Buduj zaawansowane i interaktywne strony WWW!

JavaScript i jQuery

nieoficjalny podręcznik

David Sawyer McFarland

O'REILLY®

Helion®

Tytuł oryginału: JavaScript & jQuery: The Missing Manual, Third Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-0547-2

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/jsjqn3.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jsjqn3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

14

ROZDZIAŁ

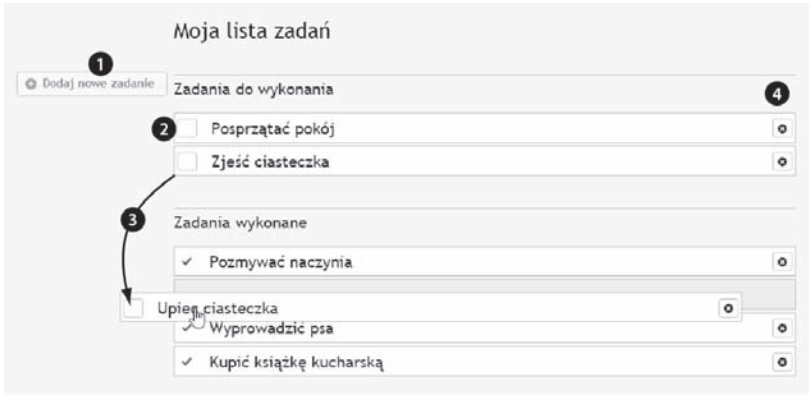
Tworzenie aplikacji do obsługi listy zadań

Biblioteki jQuery i jQuery UI udostępniają narzędzia do tworzenia zaledwie w kilku prostych krokach profesjonalnie wyglądających aplikacji internetowych. Pierwsza z nich może zadbać o wybieranie elementów strony, dodawanie do niej nowych elementów oraz aktualizowanie DOM. Z kolei biblioteka jQuery UI udostępnia wspaniale wyglądające widżety, sposoby interakcji z użytkownikiem oraz efekty animacji, rozwiązując tym samym wiele najczęściej występujących problemów związanych z opracowaniem interfejsu użytkownika. Korzystając z obu tych bibliotek, można uniknąć żmudnych oraz czasochłonnych zadań i skoncentrować się wyłącznie na tworzeniu dynamicznej, interaktywnej aplikacji. W tym rozdziale przedstawiony został proces pisania prostej aplikacji do zarządzania listą zadań.

Przegląd aplikacji

Lista zadań, którą napiszemy w tym rozdziale, będzie pozwalała użytkownikom na wykonywanie następujących operacji.

- **Dodawanie nowych zadań do listy.** W tym celu dodamy do niej widżet przycisku jQuery UI (nr 1 na rysunku 14.1), a następnie po kliknięciu wyświetlimy okno dialogowe jQuery UI.
- **Oznaczanie zadania jako wykonanego.** Każde zadanie na liście będzie mieć pole wyboru z lewej strony (numer 2 na rysunku 14.1). Kliknięcie tego pola będzie automatycznie usuwać zadanie z listy *Zadania do wykonania* i przenosić na listę *Zadania wykonane*.
- **Przeciąganie zadań z listy *Zadania do wykonania* na listę *Zadania wykonane* i na odwrót** (numer 3 na rysunku 14.1)**.** Choć klikanie pola wyboru niewątpliwie spełni swoją rolę, to niby czemu mamy się ograniczać do tylko jednego sposobu oznaczania zadań jako wykonanych? Korzystając z widżetu



Rysunek 14.1. *Biblioteki jQuery oraz jQuery UI sprawiają, że napisanie prostej listy zadań do zrobienia jest relatywnie łatwe. Pozwalają one na rozwiązywanie złożonych problemów związanych z tworzeniem interfejsów użytkownika, takich jak opracowanie okien dialogowych, list z możliwością sortowania oraz animowanie wybranych elementów stron. Dzięki nim będziemy mogli skoncentrować się wyłącznie na zaimplementowaniu możliwości funkcjonalnych aplikacji*

Sortable jQuery UI, możemy także pozwolić użytkownikom na oznaczanie zdań jako wykonanych poprzez przeciąganie ich na listę *Zadania wykonane*. Co więcej, możemy także pozwolić na przeciąganie zadań *z powrotem* na listę *Zadania do wykonania*.

- **Usuwanie zadań z listy po kliknięciu przycisku *Usuń*** (numer 4 na rysunku 14.1)**.** Aby zapewnić użytkownikom możliwość całkowitego usunięcia zadania, dodamy przycisk *Usuń*, którego kliknięcie całkowicie usunie zadanie ze strony. Zastosujemy przy tym jeden z efektów jQuery UI, by cała operacja wyglądała interesująco.

Ponieważ biblioteka jQuery UI udostępnia większość widżetów i mechanizmów interakcji, pozostaje jedynie zaimplementowanie podstawowej logiki działania aplikacji. Zadanie to wykonamy krok po kroku, zgodnie z przedstawionym powyżej planem. Zaczniemy od dodania przycisku, a następnie dokończymy pozostałą część aplikacji. Zadanie programistyczne podczas rozwiązywania warto podzielić na mniejsze elementy, które można łatwo przetestować. Podobnie będzie w tym przykładzie — najpierw zrobimy jedną rzecz, upewnimy się, że działa prawidłowo, i dopiero potem przejdziemy do kolejnej.

Dodanie przycisku

Najpierw dodamy do strony przycisk i sformatujemy go przy użyciu jQuery UI. W tym przykładzie będziemy pracować nad dwoma plikami dostępnymi w katalogu *R14*; są to *index.html* oraz *todo.js*. Cały kod JavaScript będzie umieszczany w pliku *todo.js*, natomiast kod HTML niezbędny do utworzenia przycisku oraz okna dialogowego trafi do pliku *index.html*.

Uwaga: Informacje na temat pobierania przykładów do książki można znaleźć na stronie 46.

1. **Otwórz w edytorze tekstu plik *index.html* dostępny w katalogu *R14*.**

   Plik zawiera już odwołania do potrzebnych plików CSS, jQuery oraz jQuery UI. Jednak nie ma w nim jeszcze odwołania do skryptu *todo.js* — czyli pliku, w którym zapiszesz kod JavaScript tworzonej aplikacji.

2. **W wierszu bezpośrednio poniżej `<script src="js/jqeury-ui.min.js">` `</script>` dodaj następujący wiersz kodu :**

   ```
   <script src="todo.js"></script>
   ```

   Bardzo ważne jest to, by plik *todo.js* został dołączony do strony jako ostatni, gdyż wymaga on zarówno biblioteki jQuery, jak i jQuery UI. Jeśli wczytasz go przed którąkolwiek z tych bibliotek, podczas wyświetlania strony przeglądarka zgłosi błąd syntaktyczny.

   Teraz zajmiesz się dodaniem przycisku. Użytkownik będzie mógł go kliknąć w celu dodania do listy nowego zadania.

3. **W treści pliku odszukaj komentarz `<!-- Tu dodaj przycisk. -->` i zastąp go poniższym wierszem kodu HTML:**

   ```
   <button id="add-todo">Dodaj nowe zadanie</button>
   ```

   W tym kodzie nie ma nic szczególnego — dodajesz do strony zwyczajny element `<button>`. Gdybyś teraz wyświetlił stronę w przeglądarce, przekonałbyś się, że wygląda on jak zwyczajny przycisk do wysyłania formularza — bezbarwny i nieinteresujący. Już zaraz przekształcisz go w przycisk jQuery UI.

4. **W edytorze tekstów otwórz plik *todo.js*, a następnie wewnątrz funkcji `$(document).ready(function(e) {` wpisz następujący wiersz kodu:**

   ```
   $("#add-todo").button();
   ```

   Powyższy kod spowoduje zastosowanie w przycisku formatowania określonego przez aktualnie używamy temat graficzny jQuery UI (patrz strona 407). Możesz go dodatkowo uatrakcyjnić, dodając do niego ikonę jQuery UI.

5. **Wewnątrz funkcji `button()` dodaj literał obiektowy definiujący ikonę, która będzie wyświetlona na przycisku (kod literału został wyróżniony pogrubioną czcionką):**

   ```
   $("#add-todo").button({
     icons: {
       primary: "ui-icon-circle-plus"
     }
   });
   ```

   Powyższy kod umieści z lewej strony napisu widocznego na przycisku niewielką ikonę ze znakiem „+”. Zgodnie z informacjami podanymi na stronie 390, jQuery UI pozwala na wyświetlanie na przyciskach dwóch ikon: jednej po lewej stronie (ikona główna — `primary`), a drugiej po prawej (ikona pomocnicza — `secondary`). W przypadku tego przycisku jedna ikona w zupełności wystarczy.

6. **Zapisz oba pliki, *todo.js* oraz *index.html*, a następnie wyświetl stronę *index.html* w przeglądarce.**

   Teraz przycisk powinien już wyglądać tak samo jak widżet przycisku jQuery UI widoczny na rysunku 14.1. Jeśli tak nie wygląda, oznacza to, że przygotowany kod JavaScript nie działa. Dokładnie go sprawdź, używając konsoli JavaScript, by upewnić się, czy nie ma komunikatów o błędach.

Dodanie okna dialogowego

A zatem mamy już przycisk, który jednak nic nie robi. Docelowo kliknięcie tego przycisku powinno powodować wyświetlenie okna dialogowego — jego zaimplementowanie będzie Twoim kolejnym zadaniem. Najpierw dodasz kod HTML okna dialogowego.

1. **W pliku *index.html* odszukaj komentarz `<!-- Tutaj dodaj okienko dialogowe. -->` i zastąp go poniższym kodem HTML:**

```
<div id="new-todo" title="Dodaj zadanie">
  <form>
    <p>
      <label for="task">Nazwa zadania:</label>
      <input type="text" name="task" id="task">
    </p>
  </form>
</div>
```

   Zgodnie z tym, czego dowiedziałeś się na stronie 331, w okno dialogowe możesz zamienić dowolny fragment kodu HTML. Tu użyjemy elementu `<div>`, wewnątrz którego znajduje się formularz z jednym polem tekstowym. Podczas dodawania zadania użytkownicy będą wpisywali w tym polu nazwę zadania.

   Aby przekształcić ten kod HTML w okno dialogowe, będziesz musiał użyć kodu JavaScript.

2. **Wróć do pliku *todo.js*. Poniżej kodu, który dodałeś w kroku 5. na stronie 521, wpisz:**

```
$("#new-todo").dialog();
```

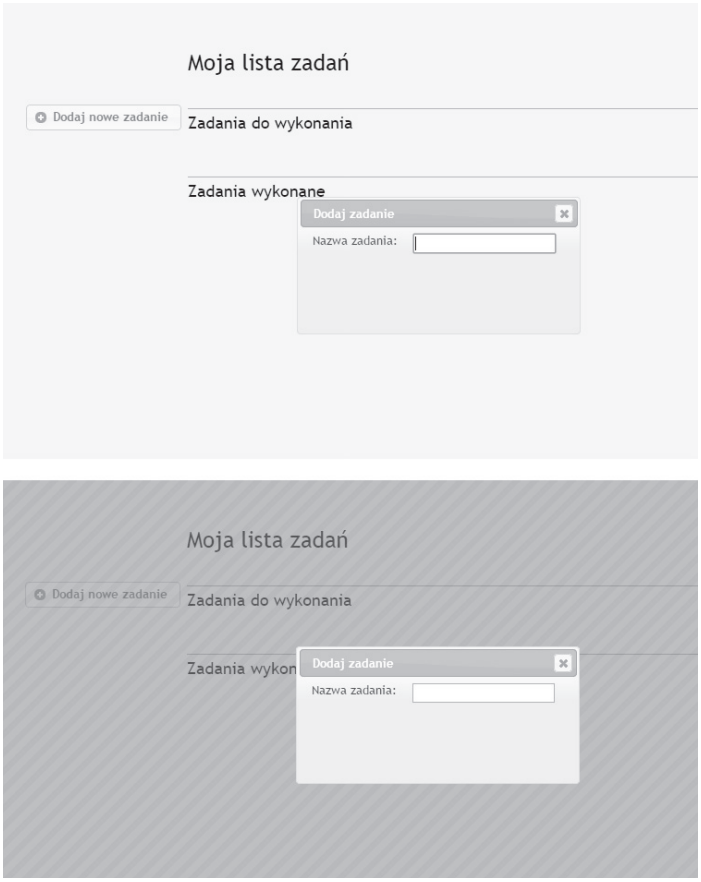
   Zapisz plik *index.html* i odśwież stronę w przeglądarce. Powinno się na niej pojawić okno dialogowe (patrz górny zrzut na rysunku 14.2). Jednak jest ono wyświetlane od razu, a nie po kliknięciu przycisku. To standardowe zachowanie okien dialogowych tworzonych przy użyciu biblioteki jQuery UI. Aby ukryć to okno, będziesz musiał przekazać w wywołaniu funkcji `dialog()` odpowiednie opcje.

3. **W wywołaniu funkcji `dialog()` umieść obiekt zawierający dwie właściwości:**

```
$("#new-todo").dialog({
  modal : true,
  autoOpen : false
});
```

   Zgodnie z tym, czego się dowiedziałeś na stronie 335, opcja `modal` zmusza użytkownika do zamknięcia okna dialogowego, zanim będzie mógł wykonać na stronie jakąkolwiek inną operację. I właśnie o to chodzi — kiedy użytkownik kliknie przycisk *Dodaj nowe zadanie*, wypełnienie okna dialogowego powinno być jedyną rzeczą, na której użytkownik ma się skoncentrować.

   Z kolei przypisanie właściwości `autoOpen` wartości `false` sprawi, że okno dialogowe nie będzie już wyświetlane natychmiast po wczytaniu strony. Teraz będzie początkowo ukryte i trzeba wyświetlić je programowo, w odpowiedzi na kliknięcie przycisku!



Rysunek 14.2. *Okna dialogowe jQuery UI są automatycznie wyświetlane zaraz po wczytaniu strony. To doskonałe rozwiązanie w przypadku prezentowania ważnych informacji, które użytkownicy muszą zobaczyć bezpośrednio po wejściu na stronę. Jednak w większości innych sytuacji, takich jak dodawanie elementów do listy zadań, nie jest już takie dobre. Zwykle będziesz chciał, by okno dialogowe było ukryte, aż do momentu wykonania jakiejś akcji przez użytkownika — na przykład takiej jak kliknięcie przycisku „Dodaj nowe zadanie"*

4. **Dodaj funkcję obsługującą zdarzenia `click`, dopisując do wywołania funkcji `.button()` wywołanie `.click()`:**

```
$("#add-todo").button({
  icons: {
    primary: "ui-icon-circle-plus"
  }
}).click(function() {
  $('#new-todo').dialog('open');
});
```

Wywołania funkcji jQuery można łączyć w sekwencje — w tym celu na końcu jednej z nich wystarczy dopisać kropkę, a za nią umieścić wywołanie kolejnej funkcji. W tym przypadku kod najpierw wybiera element o identyfikatorze `add-todo` (czyli przycisk), potem wywołuje funkcję jQuery UI `button()` i w końcu dodaje do przycisku funkcję, która będzie obsługiwać zdarzenia `click`. Dodana funkcja obsługująca zdarzenia wywołuje z kolei funkcję `open()` okna dialogowego (patrz strona 338). Innymi słowy, kliknięcie przycisku powinno teraz powodować wyświetlenie okna dialogowego.

Nadszedł czas, żeby przetestować działanie kodu.

5. **Zapisz plik *todo.js*, po czym wyświetl w przeglądarce stronę *index.html*. Kliknij przycisk *Dodaj nowe zadanie*.**

   Teraz okno dialogowe powinno być wyświetlane wyłącznie po kliknięciu przycisku (u dołu rysunku 14.2). Co więcej, cały obszar poniżej okna dialogowego powinien zostać nieco zaciemniony — przykryty przez prążkowaną, półprzezroczystą warstwę. To wizualne oznaczenie informuje użytkownika, że zostało wyświetlone *modalne* okno dialogowe, co z kolei znaczy, że dopóki go nie zamknie, nie będzie mógł zrobić nic innego. Jednak aktualnie nie ma jak zamknąć tego okna dialogowego!

   Na szczęście za pomocą jQuery UI dodawanie przycisków do okien dialogowych jest bardzo proste.

6. **Wróć do pliku *todo.js*. Do listy opcji okna dialogowego dodaj kolejną — `buttons`, jej wartością ma być pusty literał obiektowy:**

```
$("#new-todo").dialog({
  modal : true,
  autoOpen : false,
  buttons : {

  }
});
```

   Opcja `buttons` pozwala określać przyciski, które jQuery UI dynamicznie doda do okna dialogowego. Oprócz tego, do każdego z nich można dodać funkcję, dzięki czemu coś się stanie, kiedy użytkownik kliknie dany przycisk. Tworzymy kod aplikacji wolno, krok po kroku, ponieważ dzieje się w nim całkiem dużo. Teraz na przykład dysponujesz nowym obiektem (właściwością `buttons`) umieszczonym wewnątrz innego obiektu (literału obiektowego z opcjami, przekazywanego w wywołaniu funkcji `dialog()`).

   Najpierw dodaj przycisk z pustą funkcją.

7. **Do literału obiektowego wpisanego w poprzednim kroku dodaj nową właściwość (wyróżnioną pogrubioną czcionką):**

```
$("#new-todo").dialog({
  modal : true,
  autoOpen : false,
  buttons : {
    "Dodaj zadanie" : function () {

    }
  }
});
```

   Ten kod powoduje dodanie do okna dialogowego przycisku z napisem *Dodaj zadanie*. Kiedy użytkownik kliknie ten przycisk, zostanie wykonana funkcja. Obecnie funkcja jest pusta, więc nic się nie stanie. Funkcję tę zaimplementujesz do końca w następnej części przykładu. A teraz dodasz do okna dialogowego drugi przycisk.

8. **Za zamykającym nawiasem klamrowym funkcji umieszczonej we właściwości `"Dodaj zadanie"` wpisz przecinek i dodaj kolejną właściwość z funkcją anonimową:**

```
$("#new-todo").dialog({
  modal : true,
  autoOpen : false,
```

```
      buttons : {
        "Dodaj zadanie" : function () {

        },
        "Anuluj" : function () {
          $(this).dialog('close');
        }
      }
    });
```

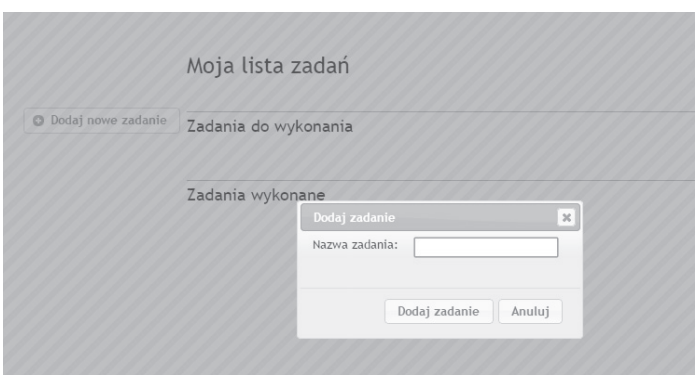
Ten nowy fragment kodu dodaje do okna dialogowego przycisk *Anuluj*. Co więcej, przycisk już coś robi. Wyrażenie `$(this)` odwołuje się do elementu, który wywołał funkcję, czyli do samego okna dialogowego. Kiedy użytkownik kliknie przycisk *Anuluj*, zostaje wywołana funkcja `close()` widżetu okna dialogowego (patrz strona 340). Powoduje ona natychmiastowe zamknięcie tego okna.

Nadszedł czas, by zobaczyć efekt wykonanej pracy.

9. **Zapisz plik *todo.js* i odśwież stronę index.html w przeglądarce. Kliknij przycisk *Dodaj nowe zadanie*.**

   Okno dialogowe powinno teraz zawierać dwa przyciski (patrz rysunek 14.3). Kliknij przycisk *Dodaj zadanie* — oczywiście nic się nie powinno stać, gdyż obsługa przycisku nie została jeszcze zaimplementowana. Kiedy jednak klikniesz drugi przycisk, czyli *Anuluj*, okno powinno zostać zamknięte.

   W następnej części przykładu zajmiesz się pisaniem kodu, który pozwoli na dodawanie zadań do listy.



Rysunek 14.3. *Wystarczyło kilka wierszy kodu, aby przygotować wszystkie komponenty prostego interfejsu użytkownika aplikacji do zarządzania listą zadań*

Dodawanie zadań

Nasza aplikacja do zarządzania listą zadań wygląda już całkiem dobrze, jednak nie dzieje się w niej zbyt dużo. Nadszedł zatem czas, byś zajął się kodem umożliwiającym dodawanie zadań. Zostanie on umieszczony w funkcji skojarzonej z przyciskiem *Dodaj zadanie*, czyli w pustej funkcji anonimowej, którą dopisałeś w kroku 5. na stronie 524. Implementację tego kodu wykonasz w opisanych poniżej czterech krokach.

1. **Pobierz nazwę zadania, którą użytkownik wpisał w polu tekstowym w oknie dialogowym.**

   Wartość ta jest przechowywana w polu tekstowym, a to oznacza, że możesz ją łatwo pobrać, używając funkcji `val()` jQuery (patrz strona 283).

2. **Utwórz element `<li>`, który następnie dodasz do treści strony.**

   Każde zadanie będzie reprezentowane przez jeden element listy wypunktowanej. Poniżej przedstawiona została struktura kodu HTML takiego elementu:

   ```
   <li>
     <span class="done">%</span>
     <span class="delete">x</span>
     <span class="task">Upiec ciasteczka</span>
   </li>
   ```

   Pierwszy element `<span>` reprezentuje obszar, który użytkownik może kliknąć, by oznaczyć zadanie jako wykonane. Drugi element `<span>` reprezentuje przycisk do usunięcie zadania. I w końcu ostatni element `<span>` zawiera nazwę zadania podaną przez użytkownika w oknie dialogowym. Taki kod HTML możesz skonstruować, łącząc ze sobą literały łańcuchowe zawierające odpowiednie znaczniki i dodając do nich wpisaną przez użytkownika nazwę zadania.

Uwaga: Postać obu przycisków określisz, stosując odpowiednie style CSS. Dzięki temu znak procenta (`%`) oraz `x` umieszczone w kodzie po wyświetleniu strony zostaną zamienione odpowiednio na znacznik charakterystyczny dla pól wyboru oraz przycisk z ikoną krzyżyka. Tajemnica tych przycisków tkwi w zastosowaniu specjalnej czcionki, w której zamiast liter są ikony.

---

3. **Dodaj do listy zadań do wykonania nowy element listy, znacznik `<li>`.**

   Biblioteka jQuery sprawia, że bardzo łatwo można zamienić łańcuch znaków na element DOM, a następnie dodać go do strony (szczegółowe informacje na ten temat można znaleźć w rozdziale 4.). Ponieważ dysponujemy potężnym zestawem efektów jQuery UI, możesz skorzystać z nich, by dodawać nowe zadania w sposób atrakcyjny wizualnie.

4. **Zamknij okno dialogowe.**

   Ta operacja jest bardzo prosta. Wykonałeś ją już wcześniej, przy okazji implementowania przycisku do zamykania okna dialogowego, w kroku 8. opisanym na stronie 524.

Skoro już znasz podstawowe kroki, jakie należy zaimplementować, czas zabrać się za pisanie kodu. Najpierw pobierzesz dane wpisane przez użytkownika, czyli nazwę zadania wpisaną w oknie dialogowym.

1. **Odszukaj funkcję skojarzoną z przyciskiem *Dodaj zadanie* okna dialogowego (została ona podana we właściwości `buttons` literału obiektowego przekazywanego w wywołaniu funkcji `dialog()`), a następnie wpisz w niej poniższy kod:**

   ```
   "Dodaj zadanie" : function () {
     var taskName = $('#task').val();
   },
   ```

   Powyższa instrukcja tworzy nową zmienną, `taksName`, i zapisuje w niej wartość pola tekstowego umieszczonego w oknie dialogowym. Pamiętasz zapewne strukturę kodu tworzącego okno dialogowe, została ona przedstawiona w kroku 1.

na stronie 522. Kod ten zawiera między innymi pole tekstowe o określonym identyfikatorze. A zatem wywołanie `$('#task').val()` pobiera wartość tego pola — czyli to, co użytkownik wpisał jako nazwę zadania.

Istnieje także możliwość, że użytkownik kliknie przycisk *Dodaj zadanie*, mimo że pole tekstowe będzie puste. Ponieważ nie chcemy dodawać do naszej listy pustych zadań, zatem powinieneś się upewnić, że wartość zmiennej `taskName` będzie inna od pustego łańcucha znaków.

2. **Poniżej wiersza kodu dodanego w poprzednim kroku wpisz trzy kolejne (wyróżnione pogrubioną czcionką):**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
    return false;
  }
},
```

Ten kod jedynie upewnia się, że zmienna `taskName` zawiera coś innego niż pusty łańcuch znaków (został on zapisany przy użyciu dwóch znaków apostrofu, z których pierwszy oznacza początek łańcucha, a drugi jego koniec), a użytkownik nie kliknął przycisku *Dodaj zadanie* bez podania tytułu zadania. Instrukcja `return false` powoduje zakończenie funkcji, co z kolei sprawia, że okno dialogowe pozostanie widoczne. A zatem użytkownik musi wpisać nazwę zadania lub zamknąć okno dialogowe (co może także zrobić, klikając niewielki przycisk „X" widoczny w jego prawym, górnym rogu).

Teraz zajmiesz się tworzeniem kodu HTML reprezentującego zadanie.

3. **Dodaj kolejną zmienną, a następnie skonstruuj i zapisz w niej łańcuch znaków; do tego celu posłużą trzy kolejne wiersze kodu (wyróżnione pogrubioną czcionką):**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
    return false;
  }
  var taskHTML = '<li><span class="done">%</span>';
  taskHTML += '<span class="delete">x</span>';
  taskHTML += '<span class="task"></span></li>';
},
```

Do utworzenia zmiennej `taskHTML` i zapisania w niej całego, długiego łańcucha znaków wystarczyłby jeden wiersz kodu. Jednak w ten sposób powstałaby instrukcja, którą trudno przeanalizować. Podzielenie długiego łańcucha znaków na części i zapisanie go w kilku kolejnych wierszach pozwala poprawić czytelność kodu. Operator += służy do dołączenia podanego łańcucha znaków do łańcucha, który już jest zapisany w zmiennej (patrz strona 72).

Zauważ, że jeszcze nie dodałeś do tego łańcucha nazwy zadania podanej przez użytkownika i przechowywanej w zmiennej `taskName`. Zrobisz to już za chwilę.

4. **Dodaj kolejną zmienną i zapisz w niej obiekt jQuery zawierający obiekt utworzony na podstawie kodu HTML (nowy kod został wyróżniony pogrubioną czcionką):**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
```

```
      return false;
    }
    var taskHTML = '<li><span class="done">%</span>';
    taskHTML += '<span class="delete">x</span>';
    taskHTML += '<span class="task"></span></li>';
    var $newTask = $(taskHTML);
  },
```

Biblioteka jQuery umożliwia przekształcenie łańcucha znaków, takiego jak `"<h1>To jest nagłówek</h1>"`, na element DOM. Innymi słowy, `taskHTML` jest zmienną zawierającą łańcuch znaków. Łańcuch ten nie jest jednak „prawdziwym" kodem HTML; natomiast przekazanie łańcucha zawierającego znaczniki HTML w wywołaniu funkcji `$()` spowoduje przekształcenie go na element DOM. A nawet jeszcze lepiej — funkcja `$()` przekształca łańcuch znaków na obiekt jQuery, dzięki czemu można na nim wywoływać standardowe funkcje tej biblioteki. Choć znak `$` na początku zmiennej `$newTask` nie jest konieczny, jednak umieszczanie go na początku nazw zmiennych stanowi praktykę powszechnie stosowaną przez programistów używających jQuery. Ten znak dolara stanowi wizualną podpowiedź informującą, że dana zmienna zawiera obiekt jQuery i można jej używać do wywoływania wszystkich metod biblioteki, takich jak `.show()` czy też `.addClass()`.

W następnym kroku dodasz nazwę zadania podaną przez użytkownika.

5. **Do kodu funkcji obsługującej przycisk *Dodaj zadanie* dopisz kolejny wiersz:**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
    return false;
  }
  var taskHTML = '<li><span class="done">%</span>';
  taskHTML += '<span class="delete">x</span>';
  taskHTML += '<span class="task"></span></li>';
  var $newTask = $(taskHTML);
  $newTask.find('.task').text(taskName);
},
```

Metoda `find()` poszukuje elementu odpowiadającego podanemu selektorowi *wewnątrz* aktualnie wybranego elementu (patrz strona 555). W tym przypadku metoda operuje na elemencie `<li>` przechowywanym w zmiennej `$newTask` i poszukuje w nim innego elementu, należącego do klasy `task` — czyli elementu `<span>`, w którym powinna zostać zapisana nazwa zadania (patrz krok 3.). Następnie wywołanie funkcji `text()` zapisuje zawartość zmiennej `taskName` w odnalezionym elemencie.

Ale dlaczego masz zadawać sobie tyle trudu, zamiast po prostu dodać wartość zmiennej `taskName` do łańcucha znaków tworzonego w kroku 4. w następujący sposób:

```
taskHTML += '<span class="task">' + taskName + '</span></li>';
```

Gdybyś postąpił w ten sposób, złośliwy użytkownik mógłby utworzyć zadanie o następującej nazwie: `<script>alert('ha, ha, ha... włamałem się do tej listy');</script>`. Ten złośliwy kod zostałby dodany bezpośrednio do strony i wykonany. Natomiast funkcja `text()` jQuery zamienia wszystkie znaczniki HTML na ich bezpieczne odpowiedniki, czyli zamienia `<script>` na `<script>`.

Jeśli nawet użytkownik nie ma żadnych złych intencji, takie rozwiązanie pozwoli mu wpisać w nazwie zadania całkowicie prawidłowy tekst, taki jak „Dodać do strony głównej znacznik <h1>", bez spowodowania awarii aplikacji.

W końcu nadszedł czas, by umieścić nowe zadanie na stronie.

6. **Do kodu funkcji dodaj kolejne dwa wiersze (wyróżnione pogrubioną czcionką):**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
    return false;
  }
  var taskHTML = '<li><span class="done">%</span>';
  taskHTML += '<span class="delete">x</span>';
  taskHTML += '<span class="task"></span></li>';
  var $newTask = $(taskHTML);
  $newTask.find('.task').text(taskName);
  $newTask.hide();
  $('#todo-list').prepend($newTask);
},
```

Ponieważ `$newTask` jest obiektem jQuery, można wywoływać na jego rzecz funkcje biblioteki. W dodanym fragmencie kodu najpierw ukrywasz nowe zadanie, dzięki czemu później będziesz mógł je wyświetlić, używając efektu animacji. Po ukryciu elementu kolejna instrukcja najpierw wybiera element listy — jest to lista wypunktowana o identyfikatorze `todo-list` — a następnie dodaje na jej początku nowy (wciąż ukryty) element (więcej informacji na temat działania metody `prepend()` można znaleźć na stronie 159).

W kolejnym kroku wyświetlisz nowy element listy i ukryjesz okno dialogowe.

7. **Do kodu funkcji dodaj kolejne dwa wiersze (zmiany zostały wyróżnione pogrubioną czcionką):**

```
"Dodaj zadanie" : function () {
  var taskName = $('#task').val();
  if (taskName === '') {
    return false;
  }
  var taskHTML = '<li><span class="done">%</span>';
  taskHTML += '<span class="delete">x</span>';
  taskHTML += '<span class="task"></span></li>';
  var $newTask = $(taskHTML);
  $newTask.find('.task').text(taskName);
  $newTask.hide();
  $('#todo-list').prepend($newTask);
  $newTask.show('clip',250).effect('highlight',1000);
  $(this).dialog('close');
},
```

Pierwsza z dodanych instrukcji operuje na dodanym do listy ukrytym elemencie i wyświetla go przy użyciu metody `show()`. Dla dodatkowego poprawienia atrakcyjności strony zastosowane zostały dwa efekty wizualne jQuery UI. Pierwszy z nich, `clip` (patrz strona 463), sprawia, że element wydaje się powiększać. Kiedy element będzie już widoczny, wywołujemy metodę `effect()`, każąc jej odtworzyć efekt `highlight`, który na chwilkę wyświetla element na żółto, przyciągając uwagę użytkownika (i nieodmiennie powodując zachwyt osób korzystających z aplikacji).

Ostatni wiersz kodu zamyka okno dialogowe (ten sam kod zastosowałeś już wcześniej, w kroku 8. na stronie 524, dodając go do przycisku *Anuluj*). Teraz trzeba sprawdzić, jak to wszystko działa.

8. **Zapisz plik todo.js, a następnie odśwież w przeglądarce stronę index.html. Kliknij przycisk *Dodaj nowe zadanie*, wpisz nazwę zadania i kliknij przycisk *Dodaj zadanie*.**

   Nowe zadanie powinno się pojawić poniżej nagłówka *Zadania do wykonania*. Jeśli jednak nie widać go, sprawdź kod i zajrzyj do konsoli JavaScript przeglądarki, by zobaczyć, czy nie ma w niej komunikatów o błędach.

   Spróbuj dodać kolejne zadanie. Przy tej okazji zauważysz zapewne kolejny, niewielki problem — w polu tekstowym w oknie dialogowym pozostała nazwa poprzedniego zadania. Aby dodać nowe, musisz najpierw zaznaczyć ten tekst i go usunąć. Na szczęście ten problem można rozwiązać bardzo łatwo.

9. **W literale obiektowym przekazywanym w wywołaniu funkcji `dialog()`, za właściwością `buttons` wpisz przecinek i dodaj poniższy fragment kodu:**

```
close: function() {
  $('#new-todo input').val('');
}
```

   Pełny kod funkcji `dialog()` powinien mieć następującą postać:

```
$("#new-todo").dialog({
  modal: true,
  autoOpen: false,
  buttons : {
    "Dodaj zadanie" : function() {
      var taskName = $('#task').val();
      if (taskName === '') {
        return false;
      }
      var taskHTML = '<li><span class="done">%</span>';
      taskHTML += '<span class="delete">x</span>';
      taskHTML += '<span class="task"></span></li>';
      var $newTask = $(taskHTML);
      $newTask.find('.task').text(taskName);
      $newTask.hide();
      $('#todo-list').prepend($newTask);
      $newTask.show('clip',250).effect('highlight',1000);
      $(this).dialog('close');
    },
    "Anuluj" : function() {
      $(this).dialog('close');
    }
  },
  close: function() {
    $('#new-todo input').val('');
  }
});
```

   To naprawdę całkiem sporo kodu. Upewnij się, że ten ostatni fragment umieściłeś poza obiektem `buttons`. Opcja `close` widżetu okna dialogowego jQuery UI pozwala na wykonanie podanej funkcji w momencie, gdy użytkownik zamknie okno dialogowe. W tym przypadku wykonywane operacje ograniczą się do wyczyszczenia pola tekstowego, kiedy zatem użytkownik następnym razem otworzy okno dialogowe, pole będzie puste i gotowe do wpisania nazwy kolejnego zadania. (Zawartość pola można by także usunąć w ramach obsługi kliknięcia przycisku *Dodaj zadanie*, jednak chodziło tu o zaprezentowanie opcji `close`).

Oznaczanie zadania jako wykonanego

Jedną z czynności, które podczas korzystania z listy zadań dają najwięcej satysfakcji, jest oznaczanie zadań jako wykonanych. Niestety nasza aplikacja jeszcze nie udostępnia tej jakże przyjemnej operacji. W tym podrozdziale naprawisz to niedopatrzenie. Ogólnie rzecz biorąc, zadanie jest całkiem proste: użytkownik zaznacza puste pole wyboru z lewej strony nazwy i zadanie — czyli element listy — jest przenoszone z jednej listy na drugą.

Delegowanie zdarzeń

Aby oznaczyć zadanie jako wykonane, użytkownik musi kliknąć pole wyboru umieszczone w danym elemencie listy. Zazwyczaj funkcje obsługujące zdarzenia są określane poprzez wybranie odpowiedniego elementu i skojarzenie z nim funkcji; operację tę można wykonać w następujący sposób:

```
$('.done').click(function () {
  // Operacja wykonywana po kliknięciu elementu.
});
```

Jednak sposób obsługi zdarzeń w tworzonej liście zadań będzie nieco inny. Bezpośrednio po wyświetleniu strony w przeglądarce lista zadań jest pusta — nie ma żadnych zadań ani żadnych pól wyboru, które użytkownik mógłby kliknąć. Gdybyśmy spróbowali określić funkcję obsługującą zdarzenia `click` bezpośrednio po wczytaniu strony, nic by się nie stało. Ponieważ w tym momencie na stronie nie ma żadnych zadań ani żadnych pól wyboru, nie byłoby na niej żadnych elementów, w których można by zastosować tę funkcję. Oczywiście można by wywoływać metodę `click()` podczas tworzenia każdego nowego zadania, jednak biblioteka jQuery udostępnia znacznie lepsze rozwiązanie, nazywane delegowaniem zdarzeń (patrz strona 200).

Ogólnie rzecz biorąc, mechanizm delegowania zdarzeń pozwala wybrać jakiś inny element strony — element już istniejący, w którym będą umieszczane elementy dynamicznie dodawane do strony po jej wczytaniu. To właśnie ten element kontenera będzie odbierał zdarzenia skierowane do elementów umieszczonych wewnątrz niego, w naszym przypadku będą to zdarzenia `click`. Kiedy takie zdarzenie zostanie odebrane, kontener sprawdzi, czy faktycznie było ono skierowane do odpowiedniego elementu (w naszym przypadku do pola wyboru w którymś z zadań) i jeśli było, wykona odpowiednią funkcję.

W tym przypadku pusta lista wypunktowana znajduje się na stronie od momentu jej pobrania:

```
<ul id="todo-list">
</ul>
```

Kiedy użytkownik tworzy nowe zadanie, aplikacja dynamicznie dodaje je do tej listy:

```
<ul id="todo-list">
  <li>
    <span class="done">%</span>
    <span class="delete">x</span>
    <span class="task">Upiec ciasto</span>
  </li>
</ul>
```

Aby zareagować na kliknięcie elementu `<span>` należącego do klasy `done`, należy delegować funkcję obsługującą zdarzenia do istniejącej już listy wypunktowanej.

1. **W pliku *todo.js* za kodem funkcji `dialog()`, lecz wciąż wewnątrz wywołania funkcji `$(document).ready()`, wpisz następujące wywołanie:**

```
$("#todo-list").on('click', '.done', function() {

});
```

Tak wygląda ogólna struktura delegowanej funkcji obsługi zdarzeń. Najpierw wybierasz element listy wypunktowanej. Następnie wywoływana jest metoda `on()` jQuery, do której przekazujesz trzy argumenty. Pierwszym z nich jest łańcuch znaków zawierający nazwę zdarzenia, `'click'`. Drugim argumentem jest selektor określający element wewnątrz listy wypunktowanej, który musi zostać kliknięty. W tym przypadku chodzi o element `<span>` klasy `done`. I w końcu ostatnim argumentem wywołania jest funkcja — kod, który ma zostać wykonany, kiedy użytkownik kliknie pole wyboru oznaczające zadanie jako wykonane.

2. **Wewnątrz funkcji obsługującej zdarzenia dodaj poniższy wiersz kodu (wyróżniony pogrubioną czcionką):**

```
$("#todo-list").on('click', '.done', function() {
  var $taskItem = $(this).parent('li');
});
```

Pamiętaj, że zadaniem tej funkcji jest przeniesienie zadania z listy *Zadania do wykonania* na listę *Zadania wykonane*. W chwili, gdy użytkownik kliknie pole wyboru, dla jQuery istnieje jedynie znacznik `<span>` umieszczony gdzieś na liście, jednak nas interesuje cały znacznik `<li>`. Możesz go pobrać, korzystając z metody `parent()` jQuery, która umożliwia pobranie elementu rodzica aktualnie wybranego elementu.

W kodzie dodanym w tym kroku wyrażenie `$(this)` odwołuje się do elementu klikniętego przez użytkownika, czyli: `<span class="done">%</span>`. A zatem całe wywołanie `$(this).parent('li')` pobiera najbliższego przodka znacznika `<span>`, którym jest znacznik `<li>`. Innymi słowy, wybiera ono dokładnie ten element, o który chodziło.

Element ten jest następnie zapisywany w zmiennej `$taskItem`, a w kolejnym kroku go ukryjesz.

Uwaga: Wyczerpujące informacje na temat różnic pomiędzy `this` i `$(this)` można znaleźć na stronie 169.

3. **Wewnątrz tej samej funkcji dodaj poniższy fragment kodu (wyróżniony pogrubioną czcionką):**

```
$("#todo-list").on('click', '.done', function() {
  var $taskItem = $(this).parent('li');
  $taskItem.slideUp(250, function() {

  });
});
```

Metoda `slideUp()` jest zabawnym sposobem ukrycia elementu (patrz strona 216). Jednak jej wywołanie nie usuwa elementu ze strony. Po zakończeniu jej wywołania element wciąż jest dostępny na stronie, choć został ukryty przy użyciu

odpowiednich stylów CSS. Zgodnie z informacjami zamieszczonymi na stronie 211, wszystkie funkcje jQuery tworzące efekty animacji (takie jak `hide()`, `show()` czy też `slideUp()`) pozwalają na podawanie argumentów. W tym przypadku pierwszym z argumentów jest liczba — `250` — określająca czas trwania animacji; w naszym przypadku zajmie ona 250 milisekund.

Drugim argumentem jest funkcja zwrotna. Jest to funkcja, która zostanie wywołana *po* zakończeniu odtwarzania animacji. Pamiętasz zapewne, że po zakończeniu wywołania funkcji `slideUp()` znacznik `<li>` reprezentujący zakończone zadanie wciąż znajduje się na liście *Zadania do wykonania*, tyle że jest niewidoczny. Musisz zatem przenieść go na drugą listę. Kod, który to zrobi, umieścisz właśnie w funkcji zwrotnej.

4. **Wewnątrz funkcji zwrotnej dodaj poniższy fragment kodu (wyróżniony pogrubioną czcionką):**

```
$("#todo-list").on('click', '.done', function() {
  var $taskItem = $(this).parent('li');
  $taskItem.slideUp(250, function() {
    var $this = $(this);
    $this.detach();
  });
});
```

Pierwszy z dwóch nowych wierszy kodu jedynie pobiera element listy — `$(this)` — i zapisuje go w kolejnej zmiennej. Robisz to dlatego, że każde wywołanie funkcji jQuery — `$( )` — zmusza przeglądarkę do wykonania pewnej pracy. Ponieważ na tym elemencie będziesz musiał wykonać kilka operacji, zatem zamiast niepotrzebnie za każdym razem wywoływać funkcję jQuery, lepiej będzie zapisać wynik jej wywołania w zmiennej. (Rozwiązanie to, opisane bardziej szczegółowo na stronie 544, stanowi jedną z ogólnie przyjętych najlepszych praktyk związanych ze stosowaniem biblioteki jQuery).

Drugi z dodanych wierszy kodu wywołuje metodę `detach()`, która usuwa wybrany element lub elementy z drzewa DOM, choć pozostawia je na stronie. Innymi słowy, wybrany element jest usuwany z listy, lecz wciąż znajduje się w pamięci przeglądarki. Co więcej, wciąż jest zapisany w zmiennej `$this`. Dzięki temu w następnym kroku będziesz mógł przenieść go w inne miejsce strony — a konkretnie, na drugą listę!

Uwaga: Wyczerpujące informacje na temat metody `detach()` można znaleźć na stronie *http://api.jquery.com/detach/*.

---

5. **Dokończ kod funkcji zwrotnej, dopisując wewnątrz niej kolejne dwa wiersze kodu (wyróżnione pogrubioną czcionką):**

```
$("#todo-list").on('click', '.done', function() {
  var $taskItem = $(this).parent('li');
  $taskItem.slideUp(250, function() {
    var $this = $(this);
    $this.detach();
    $('#completed-list').prepend($this);
    $this.slideDown();
  });
});
```

Metodę `prepend()` poznałeś już wcześniej (w kroku 6. na stronie 529). Za jej pomocą wstawisz kod HTML wewnątrz innego elementu. W tym przypadku odłączony element listy — `$this` — zostaje zapisany na początku listy *Zadania wykonane* (listy wypunktowanej o identyfikatorze `completed-list`). I w końcu, ponieważ przenoszony element został ukryty przez wywołanie metody `slideUp()`, zatem teraz możesz go wyświetlić w nowym położeniu — w tym celu wywoływana jest metoda `slideDown()` (patrz strona 216).

6. **Zapisz plik *todo.js*, a następnie odśwież stronę index.html w przeglądarce. Aby ją przetestować, najpierw dodaj kilka zadań, a następnie kliknij pole wyboru umieszczone z lewej strony nazwy zadania, aby oznaczyć je jako wykonane.**

   Teraz powinieneś już bez problemu dodawać zadania i oznaczać je jako wykonane (patrz rysunek 14.4). Jeśli przykładowa aplikacja nie działa, dokładnie sprawdź kod i zajrzyj do konsoli JavaScript, by zobaczyć, czy nie zostały w niej wyświetlone jakieś komunikaty o błędach.

Rysunek 14.4. *Pole wyboru oraz przycisk, który pozwala oznaczyć zadanie jako wykonane oraz je usunąć, zmieniają wygląd po wskazaniu ich myszą. Możliwości tej nie zapewnia jednak kod JavaScript, lecz style CSS. A konkretnie, pseudoklasa :hover pozwalająca na zmianę wyglądu elementu, na którym znajduje się wskaźnik myszy. Aby sprawdzić, jak tworzony jest ten efekt, zajrzyj do pliku todo.css umieszczonego w podkatalogu css katalogu R14*

A teraz kolejna sprawa — może warto byłoby mieć możliwość zmiany kolejności zadań na liście? Dzięki temu można by dodać kilka zadań, a następnie uporządkować je w takiej kolejności, w jakiej powinny zostać wykonane, na przykład: „Kupić książkę kucharską", „Upiec ciasteczka", „Zjeść ciasteczka". Przy użyciu widżetu Sortable jQuery UI takie możliwości funkcjonalne można uzyskać bardzo szybko i łatwo. A skoro już przy tym jesteśmy, równie łatwo możesz zapewnić użytkownikom możliwość dowolnego przenoszenia zadań między oboma listami, zatem będzie można oznaczyć zadanie jako wykonane poprzez przeciągnięcie go na listę *Zadania wykonane*.

7. **Otwórz plik *index.html* w edytorze tekstów i odszukaj wewnątrz niego znaczniki `<ul>` obu list: *Zadania do wykonania* oraz *Zadania wykonane*. Do obu dodaj atrybut `class="sortlist"`, tak by miały następującą postać:**

```
<ul id="todo-list" class="sortlist">
```

oraz

```
<ul id="completed-list" class="sortlist">
```

Przypisując tę samą nazwę klasy obu listom, będziesz mógł w prosty sposób zapewnić możliwość porządkowania elementów na każdej z nich.

8. **Zapisz plik *index.html*, po czym otwórz w edytorze plik *todo.js*. Na samym końcu funkcji `$(document).ready()`, lecz wciąż wewnątrz niej, wpisz następujące wywołanie:**

```
$('.sortlist').sortable();
```

W ten sposób obie listy zapewniają już możliwość porządkowania elementów. Jeśli chcesz sprawdzić, jak to działa, zapisz oba pliki, a następnie odśwież stronę *index.html* w przeglądarce. Zauważysz jednak, że choć faktycznie można zmieniać kolejność elementów na liście, jednak nie da się przeciągać elementów jednej listy na drugą. Bardzo łatwo można jednak połączyć ze sobą dwa widżety Sortable.

9. **Wewnątrz wywołania funkcji `sortable()` wstaw następujący literał obiektowy:**

```
$('.sortlist').sortable({
  connectWith : '.sortlist'
});
```

Opcja `connectWith` (opisana szczegółowo na stronie 452) umożliwia połączenie dwóch list. Ponieważ w naszej aplikacji obie listy należą do tej samej klasy, aby zatem je połączyć, wystarczy podać selektor tej klasy. W ten sposób użytkownik będzie już mógł dowolnie przeciągać elementy pomiędzy oboma listami. Jednak zanim definitywnie zakończysz pracę nad listami, dodasz do nich jeszcze kilka ostatnich, drobnych usprawnień wizualnych.

10. **Do tego samego literału obiektowego dodaj trzy kolejne właściwości (wyróżnione pogrubioną czcionką):**

```
$('.sortlist').sortable({
  connectWith : '.sortlist',
  cursor : 'pointer',
  placeholder : 'ui-state-highlight',
  cancel : '.delete,.done'
});
```

Nie zapomnij o przecinku za opcją `connectWith`. Opcja `cursor` (opisana na stronie 425) zmienia postać wskaźnika myszy podczas przeciągania elementów, a opcja `placeholder` (patrz strona 454) wyróżnia miejsce listy, gdzie użytkownik może upuścić przeciągany element. I w końcu opcja `cancel` (patrz strona 451) określa te elementy umieszczone wewnątrz elementu sortowalnego, które nie mogą posłużyć jako „uchwyty" do przeciągania. W naszym przypadku użytkownik nie będzie mógł przeciągać zadania, używając pola wyboru oznaczającego zadanie jako wykonane ani przycisku do usunięcia zadania.

11. **Zapisz pliki *index.html* oraz *todo.js*, po czym odśwież plik *index.html* w przeglądarce.**

Dodaj do listy kilka różnych zadań. Spróbuj je przeciągać pomiędzy obiema listami. Teraz powinieneś już oznaczyć zadanie jako wykonane poprzez samo przeciągnięcie go na listę *Zadania wykonane* (patrz rysunek 14.1).

Usuwanie zadań

Pozostała do zaimplementowania już tylko jedna możliwość, czyli usuwanie zadań. Jest ona całkiem ważna, gdyż może się zdarzyć, że użytkownik przez pomyłkę doda zadanie, które nie powinno się znaleźć na liście. Poza tym, jeśli lista wykonanych zadań stanie się zbyt długa, użytkownik może zdecydować się na usunięcie kilku z nich.

1. **W pliku *todo.js*, za kodem funkcji `sortable()`, lecz wciąż wewnątrz funkcji `$(document).ready()` wpisz następujący fragment kodu:**

   ```
   $('.sortlist').on('click', '.delete', function() {
   });
   ```

   Także w tym przypadku zastosowałeś delegowanie zdarzeń. W momencie wczytywania strony listy są puste i nie ma w nich żadnych przycisków, dlatego też do usuwania zadań używasz techniki delegowania. W tym przypadku wywołanie `$('.sortlist')` powoduje pobranie obu list wypunktowanych dostępnych na stronie (gdyż użytkownicy powinni mieć możliwość usuwania zadań z obu list), a wywołanie metody `on()` informuje jQuery, że należy obsługiwać zdarzenia kliknięcia skierowane do elementów klasy `delete`. W odpowiedzi na kliknięcie takiego elementu ma zostać wywołana przekazana funkcja.

   W kolejnym punkcie zajmiesz się zaimplementowaniem kodu tej funkcji.

2. **Znajdź kod między znacznikami `<script>` w sekcji nagłówkowej strony i usuń kod wyróżniony pogrubieniem:**

   ```
   $('.sortlist').on('click', '.delete', function() {
     $(this).parent('li').effect('puff', function() {
       $(this).remove();
     });
   });
   ```

   W tym kodzie dzieje się całkiem sporo, lecz powinieneś już być przyzwyczajony do takich rozwiązań. Poniżej zostały opisane wszystkie wykonywane operacje.

   - Wyrażenie `$(this).parent('li')` pobiera element kliknięty przez użytkownika (reprezentowany przez `$this`), a następnie wśród jego przodków odnajduje znacznik `<li>`. Innymi słowy, wyrażenie to pobiera element zadania, które należy usunąć.
   - Metoda `effect()` biblioteki jQuery UI odtwarza w elemencie określony efekt wizualny. W tym przypadku odtwarzamy efekt o nazwie `puff`, który powoduje że element się powiększa, stopniowo wygasa i w końcu znika.
   - W wywołaniu metody `effect()` została umieszczona funkcja zwrotna, która będzie wywołana po zakończeniu efektu. W naszym przypadku funkcja pobiera element listy, na którym operowała metoda `effect()` — czyli element listy pobrany przy użyciu wyrażenia `$(this)` — a następnie usuwa go całkowicie ze strony, wywołując metodę `remove()` jQuery (patrz strona 160). Metoda `remove()`, w odróżnieniu od `detach()` (opisanej na stronie 533), całkowicie usuwa wskazany element strony.

3. **Zapisz plik i odśwież w przeglądarce stronę *index.html*.**

   Teraz powinieneś już bez problemu dodawać, przenosić i usuwać zadania, a cała strona powinna wyglądać tak, jak na rysunku 14.1. Gdybyś miał jakieś problemy z wykonaniem tego przykładu, poniżej znajdziesz cały kod pliku *todo.js*.

```
$(document).ready(function(e) {
  $("#add-todo").button({
    icons: {
      primary: "ui-icon-circle-plus"
    }
  }).click(function() {
    $("#new-todo").dialog('open');
  });

  $("#new-todo").dialog({
    modal: true,
    autoOpen: false,
    buttons : {
      "Dodaj zadanie" : function() {
        var taskName = $('#task').val();
        if (taskName === '') {
          return false;
        }
        var taskHTML = '<li><span class="done">%</span>';
        taskHTML += '<span class="delete">x</span>';
        taskHTML += '<span class="task"></span></li>';
        var $newTask = $(taskHTML);
        $newTask.find('.task').text(taskName);
        $newTask.hide();
        $('#todo-list').prepend($newTask);
        $newTask.show('clip',250).effect('highlight',1000);
        $(this).dialog('close');
      },
      "Anuluj" : function() {
        $(this).dialog('close');
      }
    },
    close: function() {
      $('#new-todo input').val('');
    }
  });

  $("#todo-list").on('click','.done', function() {
    var $taskItem = $(this).parent('li');
    $taskItem.slideUp(250, function() {
      var $this = $(this);
      $this.detach();
      $('#completed-list').prepend($this);
      $this.slideDown();
    });
  });

  $('.sortlist').sortable({
    connectWith : '.sortlist',
    cursor : 'pointer',
    placeholder : 'ui-state-highlight',
    cancel : '.delete,.done'
  });

  $('.sortlist').on('click','.delete',function() {
    $(this).parent('li').effect('puff', function() {
      $(this).remove();
    });
  });
}); // Koniec funkcji ready.
```

Uwaga: Pełną, działającą kopię aplikacji napisanej w tym rozdziale znajdziesz w plikach *complete-index.*
↪html oraz *complete-todo.js*, umieszczonych w katalogu *R14*.

Dalsze kroki

Gratuluję — właśnie napisałeś swoją pierwszą aplikację internetową! Jednak tę aplikację można poprawić na kilka różnych sposobów. Być może już nawet masz przygotowaną listę ewentualnych usprawnień. W tym podrozdziale zamieszczona została lista potencjalnych poprawek wraz z odnośnikami do źródeł informacji, które mogą Ci się przydać podczas ich implementowania.

Edycja zadań

Aktualnie aplikacja nie zapewnia użytkownikowi możliwości poprawy ewentualnych błędów typograficznych w nazwach zadań. Jeśli ktoś wpisze „Uwiec ciasteczka", będzie musiał usunąć zadanie i utworzyć je od nowa. Problem edycji zadań można rozwiązać na dwa sposoby.

Pierwszym z nich jest dodanie do każdego z zadań przycisku *Edytuj*. Jego kliknięcie będzie powodować wyświetlenie okna dialogowego z polem tekstowym zawierającym nazwę zadania — czyli tekst umieszczony wewnątrz znacznika `<span>` klasy `task`.

Można także dodać do strony kolejne okno dialogowe, takie jak przedstawione na stronie 522. Kliknięcie przycisku *Edytuj* powodowałoby wyświetlenie tego okna dialogowego i umieszczenie nazwy zadania w polu tekstowym, a zamknięcie okna dialogowego — aktualizację nazwy zadania.

Innym rozwiązaniem może być skorzystanie z właściwości HTML o nazwie `contentEditable`. Jej użycie zapewnia możliwość edycji zawartości dowolnego elementu HTML. Przykładowo poniżej został przedstawiony kod HTML zapewniający możliwość edycji nazw zdań w naszej przykładowej aplikacji:

```
<span class="taks" contentEditable>
```

Właściwość tę można nawet zastosować dynamicznie, używając metody `prop()` jQuery:

```
$('.task').prop('contentEditable', true);
```

Jednak z zastosowaniem właściwości `contentEditable` wiąże się jeden problem. Widżet Sortable jQuery UI nie pozwala na zaznaczanie tekstu w sortowalnych elementach, a zatem nawet po zastosowaniu właściwości `contentEditable` użytkownik nie będzie mógł zaznaczyć tekstu do edycji. Problem ten można ominąć, nakazując jQuery UI ignorowanie elementów klasy `task` poprzez przekazanie odpowiedniej opcji w wywołaniu funkcji `sortable()`. Dokładnie w taki sam sposób postąpiłeś z polem wyboru do oznaczania zadania jako wykonanego oraz przyciskiem do usuwania zadania w kroku 10. na stronie 535; a tu wystarczy dodać selektor `.task` tak, jak pokazano na poniższym przykładzie:

```
cancel: '.delete,.done,.task'
```

Jeśli jednak zastosujesz to rozwiązanie, w elementach zadań nie pozostanie zbyt wiele miejsc, za które użytkownik będzie mógł je przeciągać. W takim przypadku powinieneś dodać do elementu `<li>` zadania jakiś wyraźnie widoczny element, którego użytkownik mógłby używać do przeciągania. Aby skorzystać z tego rozwiązania, wystarczy użyć opcji `handle` widżetu Sortable (patrz strona 453).

Potwierdzanie usunięcia

Obecnie kiedy użytkownik kliknie przycisk usuwający zadanie, zostaje ono usunięte raz na zawsze. Mógłbyś jednak dodać modalne okno dialogowe, zawierające prośbę o potwierdzenie usunięcia. Jeśli użytkownik kliknie przycisk *Tak*, zadanie zostanie usunięte, jeśli przycisk *Nie*, zadanie pozostanie na liście.

Zapisywanie listy

Jednak największym problemem naszej aplikacji jest to, że nie zapamiętuje ona zadań w momencie zamykania okna przeglądarki. Innymi słowy, lista ma charakter całkowicie tymczasowy i nie zostanie odtworzona po zamknięciu i ponownym uruchomieniu przeglądarki bądź też po wyświetleniu strony na innym komputerze. Istnieje kilka sposobów pozwalających na zapamiętanie stanu listy zadań.

Magazyn lokalny

Wszystkie nowoczesne przeglądarki udostępniają mechanizm określany jako *magazyn lokalny* (ang. *local storage*). Pozwala on na zapisywanie danych na komputerze użytkownika i odczytanie ich po ponownym wyświetleniu strony. Mógłbyś skorzystać z magazynu lokalnego, by zapisywać aktualny stan listy zadań po każdej zmianie ich stanu. W takim przypadku, kiedy użytkownik wróci na stronę, powinieneś sprawdzić, czy są zapisane jakieś dane w magazynie lokalnym, a jeśli są, musisz odpowiednio zaktualizować stronę.

Więcej informacji na temat magazynu lokalnego można znaleźć na witrynie Mozilla Developer Network: *https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage*. Dostępna jest nawet wtyczka jQuery ułatwiająca korzystanie z tego mechanizmu: *https://github.com/julien-maurel/jQuery-Storage-API*.

Zapis na serwerze

Kolejną możliwością jest zapisanie listy zadań na serwerze. To rozwiązanie ma tę zaletę, że takiej listy zadań można używać na dowolnym komputerze. Z drugiej strony jego wadą jest konieczność utworzenia jakiegoś systemu kontroli dostępu do listy, w przeciwnym razie każdy będzie mógł wejść na stronę i wyświetlić Twoje zadania (a nawet je usunąć).

Książka ta nie jest poświęcona zagadnieniom programowania aplikacji działających po stronie serwera. Jednak do przekazania zadań na serwer będziesz musiał użyć kodu JavaScript. Zapewne będzie to kod korzystający z technologii AJAX, opisanej w poprzednim rozdziale.

Najprostszym rozwiązaniem byłoby pobranie wszystkich elementów list i zastosowanie metody `.each()` do pobrania nazwy każdego z zadań. Następnie mógłbyś utworzyć obiekt jQuery zawierający listę wszystkich zadań, zarówno tych do wykonania, jak i już wykonanych. W końcu, abyś mógł je przesłać na serwer, musiałbyś *serializować* ten obiekt, czyli zapisać go w postaci łańcucha znaków. Poniżej został zamieszczony kod funkcji, która realizuje wszystkie te operacje:

```
function getData() {
  var todoData = {
    toDo : [],
    completed : []
  };
  $('#todo-list').each( function() {
    var task = $(this).find('.task').text();
    todoData.toDo.push(task);
  });
  $('#completed-list').each( function() {
    var task = $(this).find('.task').text();
    todoData.completed.push(task);
  });
  return $.param(todoData);
}
```

Tę funkcję mógłbyś wywoływać za każdym razem, kiedy będziesz chciał pobrać wszystkie zadania w postaci nadającej się do przesłania na serwer. Odczytanie tych zadań i zrobienie z nimi czegoś użytecznego należałoby już do programu wykonywanego na serwerze.

Inne pomysły

Jeśli masz więcej pomysłów na poprawienie zamieszczonej tu przykładowej listy zadań, warto, byś je dokładniej wypróbował. Ten projekt doskonale nadaje się do sprawdzenia nowych umiejętności stosowania języka JavaScript oraz bibliotek jQuery i jQuery UI. Bezustannie rozwijaną wersję tego projektu można znaleźć na serwisie GitHub, na stronie *https://github.com/sawmac/jquery-todo*.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

z, ź

JavaScript i jQuery

nieoficjalny podręcznik

JavaScript ma za sobą długą historię, w której bywały okresy lepsze i gorsze. Czasem język ten był wręcz masowo blokowany w przeglądarkach. Jednak te czasy minęły. W tej chwili nie obejdzie się bez niego żadna poważna aplikacja internetowa lub choć trochę bardziej zaawansowana strona WWW. Użytkownicy serwisów internetowych wymuszają na projektantach coraz nowsze i lepsze rozwiązania. Dlatego na rynku wciąż pojawiają się dodatkowe narzędzia dla języka JavaScript, które ułatwiają wykorzystanie jego potencjału.

Najpopularniejszym dodatkiem tego typu jest biblioteka jQuery. Genialna w swojej prostocie, z ogromnymi możliwościami, zdobyła uznanie wszystkich programistów JavaScriptu. Nie potrafią sobie oni wyobrazić programowania bez jej wykorzystania. W tej książce znajdziesz najlepsze techniki, jakie oferuje JavaScript. Nauczysz się nawigować po drzewie DOM, modyfikować zachowanie elementów oraz obsługiwać zdarzenia. Poznasz również narzędzia, które ułatwią Ci pracę oraz debugowanie kodu. Jest to obowiązkowa pozycja dla każdego projektanta stron internetowych. Musisz ją mieć!

Dzięki tej książce:

- **poznasz podstawy języka JavaScript**
- **zobaczysz, jak jQuery potrafi ułatwić pracę z JavaScriptem**
- **opanujesz mechanizm zdarzeń w jQuery**
- **zbudujesz lepszą i ciekawszą witrynę!**

Twórz atrakcyjne strony WWW.
Wzbogać kod HTML o animacje, interaktywność i dynamiczne efekty wizualne!

helion.pl
księgarnia internetowa

Nr katalogowy: 8754

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

ISBN 978-83-246-4381-3
9 788324 643813
Cena 79,00 zł

Informatyka w najlepszym wydaniu