Ajax lub JSONP. Polecam raczej to drugie rozwiązanie, ponieważ wplatanie w kod strony dużych ilości danych powoduje znaczne zwiększenie rozmiaru strony, natomiast dzięki pobieraniu danych równolegle z ładowaniem strony skraca się całkowity czas jej ładowania. Technologie AJAX i JSONP pozwalają dodatkowo umieszczać strony HTML w pamięci podręcznej, dzięki czemu nie trzeba ich ładować na nowo przy każdym kolejnym wywołaniu.

Wplatanie danych

Nie polecam stosowania tego podejścia z powodów opisanych w poprzednim akapicie, jednak technika wplatania danych może być w niektórych okolicznościach przydatna, zwłaszcza gdy ładowane są niewielkie zbiory danych. Technika wplatania danych ma tę niezaprzeczalną zaletę, że bardzo prosto się ją implementuje.

Wystarczy w tym celu generować obiekt JSON bezpośrednio w kodzie strony. W technologii Ruby on Rails odpowiednie rozwiązanie miałoby taką postać:

```
<script type="text/javascript">
  var User = {};
  User.records = <%= raw @users.to_json %>;
</script>
```

W przedstawionym kodzie wykorzystano znaczniki ERB, aby zwrócić przetworzone przez JSON dane użytkownika. Metoda `raw` zapobiega wstawianiu znaków ucieczki przez JSON. W wyniku przetworzenia strony jej kod HTML będzie wyglądał tak:

```
<script type="text/javascript">
  var User = {};
  User.records = [{"first_name": "Alex"}];
</script>
```

JavaScript może przetwarzać dane JSON bez ich dodatkowego przekształcania, ponieważ JSON ma taką samą strukturę jak obiekty języka JavaScript.

Ładowanie danych przy użyciu Ajax

Ładowanie danych przy użyciu Ajax to chyba pierwsza metoda ładowania zdalnych danych, która przychodzi na myśl programistom, gdy mowa o wywołaniach w tle. To nie przypadek: technologia ta została wypróbowana, przetestowana i udostępniona we wszystkich współczesnych przeglądarkach. Ajax nie jest oczywiście pozbawiony wad — historia rozwoju tej technologii, dla której nie istniały żadne standardy, zakończyła się powstaniem niespójnych API. Przez to właśnie, oraz ze względu na wymogi bezpieczeństwa przeglądarek internetowych, ładowanie danych z innych domen jest zadaniem dość skomplikowanym.

Programistom, którzy potrzebują krótkiego wprowadzenia do technologii Ajax i klasy `XMLHttpRequest`, warto polecić artykuł *Getting Started* w witrynie Mozilla Developer (*https://developer.mozilla.org/en/Ajax/Getting_Started*). Z dużym prawdopodobieństwem można założyć, że i tak większość programistów ostatecznie wykorzysta bibliotekę, na przykład jQuery, która będzie udostępniać warstwę abstrakcji nad API Ajax i ukrywać przed programistą różnice w obsłudze technologii między poszczególnymi przeglądarkami. Dlatego w tym punkcie będzie mowa o API biblioteki jQuery, a nie o samej klasie `XMLHttpRequest`.

API Ajax biblioteki jQuery zawiera jedną funkcję niskiego poziomu o nazwie `jQuery.ajax()` oraz kilka jej abstrakcji wyższego poziomu, dzięki którym znacznie zmniejsza się ilość wymaganego do napisania kodu źródłowego. Funkcja `jQuery.ajax()` przyjmuje między innymi zbiór ustawień parametrów wywołania, typ zawartości oraz odwołania zwrotne. W momencie wywołania funkcji żądanie jest asynchronicznie wykonywane w tle.

`url`
: Adres URL żądania. Domyślnym adresem URL jest adres bieżącej strony.

`success`
: Funkcja, która ma zostać wywołana, gdy żądanie zakończy się powodzeniem. Wszystkie dane zwrócone przez serwer są przekazywane do funkcji jako jej parametr.

`contentType`
: Ustawia nagłówek `Content-Type` żądania. Jeżeli żądanie zawiera dane, domyślnym typem zawartości jest `application/x-www-form-urlencoded`, które sprawdza się w większości przypadków.

`data`
: Dane, które należy przesłać do serwera. Jeżeli nie jest to ciąg znaków, jQuery je zserializuje i zakoduje.

`type`
: Metoda HTTP, której należy użyć do wykonania żądania: `GET`, `POST` lub `DELETE`. Metodą domyślną jest `GET`.

`dataType`
: Typ danych, jakiego oczekuje się od serwera. Biblioteka jQuery wymaga podania typu danych wynikowych, aby wiedzieć, w jaki sposób je przetwarzać po ich otrzymaniu. Jeżeli `dataType` nie zostanie określony, jQuery spróbuje ten typ odgadnąć na podstawie typu MIME odpowiedzi z serwera. Obsługiwane są następujące wartości parametru:

  `text`
  : Odpowiedź tekstowa, która nie wymaga żadnego dodatkowego przetwarzania.

  `script`
  : jQuery przyjmie, że danymi wynikowymi jest kod JavaScript, i odpowiednio te dane przetworzy.

  `json`
  : jQuery przyjmie, że danymi wynikowymi są dane JSON, i przetworzy je przy użyciu dokładnego parsera.

  `jsonp`
  : Oznacza format JSONP, który zostanie przedstawiony w dalszej części książki.

Jako przykład zdefiniujemy proste żądanie Ajax, które wyświetli dane zwrócone przez serwer:

```
jQuery.ajax({
  url: "/ajax/endpoint",
  type: "GET",
  success: function(data) {
    alert(data);
  }
});
```

Jednak definiowanie wszystkich opcji jest dość uciążliwe. Na szczęście jQuery udostępnia kilka krótszych rozwiązań. Funkcja `jQuery.get()` pobiera adres URL, opcjonalne dane oraz wywołanie zwrotne:

```
jQuery.get("/ajax/endpoint", function(data){
  $(".ajaxResult").text(data);
});
```

Natomiast aby wysłać kilka parametrów w ramach żądania `GET`, można to zrobić następująco:

```
jQuery.get("/ajax/endpoint", {foo: "bar"}, function(data){
  /* ... */
});
```

Jeżeli spodziewamy się, że serwer zwróci dane JSON, trzeba użyć funkcji `jQuery.getJSON()`, która automatycznie ustawia opcję `dataType` żądania na `"json"`:

```
jQuery.getJSON("/json/endpoint", function(json){
  /* ... */
});
```

Dostępna jest analogiczna funkcja `jQuery.post()`, która również przyjmuje adres URL, dane i wywołanie zwrotne:

```
jQuery.post("/users", {first_name: "Alex"}, function(result){
  /* Żądanie POST Ajax zakończyło się powodzeniem */
});
```

Aby skorzystać z innych metod HTTP, czyli `DELETE`, `HEAD` i `OPTIONS`, trzeba użyć niskopoziomowej funkcji `jQuery.ajax()`.

W tym punkcie przedstawiono krótki opis API Ajax biblioteki jQuery. Więcej informacji na ten temat można znaleźć w kompletnej dokumentacji dostępnej na stronie pod adresem *http://api.jquery.com/category/ajax*.

Istotnym ograniczeniem Ajax jest obowiązująca w tej technologii **zasada tożsamego pochodzenia** (ang. *same origin policy*), która ogranicza możliwość obsługi tylko do żądań o tej samej domenie, subdomenie i tym samym porcie, co adres strony, która żądanie wykonuje. Jest po temu istotna przyczyna: otóż zawsze, gdy wysyłane jest żądanie Ajax, wraz z nim wysyłane są wszystkie dane cookie domeny. Dla serwera zdalnego jest to znak, że żądanie pochodzi od zalogowanego użytkownika. Gdyby nie obowiązywała zasada tożsamego pochodzenia, potencjalny napastnik mógłby pobrać wszystkie wiadomości pocztowe innej osoby z jej skrzynki Gmail, zmieniać statusy na czyimś profilu Facebooka albo publikować wiadomości w czyimś imieniu na Twitterze — jednym słowem, byłoby to istotne zagrożenie bezpieczeństwa.

Jednak zasada tożsamego pochodzenia, która jest istotnym elementem systemu bezpieczeństwa w sieci WWW, nakłada na programistów bardzo niewygodne ograniczenia, ponieważ odbiera im możliwość korzystania z w pełni bezpiecznych zasobów zdalnych. Inne technologie, na przykład Adobe Flash i Java, udostępniają obejścia tego problemu w postaci plików z regułami dostępu do innych domen. Do tego towarzystwa dołączył ostatnio Ajax wraz ze swoim standardem o nazwie CORS (*http://www.w3.org/TR/access-control*), wyznaczającym zasady współużytkowania zasobów między domenami.

Dzięki CORS można wyłamać się z zasady tożsamego pochodzenia i uzyskać dostęp do autoryzowanych, zdalnych serwerów. Specyfikacja standardu CORS została zaimplementowana w najważniejszych przeglądarkach, dlatego jeśli tylko nie jest używana przeglądarka IE6, nie powinno być z tym standardem większych problemów.

Zakres obsługi standardu CORS w poszczególnych przeglądarkach opisano poniżej:

- IE >= 8 (z pewnymi ograniczeniami).
- Firefox >= 3.
- Safari: pełna obsługa.
- Chrome: pełna obsługa.
- Opera: brak obsługi.

Korzystanie z CORS nie sprawia żadnych problemów. Aby autoryzować dostęp do własnego serwera, wystarczy do nagłówka zwracanych odpowiedzi dodawać te dwa wiersze:

```
Access-Control-Allow-Origin: przyklad.com
Access-Control-Request-Method: GET,POST
```

Powyższy nagłówek autoryzuje żądania `GET` i `POST` do innej domeny wykonywane z domeny *przyklad.com*. Gdy wartości jest więcej niż jedna, należy oddzielić je od siebie przecinkami, tak samo jak w przypadku metod `GET`, `POST` powyżej. Aby umożliwić dostęp z więcej niż jednej domeny, wystarczy zawrzeć ich listę oddzieloną od siebie przecinkami w nagłówku `Access-Control-Allow-Origin`. Natomiast aby zezwolić na dostęp z dowolnej domeny, należy w nagłówku `Origin` wpisać znak gwiazdki (`*`).

Niektóre przeglądarki, na przykład Safari, najpierw wykonują żądanie `OPTIONS`, aby sprawdzić, czy kolejne żądanie będzie dozwolone. Z kolei Firefox prześle od razu właściwe żądanie i dopiero w przypadku braku nagłówków CORS rzuci wyjątek. W trakcie implementowania kodu źródłowego trzeba wziąć pod uwagę te różnice w działaniu serwerów.

Standard CORS pozwala nawet na autoryzowanie żądań z niestandardowymi nagłówkami. Służy do tego nagłówek `Access-Control-Request-Headers`:

```
Access-Control-Request-Headers: Authorization
```

Tak skonstruowany nagłówek oznacza, że klienty mogą dodawać własne nagłówki do żądań Ajax i na przykład podpisywać żądania za pomocą OAuth:

```
var req = new XMLHttpRequest();
req.open("POST", "/endpoint", true);
req.setRequestHeader("Authorization", oauth_signature);
```

Niestety, mimo że standard CORS jest obsługiwany w Internet Explorerze 8 i jego nowszych wersjach, Microsoft zignorował specyfikację standardu i grupę roboczą (*http://lists.w3.org/Archives/Public/public-webapps/2008AprJun/0168.html*) i utworzył własny obiekt `XDomainRequest` (*http://msdn.microsoft.com/en-us/library/cc288060%28VS.85%29.aspx*), którym w przypadku żądań między domenami należy zastąpić obiekt `XMLHttpRequest`. Obiekt `XDomainRequest` ma interfejs podobny do interfejsu obiektu `XMLHttpRequest`, jednak zdefiniowano dla niego wiele dodatkowych ograniczeń i restrykcji (*http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx*). Na przykład obiekt obsługuje tylko metody `GET` i `POST`, nie są obsługiwane nagłówki uwierzytelniania ani nagłówki niestandardowe, a w końcu — i to jest najdziwniejsze — spośród typów zawartości obsługiwany jest wyłącznie `Content-Type: text/plain`. Zatem aby skorzystać z CORS w IE, trzeba — oprócz użycia prawidłowych nagłówków `Access-Control` — opracować odpowiednie obejścia dla wspomnianych przed chwilą ograniczeń.

JSONP

JSONP (*http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp*), albo inaczej JSON z wypełnianiem, to technologia opracowana przed zdefiniowaniem standardu CORS. JSONP stanowi kolejne narzędzie do pobierania danych ze zdalnych serwerów. Mechanizm działania JSONP polega na tym, że definiowany jest znacznik skryptu, który wskazuje na końcówkę JSONP. Tam właśnie zwracane dane są opakowywane w wywołanie funkcji. Znaczniki skryptów nie podlegają ograniczeniom związanym z komunikacją między domenami, a poza tym opisywana technika jest obsługiwana właściwie przez każdą przeglądarkę.

Poniżej przedstawiono znacznik skryptu, który wskazuje na zdalny serwer:

```
<script src="http://przyklad.com/dane.json"> </script>
```

Końcówka *dane.json* zwraca obiekt JSON opakowany w wywołanie funkcji:

```
jsonCallback({"data": "foo"})
```

Następnie definiuje się funkcję dostępną globalnie. Gdy tylko skrypt zostanie załadowany, nastąpi wywołanie tej funkcji:

```
window.jsonCallback = function(result){
  // Przetwarzanie danych wynikowych
}
```

Jak widać, proces jest dość zawiły. Na szczęście jQuery udostępnia o wiele bardziej zwięzłe API do niego:

```
jQuery.getJSON("http://przyklad.com/dane.json?callback=?", function(result){
  // Przetwarzanie danych wynikowych
});
```

Biblioteka jQuery zastąpi ostatni znak zapytania w powyższym adresie URL losową nazwą utworzonej tymczasowej funkcji. Serwer będzie musiał natomiast odczytywać parametr `callback` i używać go w roli nazwy zwracanej funkcji opakowującej.

Bezpieczeństwo żądań między domenami

Jeżeli posiadany serwer jest udostępniany dla żądań JSONP z różnych domen, trzeba dogłębnie przemyśleć kwestię jego zabezpieczenia. Zazwyczaj zasada współużytkowania zasobów między domenami uniemożliwia potencjalnemu napastnikowi na przykład wywołanie API Twittera i pobranie osobistych danych innych użytkowników. Wszystko zmienia się jednak w przypadku zastosowania CORS i JSONP. Podobnie jak w przypadku zwykłego żądania Ajax, wszystkie cookie sesji są przekazywane w ramach żądania, co oznacza, że zalogowanie do API Twittera już się dokonało. Potencjalny napastnik będzie więc mógł przejąć pełną kontrolę nad kontem, a to przesądza sprawę — bezpieczeństwo wymaga najwyższej uwagi.

Jeżeli zatem nie można z góry ograniczyć zbioru domen, które będą korzystać z API i planuje się użyć technologii CORS/JSONP, należy zastosować się do następujących wytycznych:

- Nie należy ujawniać żadnych poufnych danych, takich jak adresy poczty elektronicznej.
- Nie można zezwalać na wywołanie jakichkolwiek akcji (jak na przykład „follow" na Twitterze).

Aby zapewnić odpowiedni poziom bezpieczeństwa, można także z góry wyznaczyć określone domeny, które będą mogły korzystać z API, lub użyć uwierzytelnienia OAuth.

Wypełnienie ORM danymi

Wypełnienie biblioteki ORM odpowiednimi danymi jest bardzo prostym zadaniem. Wystarczy pobrać dane z serwera, po czym uaktualnić rekordy modelu. Dodamy zatem funkcję `populate()` do obiektu `Model`. Funkcja `populate()` będzie iterować przez kolejne przekazane do niej wartości oraz tworzyć instancje i uaktualniać obiekty `records`:

```
Model.extend({
  populate: function(values){
    // Reset modelu i obiektu records
    this.records = {};

    for (var i=0, il = values.length; i < il; i++) {
      var record = this.init(values[i]);
      record.newRecord = false;
      this.records[record.id] = record;
    }
  }
});
```

Teraz można już użyć funkcji `Model.populate()` wraz z wynikiem żądania, które pobiera dane:

```
jQuery.getJSON("/assets", function(result){
  Asset.populate(result);
});
```

W ten sposób wszystkie dane zwrócone przez serwer staną się dostępne w bibliotece ORM.

Przechowywanie danych lokalnie

W przeszłości przechowywanie danych na lokalnym komputerze było ogromnym wyzwaniem. Jedynym dostępnym rozwiązaniem było wykorzystanie plików cookie oraz modułów rozszerzających, takich jak Adobe Flash. Cookie mają przestarzałe API, nie mogą przechowywać zbyt dużej ilości danych, a wszystkie znajdujące się w nich dane są przesyłane do serwera wraz z każdym żądaniem, co niepotrzebnie zwiększa transfer. Jeśli zaś chodzi o Flash, wystarczy powiedzieć, że najlepiej trzymać się z dala od wszystkich modułów rozszerzających.

Na szczęście mechanizm przechowywania danych lokalnie zawarto w specyfikacji HTML5 i jest on obsługiwany przez większość współczesnych przeglądarek. W odróżnieniu od cookies dane są w tym przypadku przechowywane wyłącznie po stronie klienta i nigdy nie są przesyłane na serwery. Ilość danych, które można w ten sposób przechowywać, jest także nieporównanie większa, a maksymalny dozwolony wolumen zależy od przeglądarki (oraz jej wersji, zgodnie z listą poniżej); nigdy jednak wolumen ten nie jest mniejszy niż 5 MB na domenę:

- IE >= 8.
- Firefox >= 3.5.
- Safari >= 4.
- Chrome >= 4.
- Opera >= 10.6.

Mechanizm przechowywania danych został opisany w specyfikacji Web Storage HTML5 (*http://www.w3.org/TR/webstorage*) i składa się z dwóch elementów: **przechowywania lokalnie** (ang. *local storage*) oraz **przechowywania w sesji** (ang. *session storage*). Dane przechowywane

lokalnie zostają utrzymane również po zamknięciu przeglądarki, natomiast dane przechowywane w sesji są dostępne wyłącznie do czasu zamknięcia okna. Wszystkie dane są przechowywane w kontekście ich domeny i udostępniane wyłącznie tym skryptom, które pochodzą z domen stanowiących źródło tych danych.

Aby skorzystać z mechanizmów przechowywania danych lokalnie i przechowywania danych w sesji, należy użyć odpowiednio obiektu `localStorage` oraz `sessionStorage`. Odpowiednie API jest bardzo podobne do w działaniu do mechanizmu ustawiania właściwości na obiektach JavaScriptu i, jeśli nie liczyć różnic w nazwach obydwóch wspomnianych obiektów, jest identyczne dla obydwóch mechanizmów:

```
// Ustawienie wartości
localStorage["someData"] = "wem";
```

API WebStorage udostępnia jeszcze kilka innych funkcji:

```
// Liczba przechowywanych elementów danych
var itemsStored = localStorage.length;

// Ustawienie danych
localStorage.setItem("someData", "wem");

// Pobranie danych, zwrócenie null, jeśli dane są nieznane
localStorage.getItem("someData"); //=> "wem";

// Usunięcie danych, zwrócenie null, jeśli dane są nieznane
localStorage.removeItem("someData");

// Wyczyszczenie wszystkich elementów danych
localStorage.clear();
```

Dane są przechowywane w postaci ciągów znaków, dlatego jeśli konieczne będzie zapisanie obiektów albo liczb całkowitych, trzeba będzie samodzielnie dokonać odpowiednich przekształceń. Aby to zrobić przy użyciu JSON, należy zserializować obiekty do postaci JSON przed ich zapisaniem, natomiast w przypadku odczytywania obiektów — zdeserializować ciągi znaków JSON:

```
var object = {some: "object"};
// Serializacja i zapisanie obiektu
localStorage.setItem("seriData", JSON.stringify(object));
// Załadowanie i deserializacja obiektu
var result = JSON.parse(localStorage.getItem("seriData"));
```

Jeżeli wolumen danych przekroczy dozwolony rozmiar (zwykle jest to 5 MB na jeden serwer źródłowy), w momencie próby zapisania kolejnych danych zwrócony zostanie błąd `QUOTA_EXCEEDED_ERR`.

Dodanie mechanizmu przechowywania danych lokalnie do ORM

Dodajmy do biblioteki ORM opcję przechowywania danych na lokalnym komputerze, aby umożliwić utrzymanie rekordów między kolejnymi odświeżeniami strony. Aby skorzystać z obiektu localStorage, należy zserializować rekordy do postaci ciągu znaków JSON. Problem polega na tym, że obecnie zserializowane obiekty wyglądają następująco:

```
var json = JSON.stringify(Asset.init({name: "foo"}));
json //=> "{"parent":{"parent":{"prototype":{}},"records":[]},"name":"foo"}"
```

Trzeba więc pokryć mechanizm serializacji JSON dla naszych modeli. W tym celu najpierw ustalmy, które właściwości trzeba zserializować. Do obiektu `Model` dodamy tablicę `attributes`, w której każdy z modeli będzie wskazywał odpowiednie atrybuty przeznaczone do serializacji:

```
Model.extend({
  created: function(){
    this.records = {};
    this.attributes = [];
  }
});

Asset.attributes = ["name", "ext"];
```

Ponieważ każdy model ma odmienne atrybuty, przez co modele nie mogą współużytkować tej samej tablicy `attributes`, właściwości tej nie ustawia się bezpośrednio na obiekcie `Model`. Zamiast tego nowa tablica jest tworzona w momencie tworzenia modelu — analogicznie jak w przypadku obiektu `records`.

Utworzymy teraz funkcję `attributes()`, która zwróci obiekt dla atrybutów i wartości:

```
Model.include({
  attributes: function(){
    var result = {};
    for(var i in this.parent.attributes) {
      var attr = this.parent.attributes[i];
      result[attr] = this[attr];
    }
    result.id = this.id;
    return result;
  }
});
```

Dla każdego modelu można już zdefiniować tablicę atrybutów:

```
Asset.attributes = ["name", "ext"];
```

Wówczas funkcja `attributes()` będzie zwracać obiekt z odpowiednimi właściwościami:

```
var asset = Asset.init({name: "document", ext: ".txt"});
asset.attributes(); //=> {name: "document", ext: ".txt"};
```

Aby pokryć funkcję `JSON.stringify()`, wystarczy wykonać metodę `toJSON()` na instancjach modeli. Biblioteka JSON wyszuka za pomocą tej funkcji obiekt przeznaczony do zserializowania, zamiast zserializować obiekt `records` w jego obecnej postaci:

```
Model.include({
  toJSON: function(){
    return(this.attributes());
  }
});
```

Spróbujmy teraz ponownie zserializować rekordy. Tym razem wynikowy ciąg znaków JSON będzie zawierał prawidłowe właściwości:

```
var json = JSON.stringify(Asset.records);
json //= "{"7B2A9E8D...":"{"name":"document","ext":".txt","id":"7B2A9E8D..."}"}"
```

Skoro serializacja JSON działa bez zarzutu, dodanie obsługi przechowywania danych lokalnie do naszego modelu będzie trywialne. Do obiektu `Model` dodamy w tym celu dwie funkcje: `saveLocal()` oraz `loadLocal()`. W trakcie zapisywania obiekt `Model.records` zostanie przekształcony w tablicę, zserializowany i wysłany do `localStorage`:

```
var Model.LocalStorage = {
  saveLocal: function(name){
    // Przekształcenie records w tablicę
    var result = [];
```

```
    for (var i in this.records)
      result.push(this.records[i])

    localStorage[name] = JSON.stringify(result);
  },

  loadLocal: function(name){
    var result = JSON.parse(localStorage[name]);
    this.populate(result);
  }
};

Asset.extend(Model.LocalStorage);
```

Zapewne dobrym pomysłem będzie odczytywanie rekordów z lokalnego repozytorium w trakcie ładowania strony i ich zapisywanie, gdy strona będzie zamykana. Zostawiam to jednak jako ćwiczenie dla Czytelników.

Przesyłanie nowych rekordów na serwer

Nieco wcześniej w tej książce pokazano, jak wykorzystywać funkcję `post()` biblioteki jQuery, aby przesyłać dane do serwera. Funkcja przyjmuje trzy argumenty: docelowy adres URL, dane żądania i wywołanie zwrotne:

```
jQuery.post("/users", {first_name: "Alex"}, function(result){
  /* Żądanie POST Ajax zakończyło sie powodzeniem */
});
```

Dzięki wcześniejszemu zaimplementowaniu funkcji `attributes()` tworzenie rekordów na serwerze jest już proste — wystarczy metodą `POST` przesłać atrybuty rekordu:

```
jQuery.post("/assets", asset.attributes(), function(result){
  /* Żądanie POST Ajax zakończyło sie powodzeniem */
});
```

Aby konsekwentnie trzymać się konwencji REST, w momencie tworzenia rekordu należy wykonywać żądanie HTTP `POST`, zaś w chwili zmiany tego rekordu — żądanie HTTP `PUT`. Dodajmy zatem do instancji `Model` dwie funkcje o nazwach `createRemote()` i `updateRemote()`, które będą odpowiadać za wysłanie do serwera żądania HTTP odpowiedniego rodzaju:

```
Model.include({
  createRemote: function(url, callback){
    $.post(url, this.attributes(), callback);
  },

  updateRemote: function(url, callback){
    $.ajax({
      url:     url,
      data:    this.attributes(),
      success: callback,
      type:    "PUT"
    });
  }
});
```

Teraz, jeśli na instancji zasobu `Asset` zostaje wywołana funkcja `createRemote()`, atrybuty instancji są przesyłane do serwera metodą `POST`:

```
// Sposób użycia:
Asset.init({name: "jason.txt"}).createRemote("/assets");
```

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

X

Y

Z

Ż

Programowanie

O'REILLY®

JavaScript. Aplikacje WWW

Język JavaScript od zawsze jest obecny w przeglądarkach internetowych. W swojej historii miewał okresy lepsze i gorsze – czasem bywał wręcz znienawidzony przez użytkowników. Te czasy jednak minęły! Obecnie jego możliwości są wprost oszałamiające. Trudno wyobrazić sobie współczesną popularną aplikację internetową, która nie korzystałaby z jego dobrodziejstw. W najnowszej odsłonie HTML5 jego potencjał jest jeszcze większy!

Sprawdź sam, jak wykorzystać JavaScript do tworzenia rozwiązań, które zaskoczą użytkowników. Budowanie w tym języku dużych internetowych aplikacji, które zaoferują użytkownikom funkcje obecne dotąd wyłącznie w programach biurkowych, wymaga utrzymywania stanu aplikacji po stronie klienta – a to nie jest łatwe zadanie. Dzięki tej książce poznasz szczegółowy opis czynności, jakie trzeba wykonać, aby zaimplementować nowoczesną aplikację, a potem opanujesz skuteczne metody korzystania z mechanizmu WebSockets, operowania na plikach oraz modelowania danych. Ponadto zgłębisz niuanse nowego API oraz dowiesz się, jakie są najlepsze techniki debugowania i analizowania wydajności Twojej aplikacji. Ta książka to długo oczekiwana pozycja, w całości poświęcona zaawansowanemu wykorzystaniu języka JavaScript!

Z tą książką błyskawicznie opanujesz:

- korzystanie z wzorca MVC
- obsługę zdarzeń
- modelowanie danych
- zarządzanie zależnościami
- wykorzystywanie zewnętrznych bibliotek
- techniki debugowania i optymalizowania Twojego oprogramowania
- najlepsze funkcje JavaScriptu!

Twórz nowoczesne aplikacje przy użyciu najlepszych dostępnych narzędzi!

helion.pl
księgarnia internetowa

Nr katalogowy: 8733

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-3887-1

Cena 39,90 zł