CHAPTER

07

더 깊은 생각과 영감은 기능에 대한 문서나 사설보다는 신문이나 잡지의 주요 광고의 구성에 깃들어 있다.
_마샬 맥루언(Marshall McLuhan)

AJAX와 부분 뷰로 뷰 구성하기

Ajax와 (플래시 같은) 임베디드 애플리케이션들은 사용자의 브라우징 경험을 바꾸어 놓았다. 마이크로소프트 기술 분야에서는 실버라이트(Silverlight), Ajax 및 수많은 자바스크립트를 이용하여 사용자의 브라우저에서 코드를 실행되는 RIA(Rich Internet Application)을 개발해 왔을 것이다. 예를 들어 최신 버전의 Firefox 4 브라우저는 작업자 스레드 개념을 지원한다. 이 개념은 데스크톱 애플리케이션들과 동일한 방식으로 백그라운드에서 자바 스크립트를 실행하기 위해 지원된다.

우선 우리는 Ajax를 통해 사용성을 향상시키는 방법부터 알아보기로 하자. MVC는 Ajax를 지원하기 위한 다양한 종류의 `ActionResult` 클래스의 서브 타입을 제공한다. 또한 Ajax를 손쉽게 지원하는 jQuery 프레임워크가 번들로 제공된다. Ajax에 대해 알아본 후에는 HTML 태그의 일부를 별도로 관리할 수 있는 부분 뷰(Partial View)에 대해 알아본다. 섹션 7.3 "중복을 제거하기 위한 부분 뷰의 활용"에서 살펴보겠지만, 부분 뷰를 이용하면 뷰 코드의 중복을 효율적으로 제거할 수 있다. GetOrganized 애플리케이션은 현재 Ajax가 적용되어 있지 않다. 애플리케이션의 응답 성능을 향상시키기 위해 우리는 여기에 몇 가지 기능을 더 추가할 것이다. 또한 GetOrganized 애플리케이션을 비롯한 대부분의 웹 애플리케이션들에 대동소이하게 적용되는 Ajax 작업들을 구현하는 방법에 대해서도 살펴본다. 먼저 버튼을 클릭하지 않고 HTTP POST 메서드를 이용해 객체를 제거하는 방법을 알아볼 것이

다. 또한 구글 Suggest처럼 매우 대중적으로 사용되는 Ajax 기능 중 하나인 자동 완성 기능에 대해서도 살펴본다. 물론 MVC에서 jQuery 플러그인을 사용하는 방법에 대해서도 학습한다[주1].

7.1 Ajax의 활용

HTTP POST 메서드를 이용해서 `Todo` 객체를 삭제하는 기능을 구현하기에 앞서 Ajax 개발에 필요한 기본적인 원칙을 먼저 짚고 넘어가보자. 지금 살펴볼 내용 중에는 jQuery API를 이용하여 코드를 작성할 때 Visual Studio의 도움을 얻을 수 있는 방법도 포함되어 있다.

간단히 말해 Ajax는 자바스크립트를 이용한 브라우저 기반의 기술이다. 일반적으로 Ajax는 새로운 페이지로 이동하거나 기존 페이지를 새로 고치지 않고도 웹 페이지를 업데이트하는 형태로 활용된다. Ajax를 활용하면 사용자가 페이지를 여러 번 새로 고치거나 필요한 작업을 위해 여러 개의 페이지 사이를 이동하는 횟수를 줄일 수 있어 전체적으로 웹 사이트의 사용자 경험을 향상시킬 수 있다. 그러나 일부 브라우저는 자바스크립트를 지원하지 않거나 보안상의 이유로 활성화하지 않고 있다. 또한 일부 모바일 장치들은 jQuery와 같은 자바스크립트 라이브러리를 완벽하게 지원하지 못한다. 그런 이유로 우리는 Ajax를 대체할 수 있는 방법이 존재하는지를 미리 확인해 두어야 한다. 이렇게 지원이 다소 빈약한 클라이언트들을 때로는 "우아한 성능 저하(Graceful degradation)"라고 부른다. 사실 간단히 생각하면 웹 페이지는 자바스크립트나 Ajax가 지원되지 않는 브라우저에서도 정상적으로 동작한다. 섹션 7.2 "자동 완성 기능의 구현"에서 이러한 부분에 대해 살펴볼 것이다.

대부분의 사용자들은 Ajax를 사용할 것이기 때문에 Ajax를 지원하지 않는 클라이언트를 함께 지원할 수 있도록 애플리케이션을 구현하려면 그만큼 개발자들은 더 많은 노력을 기울여야 한다. 이렇게 분리된 형태로 애플리케이션을 구현하는 불상사를 막으려면 코드를 일반화해야 한다. 다행히 MVC는 각각의 컨트롤러를 한 가지 방식으로 구현하는 데 도움을 줄 수 있는 `IsAjaxRequest()` 메서드를 제공한다. 이 메서드를 활용하면 컨트롤러에 `Create()` 액션과 거의 유사한 동작을 수행하는 `CreateWithAjax()`와 같은 액션 메서드를 생성하여 호

주1 http://joepoon.com/blog/2009/05/14/jquery-at-the-calgarynet-user-group/

출할 수 있다.

Ajax를 활용한다는 것은 자바스크립트와 jQuery의 활용 빈도가 높아진다는 것을 뜻한다. 이때 자바스크립트 코드를 작성하면서 도움을 받을 수 있는 방법 중 하나는 jQuery에 대한 인텔리센스 기능을 활성화하는 것이다. 그러려면 Visual Studio 2008 핫픽스를 다운로드하여 설치하면 된다[주2]. 인텔리센스를 이용하면 C# 코드를 작성할 때와 마찬가지로 자바스크립트 코드를 어렵지 않게 작성할 수 있다. 만일 jQuery에 익숙하지 않다면 http://jquery.com의 튜토리얼과 예제들을 살펴보기 바란다. Visual Studio 2010을 사용한다면 별도의 핫픽스는 필요치 않다.

자, 그러면 우아한 성능 저하와 인텔리센스를 양손에 들고 GetOrganized 애플리케이션에 첫 번째 Ajax 기능을 구현해 보자. 이번에 구현할 기능은 HTTP POST 방식으로 Todo 객체를 삭제하는 기능이다.

HTTP POST 방식으로 데이터 삭제하기

우리가 구현할 첫 번째 Ajax 기능은 섹션 3.4 "데이터 삭제하기: 뷰가 없는 액션 구현하기"에서 Todo 객체를 삭제하는 방법을 Ajax 방식으로 변경하는 것이다. 또한 액션 메서드의 호출을 HTTP POST 방식으로만 가능하도록 제한할 것이다. 우선 액션 메서드를 아래와 같이 수정해 보자.

```
Line 1  // GET: /Todo/Delete/Title={제목}
    -   [AcceptVerbs(HttpVerb.Post)]
    -   public ActionResult Delete(string title)
    -   {
    5     Todo.ThingsToBeDone.
    -       Remove(
    -       Todo.ThingsToBeDone.Find(todo => todo.Title == title));
    -
    -     if (Request.IsAjaxRequest())
   10       return new EmptyResult();
    -
    -     return RedirectToAction("Index");
    -   }
```

주2 http://code.msdn.microsoft.com/KB958502/Release/ProjectReleases.aspx?ReleaseId=1736

2번 라인에서는 Delete() 액션 메서드를 HTTP POST 방식으로만 호출할 수 있도록 제한하고 있다. 액션 메서드의 호출을 HTTP POST로 제한하는 방법 자체는 간단하지만, /Views/Todo/Index.aspx 페이지는 현재 삭제 명령을 HTTP GET 방식으로 실행하도록 구현되어 있는 것이 문제이다. 또한 9번 라인의 코드는 삭제 요청이 Ajax 방식으로 발생한 경우에만 동작하도록 구현하고 있다. 이렇게 하면 응답할 필요가 없는 요청을 다른 액션으로 리다이렉트하는 경우를 방지할 수 있다. 예제의 경우, Ajax 요청은 작업을 완료하기 위해 필요한 추가적인 데이터가 전혀 없으므로 EmptyResult를 리턴하였다.

jQuery 코드를 작성하기에 앞서 또 다른 HTML 헬퍼를 구현해야 한다. 그 이유는 우리가 /Views/Todo/Index.aspx 뷰에서 Grid 헬퍼를 사용했기 때문이다. Grid 헬퍼는 각각의 컬럼에 지정한 기능들은 실행해 주지만 직접 작성한 HTML 요소들에 대해서는 아무런 처리를 해 주지 않는다. 우리가 작성할 헬퍼는 **id** 특성과 **class** 특성이 적용된 **<A>** 태그를 렌더링한다. 여러분이 NUnit을 이용한 테스트 주도 개발에 조금 더 익숙해지도록 하기 위해 이 LinkHelper 클래스에 대한 테스트 코드를 작성하고 이 테스트 코드를 토대로 실제 코드를 구현할 수 있도록 해 보자. 새로운 헬퍼는 다른 것들과 마찬가지로 ViewHelpers 디렉터리에 구현한다.

```
Line 1  using System.Web.Mvc;
        using System.Web.Routing;

        namespace GetOrganized.ViewHelpers
        {
          public static class LinkHelper
          {
            public static string Link(this HtmlHelper helper,
              string linkText, string url, object htmlAttributes)
            {
              var builder = new TagBuilder("a");

              builder.MergeAttributte(
                new RouteValueDictionary(htmlAttributes));
              builder.Attributes.Add("href", url);
              builder.SetInnerText(linkText);

              return builder.ToString();
            }
          }
        }
```

이 HTML 헬퍼는 앞서 섹션 6.2에 나오는 "드롭다운 리스트에 색상 표시하기"에서 구현했던 것과 유사하다. 9번 라인에 선언한 마지막 매개변수는 뷰에서 익명 객체를 전달하기 위한 것이다. 섹션 6.1에 나오는 "HTML 헬퍼를 이용하여 CSS 적용하기"에서는 이 매개변수를 이용하여 **id**나 **class**와 같은 특성을 지정하였다. 헬퍼 메서드를 호출하고 스타일을 정의하는 코드는 아래와 같다.

```
Html.ActionLink("name", "action", new { @class = "SomeCssClass" })
```

이 방법을 이용하면 헬퍼 메서드를 여러 형태로 재정의할 필요가 없으며, 우리가 필요한 특성은 무엇이든 지정할 수 있다. 이 객체를 TagBuilder 클래스에 필요한 특성들로 변환하려면 14번 라인과 같이 RouteValueDictionary 객체로 변환하면 된다. 일단 객체를 IDictionary <String, Object> 타입으로 변환하면 각 요소들이 HTML 특성으로 병합되어 뷰에 렌더링된다. 이제 필요한 헬퍼의 구현을 마쳤으므로 이번 섹션의 주제인 jQuery를 이용한 데이터의 삭제 기능을 구현하기 위해 아래와 같이 코드를 작성해 보자.

```
Line 1  <head>
    -   <meta http-equiv="Content-Type"
    -     content="text/html; charset=iso-8859-1" />
    -   <link href="../../Content/colorpicker.cssv
    5     rel="Stylesheet" type="text/css" />
    -   <link href="../../Content/Site.css"
    -     rel="stylesheet" type="text/css" />
    -   <link href="../../Content/colorpicker.css"
    -     rel="stylesheet" type="text/css" />
   10   <script type="text/javascript"
    -     src="../../Scripts/jquery-1.4.1.js" ></script>
    -   <script type="text/javascript"
    -     src="../../Scripts/colorpicker.js" ></script>
    -
   15   <script type="text/javascript"
    -     src="../../Scripts/jquery.autocomplete.js" ></script>
    -   <link href="../../Content/jquery.autocomplete.css"
    -     rel="stylesheet" type="text/css" />
    -
   20   <script type="text/javascript"
    -     src="../../Scripts/jquery-ui-1.8rc3.min.js" ></script>
    -
    -   <script type="text/javascript"
    -     src="../../Scripts/jquery.livequery.js" ></script>
   25
```

```
<asp:ContentPlaceHolder ID="Head" runat="server" />
</head>
```

이번 예제 코드가 올바르게 동작하기 위해서는 몇 가지 jQuery 플러그인이 필요하다. 이 코드들은 Site.Master 파일에 작성해 두었으므로 모든 뷰에서 이 플러그인들을 참조할 수 있다. 16번 라인에서는 세 개의 jQuery 플러그인을 참조하는데, 각각 AutoComplete[주3], jQuery-UI[주4], 그리고 Live Query[주5] 등이다. AutoComplete와 같은 일부 플러그인은 적절한 스타일을 구현한 CSS 파일을 함께 제공한다. 필요한 자바스크립트 파일과 CSS 파일을 참조했으면 이제 /Views/Todo/Index.aspx 뷰에서 HTTP POST 방식으로 데이터를 삭제할 수 있도록 아래와 같이 코드를 변경하자.

```
Line 1  <%@ Page Title="" Language="C#"
        MasterPageFile="~/Views/Shared/Site.Master"
        Inherits="System.Web.Mvc.ViewPage<IEnumerable<Todo>>" %>
        <%@ Import Namespace="System.Drawing" %>
    5   <%@ Import Namespace="GetOrganized.Models" %>
        <%@ Import Namespace="GetOrganized.ViewHelpers" %>
        <%@ Import Namespace="MvcContrib.UI.Grid" %>

        <asp:Content ID="Content1"
   10     ContentPlaceHolderID="head" runat="server" >
        <title>Index</title>
        <script type="text/javascript" language="javascript" >
          $(document).ready(function() {
            $(".deleteTodoLink" ).click(function() {
              var element = $(this);
              var todoTitle = element.attr("id" );

              $.post(
                "Todo/Delete" ,
   20           { title: todoTitle },
                function() {
                  element.closest("tr" ).
                  fadeOut("slow" , function()
                  { $(this).remove(); });
```

주3 http://www.pengoworks.com/workshop/jquery/autocomplete.htm

주4 http://jqueryui.com/download

주5 http://plugins.jquery.com/project/livequery

```
25          }
-         );
-       });
-     });
- </script>
30 </asp:Content>
-
- <asp:Content ID="Content2"
-   ContentPlaceHolderID="MainContent" runat="server" >
-   <h2><%= ViewData[ "UserName" ] %>의 할 일</h2>
35
-   <%= Html.Grid(Model).Columns(column => {
-       column.For(
-         x =>
-         Html.Link("삭제" , "#" , new { id = x.Title,
40           @class = "deleteTodoLink" })).DoNotEncode();
-       column.For(
-         x =>
-         Html.ActionLink("수정" , "Edit" , new { x.Title })).
-         Named("수정" ).DoNotEncode();
45       column.For(x => x.Title);
-     })
-   .Attributes(style => "text-align: center;" )
-   .Empty("할 일을 모두 마쳤습니다!" )
-   %>
50
-   <p>
-     <%= Html.ActionLink("새로 만들기" , "Create" ) %>
-   </p>
- </asp:Content>
```

우리가 작성한 모든 jQuery 코드는 파일 상단의 head라는 ID를 갖는 **ContentPlaceHolder** 컨트롤에 작성되어 있다. 이렇게 하면 페이지의 코드를 깔끔하게 분리할 수 있을 뿐만 아니라 향후 jQuery 코드의 양이 많아지면 별도의 파일로 분리하기에도 수월하다.

6번 라인에서는 GetOrganized.ViewHelpers 네임스페이스에 대한 참조를 추가하며, 이를 통해 39번 라인에서와 같이 Grid 헬퍼 내에서 LinkHelper 클래스의 메서드를 호출할 수 있게 된다. 또한 이 코드에서는 익명 객체 초기자(Anonymous Object Initializer)를 이용하여 **class** 특성과 **id** 특성을 전달한다. 이 특성들 덕분에 jQuery의 요소 선택자를 이용하여 어떤 삭제 링크가 클릭되었는지를 알아내기가 더욱 수월해진다. 그러면 jQuery 코드에 대해 살펴보도록 하자.

15번 라인에서는 this 키워드를 jQuery 요소 선택자에 전달하여 클릭된 링크를 탐색한다. jQuery 요소 선택자는 HTML 요소의 배열을 리턴하며, 예제의 경우에는 **<A>** 요소에 대한 배열을 리턴한다. 다음으로 이 요소 배열에서 **id** 특성 값을 가져오며, 예제의 경우에 이 값은 우리가 삭제하고자 하는 `Todo` 객체의 `Title` 속성 값이다. 이제 Ajax 요청을 전달할 준비가 되었다. 18번 라인의 `$.post()` 메서드는 jQuery를 통해 컨트롤러에게 HTTP POST 요청을 보내기 위해 필요한 함수이다. 이 메서드는 세 개의 매개변수로 구성되어 있다. 첫 번째와 두 번째 매개변수는 요청을 보내고자 하는 URL과 쿼리문자열 변수이다. 세 번째 매개변수는 HTTP POST 요청이 성공적으로 실행되었을 때 추가로 실행하고자 하는 자바스크립트 함수이다. 예제에서는 삭제된 `Todo` 객체를 표현한 행에 페이드아웃 효과를 적용하여 행이 천천히 사라지도록 한다. 삭제될 행을 탐색하기 위해서 22번 라인에서와 같이 `element.closest("tr")` 구문을 사용한다. 이 구문을 통해 클릭한 링크에서 가장 가까운 행, 즉 **<TR>** 태그에 대한 참조를 얻게 된다. 이 구문은 우리가 삭제하고자 하는 전체 행을 선택한다. 24번 라인에서는 `remove()` 메서드와 jQuery의 `fade()` 효과를 이용하여 HTML 문서에서 해당 행을 제거한다.

이제 F5키를 눌러 프로그램을 실행해 보면 페이드 효과와 함께 `Todo` 객체를 삭제하는 기능이 추가된 결과를 확인할 수 있다. 이 효과는 사용자에게 어떤 항목이 삭제되는지를 시각적으로 알려줄 수 있다. 이제 Ajax에 대한 기본적인 내용을 학습했으므로 더욱 보편적인 Ajax 기능인 자동 완성 기능을 추가해 보자.

7.2 자동 완성 기능의 구현

요즘 사용자들은 사이트에 검색 기능을 추가할 때 대부분이 자동 완성 기능을 원한다. 검색을 통해 원하는 결과를 얻으려면 여러 번 다른 화면을 왔다갔다하거나 페이지 링크를 클릭해야 한다. 자동 완성 기능을 이용하면 사용자가 입력하는 즉시 검색 결과를 보여줄 수 있다.

GetOrganized 애플리케이션의 경우에도 등록된 계획들을 검색할 수 있다면 사용자가 자신의 계획을 찾는 데 도움이 될 것이다. 대부분의 사용자들은 자신들이 써 내려 갔던 생각들은 기억하겠지만, 이를 저장한 제목을 제대로 기억하지 못할 수 있다. 이런 경우에 검색은 매우 유용한 도구가 될 수 있으며, jQuery와 MVC를 이용한 자동 완성 기능의 구현 방법을

배울 수 있는 적절한 도구이다.

jQuery를 사용하면 좋은 점은 수많은 플러그인을 언제든지 다운로드하고 사용할 수 있다는 점이다. 섹션 7.1에서 배웠던 "HTTP POST 방식으로 데이터 삭제하기"에서 Site.Master 파일에 jQuery 자동 완성 플러그인을 추가했기 때문에 지금 당장이라도 사용할 수 있다. jQuery용 자동 완성 플러그인은 여러 가지가 있으며, 그 중에는 지금 우리가 사용하는 것보다 더 고급 기능을 제공하는 것들도 있다. 예를 들어 평범한 텍스트가 아닌 JSON 형식으로 데이터를 리턴하는 콜백 함수를 처리하고자 할 수도 있다. 이런 경우라면 JQ Autocomplete 플러그인이 더 나은 선택이 될 것이다[주6].

jQuery로 자동 완성 기능을 구현하기에 앞서 ThoughtController에 검색 기능을 구현해 보자. 우선 아래의 테스트 코드를 작성하자.

```
Line 1  [Test]
   2  public void Should_Find_Thoughts_By_Text_Match_Case_Insensitive()
   3  {
   4    var learnCsharp = Thought.CurrentThoughts[0];
   5    var contentResult = ((ContentResult)
   6      new ThoghtController().Search("학습");
   7
   8    Assert.AreEqual(learnCsharp.Name, contentResult.Content);
   9  }
```

이 테스트 코드는 사용자가 "학습"이라는 단어를 입력하여 그 결과로 "C# 3.5 학습"이라는 제목을 가진 Thought 객체가 검색되는 상황을 가정한다. 또한 이는 우리의 검색 알고리즘이 대소문자를 구별하지 않는다는 점을 의미한다. 주목해야 할 또 다른 사항은 이 액션은 jQuery 자동 완성 기능을 이용한다는 점이며, 따라서 평범한 텍스트를 응답으로 리턴해야 한다. 6번 라인에서 Search 액션 메서드의 실행 결과를 ContentResult 타입으로 변환하는데, 지금은 이 액션 메서드가 존재하지 않으므로 테스트가 실패하게 된다. 따라서 다음과 같이 Search 액션 메서드를 구현해 보자.

주6 http://www.devbridge.com/projects/autocomplete/jquery/

```
Line 1  public ActionResult Search(string q)
    -   {
    -     var searchResults = Thought.CurrentThoughts.FindAll(
    -       thought => thought.Name.ToLower().Contains(q.ToLower()));
    5
    -     var autoCompleteResults =
    -       String.Join("\n",
    -       searchResults.ConvertAll(g => g.Name).ToArray());
    -
   10     return Content(autoCompleteResults);
    -   }
```

1번 라인에서는 검색 키워드를 위한 매개변수로 q를 정의했다. 그 이유는 jQuery 플러그인이 검색어를 위한 변수명으로 q를 정의하고 있기 때문이다. 대체로 한 글자로 된 변수명을 사용하는 것은 (여러분을 포함한) 개발자들이 메서드나 변수의 의도를 명확히 파악하기 어렵기 때문에 그다지 권장되는 방법은 아니다. 또한 4번 라인에서는 검색어가 대소문자를 구분하지 않도록 구현했다. 대소문자를 구분하지 않도록 구현하는 가장 손쉬운 방법은 ToLower() 메서드를 이용하여 모든 문자를 소문자로 취급하는 것이다. 우리가 구현한 검색 알고리즘은 매우 기초적인 것이며, 고급 검색 기법에 대해서는 167페이지의 TIP 박스 내용을 참고하기 바란다. 다음 단계는 8번 라인과 같이 검색 결과를 줄 바꿈으로 구분하는 것이다. 자동 완성 플러그인은 각각의 검색 결과를 개별적인 한 줄로 표현하도록 구현되어 있기 때문에 Thought 객체의 Name 속성 값을 String.Join 메서드를 이용하여 하나의 문자열로 표현하되, 각 제목 뒤에 \n을 덧붙여 줄바꿈을 추가하도록 구현했다. 이렇게 만들어진 검색 결과는 Content(object result) 메서드를 통해 응답 스트림으로 보내진다.

이제 앞서 작성한 테스트 코드는 성공하겠지만, 사용자에게 그들이 선택한 Thought 객체를 전달할 방법이 필요하다. 현재 우리는 Id 속성을 통해서만 Thought 객체를 구별할 수 있다. 아래의 테스트 코드는 Thought 객체의 Name 속성 값을 바탕으로 해당 객체를 조회한 후 /Views/Thought/Detail/Id 페이지로 이동하도록 하는 방법을 테스트하기 위한 것이다.

```
Line 1  [Test]
    2   public void Should_Find_Thought_By_Name_And_Redirect_To_Details_View()
    3   {
    4     var routeValueDictionary = new ThoughtController().
    5       FindDetails("C# 3.5 학습하기").
    6       AssertActionRedirect().RouteValues;
    7
    8     Assert.AreEqual("Details", routeValueDictionary["action"]);
```

```
    Assert.AreEqual(1, routeValueDictionary["id"]);
}
```

이 테스트 코드는 ThoughtController의 FindDetails() 액션 메서드를 테스트한다. 4번 라인에서는 Dictionary<String, Object> 타입을 상속하여 구현한 RouteValueDictionary 객체를 얻어온다. 그런 후 지정된 Id 속성 값을 갖는 Thought 객체가 올바르게 상세 보기 뷰로 이동했는지를 테스트한다. FindDetails() 메서드는 아래와 같이 구현한다.

```
Line 1  //
        // GET: /Thought/FindDetails?nameOfThought={name}
        
        public ActionResult FindDetails(string nameOfThought)
    5   {
          var thought = Thought.CurrentThoughts.
            Find(x => x.Name == nameOfThought);
        
          return RedirectToAction("Details",
   10       new { id = thought.Id });
        }
```

예제에서와 같이 간단한 람다식으로 관련된 Thought 객체의 Id 속성 값을 얻을 수 있다. 10번 라인에서는 해당 Id를 가지고 상세 보기 뷰로 리다이렉트한다. 이때 Id 속성 값을 전달하기 위해 익명 객체를 사용하는 방법을 눈여겨 보기 바란다. 이 부분은 앞에서 작성했던 LinkHelper 클래스와 같은 방식으로 MVC 전반에 걸쳐 RouteValueDictionary 객체를 보다 손쉽게 전달하기 위한 방법이다. 앞서 작성한 테스트 코드가 성공하는 것을 확인했으면 이제 jQuery를 이용하여 자동 완성 기능을 아래와 같이 구현해 보자.

```
Line 1  <asp:Content ID="indexHead" ContentPlaceHolderID="head"
          runat="server" >
        <title>홈페이지</title>
        <script type="text/javascript" language="javascript" >
    5     $(document).ready(function() {
            $("#searchThoughtsTextBox" ).
              autocomplete("Thought/Search" , { minChars: 1 });
            $("#searchButton" ).click(function() {
              window.location = "Thought/FindDetails?nameOfThought=" +
   10           escape($("#searchThoughtsTextBox" )[0].value);
              });
          });
        </script>
```

```
</asp:Content>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent"
    runat="server" >
  <!-- 일부 코드 생략 -->

  <h2>계획 검색</h2>
  <input id="searchThoughtsTextBox" name="title" type="text" />
  <input id="searchButton" type="submit" value="검색" />
</asp:Content>
```

가장 먼저 해야 할 일은 21번 라인과 같이 검색어를 입력하는 텍스트 상자와 전송 버튼을 추가하는 것이다. 앞서 우리는 우아한 기능 저하에 대해 알아본 바 있다. 이 예제에는 자바스크립트를 지원하지 않는 브라우저도 지원할 수 있는데, 그 이유는 사용자가 Thought 객체의 이름을 입력할 수 있기 때문이다. 이 검색 기능에 우아한 기능 저하를 완벽하게 구현하려면 지금보다는 더 많은 작업을 해야 하지만, 중요한 것은 실제로 서비스할 웹 애플리케이션을 구현하는 데 있어 이러한 부분을 고려해야 한다는 것이다.

페이지에 텍스트 상자를 추가했으면 7번 라인과 같이 자동 완성 플러그인을 적용할 수 있다. 이 플러그인은 몇 가지 옵션을 더 제공하지만, 우리는 지금 당장은 기본적인 사용법에만 관심이 있다. 예제에서는 HTTP GET 요청을 수행할 URL을 지정하고 있으며, 사용자가 한 글자 이상 입력할 때 자동 완성 기능을 수행하기 위해 {minChars: 1} 옵션을 지정하였다. 이로써 자동 완성 기능을 완성하였지만, 우리는 전송 버튼을 클릭했을 때의 이벤트를 처리하여 브라우저가 상세 보기 페이지로 이동하도록 해야 한다. 이 부분은 9번 라인과 같이 jQuery를 사용하지 않고 window.location 속성을 이용하면 된다. 10번 라인에서는 자바스크립트의 getElementById() 메서드 대신 jQuery의 요소 선택자를 활용했다. jQuery 요소 선택자는 요소의 배열을 리턴하기 때문에 $(".class")[0]과 같은 방법으로 첫 번째 요소를 탐색하여 참조했다. 또 다른 자바스크립트 메서드인 escape() 메서드는 문자열을 URL로 인코딩한다. 자동 완성 기능이 동작하는 모습은 그림 7.1과 같다.

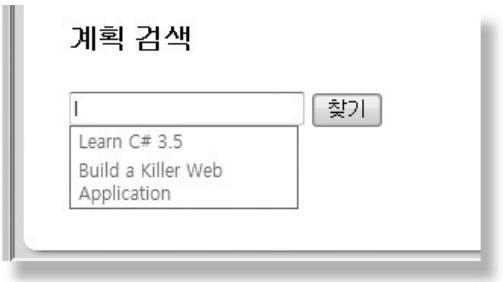

그림 7.1 자동 완성 기능을 이용하면 더욱 편리한 검색이 가능하다.

다른 유용한 jQuery 플러그인

자동 완성, jQuery-UI 및 색상 선택 플러그인(섹션 4.3 "jQuery를 이용하여 색상 대화상자 구현하기" 참고)는 수백 가지나 되는 jQuery 플러그인 중 세 가지일 뿐이다. 우리는 Ajax 요청을 생성하는 것부터 시각적으로 보다 인상적인 효과를 구현하기까지 필요한 모든 곳에 이 플러그인들을 활용할 수 있다(그림 7.2 참고).

플러그인도 훌륭하지만 MVC에도 시간과 중복 코드를 절약할 수 있는 멋진 기능들이 제공된다. 이제 MVC의 부분 뷰에 대해 알아보도록 하자.

기능	플러그인	URL
다중 파일 업로드	Uploadify	http://www.uploadify.com/
인라인 편집	In-line Text Edit	http://www.codenothing.com/archives/jquery/inline-text-edit/
드래그 앤 드롭	$.event.special.drag	http://blog.threedubmedia.com/2008/08/eventspecialdrag.html
클라이언트 측 유효성 검사	jQuery Validation	http://bassistance.de/jquery-plugins/jquery-plugin-validation/

그림 7.2 기능별 jQuery 플러그인

7.3 중복을 제거하기 위한 부분 뷰의 활용

우아한 기능 저하를 위해 우리는 궁극적으로 두 개의 서로 다른 페이지에 완전히 동일한 뷰를 구현하게 될 것이다. 이런 경우, Ajax와 함께 부분 뷰(Partial View)를 활용하면 문제를 손쉽게 해결할 수 있다.

부분 뷰는 뷰의 일부 조각과도 같다. ASP.NET에 익숙한 개발자라면 부분 뷰는 사용자 컨트롤(User Control)과 유사하다. 부분 뷰를 이용하면 뷰의 일부를 한 번만 작성하여 여러 뷰에서 공유하도록 할 수 있다. 이미 우리는 `Views/Shared` 디렉터리에 위치한 `LogOnUserControl.ascx`라는 이름의 부분 뷰를 사용한 경험이 있다. 이 부분 뷰는 로그인한 사용자의 이름을 렌더링하거나 계정 생성 링크 및 로그인 링크를 렌더링하며, `Site.Master` 페이지에 추가되어 있기 때문에 모든 뷰에서 사용이 가능하다. 이렇듯 부분 뷰를 이용하면 코드의 중복을 줄일 수 있을 뿐 아니라 코드의 유지보수도 더욱 쉬워진다.

그러면 부분 뷰에 대해 조금 더 자세히 살펴보도록 하자. GetOrganized 애플리케이션에 추가할 다음 기능은 `/Views/Todo/Create.aspx` 페이지를 벗어나지 않고 `Todo` 객체를 생성하는 기능이다. 이를 위한 반복 작업을 줄이려면 `/Views/Todo/Create.aspx` 페이지의 일부를 새 부분 뷰로 만드는 것이 좋다(DRY 원칙을 기억하기 바란다).

부분 뷰를 사용하도록 리팩토링하기

부분 뷰를 사용하도록 리팩토링하려면 `Create.aspx` 뷰의 거의 모든 코드를 새롭게 생성할 부분 뷰로 옮겨야 한다. 그런 후 `Create.aspx` 뷰가 부분 뷰를 렌더링하도록 수정하면 전체적인 뷰는 예전과 동일한 기능을 수행하게 될 것이다.

부분 뷰는 뷰 추가 마법사(그림 7.3 참고)를 이용하여 생성할 수 있다. 이 마법사 화면에서 새로 생성되는 뷰를 부분 뷰로 만들겠다는 의미로 첫 번째 체크 상자에 체크하면 된다. 이때도 마찬가지로 강력하게 형식화된 뷰를 생성할 수 있지만, 예제에서는 `Create.aspx` 뷰로부터 코드를 복사하여 `CreateElements.aspx` 뷰에 붙여 넣도록 하자.

TIP

보다 견고한 텍스트 검색

분명 텍스트 검색은 고객을 위한 B2C 사이트에서는 가장 중요한 기능 중 하나이다. 고객들이 원하는 도서를 손쉽게 검색할 수 있도록 지원하기 위해 아마존(Amazon)이 얼마나 많은 투자를 했을지 상상해 보면 그 중요성을 쉽게 알 수 있다. 아마존이나 구글이 제공하는 텍스트 검색을 활용할 수 없는 경우라 하더라도 우리가 사용할 수 있는 몇 가지 유용한 도구가 있다.

첫 번째 가장 전통적인 방법은 마이크로소프트 SQL 서버를 비롯한 대부분의 관계형 데이터베이스가 채택하고 있는 텍스트 인덱스 검색이다. 인덱스 검색의 결과는 쓸 만하지만 데이터베이스가 인덱스를 관리하기 위해 많은 비용이 소모된다. 이에 대한 보편적인 대안은 오픈 소스 프로젝트인 루신(Lucene)(http://lucene.apache.org/java/docs/index.html)을 활용하는 것이다. 루신은 개별적인 서버에서 텍스트 검색을 관리하기 때문에 인프라 관점에서 볼 때 텍스트 검색에 대한 수요의 증가에 따라 얼마든지 서버를 확장할 수 있다. 오렌 에이니(Oren Eini)는 루신과 NHibernate의 통합에 관한 글(http://ayende.com/Blog/archive/2007/03/18/Googlize-your-entities-NHibernate--Lucene.NET-Integration.aspx)을 통해 데이터베이스와 루신의 훌륭한 텍스트 검색 기능을 통합하는 방법을 소개했다.

```
Line 1  <%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
    -   <% using (Html.BeginForm()) { %>
    -   <fieldset>
    -     <legend>Fields</legend>
    5     <div class="editor-label">
    -       <%= Html.LablFor(model  => model.Title) %>
    -     </div>
    -     <div class="editor-field">
    -       <%= Html.TextBoxFor(model => model.Title) %>
   10       <%= Html.ValidationMessageFor(model => model.Title) %>
    -     </div>
    -     <p>
    -       <input type="submit" value="새로 만들기" />
    -     </p>
   15   </fieldset>
    -   <% } %>
```

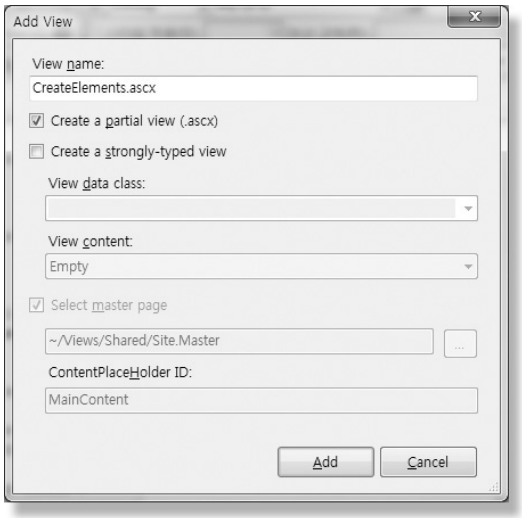

그림 7.3 부분 뷰를 사용하면 중복된 마크업을 줄일 수 있다.

예제에서는 텍스트 상자와 유효성 오류 메시지 및 전송 버튼을 포함한 HTML 폼 태그 전체를 부분 뷰로 이동하였다. 1번 라인을 보면 이 부분 뷰가 `System.Web.Mvc.ViewUserControl` 클래스를 상속한다는 것을 알 수 있다 이 클래스는 모든 부분 뷰의 기반 클래스이다. 폼 태그의 **`id`** 특성에 `CreateTodo`라는 값을 지정한 것을 제외하고는 수정한 코드는 없다. 이렇게 하면 jQuery를 이용하여 전송 버튼의 클릭 이벤트를 가로채 폼을 전송시키는 대신 Ajax 요청을 전송하도록 구현할 수 있다. 그러나 지금은 `Create.aspx` 뷰가 원래의 모습을 갖추도록 해야 한다.

```
Line 1  <asp:Content ID="Content2"
        ContentPlaceHolderID="MainContent" runat="server">
        
        <h2>새 할 일 생성</h2>
    5   <%= Html.ValidationSummary() %>
        
        <% Html.RenderPartial("CreateElements"); %>
        
        <div>
   10       <%= Html.ActionLink("목록으로", "Index") %>
        </div>
    </asp:Content>
```

부분 뷰를 렌더링하려면 7번 라인에서와 같이 RenderPartial(string partialName) 메서드를 호출하면 된다. 이때 /Views/Todo/CreateElements.aspx와 같이 부분 뷰의 전체 경로를 지정할 필요 없이 MVC 프레임워크의 규칙에 따라 뷰를 탐색하도록 뷰의 이름만을 지정한다. 기본 뷰 엔진인 WebFormViewEngine 클래스는 뷰를 찾기 위해 두 개의 디렉터리를 탐색한다. 첫 번째는 부분 뷰를 사용하는 뷰의 위치로 예제의 경우에는 /Views/Todo 디렉터리이며, 두 번째로는 /Views/Shared 디렉터리이다. 뷰 엔진은 MVC 프레임워크의 기능을 확장할 수 있는 또 다른 영역 중 하나이다(아래의 "무엇이든 물어보세요!" 박스 내용 참고).

무엇이든 물어보세요!

언제, 어떻게 다른 뷰 엔진을 활용해야 하나요?

NHaml(http://code.google.com/p/nhaml/)이나 StringTemplate(http://code.google.com/p/string-template-view-engine-mvc/)와 같은 뷰를 사용하는 것은 개인적인 취향의 문제이겠지만, 이런 뷰 엔진들을 사용할 경우에 명백한 장점이 존재한다. 우선 이들은 코드의 양을 획기적으로 줄일 수 있다. 동일한 뷰를 NHaml로 작성하면 자동 태그 닫힘과 같은 기능을 통해 더 적은 양의 코드로 뷰를 완성할 수 있다. 또 다른 장점은 뷰에 로직을 구현하지 못하도록 강제할 수 있다는 점이다. StringTemplate의 경우는 if/else 구문 같은 것이 제공되지 않기 때문에 모든 조건식은 뷰 헬퍼에 작성해야 한다.

물론 이런 뷰 엔진을 사용하거나 직접 뷰 엔진을 확장하는 방법에는 단점도 존재한다. 우선 대부분의 ASP.NET 개발자들은 NHaml이나 StringTemplate와 같은 뷰 엔진보다는 웹 폼 뷰 엔진에 훨씬 익숙하다. 만일 여러분의 팀이 경험이 많고 최종적으로 뷰를 구현하는 개발자들이 참고할 수 있는 문서를 작성해 줄 자신이 있다면, 다른 뷰 엔진을 사용하는 것이 좋을 수도 있다.

지금 당장은 필요하지 않지만, 부분 뷰는 모델이나 ViewData 컬렉션을 매개변수로 활용할 수 있다. 이런 기능은 고객의 주소와 같은 모델 객체의 일부를 부분 뷰를 통해 렌더링할 때 효과적으로 이용할 수 있다. 이로써 부분 뷰를 구현하기 위한 모든 절차를 마쳤다. F5키를 눌러 Create.aspx 뷰가 원래의 모습과 동일하게 동작하는지 확인할 수 있다. 다음으로는 Ajax 기능을 추가해 보자.

Ajax 방식으로 새로운 데이터 생성하기

앞서 우리는 CreateElement.aspx라는 이름의 부분 뷰를 생성하고 일부 코드를 옮겨왔기 때문에 이제는 /Views/Todo/Index.aspx 뷰에서 직접 새로운 Todo 객체를 생성할 수 있는 기능을 제공하여 사용자 경험을 향상시킬 수 있다. 그러려면 /Views/Todo/Index.aspx 뷰에 숨겨진 ** ** 태그를 추가하고 여기에 부분 뷰를 렌더링하면 된다.

```
Line 1  <asp:Content ID="Content1"
        -   ContentPlaceHolderID="Head" runat="server">
        - <script type="text/javascript" language="javascript">
        -   $(document).ready(function() {
        -   // 앞서 작성한 스크립트는 생략
        -
        -     $("#Create_Link").click(function() {
        -         $("#Create_Div").slideToggle("slow");
        -     });
        -     $("#Create_Link")[0].href = "#";
        -   });
        - </script>
        - </asp:Content>
        - <asp:Content ID="Content2"
        -   ContentPlaceHolderID="MainContent" runat="server">
        - <!-- Grid 코드는 생략 -->
        - <p>
        -   <%= Html.ActionLink("새로 만들기", "Create",
        -     null, new { id="Create_Link" }) %>
        - </p>
        - <div id="Create_Div" style="display:none">
        -   <% Html.RenderPartial("CreateElements"); %>
        - </div>
        - </asp:Content>
```

22번 라인에서는 부분 뷰를 추가하고 **display** 특성 값을 none으로 지정하여 페이지에 나타나지 않도록 했다. 이렇게 하면 페이지를 처음 로드했을 때 해당 영역이 보이지 않게 된다. 그런 후 19번 라인에서 새로운 링크를 추가하여 사용자가 ** ** 태그를 보이거나 숨길 수 있도록 했다. 링크와 ** ** 태그에는 **id** 특성을 지정하였기 때문에 jQuery를 이용하여 제어할 수 있다.

8번 라인의 코드에서는 페이지의 링크에 클릭 이벤트를 처리할 함수를 정의한다. 이 함수는 slideToggle(var speed) 메서드를 호출하여 슬라이드 효과와 함께 ** ** 태그를 보이거나

숨긴다. 이 메서드가 유용한 점은 지정된 요소를 보이거나 숨기는 동작을 모두 수행한다는 점이다. 10번 라인에서는 링크의 참조를 자기 자신으로 설정하여 우아한 기능 저하를 구현하고 있다. 이렇게 하면 자바스크립트를 지원하지 않는 브라우저를 사용하는 사용자는 종전처럼 Create.aspx 뷰를 사용하면 된다. 그림 7.4에서는 /Views/Todo/Create.aspx 뷰에 새로 추가한 영역이 어떻게 동작하는지를 보여준다.

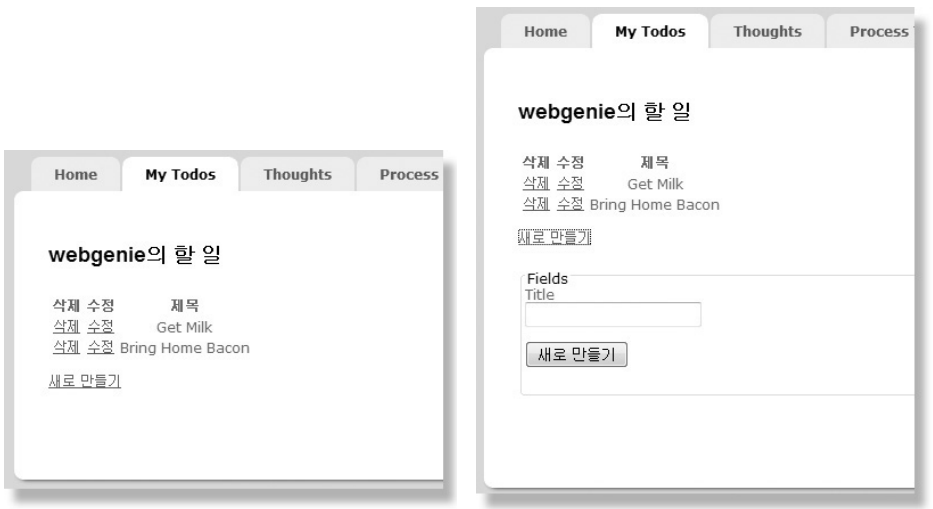

그림 7.4 jQuery의 토글 기능을 이용하면 페이지 요소가 차지하는 영역을 자유롭게 조절할 수 있다.

이제 페이지를 이동하지 않고도 새로운 데이터를 추가하는 기능이 올바르게 동작은 하지만, 새로운 Todo 객체를 생성할 때는 여전히 페이지가 다시 로드되는 문제가 있다. jQuery의 post() 메서드를 사용하기에 앞서 TodoController 클래스가 JSON을 리턴하도록 아래와 같이 수정해야 한다.

```
Line 1  // POST: /Todo/Create
      - [AcceptVerbs(HttpVerbs.Post)]
      - public ActionResult Create(Todo todo)
      - {
      5   todo.Validate(ModelState);
      -
      -   if (ModelState.IsValid)
      -   {
```

```
    CreateTodo(todo);
    if (Request.IsAjaxRequest())
      return Json(todo);
    return RedirectToAction("Index");
  }
  else
  {
    return View();
  }
}
```

10번 라인에서는 IsAjaxRequest() 메서드를 호출하여 요청이 Ajax 방식으로 발생한 것인지를 확인한다. 이 메서드에 대해서는 섹션 7.1 "Ajax의 활용"에서 소개했지만, 이 메서드가 어떻게 Ajax 요청을 구별하는지에 대해서는 아직 설명하지 않았다. 웹 요청이 Ajax 방식으로 발생하게 되면 HTTP 헤더에 X-Requested-With: XMLHttpRequest라는 항목이 추가되어 웹 서버는 이를 통해 요청이 다른 방식으로 생성되었음을 알 수 있다. IsAjaxRequest() 메서드는 요청 헤더에 이 항목이 존재하면 true를 리턴한다. 예제의 경우, Ajax 요청이 발생하면 JSON 형식의 응답을 리턴한다.

마지막으로 해야 할 일은 Index.aspx 페이지에 jQuery 코드를 추가하여 페이지를 다시 로드하지 않고 Todo 객체를 생성할 수 있도록 하는 것이다.

```
Line 1 <asp:Content ID="Content1"
    ContentPlaceHolderID="head" runat="server">
  <script type="text/javascript" language="javascript">
  $(document).ready(function() {
    $("#CreateTodo").submit(function() {
      $.post(
      $(this).attr('action'),
      $("#CreateTodo").serialize(),

      function(data, textStatus) {
        var html =
        "<tr><td><a id=\"" + data.Title + "\"" +
        "class=\"deleteTodoLink\" href=\"#\">" +
        "삭제</a></td>" +
        "<td><a href=\"/Todo/Edit?Title="
        + data.Title + "\">수정</a></td>" +
        "<td>" + data.Title  + "</td></tr>";

        $(html).appendTo($("#main table")).
```

```
10          effect("highlight", {}, 3000);
-       },
-       //응답 형식
-       "json"
-     );
25      return false;
-     });
- }
```

이 코드는 새로운 Todo 객체를 생성한 후 HTML의 문서 객체 모델(DOM: Document Object Model)을 동적으로 추가하여 페이지를 다시 로드하지 않고도 새로 추가된 데이터를 표시할 수 있도록 구현한 것이다.

우선 5번 라인에서와 같이 전송 버튼의 submit 이벤트를 가로챈다. 그런 후 jQuery의 post() 메서드를 이용하여 HTML 폼의 **action** 특성에 지정된 URL로 Ajax 요청을 보낸다. 이때 serialize()라는 특수한 메서드가 사용되었는데, 이 메서드는 폼 내의 **<input>** 태그들의 값을 쿼리 문자열로 변환해 준다. 8번 라인에서는 이 메서드를 이용하여 post() 메서드의 두 번째 매개변수를 만들어 전달한다.

이 예제에서 post() 메서드를 사용하는 방법은 Todo 객체를 삭제할 때의 방법과는 조금 다르다. 이번 예제에서는 11번 라인에서와 같이 서버로부터의 콜백 함수를 정의하고 있다. 이 메서드는 JSON 형식의 Todo 객체를 전달받아 목록에 새로운 행을 추가한다. 이 새로운 행은 20번 라인에서와 같이 jQuery의 강조 효과를 적용하여 추가된다. 서버로부터의 응답이 JSON 형식이어야 하기 때문에 23번 라인에서와 같이 post() 메서드의 세 번째 매개변수를 지정해 주어야 한다.

이제 페이지를 다시 로드하지 않고도 새로운 데이터를 추가할 수 있게 되었다. 새로 생성된 행을 강조하는 효과는 사용자에게 새로운 데이터가 생성되었음을 명확하게 알려줄 수 있다. 그런데 새로 추가된 행의 삭제 링크를 클릭하면 아무런 일도 발생하지 않는다. 그 이유는 우리가 jQuery를 이용하여 작성한 삭제 함수는 페이지가 처음 로드되는 시점에 이미 존재하는 데이터에만 연결되기 때문이다. 다행히 jQuery는 라이브 쿼리(Live Query)라는 매우 유명한 플러그인이 제공되어 이러한 문제를 손쉽게 해결할 수 있다.

```
Line 1  <script type="text/javascript" language="javascript">
-       $(document).ready(function() {
-         $(".deleteTodoLink").livequery('click', function() {
-           var element = $(this);
```

```
5       var todoTitle = element.attr("id");
-
-       $.post(
-         "Todo/Delete",
-         { title : todoTitle },
10        function() {
-           element.closest("tr").
-           fadeOut("slow", function()
-           { $(this).remove(); });
-         }
15      );
-     });
-   });
-   </script>
-   </asp:Content>
```

3번 라인에서는 `click(function())` 메서드 호출을 `livequery('click', function())` 메서드 호출로 변경하였다. 라이브 쿼리는 DOM의 변경사항을 탐지하기 위해서 `append()`나 `addClass()`와 같이 DOM 객체를 조작하는 jQuery 메서드의 호출을 모니터링하고, 필요한 경우는 이벤트를 다시 연결한다. DOM 객체가 jQuery 메서드나 플러그인을 통해 변경되면 라이브 쿼리가 여러분을 대신해 이벤트를 다시 연결해 줄 것이다.

이것으로 추가와 삭제 기능을 Ajax를 이용해 완벽하게 구현하였다. 웹 2.0이여, 영원하길!

다음 장에서는

이번 장에서 우리는 jQuery의 `$.post()` 메서드를 사용하는 방법을 학습했다. 또한 자동 완성과 같이 개발자의 생산성을 향상시켜 줄 수 있는 여러 가지 플러그인에 대해서도 알아보았다. 또한 Ajax 기능을 추가하는 과정에서 부분 뷰를 통해 마크업 코드의 중복을 제거하는 방법도 살펴보았다. 상태나 응용 프로그램의 로직이 구현되어 있지 않다는 점을 제외하면 부분 뷰는 ASP.NET의 사용자 컨트롤과 유사하다는 점을 기억하도록 하자.

다음 장에서는 목록 페이지의 보기 흉한 코드를 대체할 ORM에 대해 알아볼 것이다. 그렇다고 SQL 기술을 다시 공부할 필요는 없다. 왜냐하면 우리는 NHibernate라는 걸출한 ORM 도구를 이용할 것이기 때문이다.