

1장 차례

| | |
|---|----|
| Chapter 1 NerdDinner 애플리케이션..... | 4 |
| NerdDinner | 4 |
| 파일 -> 새 프로젝트..... | 11 |
| NerdDinner 애플리케이션의 디렉터리 구조..... | 13 |
| NerdDinner 애플리케이션 실행하기..... | 18 |
| NerdDinner 애플리케이션의 테스트..... | 21 |
| 데이터베이스 생성하기..... | 22 |
| SQL Server Express 데이터베이스 생성하기..... | 23 |
| 데이터베이스에 테이블 생성하기..... | 25 |
| 테이블 간의 외래 키 관계 설정하기..... | 29 |
| 테이블에 데이터 추가하기..... | 32 |
| 모델 객체 구현하기..... | 33 |
| LINQ to SQL..... | 34 |
| 프로젝트에 LINQ to SQL 클래스 추가하기..... | 34 |
| LINQ to SQL을 이용하여 데이터 모델 클래스 생성하기..... | 37 |
| NerdDinnerDataContext 클래스..... | 41 |
| DinnerRepository 클래스 구현하기..... | 42 |
| DinnerRepository 클래스를 이용하여 데이터베이스 작업 수행하기..... | 45 |
| 데이터 조회 예제..... | 45 |
| 데이터의 추가와 수정 예제..... | 45 |
| 데이터 삭제 예제..... | 46 |
| 모델 클래스에 유효성 검사 및 비즈니스 규칙 추가하기..... | 47 |
| 스키마 유효성 검사..... | 47 |
| 유효성 검사와 비즈니스 규칙..... | 47 |
| 컨트롤러와 뷰..... | 51 |
| DinnersController 컨트롤러 클래스 추가하기..... | 52 |
| Index()와 Details() 액션 메서드 추가하기..... | 54 |
| ASP.NET MVC의 URL 라우팅..... | 55 |
| DinnerController 클래스에서 DinnerRepository 클래스 활용하기..... | 57 |
| 뷰 활용하기..... | 58 |
| “NotFound” 뷰 템플릿 구현하기..... | 60 |
| “Details” 뷰 템플릿 구현하기..... | 64 |
| “Index” 뷰 템플릿 구현하기..... | 70 |
| Views 디렉터리의 구조와 뷰의 명명 규칙..... | 77 |
| 데이터의 생성, 수정 및 삭제 양식을 위한 시나리오..... | 79 |

| | |
|---|-----|
| DinnersController에 의해 처리되는 URL들..... | 79 |
| HTTP-GET 방식을 위한 Edit 액션 메서드 구현하기..... | 80 |
| Html.BeginForm() 메서드와 Html.EndForm() 메서드..... | 86 |
| Html.BeginForm() 메서드..... | 86 |
| Html.TextBox() 메서드..... | 87 |
| HTTP-POST 방식을 위한 Edit 액션 메서드 구현하기..... | 88 |
| 전송된 값의 조회..... | 89 |
| 오류의 처리..... | 91 |
| 유효성 검사에 관련된 메서드들과 ModelState 속성..... | 94 |
| ModelState 속성과 HTML 헬퍼 메서드들..... | 94 |
| Html.ValidationMessage() 메서드..... | 95 |
| Html.ValidationSummary() 메서드..... | 96 |
| AddModelError 메서드 활용하기..... | 96 |
| 완벽하게 구현된 Edit 액션 메서드..... | 97 |
| HTTP GET 방식을 위한 Create 액션 메서드 구현하기..... | 98 |
| HTTP POST 방식을 위한 Create 액션 메서드 구현하기..... | 102 |
| HTTP GET 방식을 위한 Delete 액션 메서드 구현하기..... | 106 |
| HTTP POST 방식을 위한 Delete 액션 메서드 구현하기..... | 109 |
| 모델 바인딩시의 보안에 대한 고려..... | 111 |
| 사용 방식 기반의 바인딩 제한..... | 111 |
| 타입 기반의 바인딩 제한..... | 112 |
| CRUD 기능을 통합하기..... | 112 |
| ViewData와 ViewModel..... | 114 |
| 컨트롤러에서 뷰 템플릿으로 데이터 전달하기..... | 114 |
| ViewData 속성의 활용..... | 115 |
| ViewModel 패턴의 활용..... | 117 |
| 맞춤형 ViewModel 클래스들..... | 120 |
| 부분 뷰와 마스터 페이지..... | 120 |
| Edit과 Create 뷰 템플릿 다시 보기..... | 121 |
| 부분 뷰 템플릿의 활용..... | 122 |
| 부분 뷰 템플릿으로 깔끔한 코드 유지하기..... | 126 |
| 마스터 페이지..... | 127 |
| 페이징 기능 구현하기..... | 131 |
| Index() 액션 메서드 다시 살펴보기..... | 132 |
| IQueryable<T> 인터페이스 이해하기..... | 132 |
| URL에 "page" 변수 추가하기..... | 133 |
| 쿼리 문자열 활용하기..... | 134 |
| URL에 포함된 값 활용하기..... | 134 |
| 페이지 이동을 위한 UI 추가하기..... | 137 |

| | |
|--|-----|
| 인증과 권한 설정 | 142 |
| 인증과 권한 부여 | 142 |
| 폼 인증과 AccountController 클래스 | 143 |
| [Authorize] 필터로 /Dinners/Create URL에 인증 구현하기 | 148 |
| 새로운 모임 데이터를 추가할 때 User.Identity.Name 속성 활용하기 | 150 |
| 기존의 모임 데이터를 수정할 때 User.Identity.Name 속성 활용하기 | 151 |
| 수정/삭제 링크를 보이거나 숨기기 | 153 |
| 모임 참여 기능에 AJAX 적용하기 | 154 |
| 사용자가 이미 참여 중인지 확인하기 | 154 |
| Register 액션 메서드 구현하기 | 157 |
| AJAX 방식으로 Register 액션 메서드 호출하기 | 158 |
| jQuery를 이용한 애니메이션 추가하기 | 161 |
| 코드 정리 - RSVP 부분 뷰 템플릿 리팩토링하기 | 163 |
| AJAX를 이용하여 지도 통합하기 | 163 |
| 지도를 위한 부분 뷰 템플릿 생성하기 | 163 |
| Map.js 유틸리티 라이브러리 구현하기 | 165 |
| 모임 생성 및 수정 페이지에 지도 기능을 추가하기 | 166 |
| 모임 정보 상세 보기 페이지에 지도 추가하기 | 172 |
| 데이터베이스와 저장소 클래스에서 위치를 검색하기 | 174 |
| JSON을 이용한 AJAX 검색 메서드 구현하기 | 178 |
| jQuery를 이용하여 JSON 기반의 AJAX 메서드 호출하기 | 179 |
| 단위 테스트 수행하기 | 183 |

Chapter 1 NerdDinner 애플리케이션

NerdDinner

새로운 프레임워크를 학습하는 가장 좋은 방법은 그 새로운 프레임워크를 이용하여 무언가를 만들어 보는 방법이다. 이 책의 첫 번째 장에서는 ASP.NET MVC를 이용하여 작지만 완벽하게 동작하는 애플리케이션을 개발하는 방법을 설명하며 그 배경에 깔려있는 핵심 개념들에 대해서 소개하고자 한다.

우리가 구현할 "NerdDinner"라는 애플리케이션은 사람들이 저녁 모임 약속을 온라인 상에서 손쉽게 검색하거나 정리할 수 있는 애플리케이션이다.

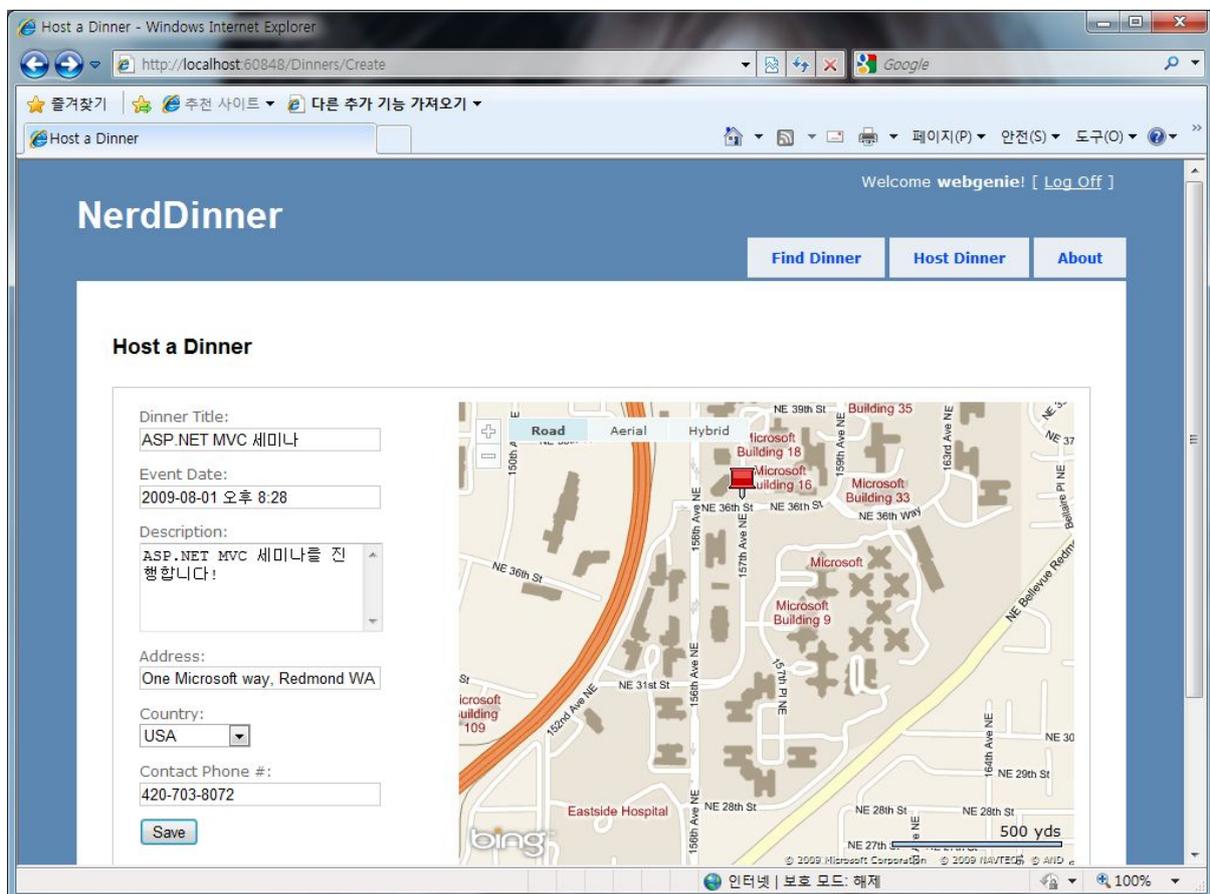


그림 1-1

NerdDinner 애플리케이션 사용자들은 저녁 모임을 생성하거나 편집 혹은 삭제할 수 있다. 또한 애플리케이션 전반에 걸쳐 탄탄한 유효성 검사와 비즈니스 규칙이 적용되어 있다.

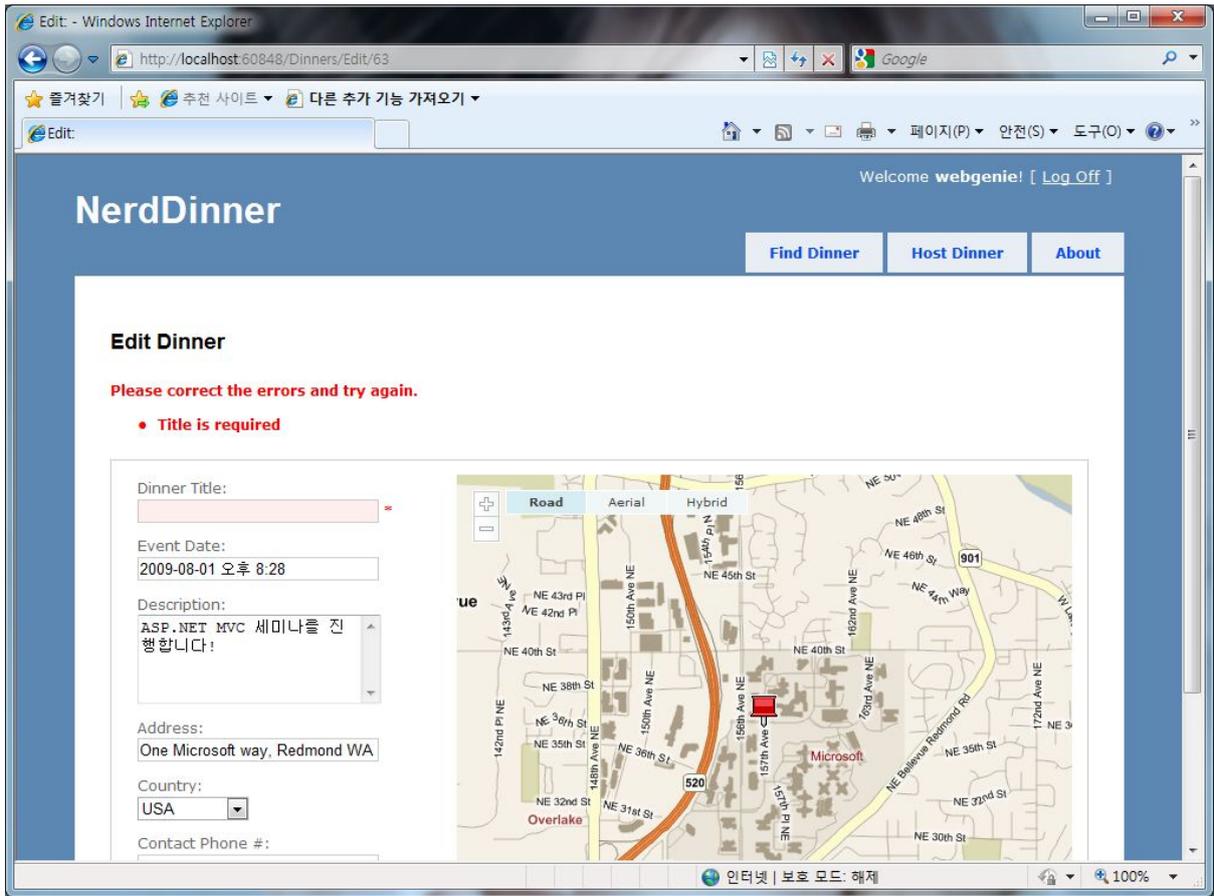


그림 1-2

이 사이트에 방문한 사용자들은 예정된 저녁 모임을 손쉽게 검색할 수도 있다.

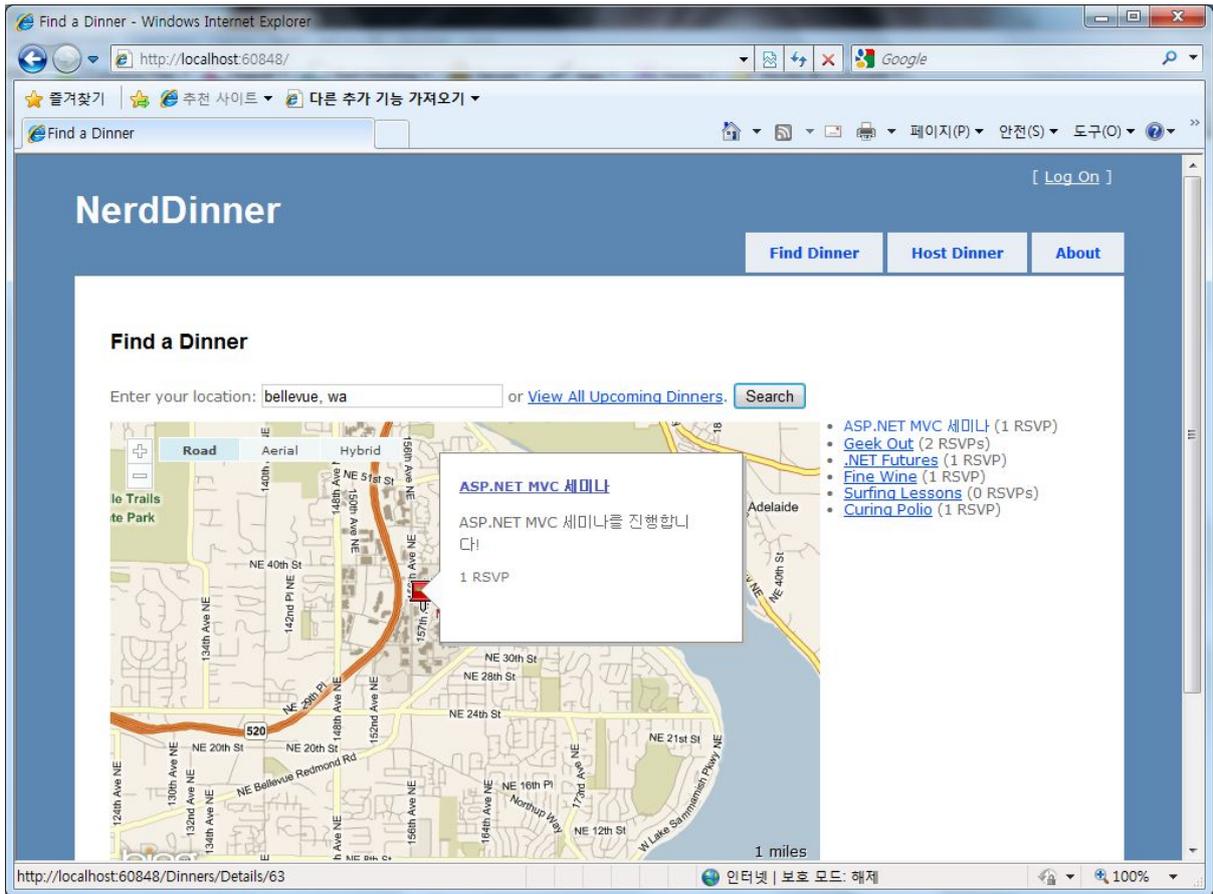


그림 1-3

저녁 모임 항목을 클릭하면 더욱 상세한 정보를 보여주는 상세 보기 페이지로 이동한다.

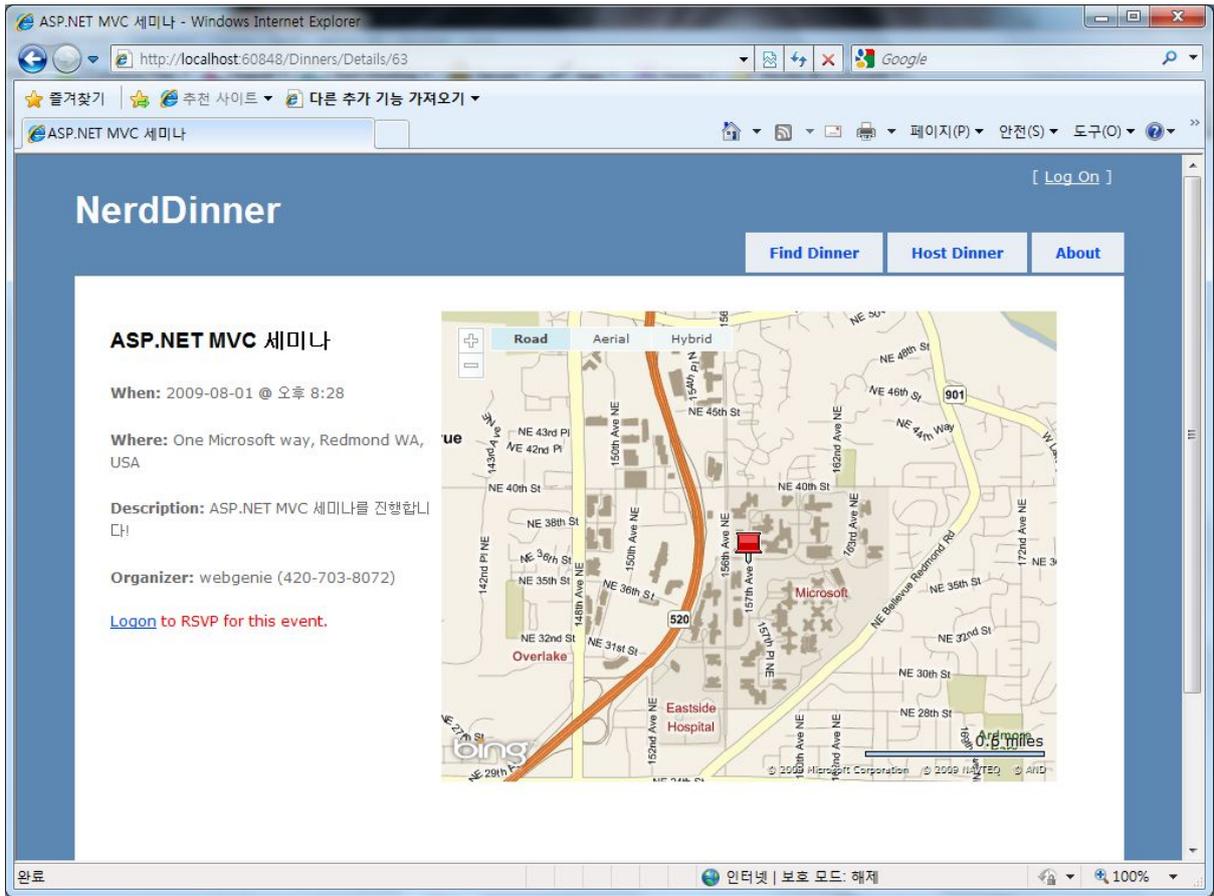


그림 1-4

사용자들이 저녁 모임에 참석하고자 한다면 로그인을 하거나 혹은 회원으로 가입을 해야 한다.

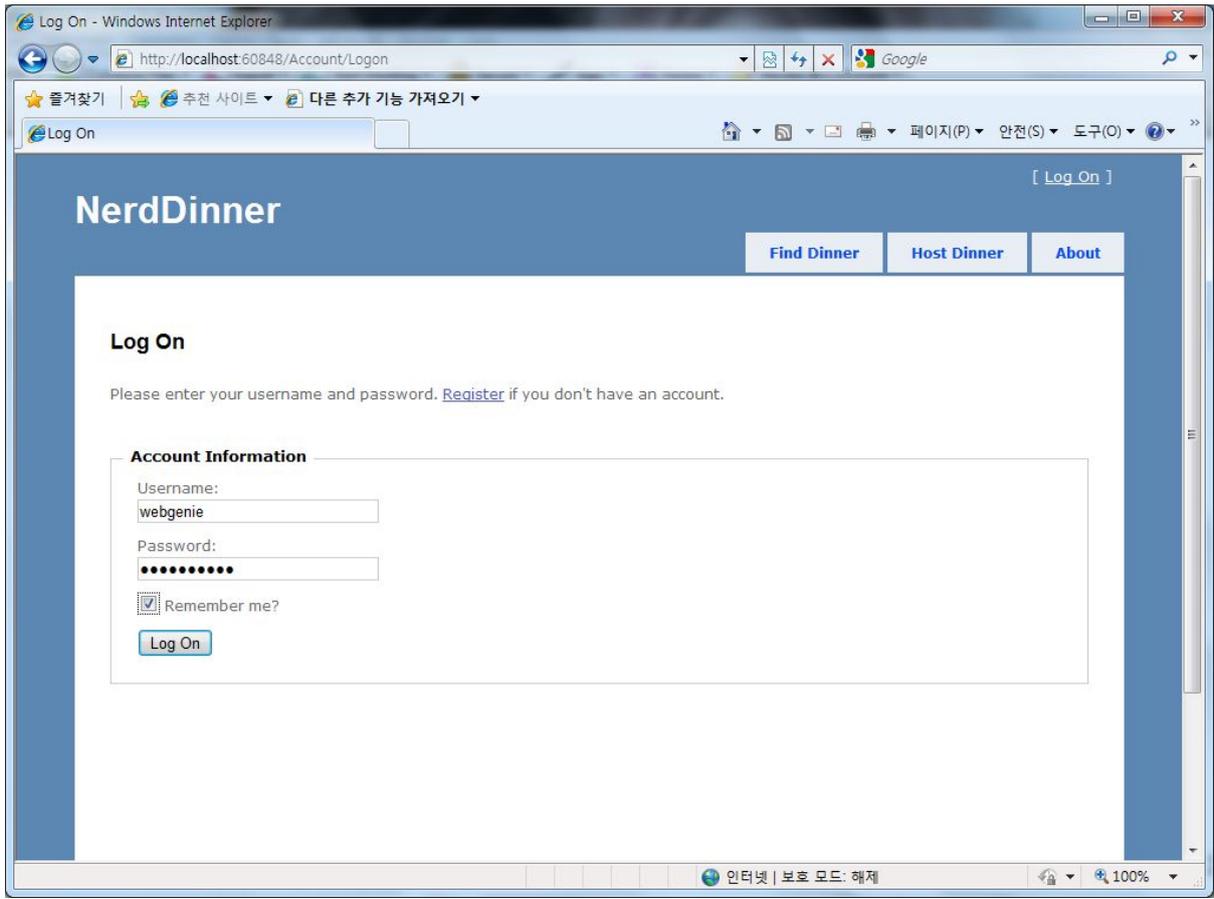


그림 1-5

또한 해당 약속에 참석하기 위해 손쉽게 응답을 남길 수도 있다.

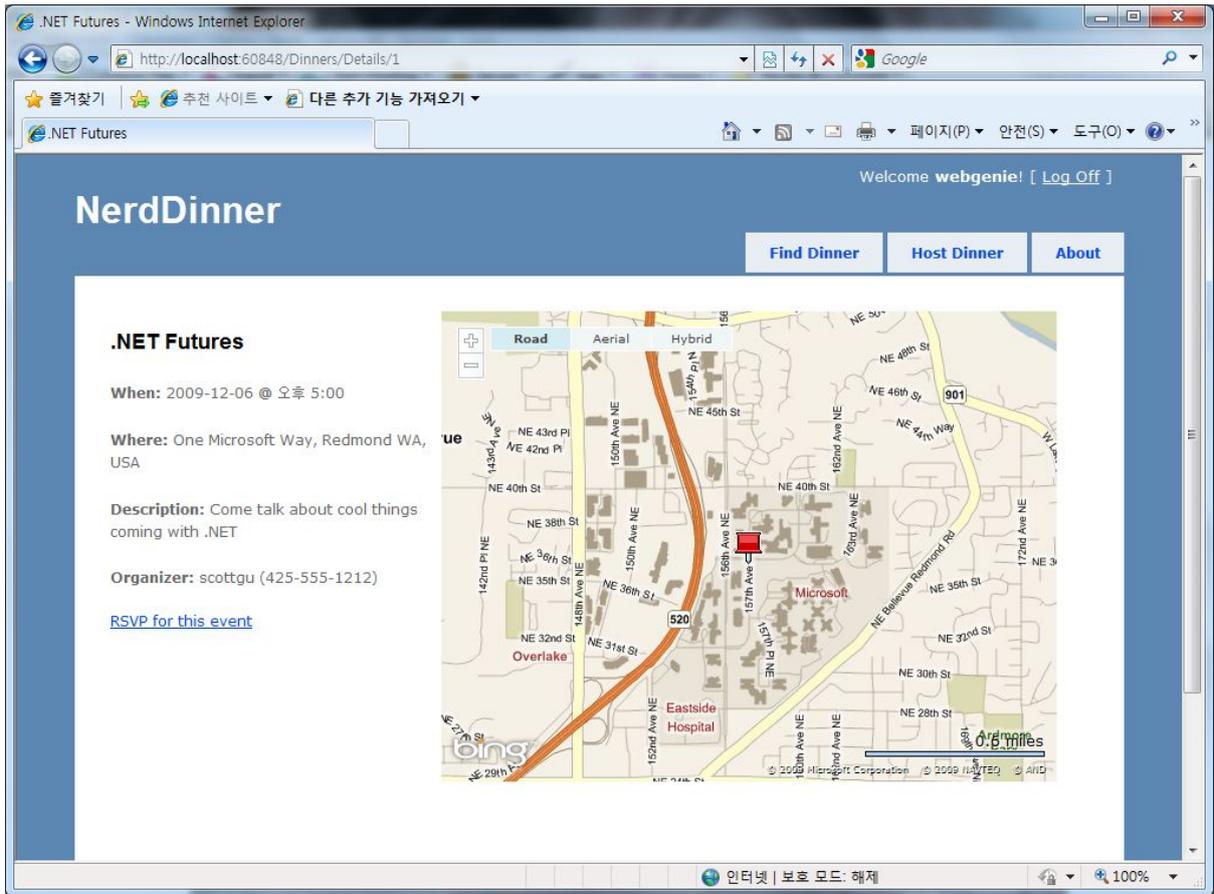


그림 1-6

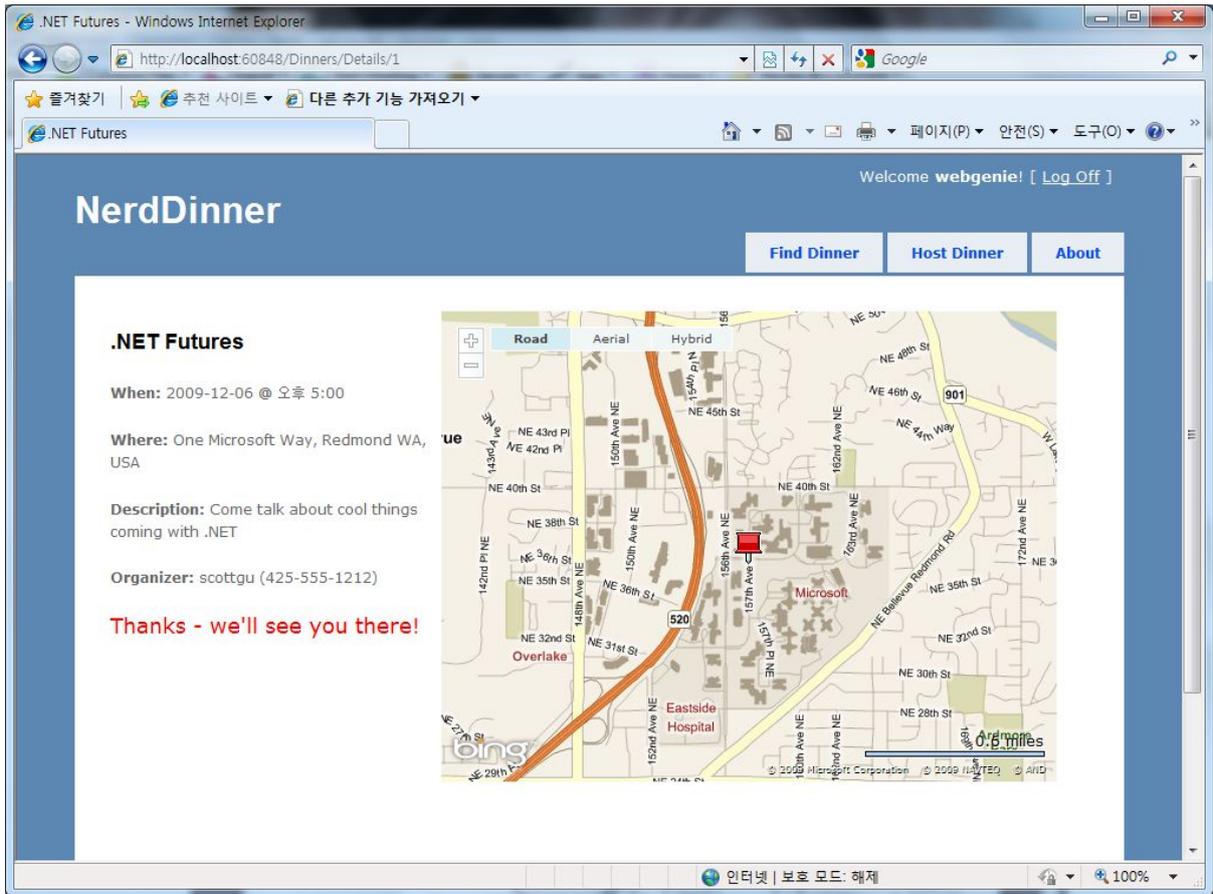


그림 1-7

NerdDinner 애플리케이션을 구현하려면 Visual Studio에서 [파일 > 새 프로젝트] 항목을 선택하여 새로운 ASP.NET MVC 프로젝트를 생성하면 된다. 그런 후에는 차차 필요한 기능들을 구현해 나갈 것이다. 물론 그 과정에서 데이터베이스의 생성, 비즈니스 규칙에 대한 유효성 검사를 수행할 수 있는 모델 객체의 구현, 데이터의 나열 및 상세 보기 UI의 구현, CRUD(Create, Read, Update, Delete: 추가, 조회, 수정, 삭제) 양식을 위한 항목들의 제공, 효과적인 데이터 페이징, 마스터 페이지와 Partial 뷰를 이용한 UI의 재사용, 인증 및 권한을 이용한 애플리케이션 보안, 동적 업데이트와 인터랙티브 지도의 지원을 위한 AJAX의 활용, 그리고 자동화된 단위 테스트의 구현 등을 소개할 것이다.

이번 장에서 살펴볼 NerdDinner 애플리케이션은 여러분이 직접 구현할 수도 있다. 혹은 원한다면 <http://tinyurl.com/aspnetmvc>에서 전체 소스 코드를 다운로드할 수도 있다.

여러분은 Visual Studio 2008뿐만 아니라 무료로 제공되는 Visual Web Developer 2008 Express를 사용하여 애플리케이션을 구현할 수 있다. 또한 SQL Server 혹은 무료로 제공되는 SQL Server Express를 사용하여 데이터베이스를 서비스할 수 있다.

ASP.NET MVC와 Visual Web Developer 2008 그리고 SQL Server Express는 마이크로소프트의 Web Platform Installer를 이용하여 설치할 수도 있다. Web Platform Installer는

http://www.microsoft.com/web/downloads에서 다운로드할 수 있다.

파일 -> 새 프로젝트

자, 그러면 Visual Studio 2008 혹은 무료 버전인 Visual Web Developer 2008 Express를 실행하고 [파일 > 새 프로젝트] 메뉴를 선택하여 NerdDinner 애플리케이션의 개발을 시작해보자.

[새 프로젝트] 대화 상자가 나타나면 새로운 ASP.NET MVC 애플리케이션을 생성하기 위해 대화 상자 왼쪽의 “웹” 노드를 선택하고 오른쪽의 프로젝트 템플릿 항목에서 “ASP.NET MVC Web Application” 템플릿을 선택한다.

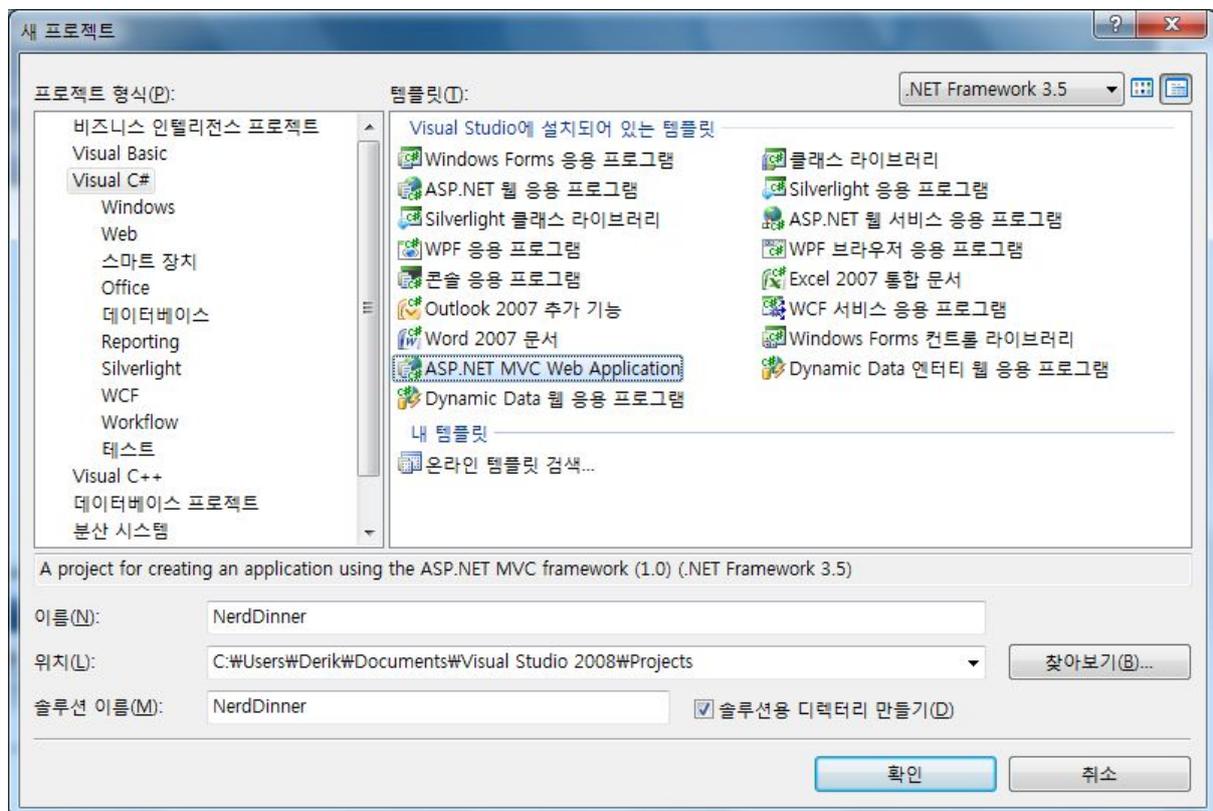


그림 1-8

새로운 프로젝트의 이름에 “NerdDinner”를 입력하고 [확인] 버튼을 클릭하면 프로젝트가 생성된다.

[확인] 버튼을 클릭하면 Visual Studio는 새로운 애플리케이션을 위한 단위 테스트 프로젝트를 함께 생성할 것인지를 묻는 대화 상자를 보여준다. 이 단위 테스트 프로젝트를 이용하면 우리가 구현할 애플리케이션의 기능과 동작을 검사할 수 있는 테스트들을 자동으로 구현할 수 있다. (물론 나중에 다시 설명할 것이다.)

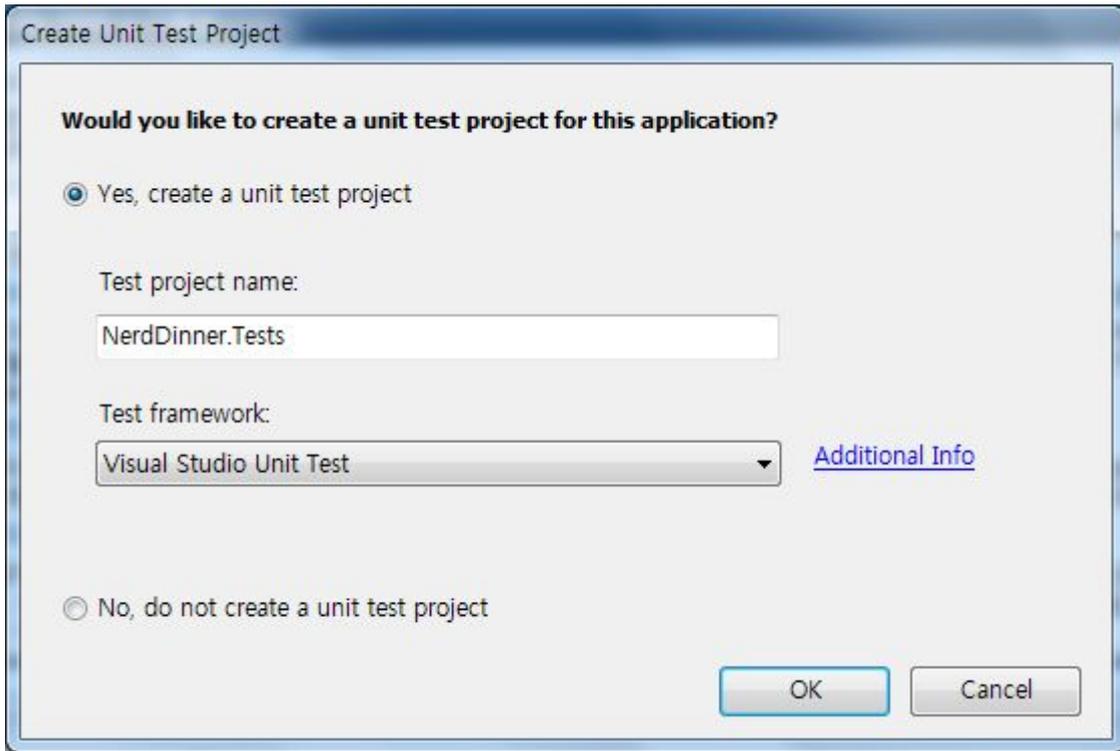


그림 1-9

위 그림의 대화 상자에서 “Test framework” 드롭다운 목록은 현재 시스템에 설치된 사용 가능한 모든 ASP.NET MVC 단위 테스트 프로젝트 템플릿을 보여준다. NUnit과 MBUnit, XUnit 등이 지원되며 Visual Studio에 내장된 단위 테스트 프레임워크 역시 지원된다.

=====
Note: Visual Studio의 단위 테스트 프레임워크는 Visual Studio 2008 Professional과 그 상위 버전에서만 사용할 수 있다. 만일 Visual Studio 2008 Standard 에디션이나 Visual Web Developer 2008 Express를 사용한다면 이 대화 상자에 NUnit이나 MBUnit 혹은 XUnit 프레임워크를 다운로드하여 설치해야 한다. 만일 설치된 테스트 프레임워크가 없다면 이 대화 상자는 아무것도 보여주지 않는다.
=====

테스트 프로젝트는 기본적으로 제공되는 “NerdDinner.Tests”라는 이름을 사용하며 “Visual Studio Unit Test” 프레임워크 옵션을 사용한다. [확인] 버튼을 클릭하면 Visual Studio는 웹 애플리케이션 프로젝트와 단위 테스트 프로젝트 등 두 개의 프로젝트가 포함된 솔루션을 생성하게 된다.

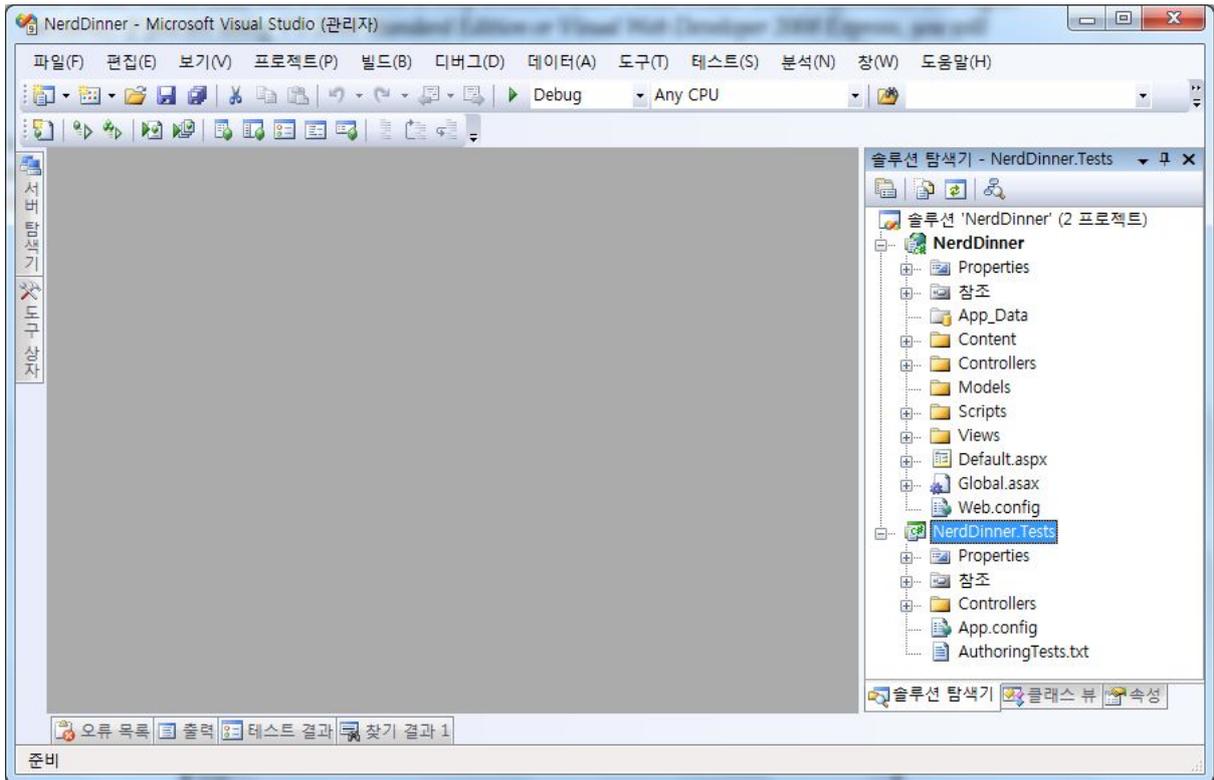


그림 1-10

NerdDinner 애플리케이션의 디렉터리 구조

Visual Studio를 사용하여 ASP.NET MVC 애플리케이션을 생성하면 자동으로 프로젝트에 몇 개의 파일과 디렉터리가 추가된다.

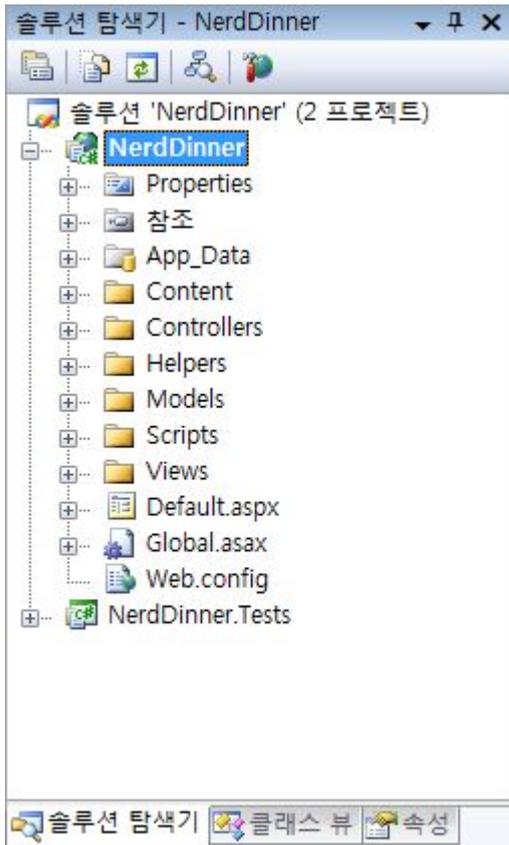


그림 1-11

ASP.NET MVC 프로젝트는 기본적으로 여섯 가지의 최상위 디렉터리를 정의하고 있다.

| 디렉터리 | 설명 |
|--------------|-------------------------------------|
| /Controllers | URL 요청을 처리할 컨트롤러 클래스들을 보관하는 디렉터리 |
| /Models | 데이터를 표현하고 조작하는 클래스들을 보관하는 디렉터리 |
| /Views | 페이지를 렌더링할 UI 템플릿 파일들을 보관하는 디렉터리 |
| /Scripts | 자바스크립트 라이브러리 파일들을 보관하는 디렉터리 |
| /Content | 이미지와 CSS를 포함한 정적 콘텐츠 파일들을 보관하는 디렉터리 |
| /App_Data | 읽거나 쓸 데이터 파일을 보관하는 디렉터리 |

ASP.NET MVC 프로젝트가 반드시 이와 같은 구조를 가질 필요는 없다. 사실 대형 애플리케이션을 개발하는 개발자들은 애플리케이션을 여러 개의 프로젝트로 나누어 보다 쉽게 관리하곤 한다. (예를 들어 데이터 모델 클래스들은 웹 애플리케이션 프로젝트와는 별개의 클래스 라이브러리 프로젝트에 구현하는 식이다). 그러나 이 기본적인 프로젝트 구조는 애플리케이션을 제법 깔끔하게 유지할 수 있는 쓸 만한 기본 디렉터리 규칙을 제공한다.

/Controllers 디렉터리를 확장해보면 Visual Studio가 HomeController와 AccountController라는 이름의 두 개의 컨트롤러 클래스를 기본적으로 프로젝트에 추가해둔 것을 발견할 수 있다.

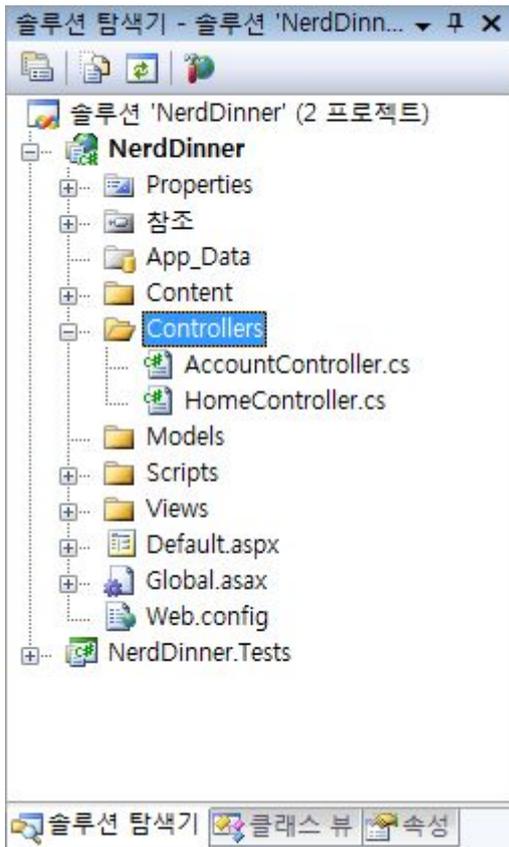


그림 1-12

또한 /Views 디렉터리를 확장해보면 프로젝트에 기본적으로 추가된 몇 개의 템플릿 파일들과 /Home, /Account, /Shared 등 세 개의 하위 디렉터리를 볼 수 있다.

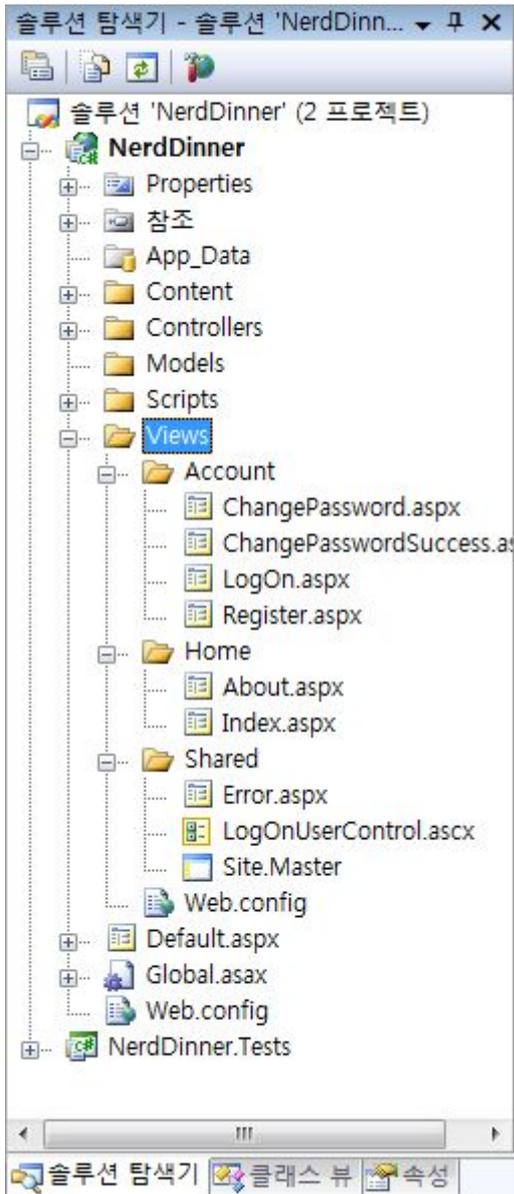


그림 1-13

/Content 디렉터리와 /Scripts 디렉터리를 확장해 보면 사이트의 모든 HTML에 적용될 스타일이 정의된 Site.css 파일과 애플리케이션에 ASP.NET AJAX 및 jQuery 지원을 가능하게 해주는 자바스크립트 라이브러리들이 포함되어 있음을 볼 수 있다.

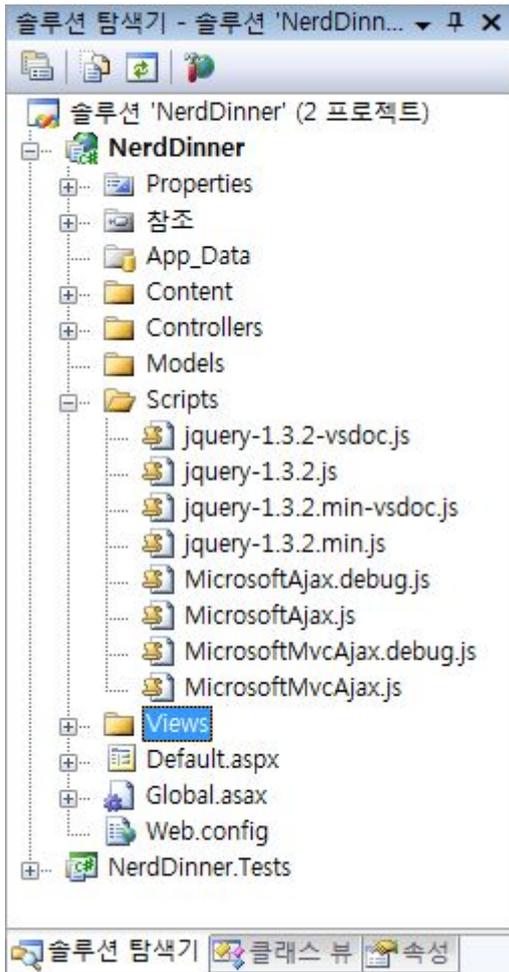


그림 1-14

이번에는 NerdDinner.Tests 프로젝트를 확장해보자. 이 프로젝트에는 컨트롤러 클래스들에 대한 단위 테스트가 구현된 두 개의 클래스가 포함되어 있다.

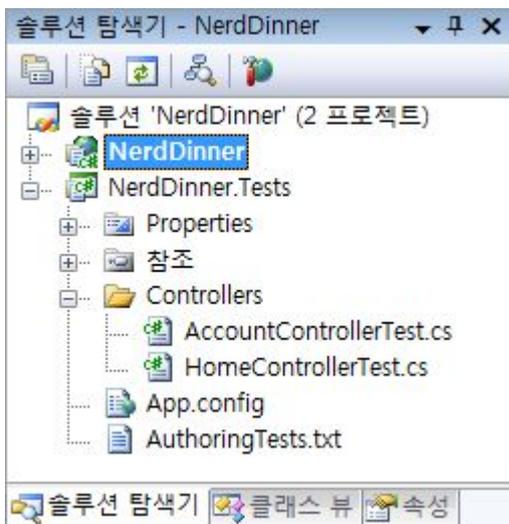


그림 1-15

지금까지 설명한 파일들은 실제로 동작하는 애플리케이션의 기본 구조를 제공하기 위해 Visual Studio에 의해 기본적으로 포함된다. 기본적으로 홈페이지와 소개 페이지, 계정 로그인/로그아웃 및 등록 페이지 그리고 처리되지 않은 예외를 위한 에러 페이지들이 포함되며 이들은 모두 완벽하게 동작한다.

NerdDinner 애플리케이션 실행하기

애플리케이션 프로젝트를 실행하려면 [디버그 > 디버깅 시작] 메뉴나 [디버그 > 디버깅하지 않고 시작] 메뉴를 선택하면 된다.

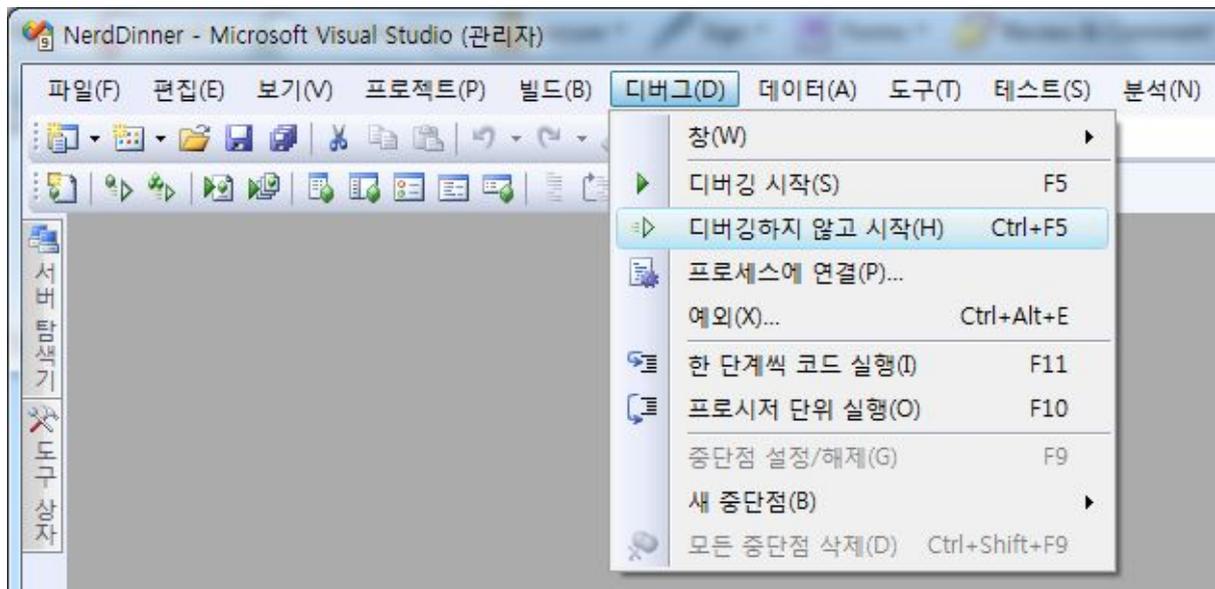


그림 1-16

그러면 다음 그림과 같이 Visual Studio에 내장된 ASP.NET 웹 서버가 실행되며 애플리케이션이 동작하게 된다.



그림 1-17

다음의 그림은 애플리케이션 프로젝트의 홈 페이지가 실행된 모습이다. (URL은 "/"이다.)

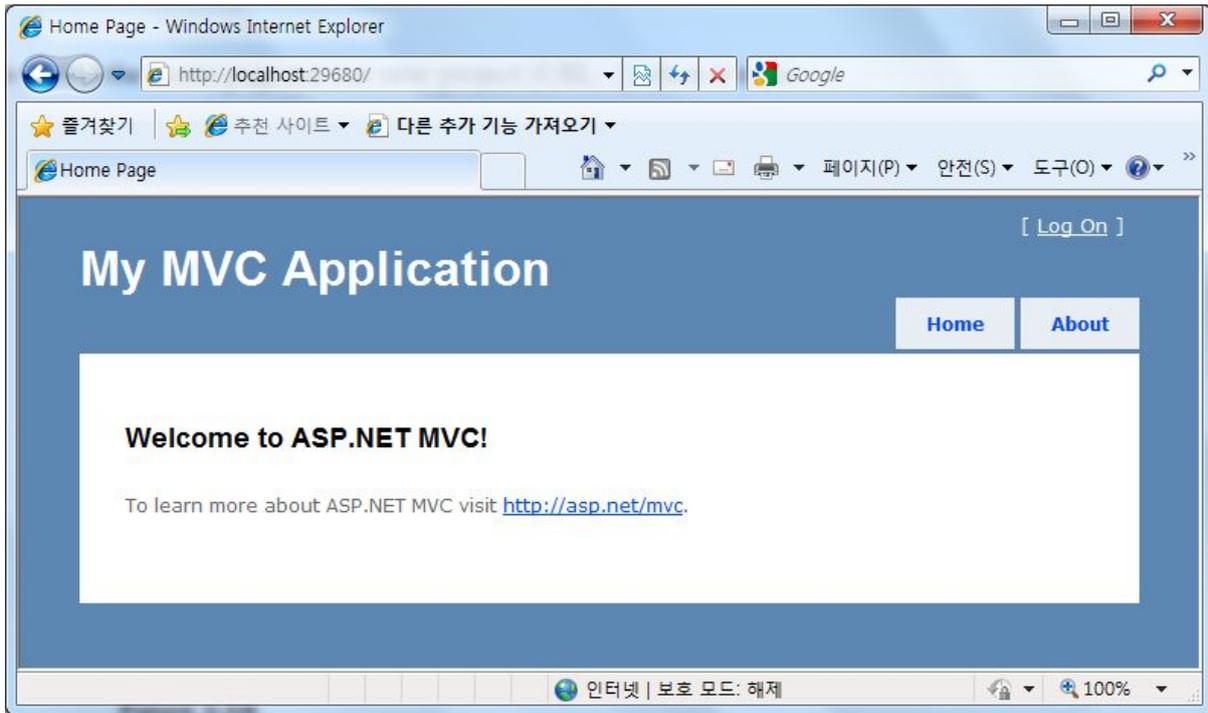


그림 1-18

[About] 메뉴를 클릭하면 소개 페이지(URL은 "/Home/About")로 이동하게 된다.

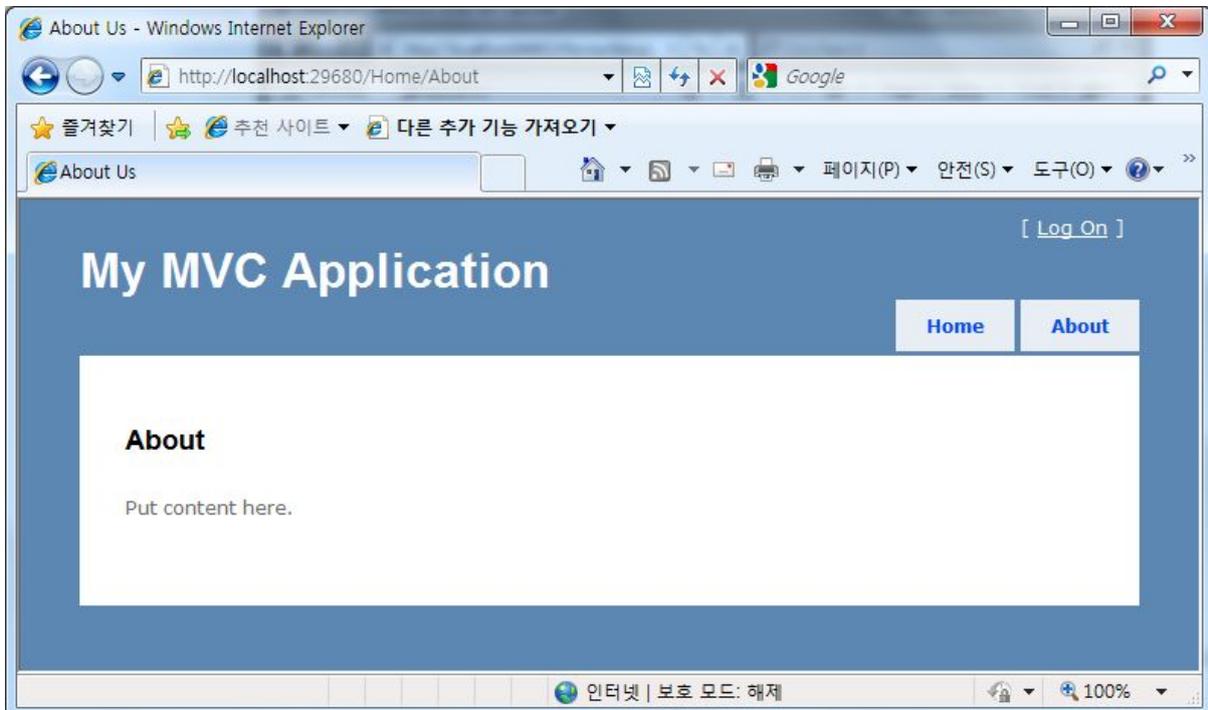


그림 1-19

우측 상단의 [Log On] 링크를 클릭하면 로그인 페이지(URL은 "/Account/Logon")로 이동한다.

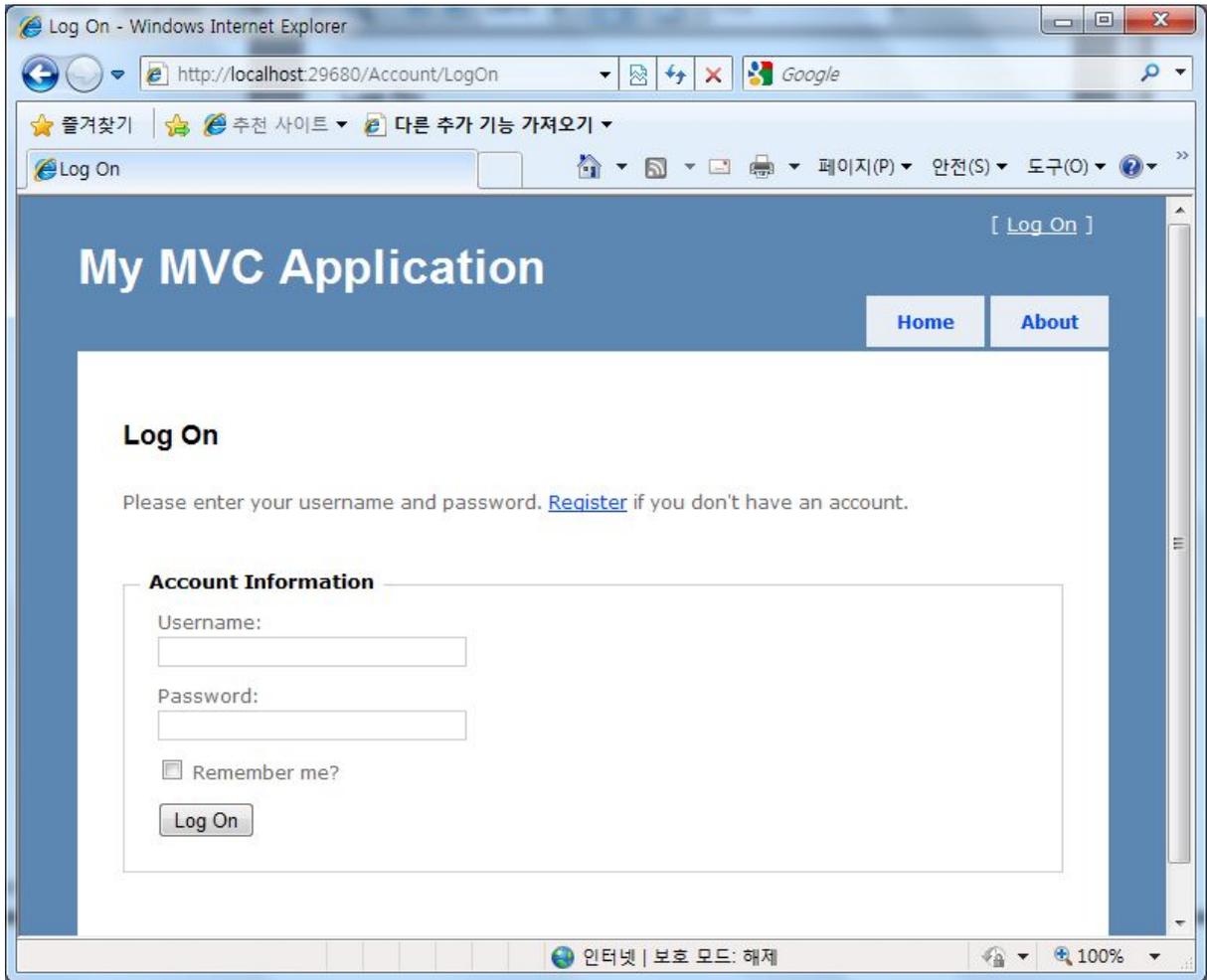


그림 1-20

만일 로그인 계정을 생성하지 않았다면 [Register] 링크를 클릭하여 계정 생성 페이지(URL은 "/Account/Register")로 이동한다.

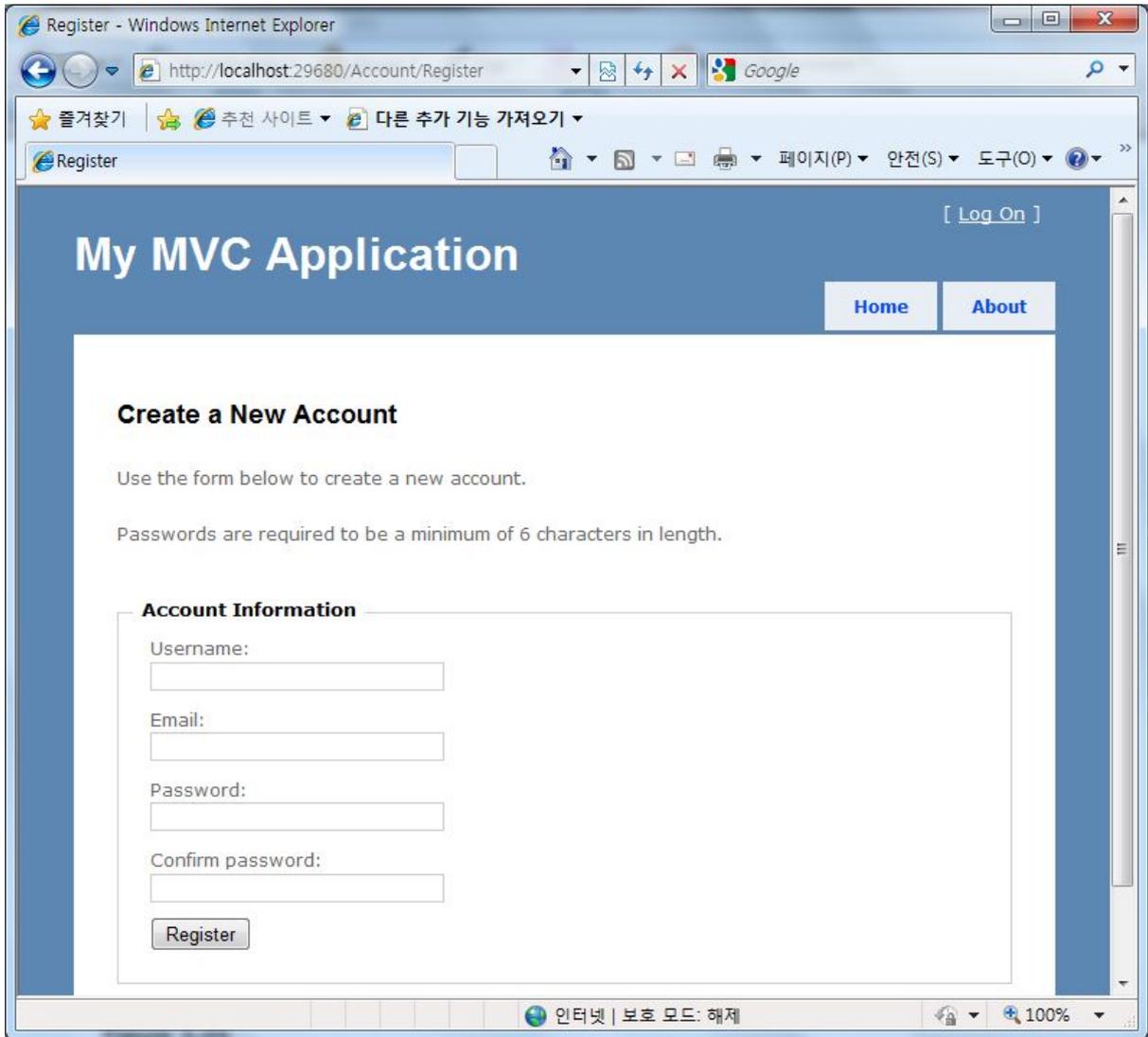


그림 1-21

지금까지 설명한 홈페이지와 소개 페이지, 그리고 로그인/로그아웃 기능들은 우리가 새로운 ASP.NET MVC 프로젝트를 생성할 때 자동으로 구현되며 이 기능들을 바탕으로 우리만의 애플리케이션을 개발하게 될 것이다.

NerdDinner 애플리케이션의 테스트

만일 Visual Studio 2008 Professional Edition 혹은 그 상위 버전을 사용 중이라면 프로젝트를 테스트하기 위해 Visual Studio에 내장된 단위 테스트 IDE를 사용할 수 있다.

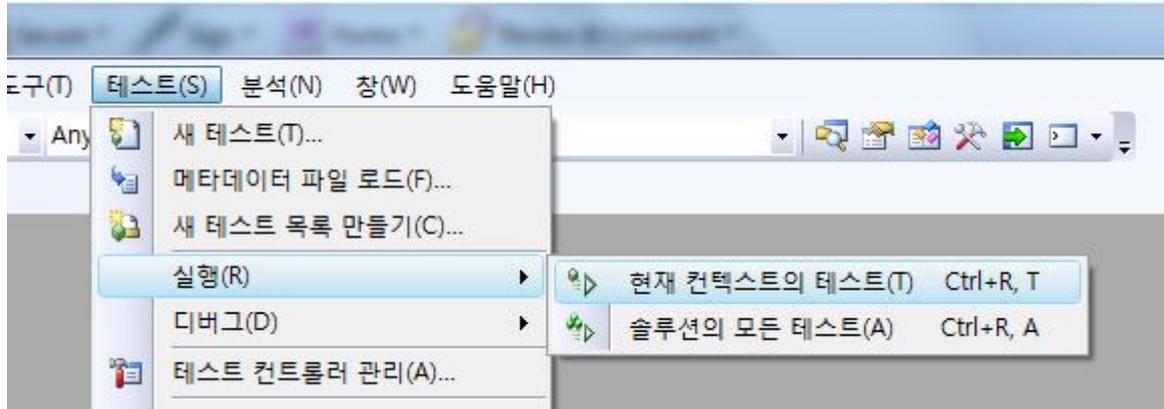


그림 1-22

그림에 나타난 옵션 중 하나를 선택하면 IDE 내에 “테스트 결과” 패널이 나타나게 되며 내장된 테스트 기능을 이용하여 우리의 프로젝트에 대한 27가지 단위 테스트를 수행한 결과를 보여주게 된다.

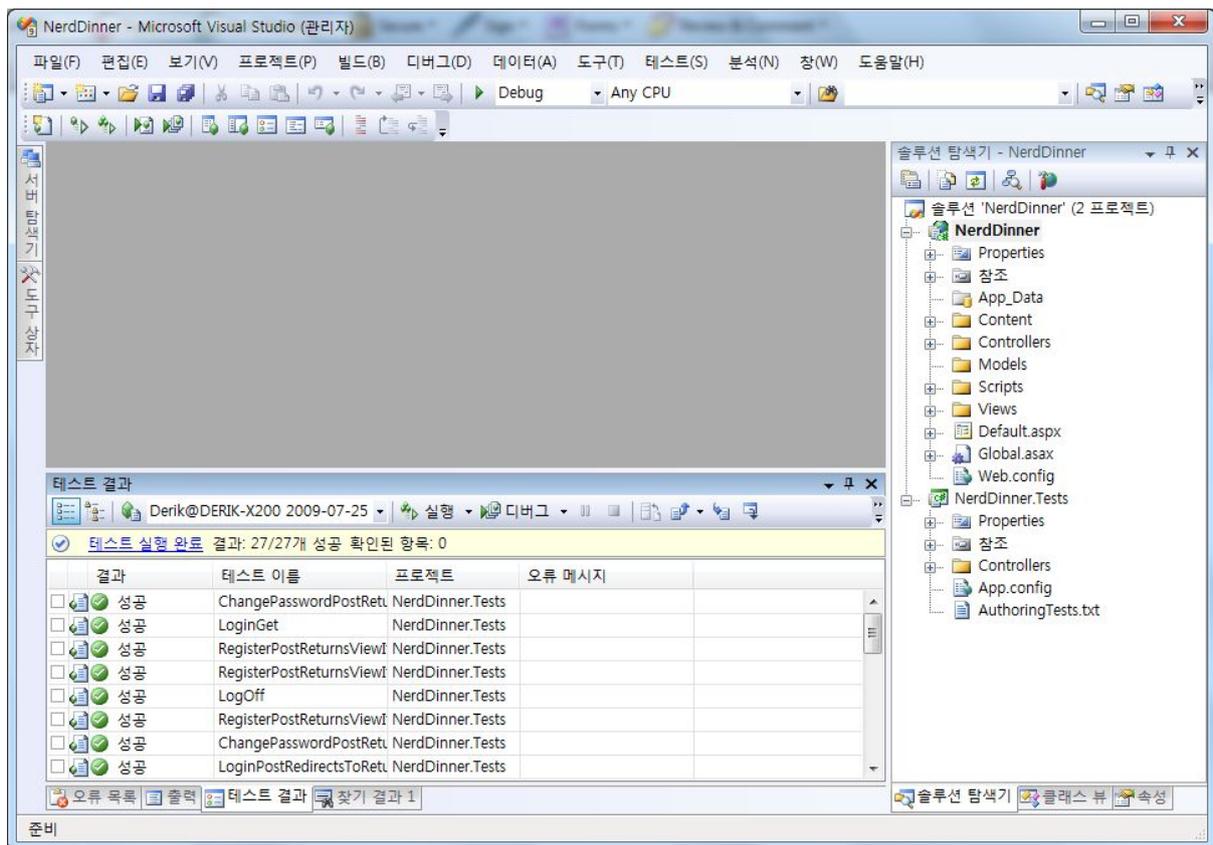


그림 1-23

데이터베이스 생성하기

우리는 NerdDinner 애플리케이션에서 사용할 저녁 모임 및 참여자 데이터를 데이터베이스에 저장할 것이다.

다음의 과정은 무료로 배포되는 SQL Server Express를 이용하여 데이터베이스를 생성하는 방법을 보여준다. 앞으로 우리가 작성할 모든 코드는 SQL Server Express는 물론 SQL Server 상용 버전에서도 문제없이 동작한다.

SQL Server Express 데이터베이스 생성하기

새로운 데이터베이스를 생성하기 위해 다음 그림과 같이 [추가 > 새 항목] 메뉴를 선택하자.

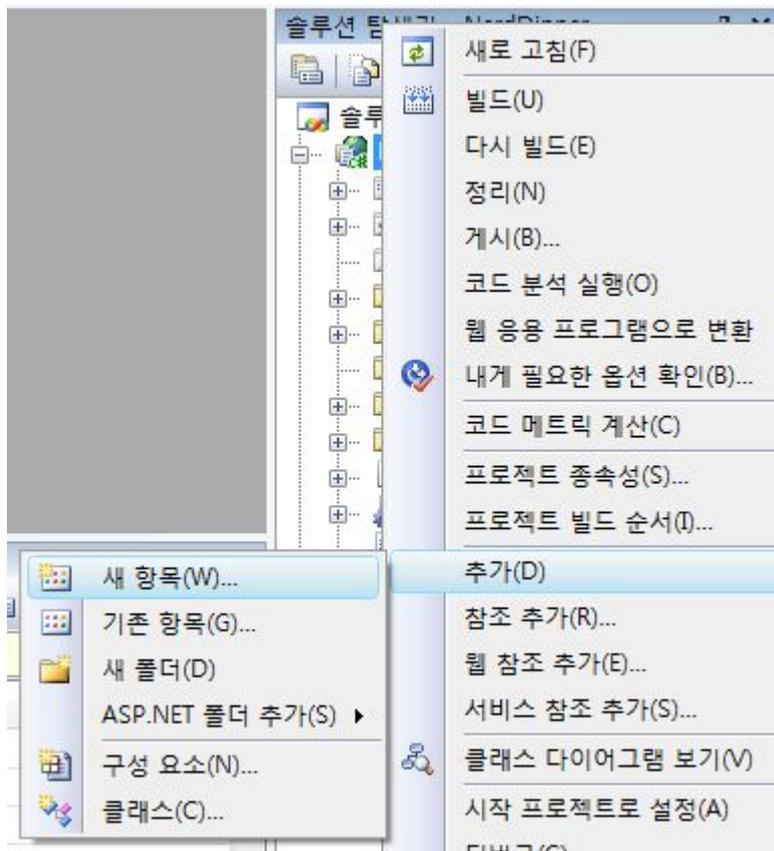


그림 1-24

그러면 “새 항목 추가” 대화 상자가 나타나게 될 것이다. 카테고리 항목에서 [Data] 항목을 선택하고 우측의 템플릿 항목 중에서 [SQL Server 데이터베이스] 항목을 선택하자.

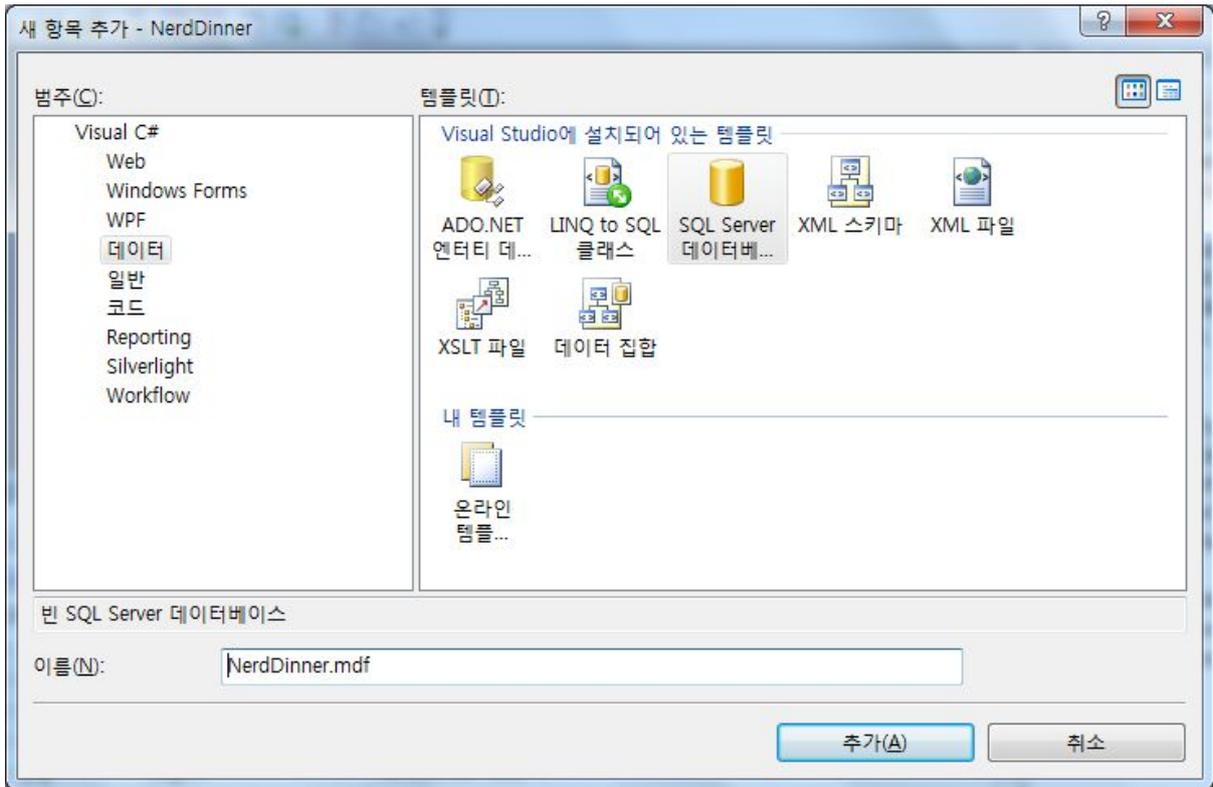


그림 1-25

SQL Server Express 데이터베이스의 이름은 우리가 원하는 "NerdDinner.mdf"로 지정하고 [추가] 버튼을 클릭한다. 그러면 Visual Studio는 이 파일을 WApp_Data 디렉터리(이 디렉터리에는 읽기 및 쓰기 권한이 미리 주어져 있다)에 추가할 것인지를 물어올 것이다.

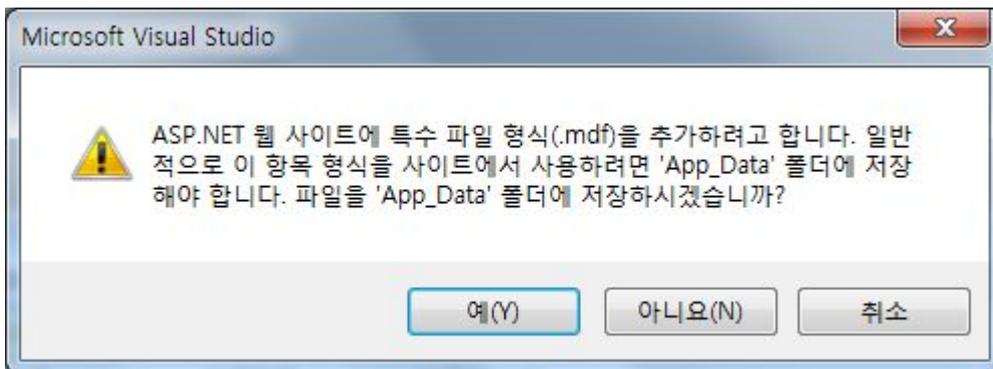


그림 1-26

[예] 버튼을 클릭하면 다음 그림과 같이 솔루션 탐색기에 새로운 데이터베이스가 생성된다.

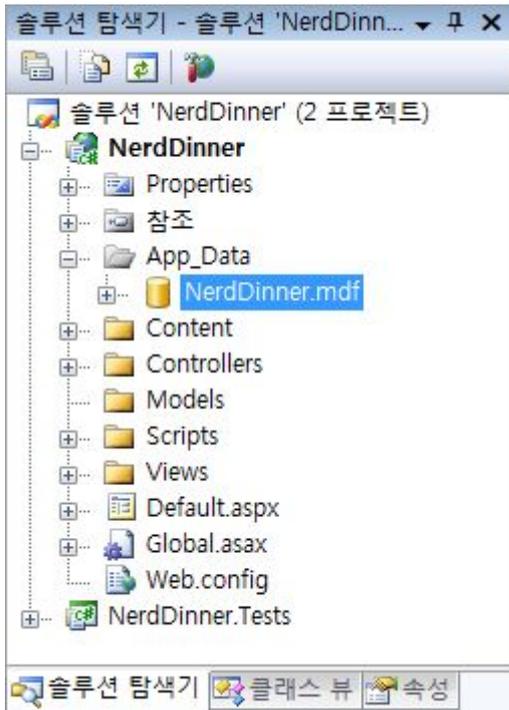


그림 1-27

데이터베이스에 테이블 생성하기

이제 빈 데이터베이스를 생성했으므로 몇 개의 테이블을 추가해보자.

테이블을 추가하기 위해 Visual Studio의 탭 문서 인터페이스에서 “서버 탐색기” 탭을 선택하자. 서버 탐색기 탭은 데이터베이스 객체와 서버들을 관리하기 위한 인터페이스를 제공한다. 우리가 구현할 애플리케이션의 App_Data 폴더에 저장된 SQL Server Express 데이터베이스들은 서버 탐색기에 자동적으로 나타나게 된다. 물론 “서버 탐색기” 창의 상단에 위치한 “데이터베이스 연결” 아이콘을 클릭하여 – 로컬 혹은 원격지의 – SQL Server 데이터베이스에 연결할 수도 있다.

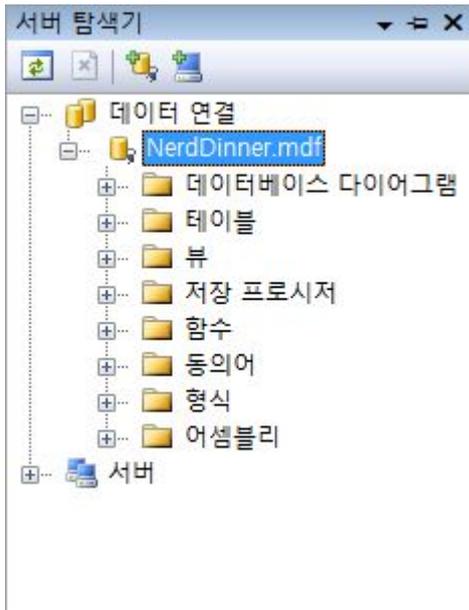


그림 1-28

우리는 NerdDinner 데이터베이스에 두 개의 테이블을 추가할 것이다. 하나는 저녁 모임 데이터를 저장할 테이블이며, 다른 하나는 모임에 참여하고자 하는 참여자 데이터를 저장할 테이블이다. 새 테이블을 추가하려면 데이터베이스 아래의 “테이블” 폴더를 마우스 오른쪽 버튼으로 클릭하고 “새 테이블 추가” 메뉴를 선택한다.

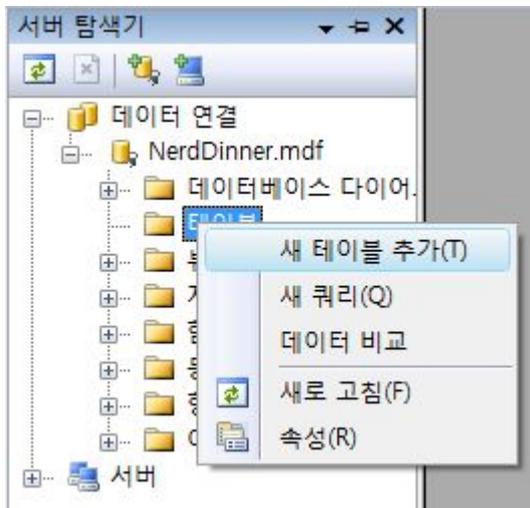


그림 1-29

그러면 다음 그림과 같이 테이블의 스키마를 정의할 수 있는 테이블 디자이너가 나타난다. 새로 추가할 “Dinners” 테이블에는 그림과 같이 10개의 컬럼을 추가할 것이다.

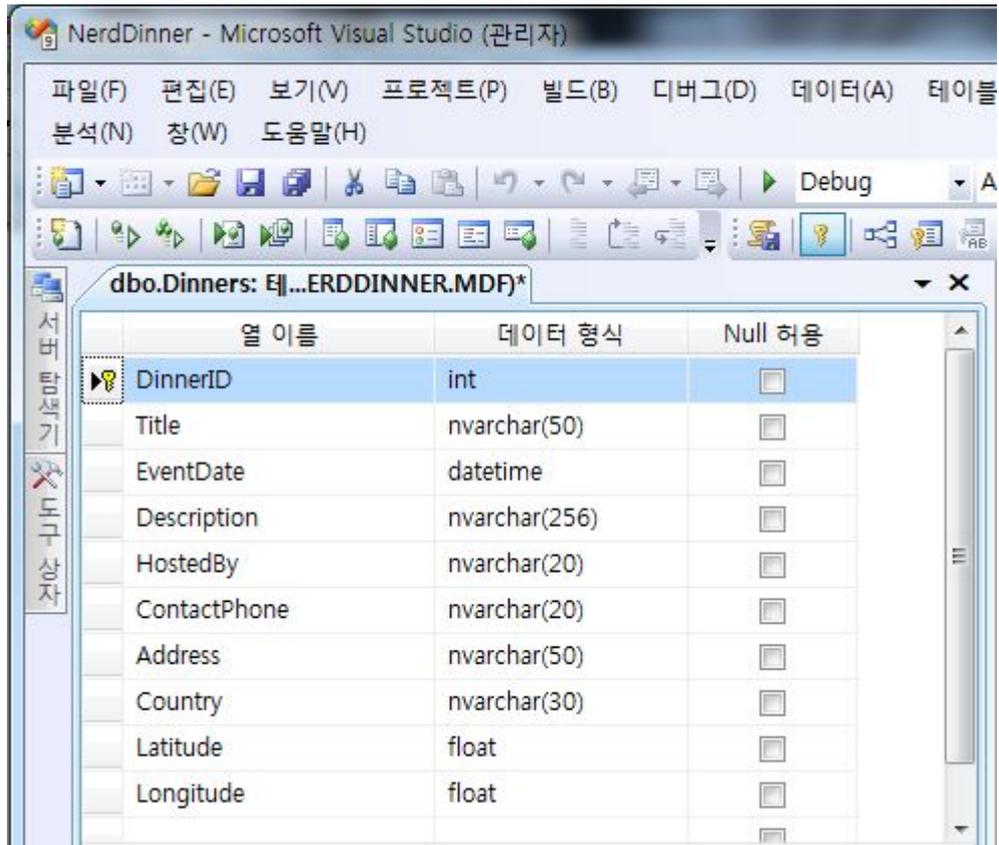


그림 1-30

우리는 DinnerID 컬럼을 테이블의 유일한 기본 키로 사용할 것이다. 컬럼을 기본 키 컬럼을 지정하려면 DinnerID 컬럼을 마우스 오른쪽 버튼으로 클릭한 후 “기본 키 설정” 메뉴 항목을 선택하면 된다.

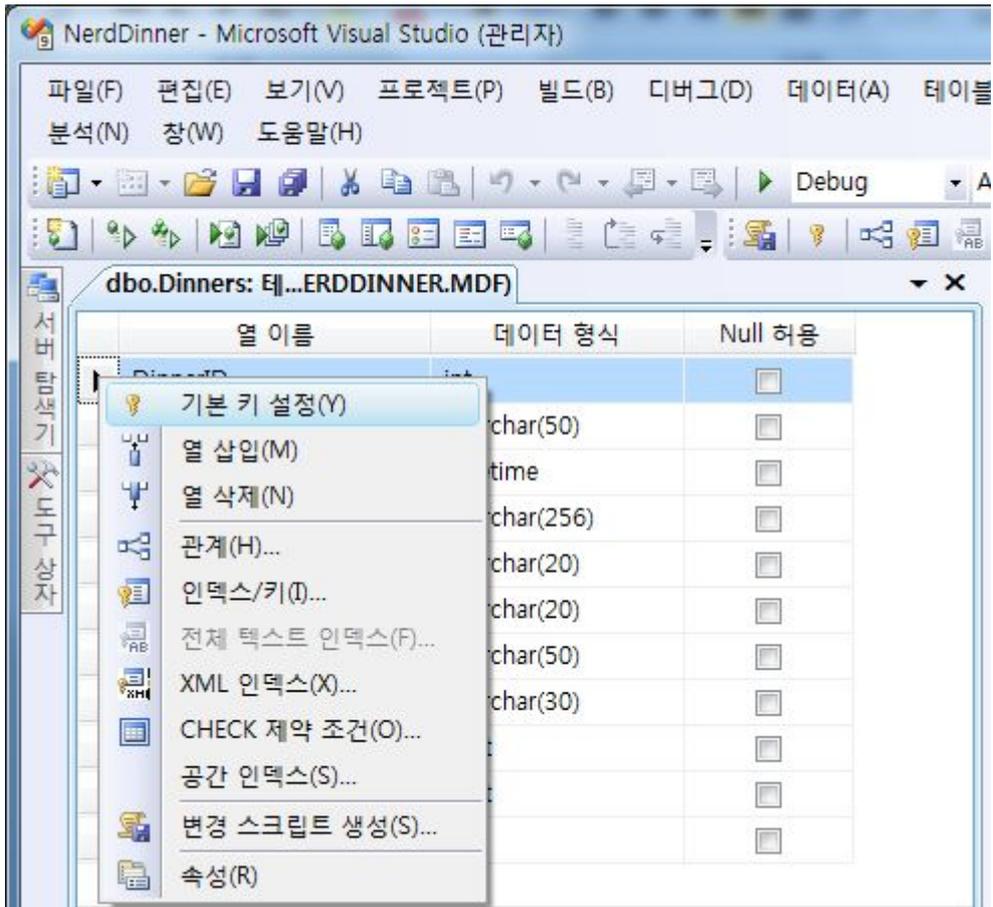


그림 1-31

DinnerID 컬럼은 기본 키 컬럼인 동시에 자동 증가 컬럼으로 구성해야 한다. 자동 증가 컬럼은 테이블에 새로운 행이 추가될 때마다 그 값이 자동으로 증가하는 컬럼이다. (즉, 첫 번째 행이 추가되면 DinnerID 컬럼의 값은 1이 되며, 두 번째 행이 추가되면 DinnerID 컬럼의 값이 2가 되는 식이다.)

DinnerID 컬럼을 자동 증가 컬럼으로 지정하려면 DinnerID 컬럼을 클릭하고 “열 속성” 편집기에서 “ID 사양” 속성의 값을 “예”로 선택한다. 그리고 나머지 옵션들은 기본 값을 유지한다. (그러면 새로운 행이 추가될 때 DinnerID 컬럼의 값은 1부터 시작하여 1씩 증가하는 값을 갖게 된다.)

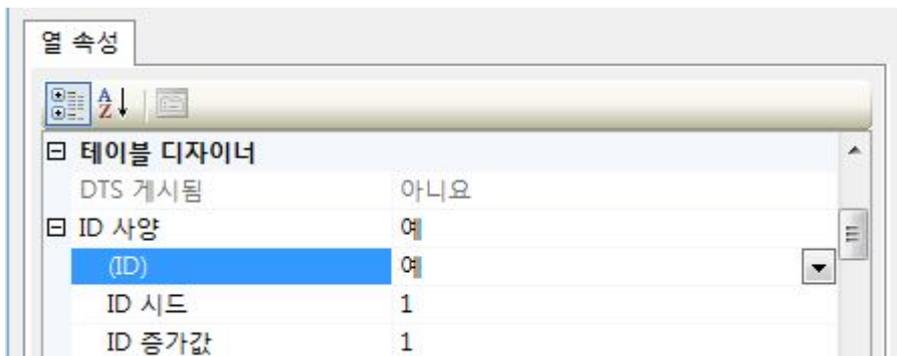
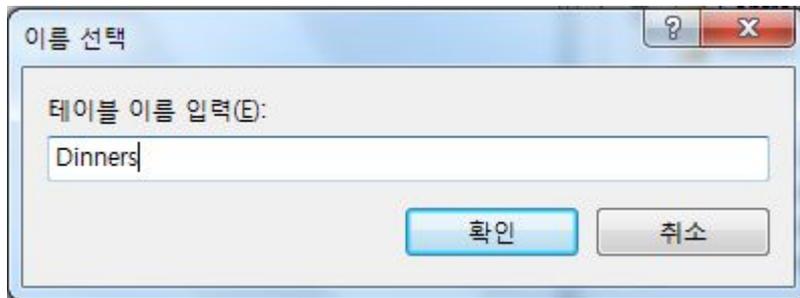


그림 1-32

이제 Ctrl + S 키를 누르거나 혹은 [파일 > 저장] 메뉴를 선택하면 테이블을 저장할 수 있다. 이 때 다음 그림과 같이 저장할 테이블의 이름을 묻는 대화 상자가 나타난다. 이 책에서는 “Dinners” 라는 이름으로 테이블을 저장할 것이다.



이제 서버 탐색기에는 Dinners라는 새로운 테이블이 나타나게 된다.

그러면 지금까지의 단계를 반복하여 “RSVP” 테이블을 추가해보자. 이 테이블은 다음 그림과 같이 세 개의 컬럼을 갖는다. 마찬가지로 RsvpID 컬럼 역시 기본 키 컬럼인 동시에 자동 증가 컬럼으로 설정해야 한다.

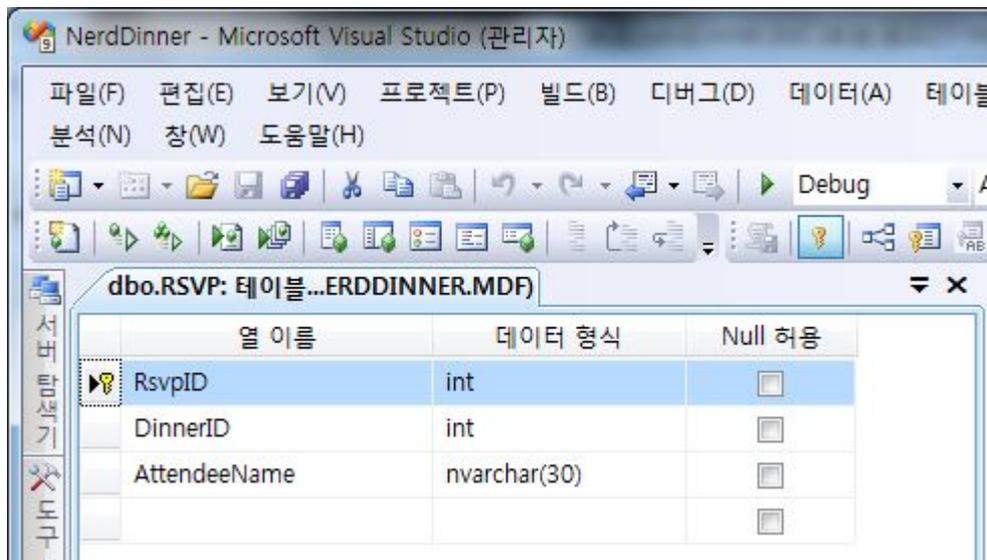


그림 1-34

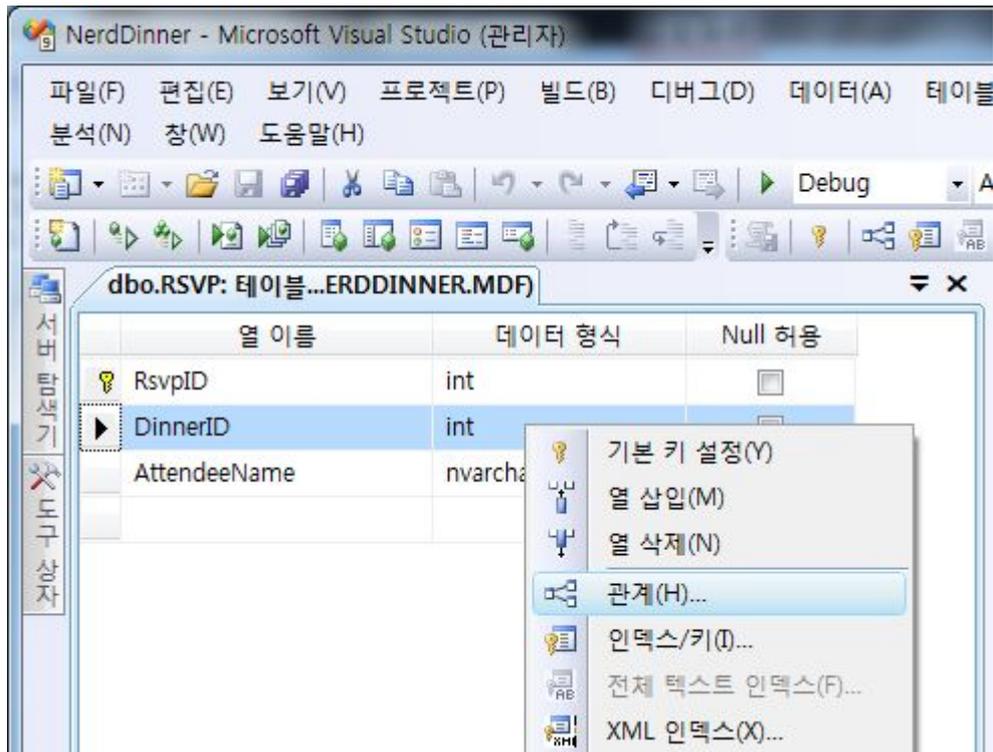
이 테이블은 “RSVP”라는 이름으로 저장하도록 하자.

테이블 간의 외래 키 관계 설정하기

이제 우리는 데이터베이스에 두 개의 테이블을 생성하였다. 마지막으로 해야 할 일은 이 두 테이블 사이에 “일 대 다” 관계를 설정하는 것이다. 이렇게 함으로써 하나의 저녁 모임에 대한 데이터

에 여러 개의 참여자 데이터가 연결될 수 있다. 그러려면 RSVP 테이블의 "DinnerID" 컬럼에 "Dinners" 테이블의 "DinnerID" 컬럼에 대한 외래 키 관계를 설정해야 한다.

외래 키 관계를 설정하려면 우선 서버 탐색기에서 RSVP 테이블을 더블 클릭하여 테이블 디자이너를 열어야 한다. 그런 후 "DinnerID" 컬럼을 마우스 오른쪽 버튼으로 클릭하고 "관계..." 메뉴를 선택한다.



그러면 다음 그림과 같이 두 테이블 사이의 관계를 설정할 수 있는 대화 상자가 나타난다.

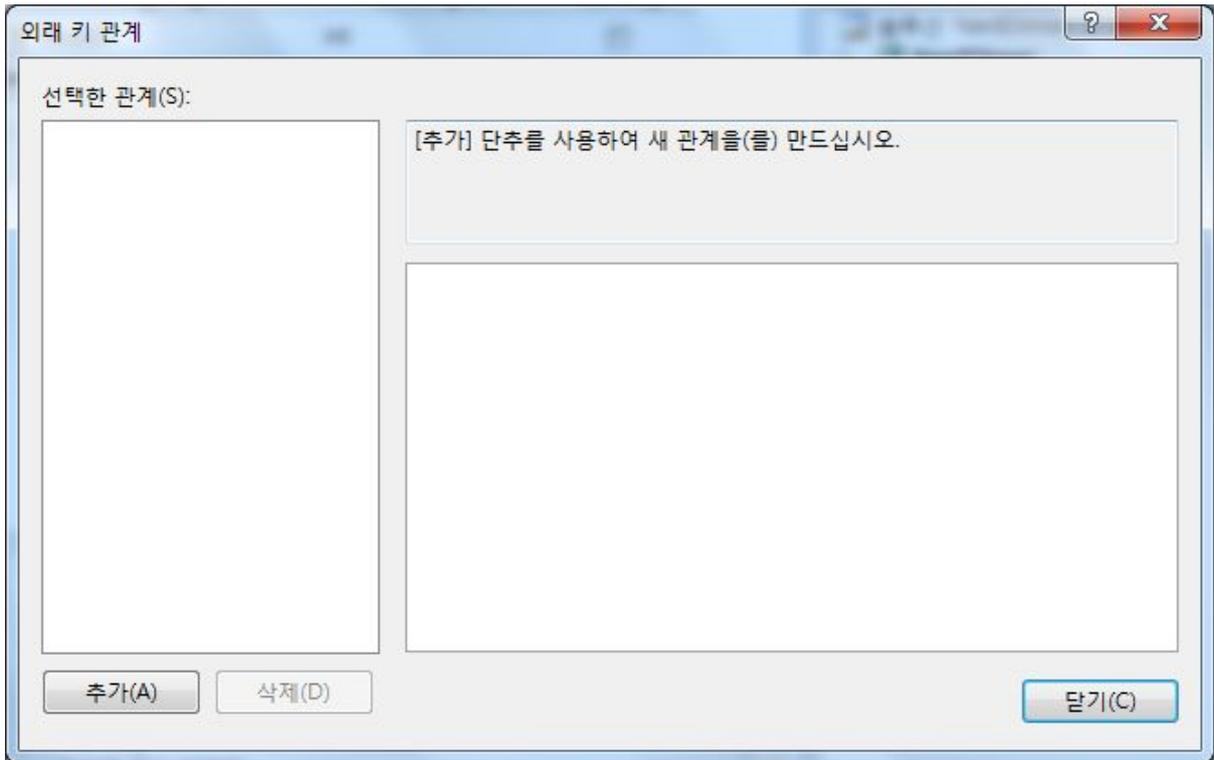


그림 1-36

새로운 관계를 추가하기 위해서는 우선 [추가] 버튼을 클릭한다. 일단 관계가 생성되면 대화 상자 오른쪽의 "테이블 및 컬럼 정의" 트리 뷰 노드의 속성 그리드를 확장하고 오른쪽에 있는 "..." 버튼을 클릭하자.

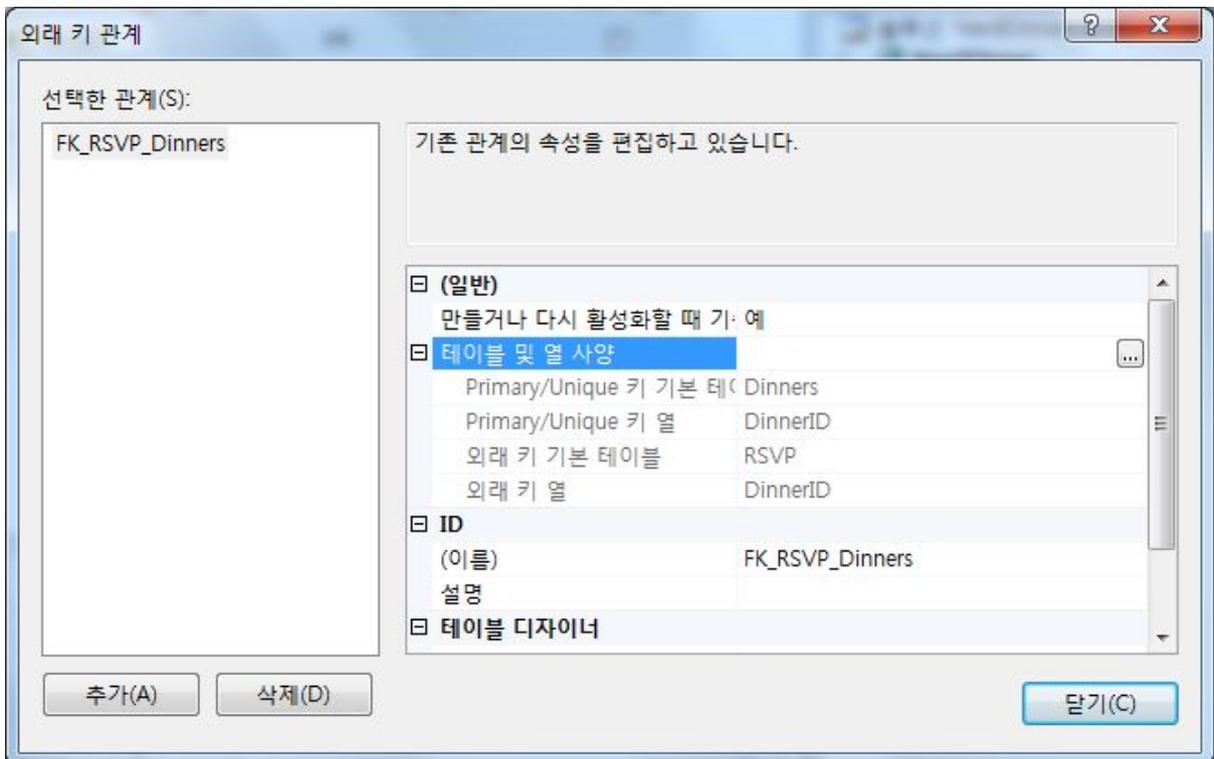


그림 1-37

"..." 버튼을 클릭하면 관계를 맺을 테이블과 컬럼들을 보여주는 또 다른 대화 상자가 나타난다. 이 대화 상자에서는 관계의 이름을 지정할 수도 있다.

이 대화 상자에서는 다음 그림과 같이 기본 키 테이블을 "Dinners" 테이블로 바꾸고 "DinnerID" 컬럼을 기본 키 컬럼으로 선택한다. 외래 키 테이블은 RSVP 테이블로 선택하면 RSVP.DinnerID 컬럼이 외래 키로 자동으로 설정된다.

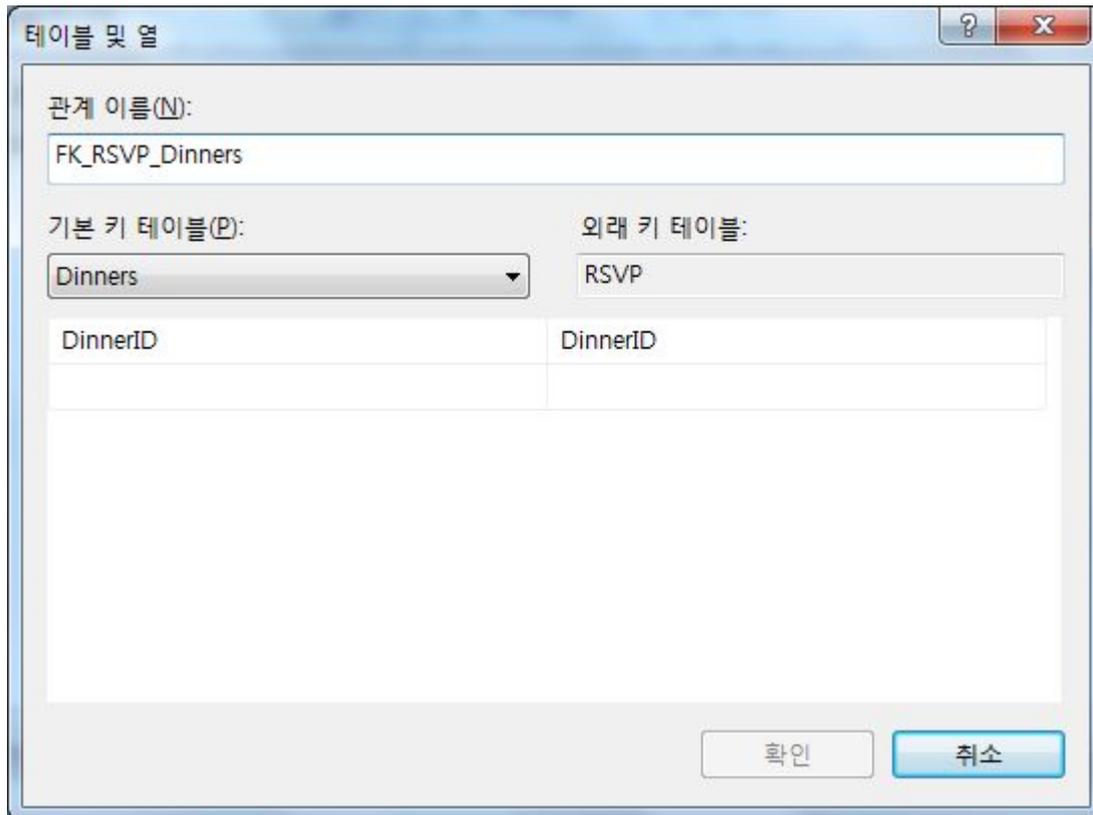


그림 1-38

이제 RSVP 테이블에 저장될 각각의 행들은 Dinners 테이블의 행과 관계가 맺어지게 되었으며 이 관계는 SQL Server가 자동으로 관리해준다. 이렇게 하면 Dinners 테이블에 존재하지 않는 값을 가진 행을 RSVP 테이블에 추가하지 못하며, 또한 RSVP 테이블의 행이 참조하고 있는 Dinners 테이블의 행을 삭제하지도 못하게 된다. (역주: 테이블 사이에 외래 키 관계를 설정하는 것은 데이터의 무결성을 보장하기 위한 기본적인 방법이다.)

테이블에 데이터 추가하기

이제 Dinners 테이블에 몇 개의 예제 데이터를 추가해보자. 서버 탐색기에서 테이블을 마우스 오른쪽 버튼으로 클릭하고 "테이블 데이터 표시" 메뉴를 선택하면 새로운 데이터를 테이블에 추가할 수 있다.

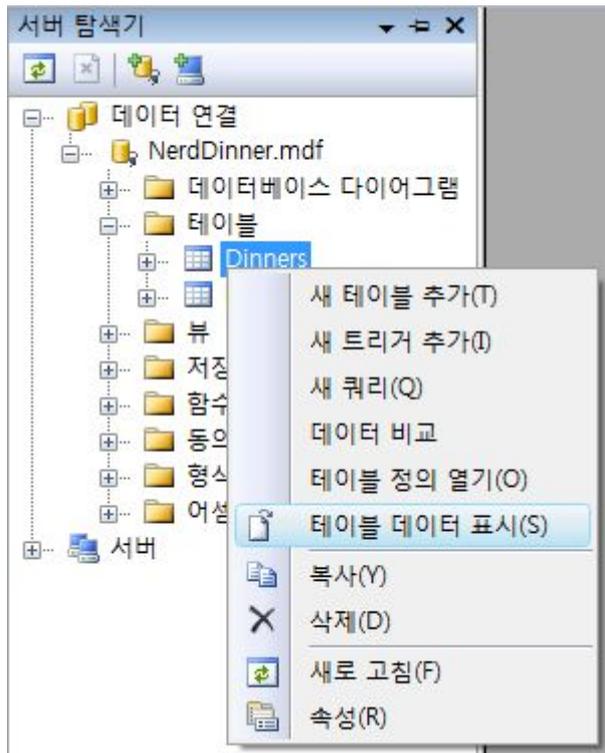


그림 1-39

향후에 애플리케이션의 개발에 사용할 수 있도록 다음 그림과 같이 몇 개의 행을 추가해두자.

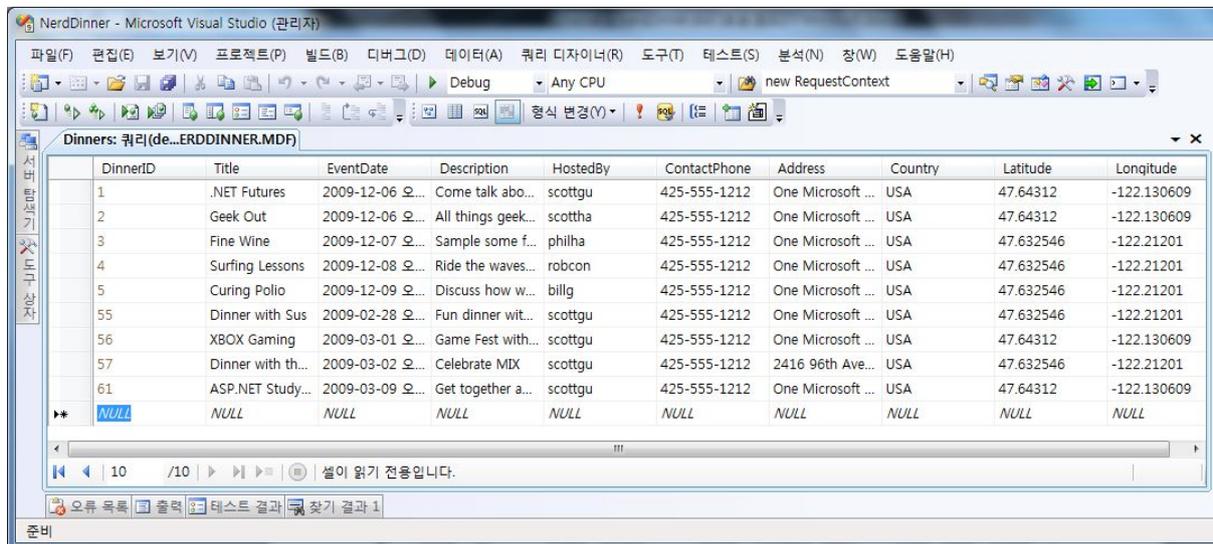


그림 1-40

모델 객체 구현하기

모델 - 뷰 - 컨트롤러 프레임워크에서 "모델"이라는 단어는 애플리케이션의 데이터를 표현할 뿐만 아니라 그에 해당하는 유효성 검사 및 비즈니스 규칙 등의 도메인 로직을 의미하기도 한다.

모델은 여러 가지 방법으로 MVC 기반 애플리케이션의 핵심적인 역할을 수행하며 그 기본적인 기능에 대해서는 잠시 후에 살펴보도록 하겠다.

ASP.NET MVC 프레임워크는 모든 종류의 데이터 액세스 기술을 지원하기 때문에 개발자들은 LINQ to Entities, LINQ to SQL, NHibernate, LLBLGen Pro, SubSonic, WilsonORM 혹은 ADO.NET DataReader나 DataSet 등 .NET의 풍부한 데이터 옵션들을 이용하여 모델 객체를 구현할 수 있다.

우리가 구현할 NerdDinner 예제 애플리케이션은 LINQ to SQL 기술을 활용하여 앞서 디자인한 데이터베이스와 유사한 단순한 도메인 모델을 생성하고 약간의 유효성 검사 로직과 비즈니스 규칙들을 추가할 것이다. 그런 후 데이터의 상세한 구현을 추상화하는 저장소 클래스(Repository Classes)들을 구현함으로써 데이터 계층을 애플리케이션으로부터 분리함과 동시에 단위 테스트를 보다 손쉽게 적용할 수 있도록 할 것이다.

LINQ to SQL

LINQ to SQL은 .NET 3.5에 포함된 ORM (Object-Relational Mapper, 객체-관계 매퍼) 도구이다.

LINQ to SQL을 이용하면 데이터베이스 테이블을 .NET 클래스로 손쉽게 매핑할 수 있다. NerdDinner 애플리케이션의 경우, Dinners 테이블과 RSVP 테이블을 각각 Dinners와 RSVP 모델 클래스로 매핑하기 위해 LINQ to SQL을 이용한다. Dinners 테이블과 RSVP 테이블의 각 컬럼들은 Dinners 클래스와 RSVP 클래스의 속성으로 구현되며, 각각의 Dinner와 RSVP 객체들은 Dinners 테이블과 RSVP 테이블의 각 행을 표현하게 된다.

LINQ to SQL을 사용하면 Dinners 객체와 RSVP 객체를 이용하여 데이터베이스의 데이터를 조회하거나 업데이트하기 위한 SQL 구문을 굳이 작성할 필요가 없다. 대신 우리는 Dinners 클래스와 RSVP 클래스를 정의하고 이들이 데이터베이스의 테이블과 어떻게 매핑되는지를 정의한 후 이들 사이의 관계를 설정하면 된다. 그러면 LINQ to SQL은 우리가 런타임에 이들 객체들과 상호 작용하는데 필요한 SQL 실행 로직을 생성하고 관리한다.

Dinners 객체와 RSVP 객체를 조회하는 데 필요한 쿼리는 VB와 C#에서 지원되는 LINQ 언어를 이용하여 작성할 수 있다. 이렇게 하면 우리가 작성해야 하는 데이터 관련 코드가 획기적으로 줄어들며 보다 깔끔한 애플리케이션을 작성할 수 있게 된다.

프로젝트에 LINQ to SQL 클래스 추가하기

그러면 프로젝트의 "Models" 폴더를 마우스 오른쪽 버튼으로 클릭하고 [추가 > 새 항목] 메뉴를 선택해보자.

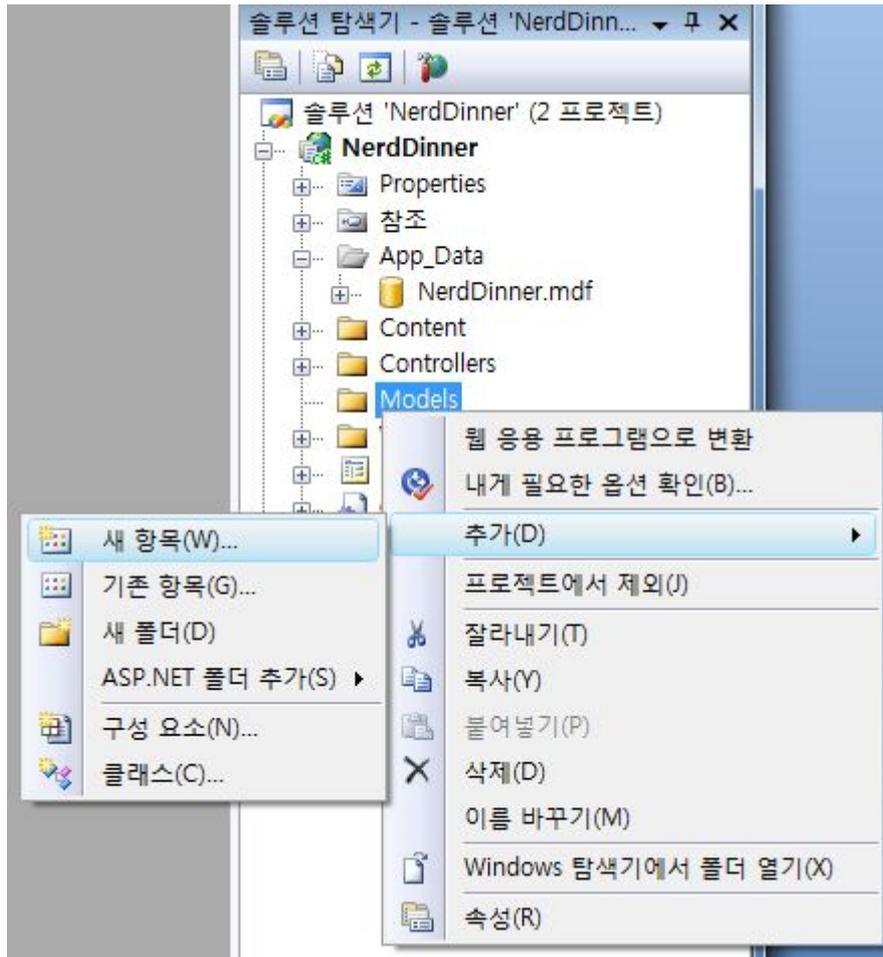


그림 1-41

다음 그림과 같이 “새 항목 추가” 대화 상자가 나타나면 “Data” 카테고리를 선택하고 “LINQ to SQL 클래스” 템플릿을 선택한다.

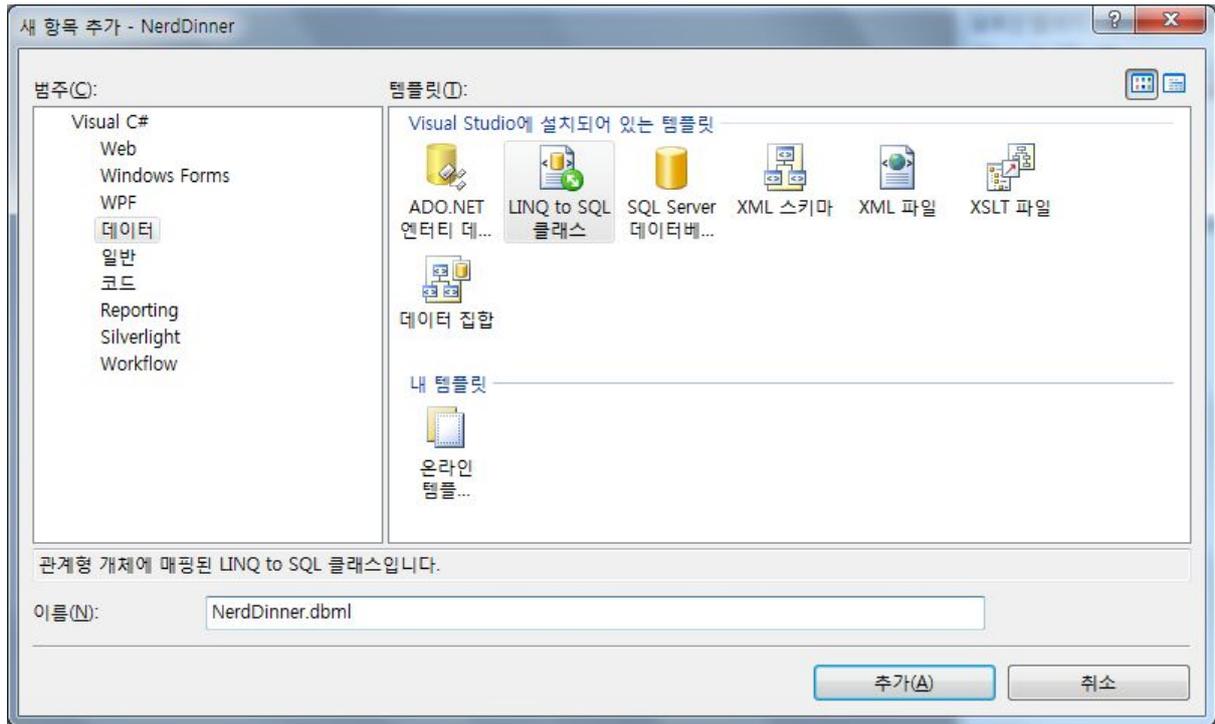


그림 1-42

새로 추가할 객체의 이름을 "NerdDinner"라고 입력한 후 "추가" 버튼을 클릭하자. 그러면 Visual Studio는 Models 디렉터리에 NerdDinner.dbml이라는 이름의 파일을 추가하고 LINQ to SQL 객체 관계 디자이너를 보여준다.

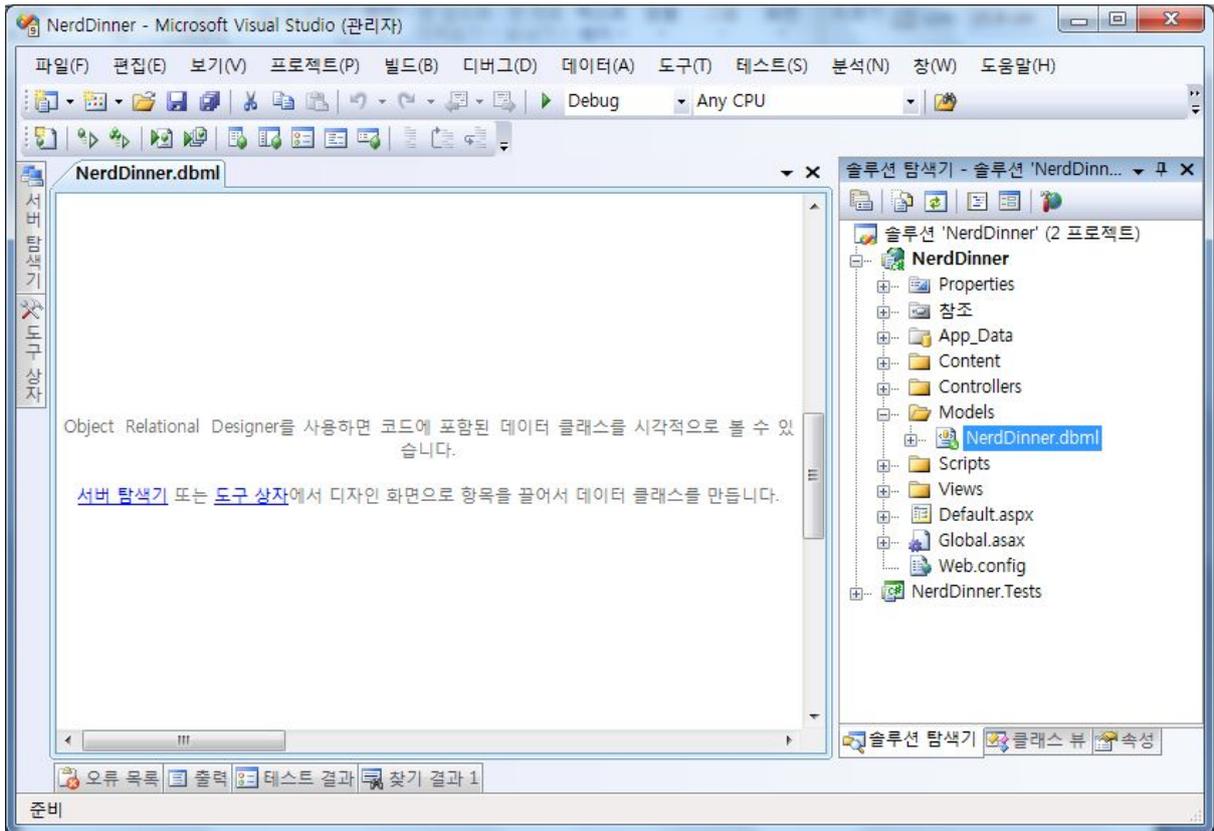


그림 1-43

LINQ to SQL을 이용하여 데이터 모델 클래스 생성하기

LINQ to SQL을 사용하면 이미 존재하는 데이터베이스 스키마를 이용하여 데이터 모델 클래스를 빠르게 생성할 수 있다. 우선 서버 탐색기에서 NerdDinner 데이터베이스를 열고 모델 클래스로 구현할 테이블들을 선택한다.

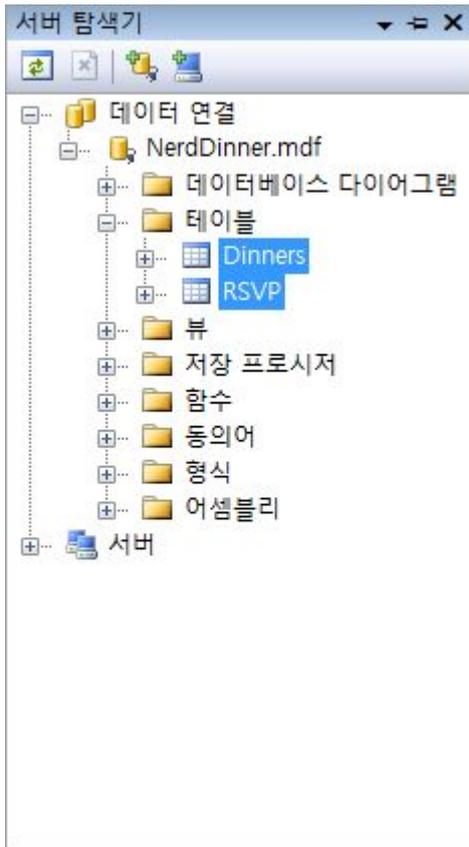


그림 1-44

그런 후 선택한 테이블들을 LINQ to SQL 디자이너로 드래그하면 LINQ to SQL은 선택된 테이블들의 스키마를 이용하여 자동으로 Dinner과 RSVP 클래스를 생성해 준다. (물론 클래스의 속성들은 데이터베이스 테이블의 컬럼들과 매핑된다.)

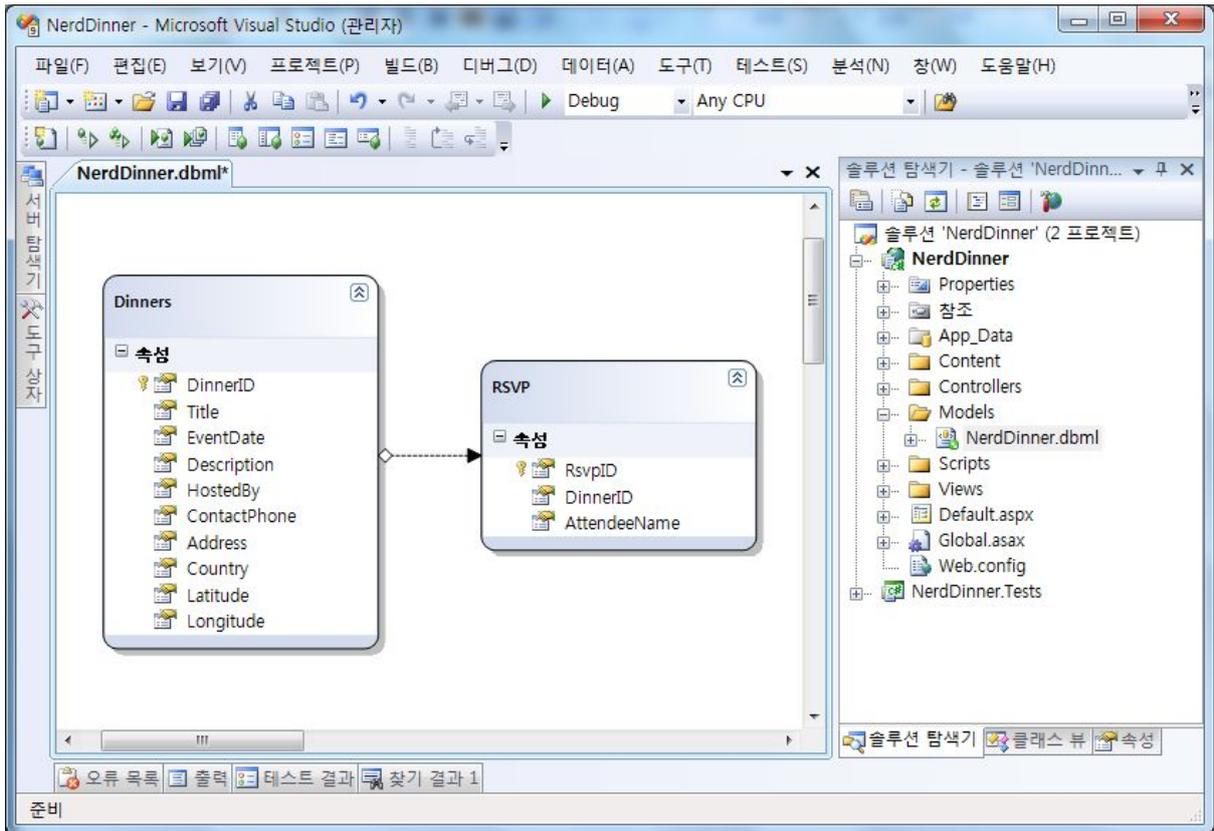


그림 1-45

LINQ to SQL 디자이너는 데이터베이스 스키마를 토대로 클래스들을 생성할 때 테이블과 컬럼 이름에 복수형을 제거한다. 예를 들어 예제의 “Dinners” 테이블은 “Dinner”라는 이름의 클래스로 만들어진다(역자 주: 한글 버전의 Visual Studio를 사용한다면 이 기능은 동작하지 않는다. 즉 Dinners 테이블은 Dinners 클래스로 생성된다). 이와 같은 클래스 명명 규칙은 생성되는 모델 클래스들이 .NET의 명명 규칙을 따르도록 하기 위함이며, 필자의 경우 이런 면에서 디자이너가 사용하기 편리하다고 느끼고 있다(특히 많은 수의 테이블을 추가하는 경우에는 더욱 그렇다).

만일 디자이너가 생성한 클래스나 속성의 이름이 마음에 들지 않는다면 언제든지 여러분이 원하는 이름으로 변경이 가능하다. 이름의 변경은 엔티티/속성 이름을 디자이너에서 직접 수정하거나 속성 창을 이용하여 변경하면 된다.

기본적으로 LINQ to SQL 디자이너는 테이블들의 기본 키와 외래 키를 검색하여 모델 클래스를 생성할 때 이들 사이의 기본적인 관계를 자동으로 설정한다. 예를 들어, Dinners 테이블과 RSVP 테이블을 LINQ to SQL 디자이너에 추가하면 RSVP 테이블이 Dinners 테이블에 대한 외래 키를 가지고 있기 때문에 이 두 클래스 사이의 일 대 다 관계가 형성된다(디자이너에 나타난 화살표가 이것을 의미한다).

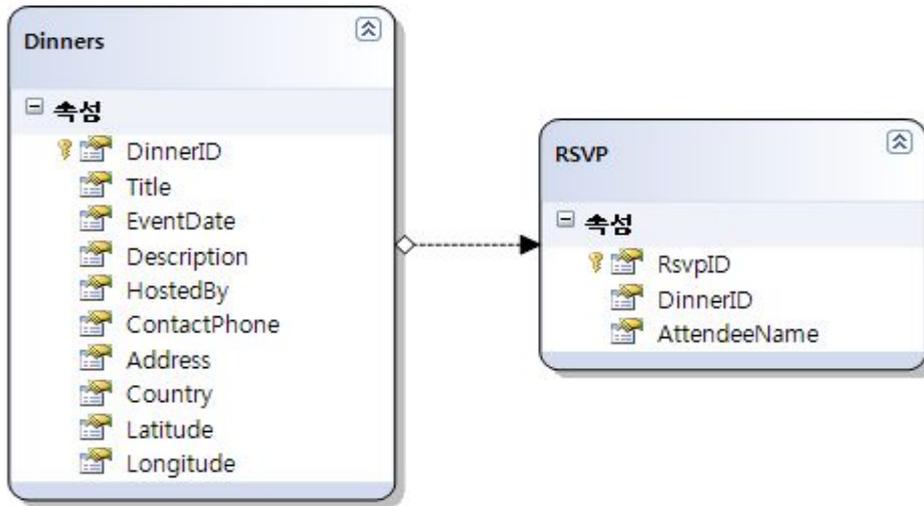


그림 1-46

LINQ to SQL은 위 그림에 표시된 관계를 표현하기 위해 RSVP 클래스에 "Dinner" 타입의 속성을 추가하며 개발자들은 이 속성을 통해 특정 RSVP 데이터와 관련된 Dinner 엔티티 객체에 손쉽게 액세스할 수 있게 된다. 또한 Dinners 클래스에는 "RSVP"라는 이름의 컬렉션 속성이 추가되어 개발자들이 특정 Dinners 클래스에 관계된 RSVP 객체들을 손쉽게 조회하고 업데이트할 수 있게 된다.

다음 그림은 Dinners 클래스의 RSVP 컬렉션에 새로운 RSVP 객체를 추가할 때 Visual Studio의 인텔리센스가 동작한 모습을 보여준다.

```
Dinners dinner = db.Dinners.Single(d => d.DinnerID == 1);
RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "webgenie";
```

dinner.R|

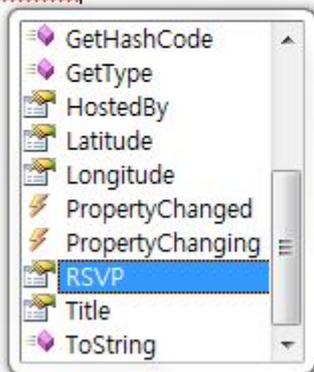


그림 1-47

위 그림에서 우리는 LINQ to SQL이 Dinners 객체에 "RSVP" 컬렉션을 추가했음을 알 수 있다. 이 속성은 아래 코드에서 볼 수 있듯이 데이터베이스 내의 Dinner와 RSVP 행 사이의 외래 키 관계를 표현할 때 사용할 수 있다.

```
Dinners dinner = db.Dinners.Single(d => d.DinnerID == 1);
RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "webgenie";

dinner.RSVP.Add(myRSVP);
```

그림 1-48

만일 디자이너가 생성하는 테이블 사이의 관계 모델링이나 이름이 마음에 들지 않는다면 여러분 임의로 변경이 가능하다. 디자이너의 관계 화살표를 클릭하고 속성 창을 이용하여 속성의 값을 변경하거나 삭제 또는 수정이 가능하다. 그러나 NerdDinner 애플리케이션의 경우 기본적으로 생성된 관계 규칙이 우리가 구현한 데이터에 대한 모델 클래스들과 잘 동작하고 있으므로 기본적인 동작을 사용할 것이다.

NerdDinnerDataContext 클래스

Visual Studio는 LINQ to SQL 디자이너를 사용하여 정의한 모델과 데이터 관계들을 표시하기 위한 .NET 클래스들을 자동으로 생성한다. 이때 각각의 LINQ to SQL 디자이너 파일을 위한 LINQ to SQL DataContext 클래스 역시 함께 생성된다. 이 예제의 경우 LINQ to SQL 클래스의 이름이 "NerdDinner"이기 때문에 DataContext 클래스는 "NerdDinnerDataContext"라는 이름으로 생성된다. 이 NerdDinnerDataContext 클래스는 데이터베이스와의 상호 작용을 위한 가장 기본적인 방법이다.

NerdDinnerDataContext 클래스는 "Dinners"와 "RSVPs" 등 두 가지 속성을 제공하며 이들은 데이터베이스에 생성한 두 개의 테이블을 표현한다. 우리는 데이터베이스로부터 Dinners 객체와 RSVP 객체를 조회하기 위해 이 두 속성에 LINQ 쿼리를 C# 문법으로 작성할 수 있다.

다음의 코드는 NerdDinnerDataContext 객체의 인스턴스를 생성하고 LINQ 쿼리를 이용하여 Dinners 객체의 컬렉션을 얻어오는 방법을 보여준다.

```

NerdDinnerDataContext db = new NerdDinnerDataContext ();

var upcomingDinners = from dinner in db.Dinners
                       where dinner.EventDate > DateTime.Now
                       orderby dinner.EventDate
                       select dinner;

foreach (Dinners dinner in upcomingDinners)
{
    dinner.
}

```

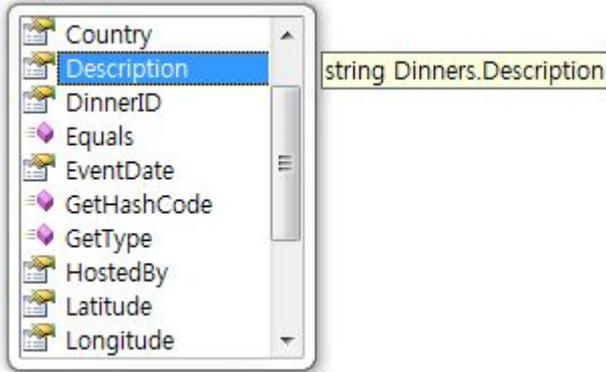


그림 1-49

NerdDinnerDataContext 객체는 자신이 리턴한 Dinners 객체 및 RSVP 객체의 변경 사항을 모두 추적하기 때문에 변경 사항을 손쉽게 데이터베이스에 저장할 수 있다. 다음의 코드는 LINQ 쿼리를 이용하여 하나의 Dinners 객체를 데이터베이스로부터 조회하고 이 객체의 두 속성 값을 변경한 후 이를 다시 데이터베이스에 적용하는 방법을 보여준다.

```

NerdDinnerDataContext db = new NerdDinnerDataContext ();
// DinnerID가 1인 레코드를 표현하는 Dinners 객체를 찾는다.
Dinners dinner = db.Dinners.Single(d => d.DinnerID == 1);
// Dinners 객체의 속성 값들을 변경한다
dinner.Title = "불타는 토요일밤의 모임";
dinner.Description = "재미있을거예요!";
// 데이터베이스에 변경사항을 저장한다.
db.SubmitChanges ();

```

위의 코드에서 사용된 NerdDinnerDataContext 객체는 Single 메서드에 의해 리턴된 Dinners 객체에 가해진 변경 사항을 자동적으로 추적한다. 그런 후 우리가 "SubmitChanges()" 메서드를 호출하면 업데이트된 값들을 데이터베이스에 저장하기 위한 "UPDATE" SQL 구문을 실행한다.

DinnerRepository 클래스 구현하기

작은 규모의 애플리케이션이라면 LINQ 쿼리를 컨트롤러 객체 내에 작성하여 컨트롤러 객체가 LINQ to SQL DataContext 클래스와 직접 상호 작용하도록 구현하는 것도 나쁘지는 않다. 그러나

애플리케이션의 규모가 커진다면 이와 같은 방법은 오히려 유지보수 및 테스트에 악영향을 주게 되며 동일한 LINQ 쿼리를 여러 곳에서 중복 사용하게 될 가능성도 있다.

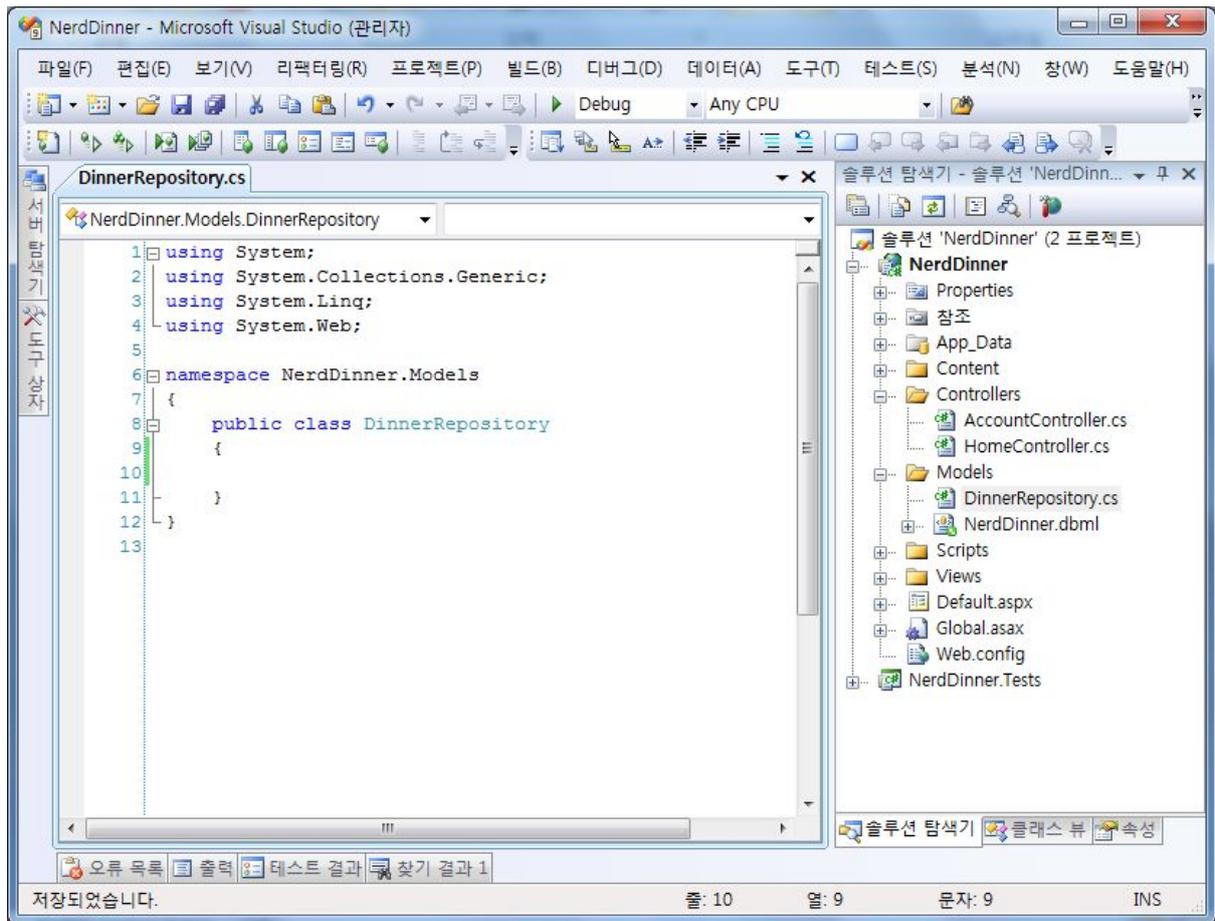
애플리케이션의 유지보수 및 테스트를 보다 손쉽게 하기 위해 사용되는 방법 중 하나는 “저장소 (Repository)” 패턴을 사용하는 것이다. 저장소 클래스들은 데이터와 관련된 쿼리와 로직을 캡슐화 하며 애플리케이션으로부터 데이터베이스와 관련된 상세 구현을 추상화한다. 또한 애플리케이션의 코드를 보다 깔끔하게 유지할 수 있을 뿐 아니라 향후에 데이터 저장소의 구현을 변경하기에도 쉬우며 실제 데이터베이스가 없어도 단위 테스트를 수행할 수 있는 등 여러 장점을 제공한다.

NerdDinner 애플리케이션의 경우 다음과 같이 DinnerRepository 클래스를 구현한다.

```
public class DinnerRepository {  
    // 검색 메서드  
    public IQueryable<Dinner> FindAllDinners ();  
    public IQueryable<Dinner> FindUpcomingDinners ();  
    public Dinner GetDinner (int id);  
    // 추가/삭제 메서드  
    public void Add (Dinner dinner);  
    public void Delete (Dinner dinner);  
    // 저장 메서드  
    public void Save ();  
}
```

=====
Note: 이 장의 후반부에서는 DinnerRepository 클래스로부터 IDinnerRepository 인터페이스를 추출하여 컨트롤러 클래스에 의존성 삽입(DI: Dependency Injection) 기능을 지원할 것이다. 그러나 우선은 DinnerRepository 클래스를 직접 사용할 것이다.
=====

이 클래스를 구현하려면 “Models” 폴더를 마우스 오른쪽 버튼으로 클릭하고 “추가 > 새 항목” 메뉴를 선택한다. 그런 후 “새 항목 추가” 대화 상자에서 “클래스” 템플릿을 선택하고 “DinnerRepository.cs”라는 이름을 입력한다.



그런 후 다음과 같이 DinnerRepository 클래스를 구현한다.

```
public class DinnerRepository
{
    private NerdDinnerDataContext db = new NerdDinnerDataContext();
    //
    // 쿼리 메서드
    public IQueryable<Dinners> FindAllDinners()
    {
        return db.Dinners;
    }
    public IQueryable<Dinners> FindUpcomingDinners()
    {
        return from dinner in db.Dinners
               where dinner.EventDate > DateTime.Now
               orderby dinner.EventDate
               select dinner;
    }
    public Dinners GetDinner(int id)
    {
        return db.Dinners.SingleOrDefault(d => d.DinnerID == id);
    }
    //
    // 삽입/삭제 메서드
    public void Add(Dinner dinner)
    {
```

```

        db.Dinners.InsertOnSubmit(dinner);
    }
    public void Delete(Dinner dinner)
    {
        db.RSVP.DeleteAllOnSubmit(dinner.RSVPs);
        db.Dinners.DeleteOnSubmit(dinner);
    }
    //
    // 저장 메서드
    public void Save()
    {
        db.SubmitChanges();
    }
}

```

DinnerRepository 클래스를 이용하여 데이터베이스 작업 수행하기

이제 DinnerRepository 클래스를 생성했으므로 이 클래스를 이용하여 우리가 할 수 있는 작업을 보여주는 간단한 예제들을 살펴보자.

데이터 조회 예제

다음의 예제 코드는 DinnerID 값을 이용하여 하나의 Dinners 객체를 조회하는 코드이다.

```

DinnerRepository dinnerRepository = new DinnerRepository();
// 지정된 DinnerID 값을 갖는 Dinners 객체를 조회한다.
Dinners dinner = dinnerRepository.GetDinner(5);

```

다음의 코드는 아직 완료되지 않은 저녁 모임 목록을 모두 가져와 루프를 실행한다.

```

DinnerRepository dinnerRepository = new DinnerRepository();
// 완료되지 않은 저녁 모임 데이터를 모두 가져온다.
var upcomingDinners = dinnerRepository.FindUpcomingDinners();
// 각각의 Dinners 객체에 대한 루프를 실행한다.
foreach (Dinners dinner in upcomingDinners) {
}

```

데이터의 추가와 수정 예제

다음의 예제 코드는 새로운 두 개의 저녁 모임을 추가한다. 저장소 클래스를 통해 추가 혹은 수정된 값들은 "Save()" 메서드를 호출하기 전까지는 데이터베이스에 적용되지 않는다. LINQ to SQL 은 데이터베이스 트랜잭션 내의 변경 사항들을 자동으로 래핑하기 때문에 저장소에 변경 사항을 저장할 때는 모든 변경 사항이 적용되거나 혹은 아무런 변화도 일어나지 않게 된다. (역자 주: 데이터베이스 트랜잭션은 데이터베이스에 변경을 가하는 명령이 모두 성공적으로 완료되면 모든 변경 사항들을 적용하며 단 하나라도 실패하면 아무런 변경 사항도 적용하지 않는다.)

```

DinnerRepository dinnerRepository = new DinnerRepository();
// 첫 번째 Dinners 객체를 생성한다.
Dinners newDinner1 = new Dinners();
newDinner1.Title = "Dinner with Scott";
newDinner1.HostedBy = "ScotGu";
newDinner1.ContactPhone = "425-703-8072";
// 두 번째 Dinners 객체를 생성한다.
Dinners newDinner2 = new Dinners();
newDinner2.Title = "Dinner with Bill";
newDinner2.HostedBy = "BillG";
newDinner2.ContactPhone = "425-555-5151";
// 저장소 클래스에 새로운 Dinners 객체들을 추가한다.
dinnerRepository.Add(newDinner1);
dinnerRepository.Add(newDinner2);
// 변경 사항을 저장한다.
dinnerRepository.Save();

```

다음의 코드는 이미 존재하는 Dinners 객체를 데이터베이스에서 조회한 후 두 개의 속성 값을 변경한다. 이들 변경 사항 역시 "Save()" 메서드를 호출할 때 데이터베이스에 적용된다.

```

DinnerRepository dinnerRepository = new DinnerRepository();
// DinnerID 값을 이용하여 데이터베이스에서 Dinners 객체를 조회한다.
Dinners dinner = dinnerRepository.GetDinner(5);
// 속성 값을 변경한다.
dinner.Title = "Update Title";
dinner.HostedBy = "New Owner";
// 변경 사항을 적용한다.
dinnerRepository.Save();

```

다음의 예제 코드는 저녁 모임 데이터를 조회한 후 그에 대한 참여자 데이터를 추가한다. 이와 같은 작업은 LINQ to SQL이 Dinners 클래스에 추가한 RSVPs 컬렉션 속성을 통해 이루어진다(왜냐하면 두 테이블 사이에는 기본 키/외래 키 관계가 성립되어 있기 때문이다). 새로 추가된 참여자 데이터들은 "Save()" 메서드를 호출할 때 RSVP 테이블에 새로운 행으로 추가된다.

```

DinnerRepository dinnerRepository = new DinnerRepository();
// DinnerID 값을 이용하여 Dinners 객체를 조회한다.
Dinners dinner = dinnerRepository.GetDinner(5);
// 새로운 참여자 데이터를 생성한다.
RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";
// 생성된 참여자 데이터를 Dinners 객체의 RSVP 컬렉션에 추가한다.
dinner.RSVPs.Add(myRSVP);
// 변경 사항을 적용한다.
dinnerRepository.Save();

```

데이터 삭제 예제

다음의 예제 코드는 이미 존재하는 Dinners 객체를 조회한 후 이를 삭제된 것으로 표시한다. 저장소 클래스의 "Save()" 메서드를 호출하면 삭제된 것으로 표시된 객체들은 실제로 데이터베이스에서 삭제된다.

```
DinnerRepository dinnerRepository = new DinnerRepository();  
// DinnerID 값을 이용하여 Dinners 객체를 조회한다.  
Dinners dinner = dinnerRepository.GetDinner(5);  
// 조회된 객체를 삭제된 것으로 표시한다.  
dinnerRepository.Delete(dinner);  
// 변경 사항을 적용한다.  
dinnerRepository.Save();
```

모델 클래스에 유효성 검사 및 비즈니스 규칙 추가하기

유효성 검사와 비즈니스 규칙을 적용하는 것은 데이터를 처리하는 애플리케이션에 있어서는 필수 불가결한 요구 사항이다.

스키마 유효성 검사

LINQ to SQL 디자이너를 이용하여 모델 클래스를 정의하는 경우 데이터 모델 클래스의 속성이 사용하는 데이터 타입은 데이터베이스 테이블의 데이터 타입과 일치하게 된다. 예를 들어, Dinners 테이블의 "EventDate" 컬럼의 데이터 타입이 "datetime" 타입이라면 LINQ to SQL 디자이너에 의해 생성된 데이터 모델 클래스의 속성은 DateTime 타입(.NET의 내장 데이터 타입이다)을 사용하게 된다. 즉, 이 속성에 정수나 불린 값을 대입하려 하면 컴파일 오류가 발생하게 되며, 런타임에 유효하지 않은 문자열을 묵시적으로 DateTime 타입으로 변환하려고 시도하는 경우에도 오류를 발생하게 된다.

또한 LINQ to SQL은 문자열을 사용하는 경우 SQL 구문과 관련된 문자열들을 자동으로 처리한다. 따라서 LINQ to SQL을 사용하면 SQL 인젝션 공격에 대해 걱정할 필요가 없다.

유효성 검사와 비즈니스 규칙

기본적인 사항이라 할 수 있는 데이터 타입에 대한 유효성 검사는 유용하기는 하지만 충분하다고 보기는 어렵다. 대부분 실제 환경에서의 시나리오들은 여러 개의 속성에 동시에 적용되거나 코드의 실행, 그리고 모델의 상태(예를 들면, 모델이 추가 수정 혹은 삭제된 상태인지 아니면 경우에 따라 보관된 상태인지 등) 인식 등에 사용될 수 있는 보다 풍부한 유효성 검사 로직을 요구한다.

이러한 유효성 검사 규칙을 정의하고 적용하기 위한 다양한 패턴과 프레임워크들이 존재하며 .NET 프레임워크를 기반으로 하는 것들도 상당수 존재한다. 물론 ASPNET MVC 애플리케이션 내에서는 이러한 패턴이나 프레임워크의 대부분을 그대로 활용할 수 있다.

NerdDinner 애플리케이션의 목적 상 우리는 Dinner 모델 객체에 IsValid 속성과 GetRuleViolations() 메서드를 정의하는 비교적 단순하며 직관적인 방법을 사용할 것이다. IsValid 속성은 유효성 검사 및 비즈니스 규칙의 유효성 여부에 따라 true 혹은 false를 리턴하게 될 것이며 GetRuleViolations() 메서드는 규칙에 어긋난 에러들의 목록을 리턴한다.

IsValid 속성과 GetRuleViolations() 메서드를 구현하기 위해서는 프로젝트에 부분 클래스(Partial Class)를 추가해야 한다. 부분 클래스는 VS 디자이너에 의해 관리되는 클래스들(LINQ to SQL 디자이너에 의해 관리되는 Dinners 객체와 같은 클래스들)에 속성이나 메서드 혹은 이벤트를 추가하기 위한 용도로 활용되며, 디자이너 도구들이 우리가 작성한 코드를 날려버리지 않도록 방지하기 위한 목적으로도 활용할 수 있다.

새로운 부분 클래스를 추가하려면 프로젝트의 WModels 폴더를 마우스 오른쪽 버튼으로 클릭하고 "새 항목 추가" 메뉴를 선택한다. 그런 후 "새 항목 추가" 대화 상자에서 "클래스" 템플릿을 선택한 후 파일의 이름을 Dinner.cs라고 입력한다.

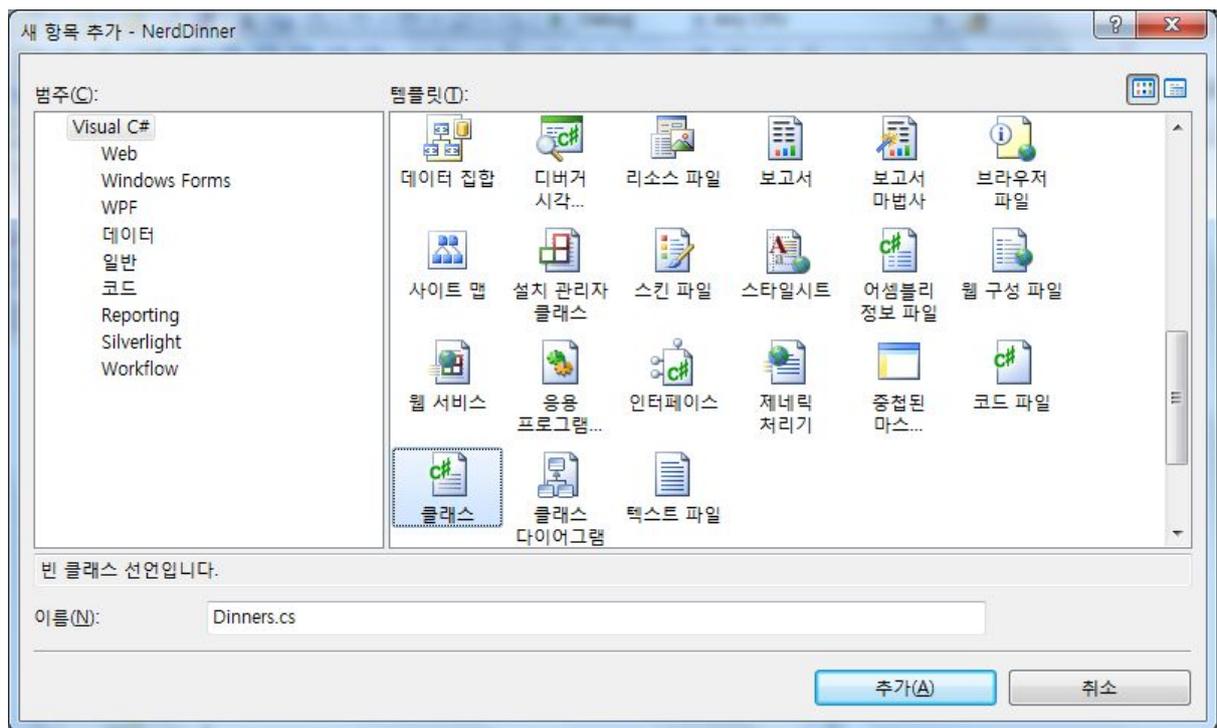


그림 1-51

"추가" 버튼을 클릭하면 Dinner.cs 파일이 생성하고 해당 파일을 IDE를 통해 열어준다. 그런 후 다음의 코드와 같이 기본적인 유효성 검사 및 비즈니스 규칙을 수행하는 프레임워크를 구현할 수 있다.

```
public partial class Dinners
{
```

```

public bool IsValid
{
    get { return (GetRuleViolations().Count() == 0); }
}
public IEnumerable<RuleViolation> GetRuleViolations()
{
    yield break;
}
partial void OnValidate(ChangeAction action)
{
    if (!IsValid)
        throw new ApplicationException("Rule violations prevent saving");
}
}

public class RuleViolation
{
    public string ErrorMessage { get; private set; }
    public string PropertyName { get; private set; }
    public RuleViolation(string errorMessage)
    {
        ErrorMessage = errorMessage;
    }
    public RuleViolation(string errorMessage, string propertyName)
    {
        ErrorMessage = errorMessage;
        PropertyName = propertyName;
    }
}

```

이 코드에 대해 간략히 설명하자면 다음과 같다.

- Dinners 클래스는 "partial" 키워드를 이용하여 부분 클래스로 선언된다. 따라서 Dinners 클래스에 구현한 코드는 LINQ to SQL 디자이너가 생성해준 코드와 병합되어 하나의 클래스로 컴파일된다.
- GetRuleViolations() 메서드는 (우리가 간단하게 구현할) 유효성 검사 및 비즈니스 규칙을 실행하며, 각각의 규칙 오류에 대한 보다 상세한 정보를 제공하는 RuleViolation 클래스의 컬렉션을 리턴한다.
- IsValid 속성은 Dinners 객체가 유효성 검사에 실패했을 때 생성되는 RuleValidation 객체를 가지고 있는지 여부를 간편하게 확인하기 위해 정의하였다. 개발자는 이 속성을 이용하여 언제든지 (예외를 발생시키지 않고) Dinners 객체가 올바른 상태인지를 확인할 수 있다.
- 부분 메서드(Partial Method)인 OnValidate() 메서드는 Dinners 객체가 데이터베이스에 보관될 수 있는 상태인지를 개발자가 언제든지 알 수 있도록 하기 위해 LINQ to SQL이 제공하는 메서드이다. 우리가 앞서 구현한 OnValidate() 메서드는 Dinners 객체가 데이터베이스에 저장되기 전에 유효성 검사 및 비즈니스 규칙에 오류가 없는지를 확인한다. 만일 Dinners 객체가 유효한 상태가 아니라면 OnValidate() 메서드는 LINQ to SQL이 트랜잭션을 취소할 수 있도록 예외를 발생시킨다.

이와 같은 방법을 통해 유효성 검사 및 비즈니스 규칙 검사를 위한 간단한 프레임워크를 애플리케이션에 통합할 수 있다. 이제 GetRuleViolations() 메서드에 다음과 같이 규칙을 추가해보자.

```
public IEnumerable<RuleViolation> GetRuleViolations() {
    if (String.IsNullOrEmpty(Title))
        yield return new RuleViolation("Title required", "Title");
    if (String.IsNullOrEmpty(Description))
        yield return new RuleViolation("Description required", "Description");
    if (String.IsNullOrEmpty(HostedBy))
        yield return new RuleViolation("HostedBy required", "HostedBy");
    if (String.IsNullOrEmpty(Address))
        yield return new RuleViolation("Address required", "Address");
    if (String.IsNullOrEmpty(Country))
        yield return new RuleViolation("Country required", "Country");
    if (String.IsNullOrEmpty(ContactPhone))
        yield return new RuleViolation("Phone# required", "ContactPhone");
    if (!PhoneValidator.IsValidNumber(ContactPhone, Country))
        yield return new RuleViolation("Phone# does not match country",
            "ContactPhone");
    yield break;
}
```

이 예제에서는 C#의 "yield return" 구문을 이용하여 RuleViolation 객체들을 순차적으로 리턴할 수 있도록 구현하였다. 위의 코드에서 처음 6개의 규칙은 단순히 Dinners 객체의 문자열 속성들이 빈 문자열이나 null 값을 가지지 못하도록 하기 위해 사용하였다. 마지막 규칙은 PhoneValidator.IsValidNumber() 메서드를 호출하여 Dinners 객체의 ContactPhone 속성에 대입된 값이 Dinners 객체의 Country 속성 값에 보관된 국가별 설정을 토대로 올바른 전화 번호인지 여부를 판단한다.

전화 번호와 같은 데이터의 유효성을 검사하기 위해서는 .NET의 정규 표현식(Regular Expression)을 활용할 수도 있다. 다음의 코드는 PhoneValidator 클래스가 국가별 설정에 따른 유효한 전화 번호임을 확인하기 위해 정규 표현식을 사용하는 방법을 보여준다.

```
public class PhoneValidator {
    static IDictionary<string, Regex> countryRegex =
        new Dictionary<string, Regex>() {
            { "USA", new Regex("^([2-9]\\d{2}-\\d{3}-\\d{4}$)"),
            { "UK", new
                Regex("(^1300\\d{6}$)|(^1800|1900|1902\\d{6}$)|(^0[2|3|7|8]{1}[0-9]{8}$)|(^13\\d{4}$)|(^04\\d{2,3}\\d{6}$)"),
            { "Netherlands", new Regex("(^\\+[0-9]{2}|^\\+[0-9]{2}\\(0\\)|^\\+(\\+[0-9]{2}\\)\\(0\\)|^00[0-9]{2}|^0([0-9]{9}$|[0-9]\\-\\s){10}$)"),
            };
    public static bool IsValidNumber(string phoneNumber, string country) {
        if (country != null && countryRegex.ContainsKey(country))
            return countryRegex[country].IsMatch(phoneNumber);
        else
            return false;
    }
    public static IEnumerable<string> Countries {
        get {
```

```
return countryRegex.Keys;
}
}
}
```

이제 우리가 Dinners 객체를 생성하거나 수정하려 할 때 지금까지 구현한 유효성 검사 규칙이 적용된다. 개발자들은 Dinners 객체가 유효한 상태인지를 미리미리 파악할 수 있으며, 예외를 발생시키지 않고도 어떤 문제로 인해 Dinners 객체가 유효한 상태가 되지 못했는지 손쉽게 조회할 수 있게 된다.

```
Dinner dinner = dinnerRepository.GetDinner(5);
dinner.Country = "USA";
dinner.ContactPhone = "425-555-BOGUS";
if (!dinner.IsValid) {
var errors = dinner.GetRuleViolations();
// 오류를 수정하기 위한 코드를 실행한다.
}
```

만일 유효하지 않은 상태인 Dinners 객체를 저장하려고 하면 DinnerRepository 클래스의 Save() 메서드를 호출할 때 예외가 발생하게 된다. 이는 Dinner.OnValidate() 메서드가 Dinners 객체가 규칙에 위반되는 경우 예외를 발생시키도록 구현되어 있기 때문이다. 우리는 이 예외를 캐치하여 다음과 같이 오류를 수정하는 코드를 작성할 수 있다.

```
Dinner dinner = dinnerRepository.GetDinner(5);
try {
dinner.Country = "USA";
dinner.ContactPhone = "425-555-BOGUS";
dinnerRepository.Save();
}
catch {
var errors = dinner.GetRuleViolations();
// 오류를 수정하기 위한 코드를 실행한다.
}
```

우리가 지금까지 구현한 유효성 검사 및 비즈니스 규칙 검사 로직은 UI 수준이 아니라 모델 수준에서 구현되었기 때문에 애플리케이션에 대해 전역적으로 적용할 수 있다. 만일 규칙을 수정하거나 새로운 규칙을 추가한다면 이는 Dinners 객체를 사용하는 모든 코드에 적용된다. 애플리케이션과 UI 로직에 어떠한 영향을 미치지 않고도 비즈니스 규칙을 한 곳에서 변경할 수 있는 유연함을 얻었다는 것은 애플리케이션이 잘 설계되었다는 것을 의미하며, 이는 결국 MVC 프레임워크를 사용함으로써 얻을 수 있는 이점이다.

컨트롤러와 뷰

지금까지의 웹 프레임워크(ASP나 PHP, ASP.NET 웹 폼 모델 등) 환경에서는 요청된 URL은 주로 서버의 디스크에 저장된 파일과 매칭되었다. 예를 들어, "/Products.aspx"나 "/Products.php"와 같은

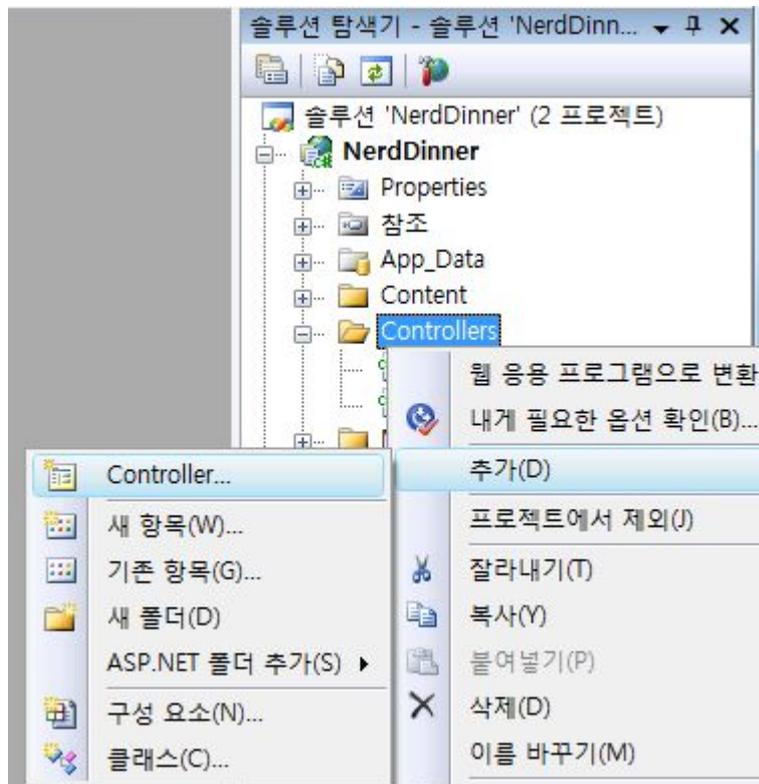
URL은 "Products.aspx" 파일이나 "Products.php" 파일에 의해 처리되었다.

반면 웹 기반의 MVC 프레임워크는 예전과는 조금 다른 방법을 사용하여 URL을 서버 측 코드로 매핑한다. 즉, 요청된 URL을 파일에 매핑하는 대신 특정 클래스에 매핑한다. 이처럼 요청을 처리하도록 매핑되는 클래스들을 "컨트롤러(Controller)"라고 하며 컨트롤러 클래스들은 HTTP 요청을 수행하고 사용자의 입력을 처리하며 데이터를 조회하거나 저장하고 클라이언트에 되돌려 줄 응답(HTML을 표시하거나 파일을 다운로드 하거나 다른 URL로 이동하거나 하는 등의 응답)을 결정한다.

지금까지 우리는 NerdDinner 애플리케이션을 위한 모델 클래스들을 구성하였으므로 이제는 사이트를 통해 저녁 모임 데이터를 나열하거나 혹은 보다 상세한 정보를 보여주기 위한 기능을 제공할 컨트롤러 클래스들을 애플리케이션에 추가해 보기로 하자.

DinnersController 컨트롤러 클래스 추가하기

Visual Studio 2008의 솔루션 탐색기에서 웹 프로젝트의 "Controllers" 폴더를 마우스 오른쪽 버튼으로 클릭하고 다음 그림과 같이 "추가 > Controller" 메뉴를 선택하자(Ctrl + M, Ctrl + C 단축키를 이용해도 된다).



=====

잠깐만

ASP.NET MVC 1.0은 현재 한글 리소스를 지원하지 않는다. 따라서 Visual Studio 2008 한글 버전을 사용하고 있다 하더라도 ASP.NET MVC 1.0을 설치하면 그와 관련된 메뉴나 대화 상자 등의 리소스는 모두 영문으로 표시된다.

그러면 다음 그림과 같이 "Add Controller" 대화 상자가 나타날 것이다.

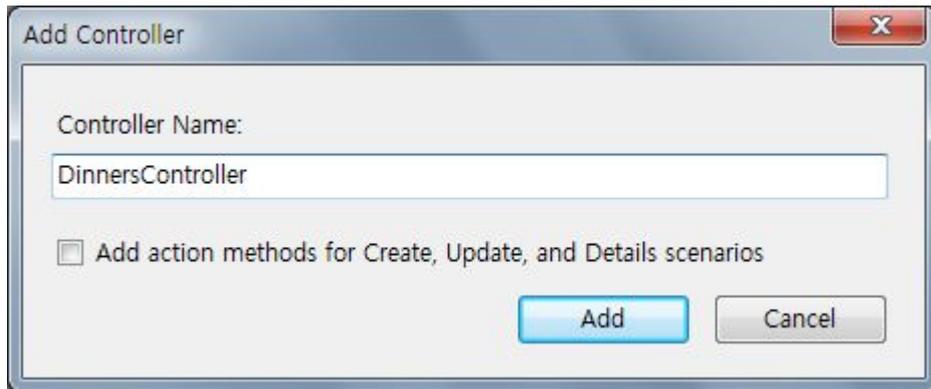


그림 1-53

새로운 컨트롤러 클래스의 이름은 "DinnersController"라고 입력하고 "Add" 버튼을 클릭하자. 그러면 Visual Studio는 다음 그림과 같이 /Controller 폴더에 DinnersController.cs 라는 이름의 파일을 추가한다.

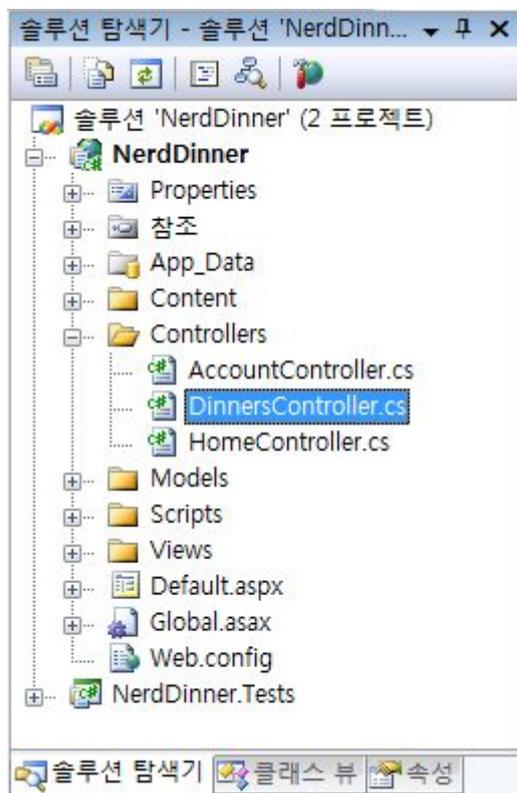


그림 1-54

추가된 새로운 클래스 파일은 자동적으로 IDE의 코드 편집기를 이용하여 열리게 된다.

Index()와 Details() 액션 메서드 추가하기

우리는 애플리케이션을 사용하는 사용자가 앞으로 예정된 저녁 모임 데이터의 목록을 살펴보고 해당 목록에서 특정 데이터를 클릭하여 상세 정보를 열람할 수 있도록 구현할 것이다. 이 두 가지 기능을 제공하기 위해서 우리는 다음과 같은 두 개의 URL을 제공할 예정이다.

| URL | 설명 |
|-----------------------|--|
| /Dinners/ | 예정된 저녁 모임 데이터를 나열하는 HTML을 표시한다. |
| /Dinners/Details/[id] | "id" 매개 변수를 통해 지정된 데이터의 상세 정보를 표시한다. URL에 포함되어 전달될 이 매개 변수는 데이터베이스의 DinnerID 컬럼에서 사용하는 값이다. 예를 들어, /Dinners/Details/2라는 URL은 DinnerID 컬럼의 값이 2인 레코드의 상세 정보를 보여주는 HTML을 표시하게 된다. |

위의 두 URL을 제공하려면 우선은 DinnersController 클래스에 다음과 같이 두 개의 "액션 메서드 (Action Method)"를 public 메서드로 추가하면 된다.

```
public class DinnersController : Controller {  
    //  
    // GET: /Dinners/  
    public void Index() {  
        Response.Write("<h1>예정된 저녁 모임 </h1>");  
    }  
    //  
    // GET: /Dinners/Details/2  
    public void Details(int id) {  
        Response.Write("<h1>상세 보기: " + id + "</h1>");  
    }  
}
```

이제 애플리케이션을 실행하고 브라우저를 통해 URL을 호출해보자. 브라우저의 주소 표시줄에 "/Dinners/" URL을 입력하면 Index() 메서드가 호출되어 다음과 같은 응답을 보여줄 것이다.

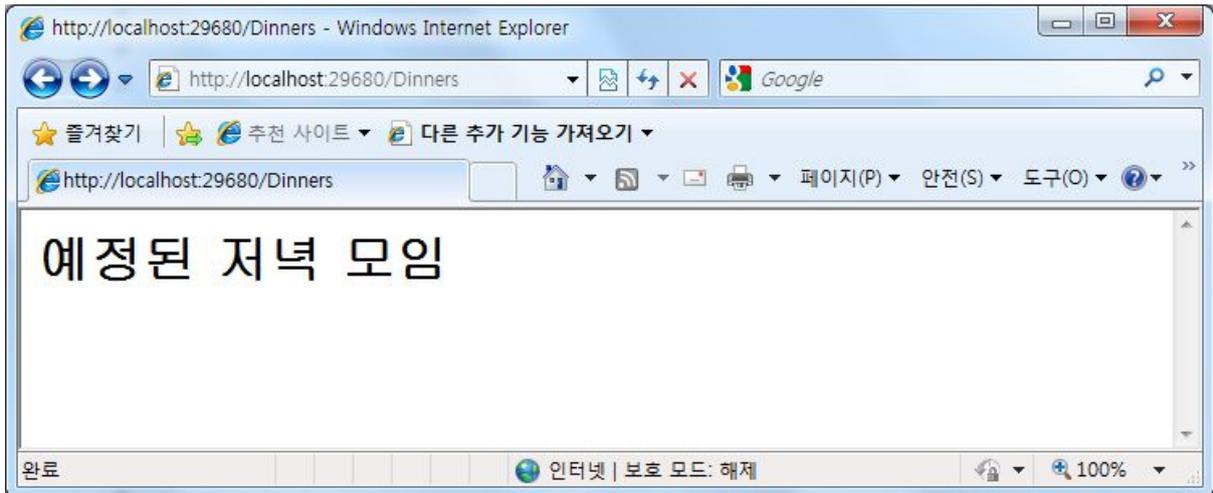
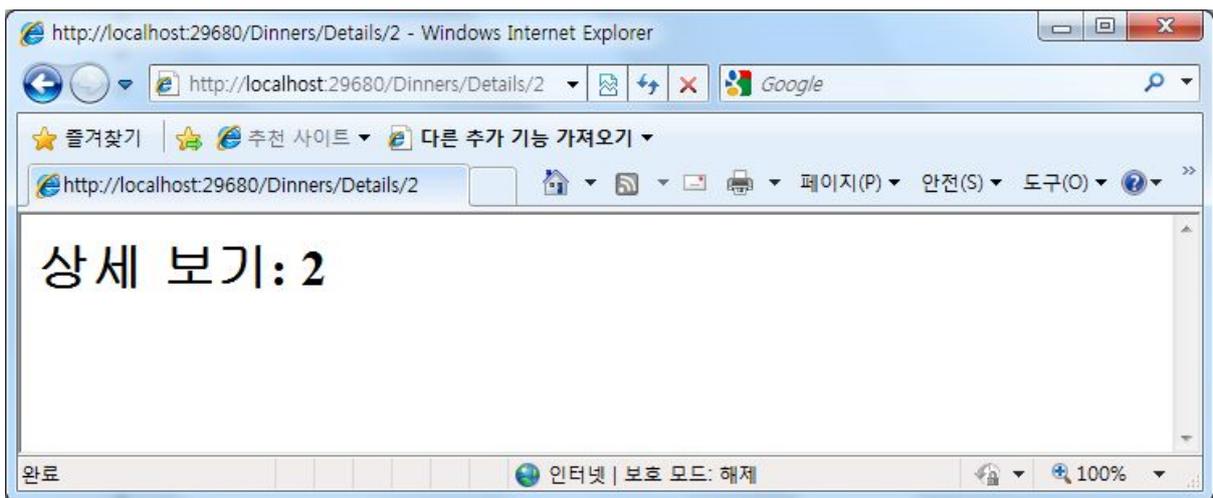


그림 1-55

마찬가지로 `/Dinners/Details/2` URL을 입력하면 `Details()` 메서드가 호출되어 다음과 같은 응답을 보여준다.



이 시점에서 여러분은 한 가지 의문이 생길 것이다. “대체 MVC 프레임워크는 URL만 보고 어떻게 `DinnersController` 클래스의 인스턴스를 생성하고 적절한 메시지를 호출해야 한다는 것을 알 수 있었을까?” 이와 같은 동작을 이해하기 위해서는 MVC 프레임워크의 URL 라우팅이 어떻게 동작하는지를 우선 간략하게나마 살펴보아야 한다.

ASP.NET MVC의 URL 라우팅

ASP.NET MVC는 URL을 컨트롤러 클래스와 유연하게 연결할 수 있는 강력한 URL 라우팅 엔진을 내장하고 있다. (역자 주: ASP.NET MVC의 URL 라우팅 기능은 .NET 프레임워크 3.5 서비스 팩 1에 포함되어 있으며 기존의 ASP.NET 웹 폼 환경에서도 사용이 가능하다.) 이 URL 라우팅 엔진을 활용하면 ASP.NET MVC가 어떤 컨트롤러 클래스를 생성하고 어떤 메시지를 호출할 것인지를 선택하는 방법을 완벽하게 제어할 수 있으며, 서로 다른 방법으로 URL/쿼리문자열 형태로 전달되는

변수들을 자동적으로 파싱하여 메서드의 매개 변수로 대입하도록 할 수 있다. 이를 통해 사이트가 SEO(검색 엔진 최적화: Search Engine Optimization)에 완벽하게 최적화될 수 있으며 우리가 원하는 어떤 형태의 URL도 제공할 수 있게 된다.

새로운 ASP.NET MVC 프로젝트에는 기본적으로 미리 정의된 URL 라우팅 규칙이 이미 등록되어 있다. 덕분에 우리는 별도의 설정을 추가하지 않고도 손쉽게 애플리케이션의 개발을 시작할 수 있다. 기본적으로 등록된 URL 라우팅 규칙은 프로젝트의 "Application" 클래스에서 찾을 수 있다. Visual Studio의 솔루션 탐색기에서 프로젝트의 루트에 위치한 "Global.asax.cs" 파일을 더블 클릭하여 열어보자.

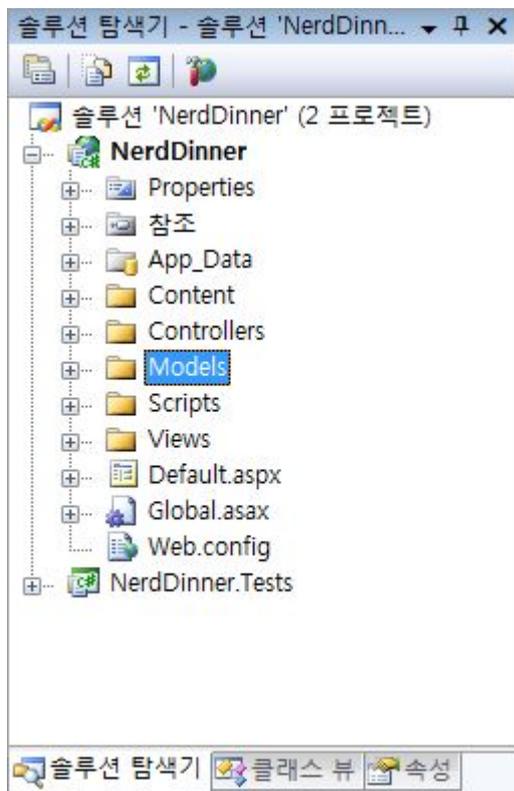


그림 1-57

기본적으로 등록된 URL 라우팅 규칙은 Application 클래스의 "RegisterRoutes" 메서드에 다음과 같이 작성되어 있다.

```
public void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Default", // 라우팅 규칙의 이름
        "{controller}/{action}/{id}", // URL과 매개 변수
        new { controller="Home", action="Index", id="" } // 매개 변수들의 기본 값
    );
}
```

위의 코드에서 호출하는 "routes.MapRoute()" 메서드는 "{controller}/{action}/{id}" 형식을 이용하여 요청된 URL을 컨트롤러 클래스와 매핑하는 기본 URL 라우팅 규칙을 등록한다. 여기서 "controller"는 인스턴스가 생성될 컨트롤러 클래스의 이름이며 "action"은 호출될 public 메서드의 이름이다. 마지막으로 "id"는 URL에 포함되어 메서드의 인자로 전달될 선택적인 매개 변수이다. "MapRoute()" 메서드를 호출할 때 사용한 세 번째 매개 변수는 controller/action/id 변수들이 URL에 명시되지 않은 경우에 적용될 기본 값들을 정의한 객체이다. (예제의 경우 controller 변수에는 "Home"이 지정되며 action 변수에는 "Index", id 변수에는 빈 문자열이 지정된다.)

다음의 표는 기본적으로 등록된 "{controller}/{action}/{id}" 형식의 URL 라우팅 규칙에 의해 매핑될 수 있는 다양한 URL들을 보여준다.

| URL | 컨트롤러 클래스 | 액션 메서드 | 매개 변수 |
|--------------------|-------------------|-------------|----------|
| /Dinners/Details/5 | DinnersController | Details(id) | id=5 |
| /Dinners/Edit/5 | DinnersController | Edit(id) | id=5 |
| /Dinners/Create | DinnersController | Create() | 매개 변수 없음 |
| /Dinners | DinnersController | Index() | 매개 변수 없음 |
| /Home | HomeController | Index() | 매개 변수 없음 |
| / | HomeController | Index() | 매개 변수 없음 |

마지막 세 개의 행은 기본 값이 사용된 경우를 보여준다. 즉, controller 변수에는 "Home"이, action 변수에는 "Index"가, 그리고 id 변수에는 빈 문자열이 지정된 경우를 뜻한다. 만일 액션 메서드의 이름을 지정하지 않는다면 "Index" 메서드가 기본 값으로 사용되기 때문에 "/Dinners"와 "/Home"과 같은 URL들은 각각의 컨트롤러 클래스에 정의된 Index() 메서드를 호출하게 된다. 또한 컨트롤러의 이름을 지정하지 않는다면 기본 값은 "Home" 컨트롤러이기 때문에 "/" URL은 HomeController 클래스의 인스턴스를 생성하고 Index() 메서드를 호출하게 된다.

만일 이 기본 라우팅 규칙이 마음에 들지 않는다면 RegisterRoutes 메서드를 수정하여 손쉽게 라우팅 규칙을 변경할 수 있다. 그러나 NerdDinner 애플리케이션의 경우에는 기본 라우팅 규칙을 수정하지 않고 그대로 사용할 것이다.

DinnerController 클래스에서 DinnerRepository 클래스 활용하기

이제 Index() 메서드와 Details() 메서드가 모델 클래스들을 이용하여 실제로 기능을 수행할 수 있도록 구현해보자.

실제 동작을 구현하기 위해 앞서 구현했던 DinnerRepository 클래스를 활용할 것이다. 우선 "using" 구문을 이용하여 "NerdDinner.Models" 네임스페이스에 대한 참조를 추가한 후 DinnersController 클래스에 DinnerRepository 클래스의 인스턴스를 필드로 선언한다.

이 장의 후반부에서는 “의존성 주입(DI: Dependency Injection)”의 개념에 대해 설명하고 컨트롤러 클래스들에 대한 단위 테스트를 보다 손쉽게 실행할 수 있는 여러 가지 방법을 소개할 것이다. 그러나 지금 당장은 DinnerRepository 클래스의 인스턴스를 다음과 같이 생성하여 사용하기로 하자.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using NerdDinner.Models;
namespace NerdDinner.Controllers {
public class DinnersController : Controller {
DinnerRepository dinnerRepository = new DinnerRepository();
//
// GET: /Dinners/
public void Index() {
var dinners = dinnerRepository.FindUpcomingDinners().ToList();
}
//
// GET: /Dinners/Details/2
public void Details(int id) {
Dinners dinner = dinnerRepository.GetDinner(id);
}
}
}
```

여기까지 구현했으면 이제 클라이언트에 응답으로 전달할 HTML 코드를 생성할 준비를 마친 셈이다.

뷰 활용하기

액션 메서드 내에서 Response.Write() 메서드를 이용하여 HTML을 렌더링할 수도 있지만 이 방법은 그다지 보기 좋은 방법은 아니다. 이보다 더 나은 방법은 DinnersController 클래스의 액션 메서드에서는 애플리케이션 로직과 데이터 로직만을 수행하고 HTML을 표현하는 데 필요한 데이터를 별도의 “뷰(View)” 템플릿에 전달하여 뷰로 하여금 HTML의 출력을 담당하도록 하는 것이다. 잠시 후에 살펴보겠지만 “뷰” 템플릿은 주로 HTML 마크업 코드와 렌더링 코드를 포함하는 텍스트 파일이다.

컨트롤러의 로직과 뷰의 로직을 분리하면 여러 가지 이점을 얻을 수 있다. 특히 애플리케이션 로직과 UI 렌더링 로직 사이의 “관계”를 명확하게 분리하여 서로 독립적으로 동작하도록 할 수 있다. 이렇게 함으로써 애플리케이션 로직에 대한 단위 테스트를 UI 렌더링 로직과는 무관하게 수행할 수 있다. 또한 애플리케이션 로직을 수정하지 않고도 UI 렌더링 로직을 수정하는 것도 가능하다. 그럼으로써 하나의 프로젝트를 수행하는 개발자와 디자이너의 협업에도 도움이 된다.

DinnersController 클래스에 정의한 액션 메서드의 리턴 타입을 "void"에서 "ActionResult"로 변경하여 HTML UI의 응답을 뷰 템플릿을 이용하여 수행할 수 있다. 메서드의 시그니처(Signature)를 변경했으면 아래 코드와 같이 Controller 기반 클래스가 제공하는 View() 메서드를 호출하여 "ViewResult" 객체를 리턴할 수 있다.

```
public class DinnersController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
    // GET: /Dinners/
    public ActionResult Index() {
        var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        return View("Index", dinners);
    }
    //
    // GET: /Dinners/Details/2
    public ActionResult Details(int id) {
        Dinners dinner = dinnerRepository.GetDinner(id);
        if (dinner == null)
            return View("NotFound");
        else
            return View("Details", dinner);
    }
}
```

위의 코드에서 사용한 View() 메서드의 시그니처는 다음과 같다.

```
ViewResult View(string viewName, object model);
```

그림 1-58

View() 메서드의 첫 번째 매개 변수는 우리가 HTML 응답을 렌더링하기 위해 사용할 뷰 템플릿의 이름이며 두 번째 매개 변수는 뷰 템플릿이 HTML 응답을 렌더링하기 위해 필요로 하는 데이터를 가진 모델 객체이다.

Index() 액션 메서드에서는 View() 메서드를 호출하여 저녁 모임에 대한 목록을 나열하는 HTML 응답을 "Index" 뷰 템플릿을 이용하여 수행하도록 지정하고 있으며 다음과 같이 Dinners 객체의 컬렉션을 뷰 템플릿에 전달한다.

```
//
// GET: /Dinners/
public ActionResult Index() {
    var dinners = dinnerRepository.FindUpcomingDinners().ToList();
    return View("Index", dinners);
}
```

Details() 액션 메서드에서는 URL에 지정된 ID 값을 이용하여 Dinners 객체를 조회한다. 조회한

Dinners 객체가 유효한 상태라면 View() 메서드를 호출하여 "Details" 뷰 템플릿이 조회된 Dinners 객체를 렌더링하도록 한다. 만일 조회된 Dinners 객체가 유효하지 않은 상태라면 (단순히 뷰 템플릿 이름만 지정하도록 재정의된 View() 메서드를 이용하여) "NotFound" 뷰 템플릿을 출력함으로써 원하는 Dinners 객체를 얻어오지 못했음을 알리는 에러 메시지를 출력한다.

```
//  
// GET: /Dinners/Details/2  
public ActionResult Details(int id) {  
    Dinners dinner = dinnerRepository.FindDinner(id);  
    if (dinner == null)  
        return View("NotFound");  
    else  
        return View("Details", dinner);  
}
```

이제 "NotFound", "Details" 그리고 "Index" 뷰 템플릿을 구현해보자.

"NotFound" 뷰 템플릿 구현하기

우선은 요청한 Dinners 객체를 찾을 수 없다는 오류 메시지를 보여줄 "NotFound" 뷰 템플릿부터 구현해보자.

새로운 뷰 템플릿을 추가하려면 컨트롤러의 액션 메서드 내에 텍스트 커서를 위치하고 마우스 오른쪽 버튼을 클릭한 후 "Add View" 메뉴를 선택한다 (또는 Ctrl + M, Ctrl + V 단축키를 사용해도 된다).

```
//  
// GET: /Dinners/Details/2  
public ActionResult Details(int id)  
{  
    Dinners dinner = dinnerRepository.GetDinner(id);  
  
    if (dinner == null)  
        return View("NotFound");  
  
    return View(dinner);  
}
```

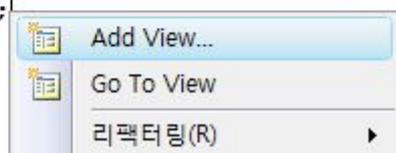


그림 1-59

그러면 다음 그림과 같이 "Add View" 대화 상자가 나타난다. 기본적으로 이 대화 상자에는 커서가 위치한 액션 메서드와 동일한 이름의 뷰를 생성하도록 설정되어 있다 (예제의 경우에는 "Details"이라는 이름이 지정되어 있다). 그러나 우리가 처음으로 생성할 뷰 템플릿은 "NotFound" 템플릿이므로 다음과 같이 뷰 템플릿 이름을 "NotFound"로 변경해야 한다.

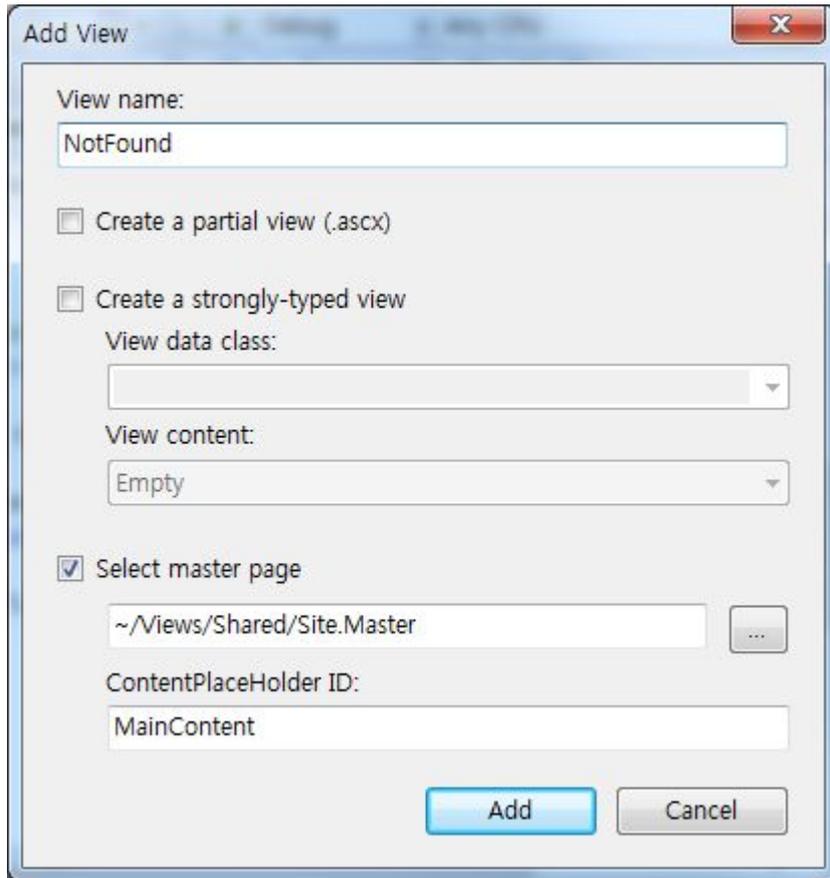
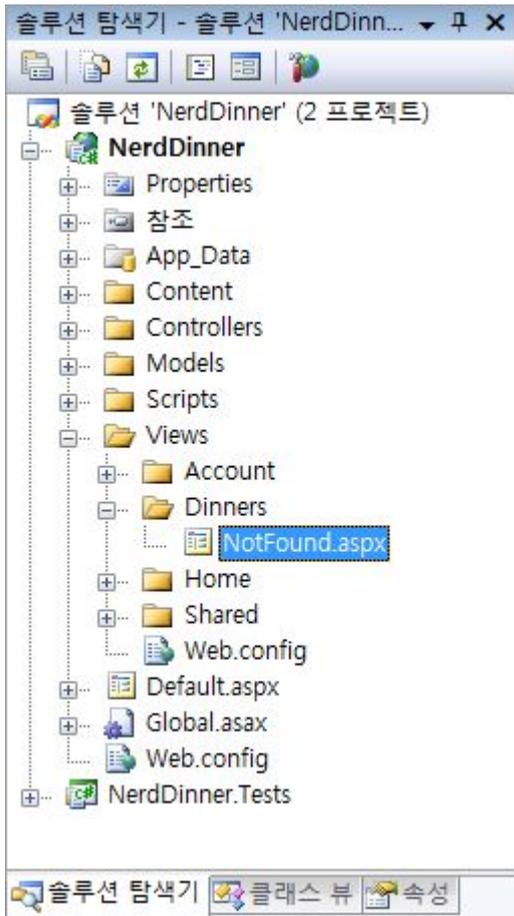


그림 1-60

“Add” 버튼을 클릭하면 Visual Studio는 프로젝트의 “\Views\Dinners\” 폴더에 “NotFound.aspx” 뷰 템플릿을 추가한다 (만일 디렉터리가 존재하지 않는다면 디렉터리도 생성된다).



또한 다음과 같이 “NotFound.aspx” 뷰 템플릿이 코드 윈도우에 열리게 된다.

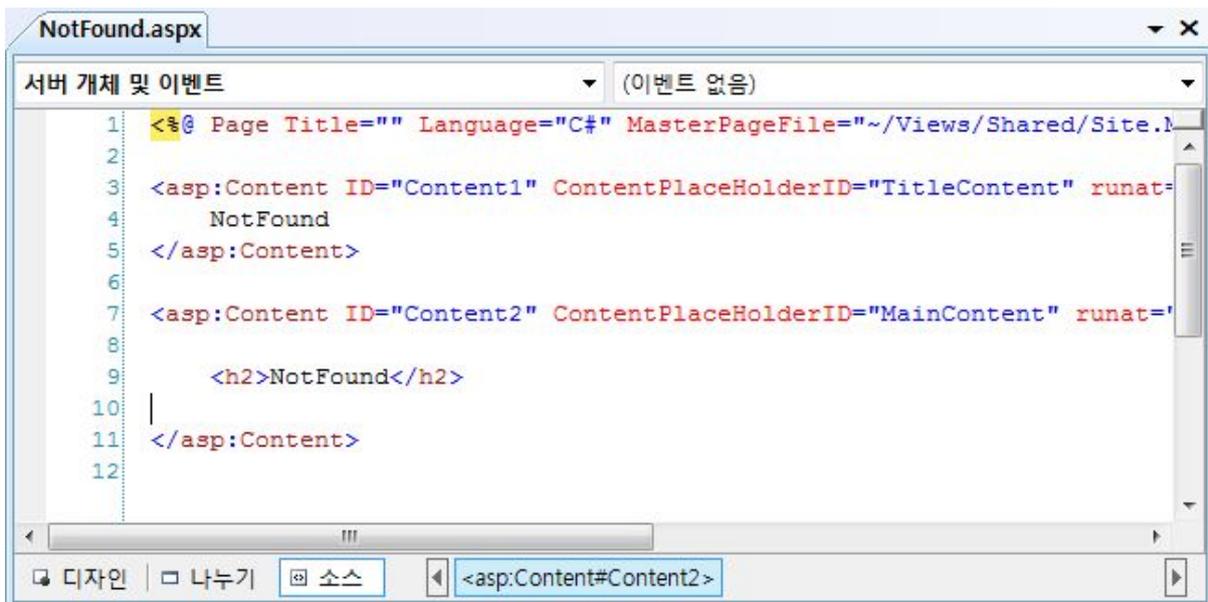


그림 1-62

기본적으로 뷰 템플릿 파일들은 우리가 콘텐츠와 코드를 추가할 수 있는 두 개의 “콘텐츠 영역

(Content Regions)"을 가지고 있다. 첫 번째 콘텐츠 영역은 응답에 사용할 HTML 페이지의 "제목 (Title)"을 변경할 수 있는 영역이며 두 번째 영역은 HTML 페이지의 "주요 콘텐츠(Main Content)"를 표시하는 영역이다.

"NotFound" 뷰 템플릿을 구현하기 위해 다음과 같이 기본적인 콘텐츠를 추가해보자.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
데이터를 찾을 수 없습니다.
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>데이터를 찾을 수 없습니다.</h2>
<p>죄송합니다. 요청하신 데이터는 삭제되었거나 존재하지 않습니다.</p>
</asp:Content>
```

이제 브라우저를 통해 이 페이지를 실행해보자. 브라우저의 주소 표시줄에 "/Dinners/Details/9999"와 같이 입력하면 현재 데이터베이스에 존재하지 않는 데이터를 조회하려고 시도했기 때문에 DinnersController.Details() 메서드는 다음과 같이 "NotFound" 뷰 템플릿을 렌더링할 것이다.

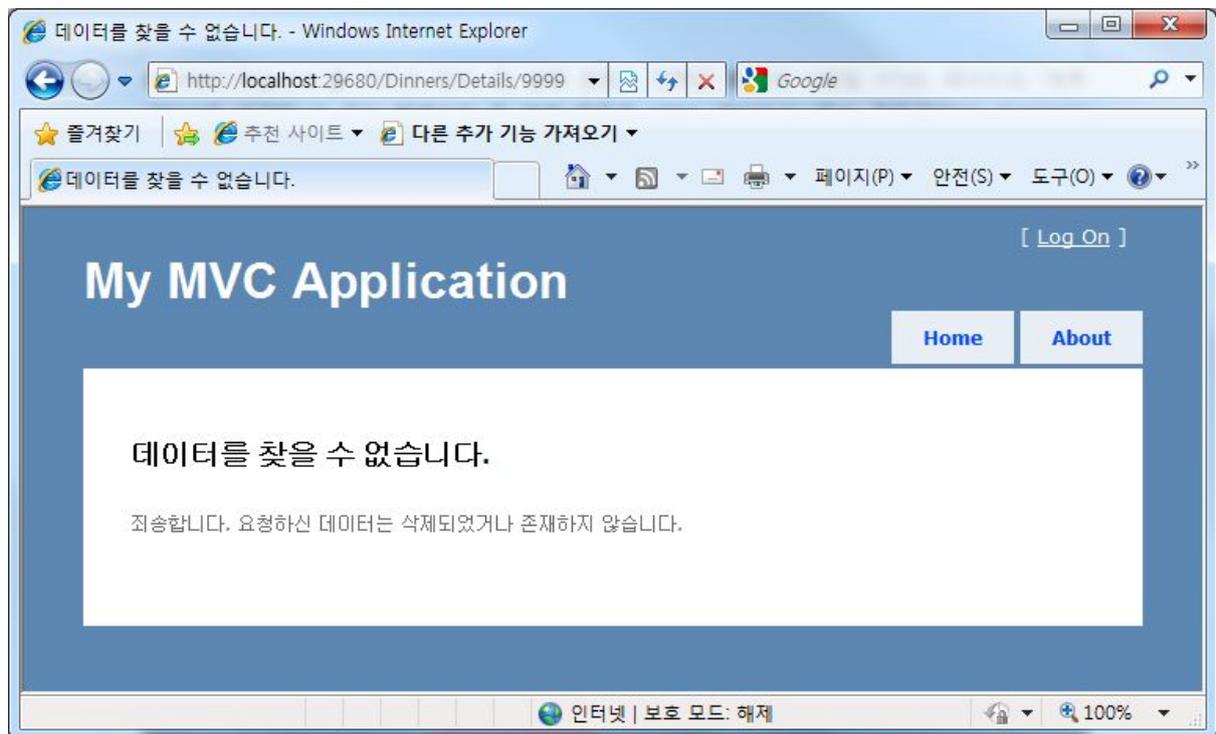


그림 1-63

위의 그림을 통해 알 수 있는 것은 우리가 렌더링한 HTML 코드를 둘러싼 일련의 HTML 코드가 존재한다는 사실이다. 이는 우리가 생성한 뷰 템플릿이 사이트를 구성하는 뷰 전체의 일관성을 유지하기 위해 사용되는 "마스터 페이지(Master Page)" 템플릿을 사용하고 있기 때문이다. 마스터 페이지의 동작 방식에 대해서는 이 장의 후반부에서 다시 설명하도록 하겠다.

“Details” 뷰 템플릿 구현하기

이번에는 Dinners 객체의 상세 정보를 보여주는 “Details” 뷰 템플릿을 구현해보자.

우선 Details 액션 메서드에 텍스트 커서를 옮기고 마우스 오른쪽 버튼을 클릭한 후 “Add View” 메뉴를 선택한다 (혹은 Ctrl + M, Ctrl + V 단축키를 사용해도 된다).

```
//  
// GET: /Dinners/Details/2  
public ActionResult Details(int id)  
{  
    Dinners dinner = dinnerRepository.GetDinner(id);  
  
    if (dinner == null)  
        return View("NotFound");  
  
    return View(dinner);  
}
```



그림 1-64

그러면 “Add View” 대화 상자가 나타나게 된다. 이번에는 기본적으로 지정된 뷰 이름 (“Details”) 를 그대로 사용한다. 또한 “Create strongly-typed View” 체크 상자를 클릭하여 체크 표시를 하고 (콤보 상자에서) 우리가 컨트롤러에서 뷰로 전달할 모델 객체의 타입 이름을 선택한다. 이 뷰에는 Dinners 객체를 전달할 것이므로 Dinners 객체의 전체 이름인 “NerdDinner.Models.Dinner” 항목을 선택한다.

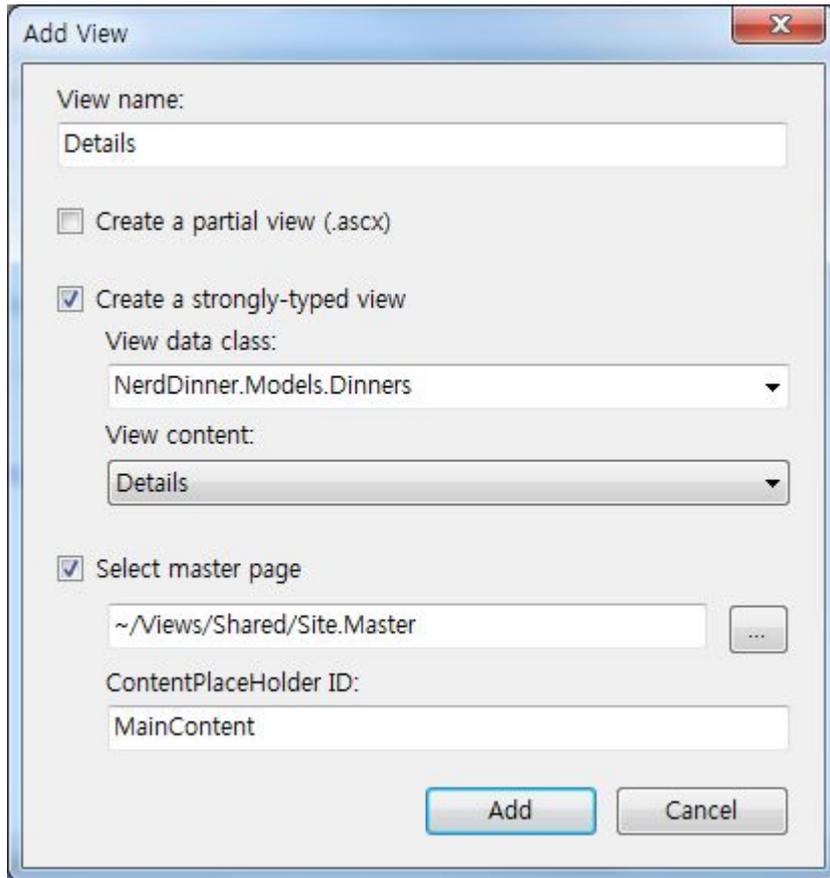


그림 1-65

앞서 구현했던 “NotFound” 뷰 템플릿을 생성할 때 “Empty View” 항목을 사용했던 것과 달리 이번에는 “View Content” 항목의 드롭다운 목록에서 “Details” 템플릿을 선택하여 자동적으로 뼈대 코드가 완성된 뷰 템플릿을 생성하도록 한다.

“자동 코드 생성(Scaffolding)” 기능을 선택하면 Details 뷰 템플릿에는 우리가 전달하는 Dinners 객체를 이용한 상세 보기 뷰 템플릿을 위한 코드가 자동적으로 생성되어 뷰 템플릿을 손쉽게 구현할 수 있다.

“Add” 버튼을 클릭하면 Visual Studio는 “\Views\Dinners\” 디렉터리에 “Details.aspx” 뷰 템플릿 파일을 생성한다.

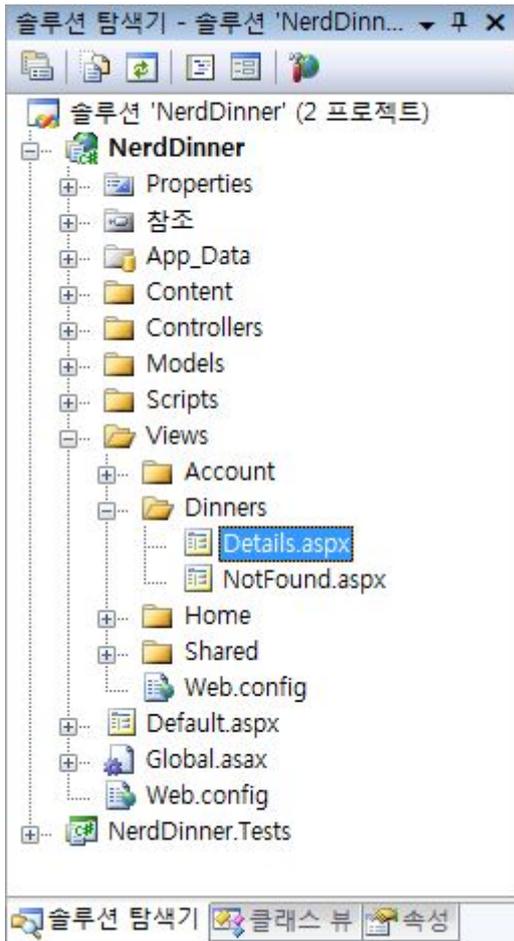


그림 1-66

또한 코드 편집기에 "Details.aspx" 파일이 열리게 되며 자동 코드 생성 기능에 의해 생성된 Dinner 모델 객체의 상세 보기 코드가 구현되어 있음을 볼 수 있다. 자동 코드 생성 기능은 .NET의 Reflection 기능을 이용하여 뷰 템플릿에 전달된 모델 객체가 제공하는 모든 public 속성들을 탐색하여 다음과 같이 적절한 콘텐츠를 표시한다.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
Details
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>Details</h2>
<fieldset>
<legend>Fields</legend>
<p>
DinnerID:
<%= Html.Encode(Model.DinnerID) %>
</p>
<p>
Title:
<%= Html.Encode(Model.Title) %>
</p>
<p>
EventDate:
<%= Html.Encode(String.Format("{0:g}", Model.EventDate)) %>
```

```

</p>
<p>
Description:
<%= Html.Encode (Model.Description) %>
</p>
<p>
HostedBy:
<%= Html.Encode (Model.HostedBy) %>
</p>
<p>
ContactPhone:
<%= Html.Encode (Model.ContactPhone) %>
</p>
<p>
Address:
<%= Html.Encode (Model.Address) %>
</p>
<p>
Country:
<%= Html.Encode (Model.Country) %>
</p>
<p>
Latitude:
<%= Html.Encode (String.Format ("{0:F}", Model.Latitude)) %>
</p>
<p>
Longitude:
<%= Html.Encode (String.Format ("{0:F}", Model.Longitude)) %>
</p>
</fieldset>
<p>
<%=Html.ActionLink("Edit", "Edit", new { id=Model.DinnerID }) %> |
<%=Html.ActionLink("Back to List", "Index") %>
</p>
</asp:Content>

```

이제 브라우저의 주소 표시줄에 "/Dinners/Details/1"과 같은 URL을 요청하면 상세 보기 기능이 자동으로 구현한 콘텐츠를 볼 수 있다. 이 URL은 우리가 데이터베이스를 생성할 때 수동으로 추가했던 데이터 중 첫 번째 데이터를 보여준다.

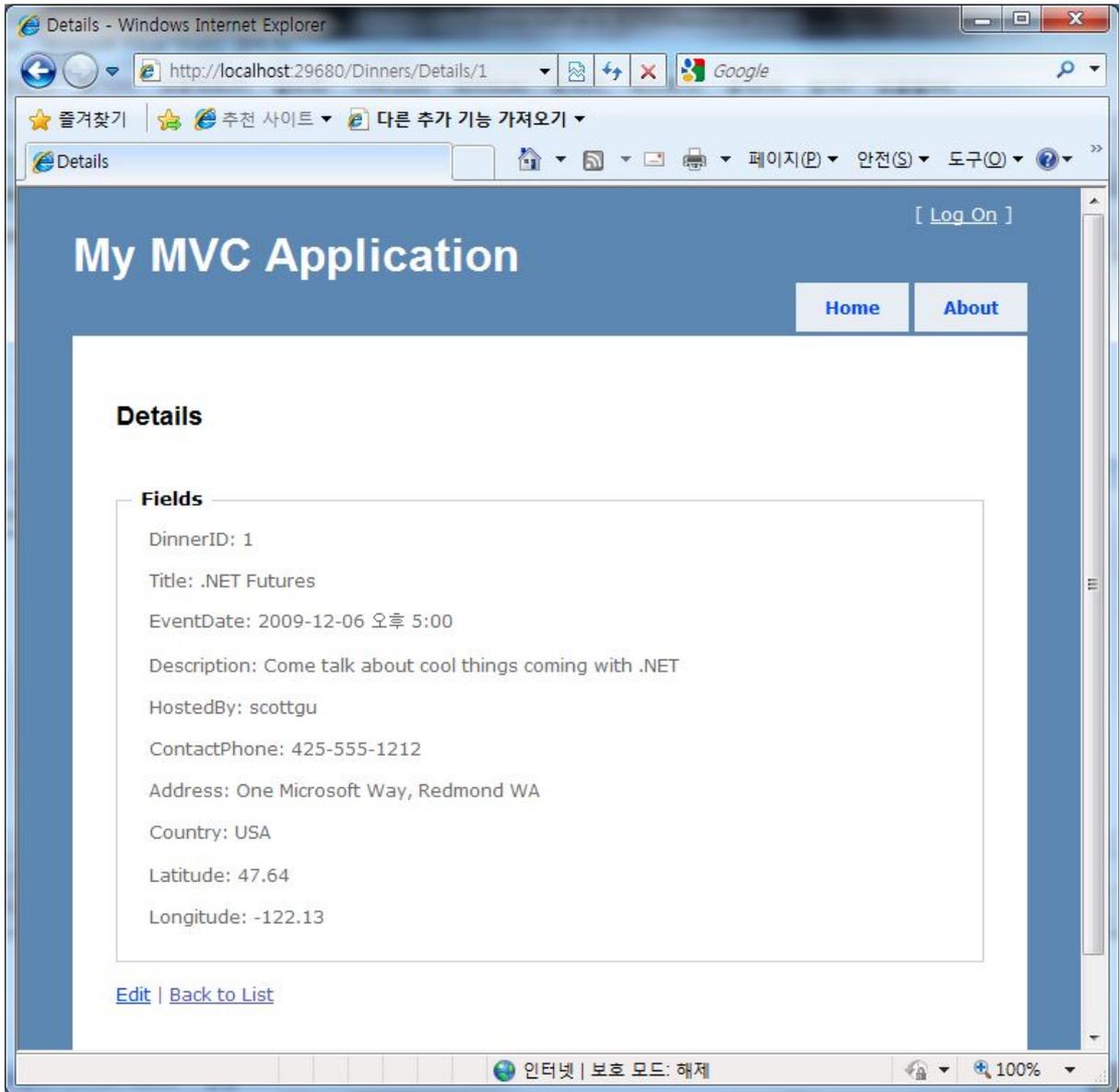


그림 1-67

그림에서 보듯이 자동 코드 생성 기능을 통해 매우 손쉽게 Details.aspx 뷰를 구현할 수 있다. 이를 바탕으로 우리가 원하는대로 UI를 재정의해보자.

Details.aspx 템플릿을 자세히 살펴보면 이 템플릿이 정적인 HTML 코드 외에도 렌더링 코드를 포함하고 있음을 알 수 있다. <% %> 코드 블록은 뷰 템플릿이 렌더링될 때 실행되며 <%= %> 코드 블록은 작성된 코드를 실행하고 그 결과를 템플릿의 응답 스트림에 출력한다.

우리는 컨트롤러에서 전달된 모델 객체를 강력하게 형식화된 "Model" 속성을 이용하여 액세스하는 코드를 뷰에 작성할 수 있다. Visual Studio는 강력하게 형식화된(Strongly-typed) "Model" 속성에 대해서는 다음 그림과 같이 완벽하게 인텔리센스 기능을 제공한다.



그림 1-68

이제 다음과 같이 Details 뷰 템플릿의 소스 코드를 수정해보자.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
모임명: <%= Html.Encode (Model.Title) %>
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2><%= Html.Encode (Model.Title) %></h2>
<p>
<strong>시간:</strong>
<%= Model.EventDate.ToShortDateString () %>
<strong>@</strong>
<%= Model.EventDate.ToShortTimeString () %>
</p>
<p>
<strong>장소:</strong>
<%= Html.Encode (Model.Address) %>,
<%= Html.Encode (Model.Country) %>
</p>
<p>
<strong>설명:</strong>
<%= Html.Encode (Model.Description) %>
</p>
<p>
<strong>주선자:</strong>
<%= Html.Encode (Model.HostedBy) %>
(<%= Html.Encode (Model.ContactPhone) %>)
</p>
<%= Html.ActionLink ("Edit Dinner", "Edit", new { id=Model.DinnerID })%> |
<%= Html.ActionLink ("Delete Dinner", "Delete", new { id=Model.DinnerID })%>
</asp:Content>
```

그런 후 `/Dinners/Details/1` URL을 다시 요청해보면 브라우저가 다음 그림과 같은 페이지를 렌더링하는 것을 볼 수 있다.

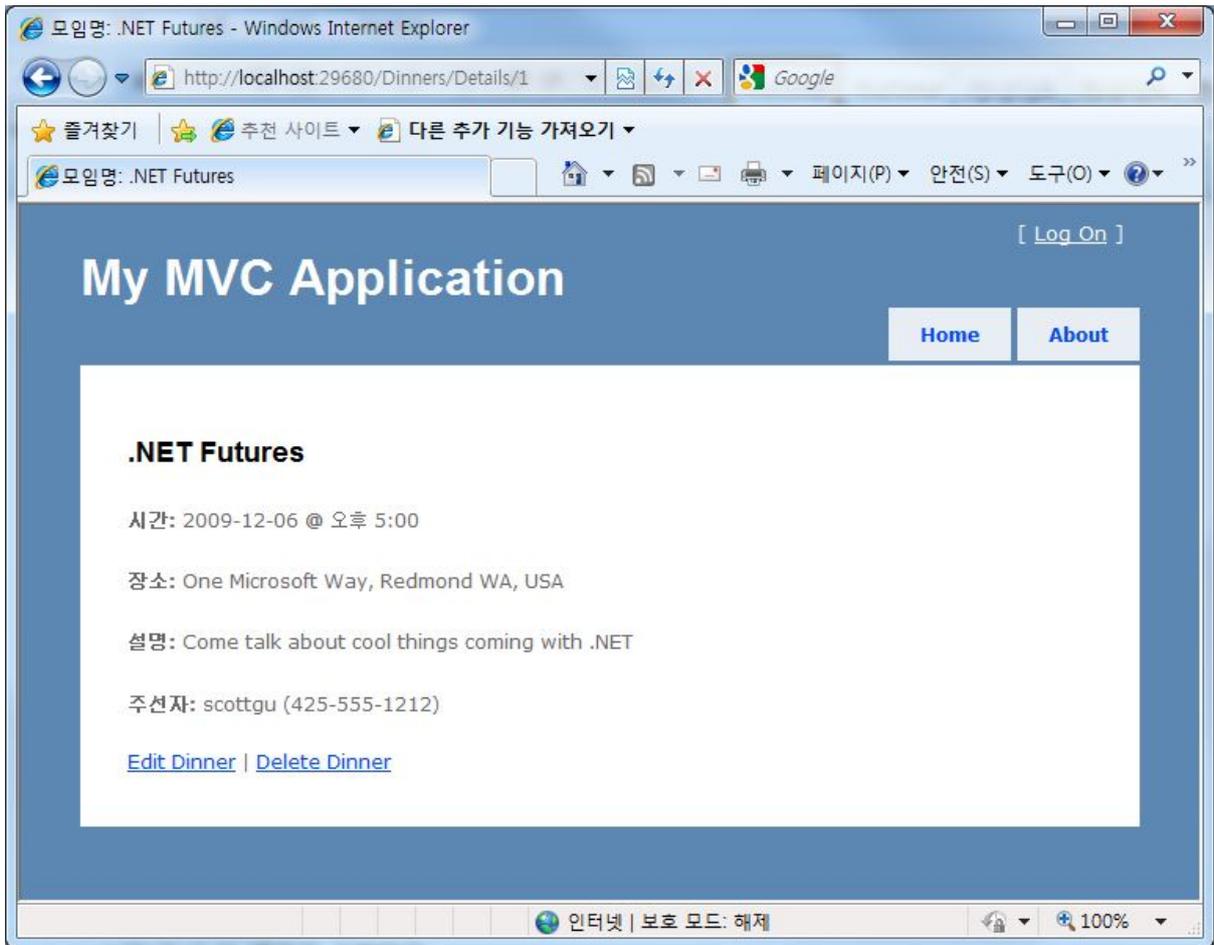


그림 1-69

“Index” 뷰 템플릿 구현하기

이제 예정된 저녁 모임 데이터의 목록을 보여주는 “Index” 뷰 템플릿을 구현해보자. 먼저 Index 액션 메서드에 포인터 커서를 옮긴 후 마우스 오른쪽 버튼을 클릭하여 “Add View” 메뉴를 선택한다 (또는 Ctrl + M, Ctrl + V 단축키를 사용해도 된다).

“Add View” 대화 상자에서 뷰 템플릿의 이름은 기본 값인 “Index”를 그대로 사용하고 “Create strongly-typed View” 옵션을 선택한다. 그리고 이번에는 자동으로 “List” 형식의 코드를 생성하도록 선택하고 모델 객체로는 “NerdDinner.Models.Dinner” 객체를 선택한다 (왜냐하면 “List” 페이지를 자동으로 생성하는 기능은 컨트롤러가 뷰 템플릿에 Dinners 객체의 컬렉션을 전달할 것이라고 가정하기 때문이다).

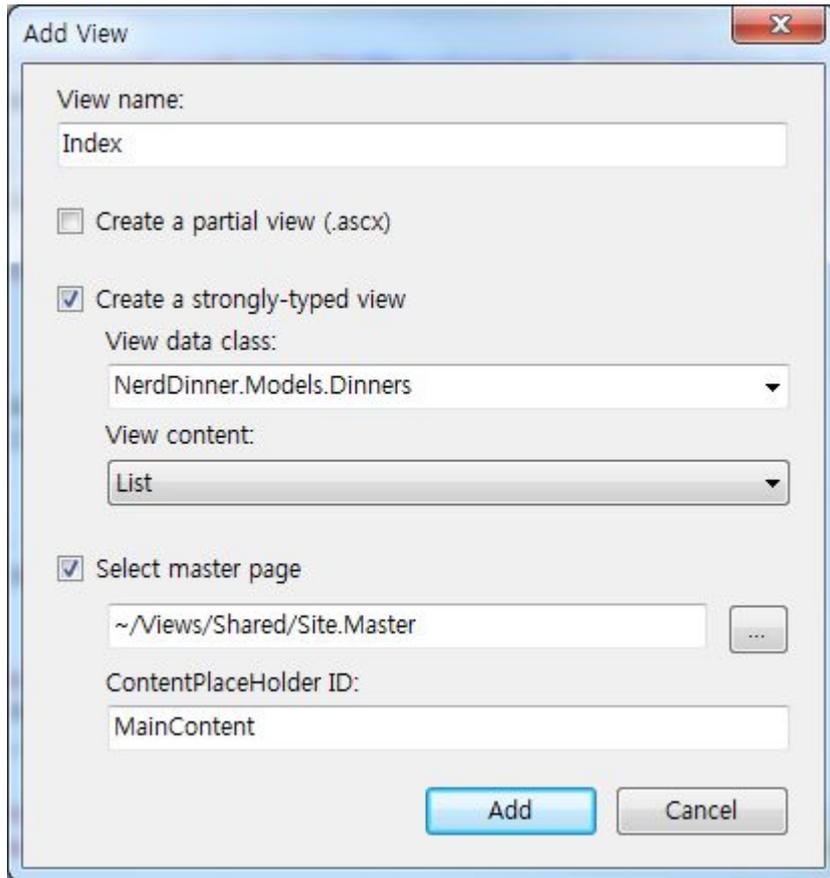


그림 1-70

"Add" 버튼을 클릭하면 Visual Studio는 "Views\Dinners" 디렉터리에 "Index.aspx" 뷰 템플릿을 생성한다. 또한 자동 코드 생성 기능에 의해 Dinners 객체들을 나열하기 위해 HTML의 <TABLE> 태그를 사용하는 기본적인 코드를 생성해 준다.

이제 애플리케이션을 실행하고 "/Dinners/" URL에 접근하면 다음과 같이 데이터가 나열된 페이지를 볼 수 있다.

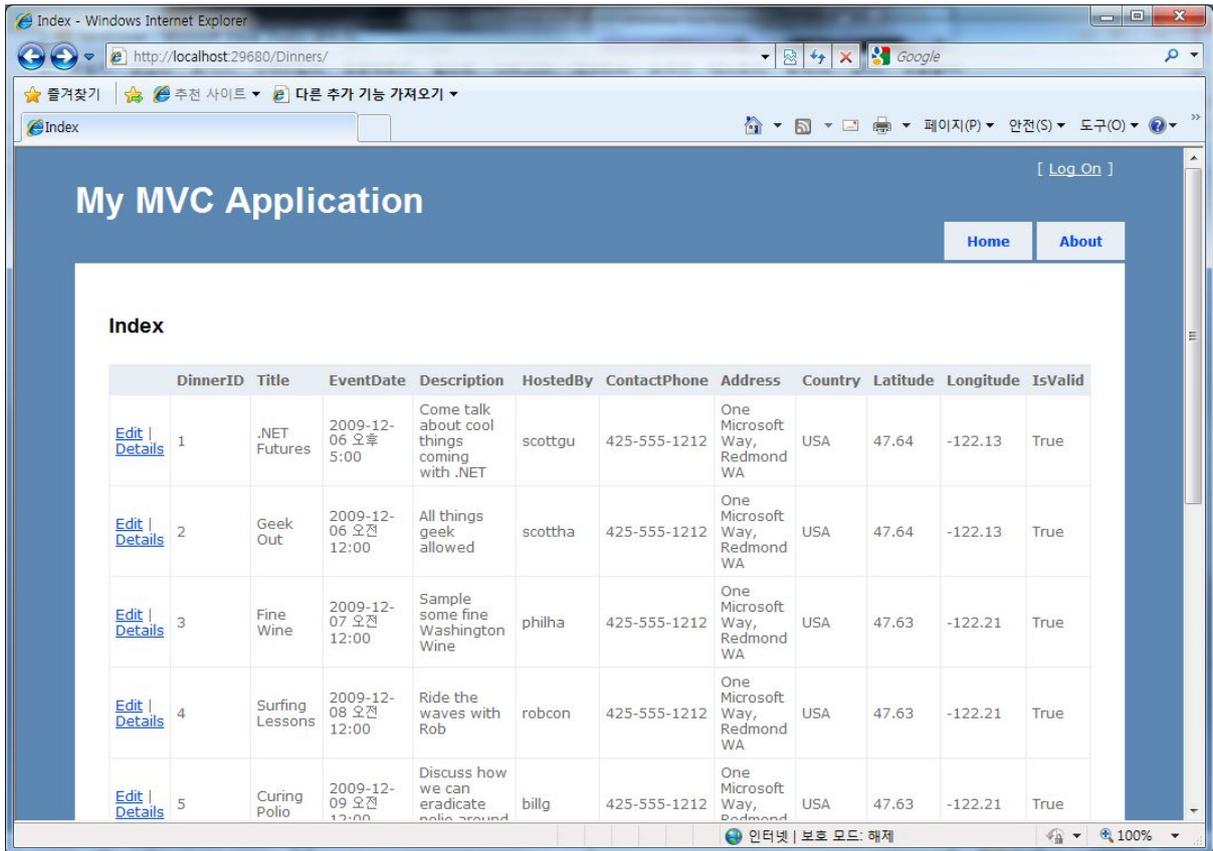


그림 1-71

위의 그림에서 보듯이 자동으로 생성된 표는 마치 그리드(Grid) 레이아웃과 유사한 형태로 Dinners 객체들을 나열하는데 사실 이는 우리가 사용자에게 보여주고자 하는 모습은 아니다. 따라서 다음과 같이 뷰 템플릿의 코드를 수정하여 몇 개의 컬럼을 제거하고 표 대신 태그를 이용하여 데이터를 나열하도록 수정하도록 하자.

```
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>예정된 모임들</h2>
<ul>
<% foreach (var dinner in Model) { %>
<li>
<%= Html.Encode(dinner.Title) %>
on
<%= Html.Encode(dinner.EventDate.ToShortDateString()) %>
@
<%= Html.Encode(dinner.EventDate.ToShortTimeString()) %>
</li>
<% } %>
</ul>
</asp:Content>
```

위의 코드에 사용된 foreach 구문에서는 "var" 키워드를 이용하여 모델 객체로부터 각각의 Dinners 객체를 얻어와 데이터를 출력하였다. C# 3.0에 익숙하지 않은 독자라면 "var" 키워드가 Dinners 객체에 대한 지연 바인딩(Late Binding)을 의미하는 것이라고 생각할지도 모르겠다. 그러

나 사실 이는 컴파일러가 강력하게 형식화된 ("IEnumerable<Dinner>" 타입을 사용하는) "Model" 속성에 대해 타입 추측(Type Inferencing)을 실행한다는 것을 의미하며 따라서 다음 그림과 같이 코드 블록 내에서 완벽한 인텔리센스의 지원 및 컴파일 시점에서의 타입 검사가 가능하다.

```
<ul>  
  <li>foreach (var dinner in Model) {  
    <li>Html.Encode(dinner.)  
  }  
</li>  
</ul>
```

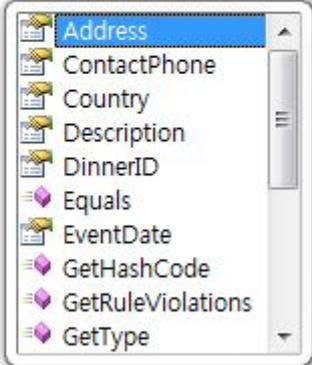


그림 1-72

/Dinners/ 페이지를 새로 고침해보면 이제 수정된 페이지가 다음과 같이 나타난다.



그림 1-73

수정된 페이지는 예전보다 조금 나아보이지만 아직 완벽하지는 않다. 마지막으로 해야 할 일은 사용자가 각각의 항목들을 클릭하여 상세 보기 페이지로 이동할 수 있도록 하는 것이다. 따라서 DinnersController 컨트롤러 클래스의 Details 액션 메서드를 호출하는 링크를 가진 HTML 하이퍼링크 요소를 렌더링해야 한다.

Index 뷰 템플릿에서 하이퍼링크를 생성하는 방법은 크게 두 가지로 나뉜다. 첫 번째 방법은 다음 그림과 같이 HTML <A> 태그를 직접 생성하는 방법으로 <A> 요소 내에서 <% %> 블록을 사용할 수도 있다.

```

<% foreach (var dinner in Model) { %>
  <li>
    <a href="/Dinners/Details/<%= dinner.DinnerID %>
      <%= Html.Encode(dinner.Title) %>
    </a>
    on
    <%= Html.Encode(dinner.EventDate.ToShortDateString()) %>
    @
    <%= Html.Encode(dinner.EventDate.ToShortTimeString()) %>
  </li>
<% } %>

```

그림 1-74

또 다른 방법은 아래와 같이 ASPNET MVC 프레임워크가 제공하는 "Html.ActionLink()" 헬퍼 메서드(Helper Method)를 이용하는 방법이다. 이 메서드를 이용하면 특정 컨트롤러의 액션 메서드를 호출하는 링크를 사용하는 <A> 요소를 프로그래밍적으로 생성할 수 있다.

```

<%= Html.ActionLink(dinner.Title, "Details", new { id=dinner.DinnerID }) %>

```

Html.ActionLink() 메서드의 첫 번째 매개 변수는 링크가 보여질 텍스트 (예제의 경우에는 Dinners 객체의 제목이다)이며 두 번째 매개 변수는 링크를 생성할 컨트롤러의 액션 메서드의 이름 (이 경우에는 Details 메서드이다)이고 마지막 세 번째 매개 변수는 액션 메서드에 전달될 변수들의 집합이다(속성의 이름과 값을 익명 타입(Anonymous Type)으로 구현하였다). 예제의 경우에는 우리가 연결하고자 하는 id 변수의 값을 지정하였으며 ASPNET MVC의 기본 URL 라우팅 규칙이 "{controller}/{action}/{id}"이기 때문에 Html.ActionLink() 메서드는 다음과 같은 URL을 생성하게 된다.

```

<a href="/Dinners/Details/1">.NET Futures</a>

```

Index.aspx 뷰에서는 다음과 같이 Html.ActionLink() 메서드를 이용하여 각각의 Dinners 객체에 연결될 링크들을 생성한다.

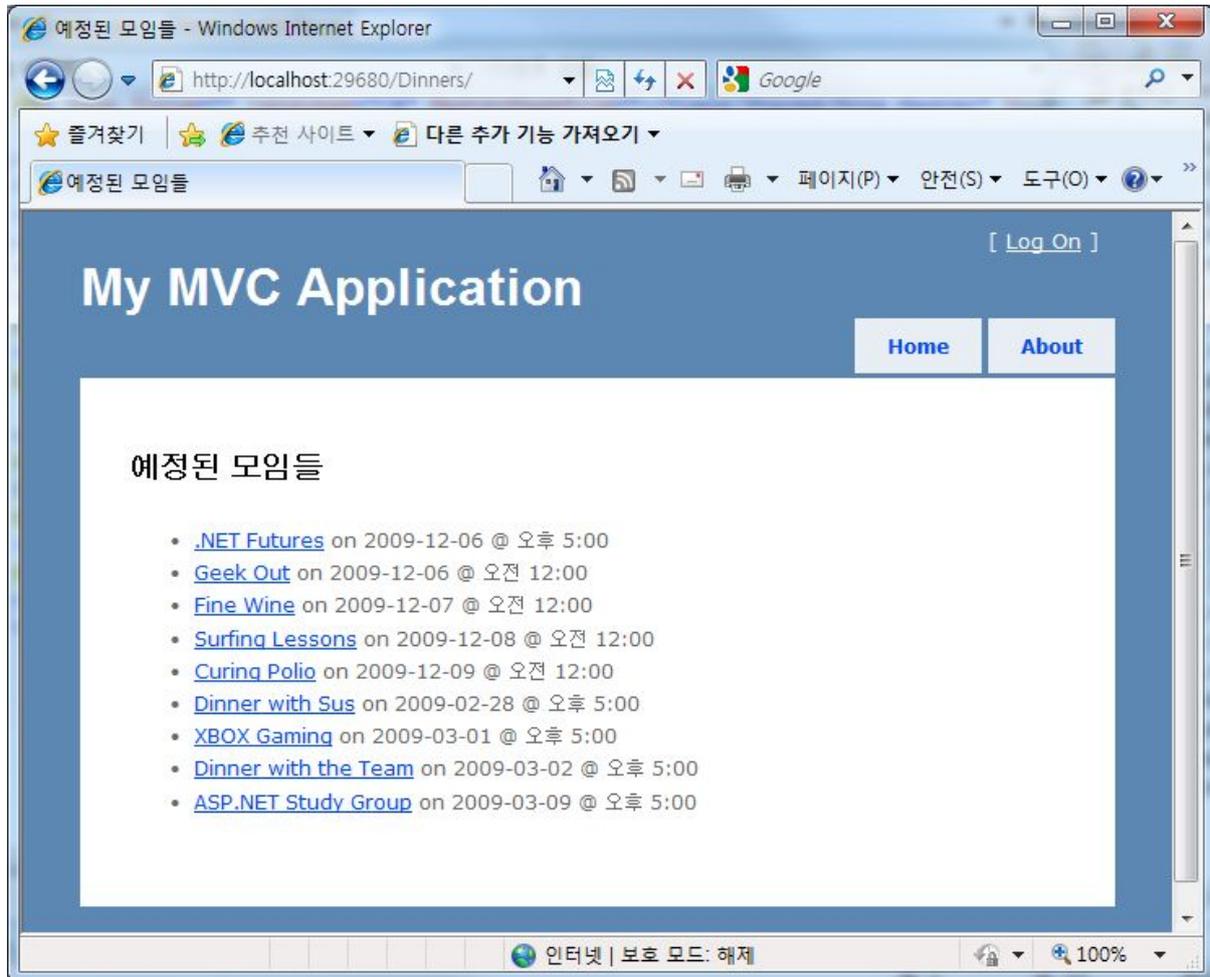
```

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
예정된 모임들
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>예정된 모임들</h2>
<ul>
<% foreach (var dinner in Model) { %>
<li>
<%= Html.ActionLink(dinner.Title, "Details",
new { id=dinner.DinnerID }) %>
on
<%= Html.Encode(dinner.EventDate.ToShortDateString()) %>
@
<%= Html.Encode(dinner.EventDate.ToShortTimeString()) %>

```

```
</li>
<% } %>
</ul>
</asp:Content>
```

이제 /Dinners URL을 다시 접근해 보면 아래와 같은 결과를 볼 수 있다.



이 페이지에서 아무 항목이나 클릭해보면 다음과 같이 상세 보기 페이지로 이동할 수 있다.

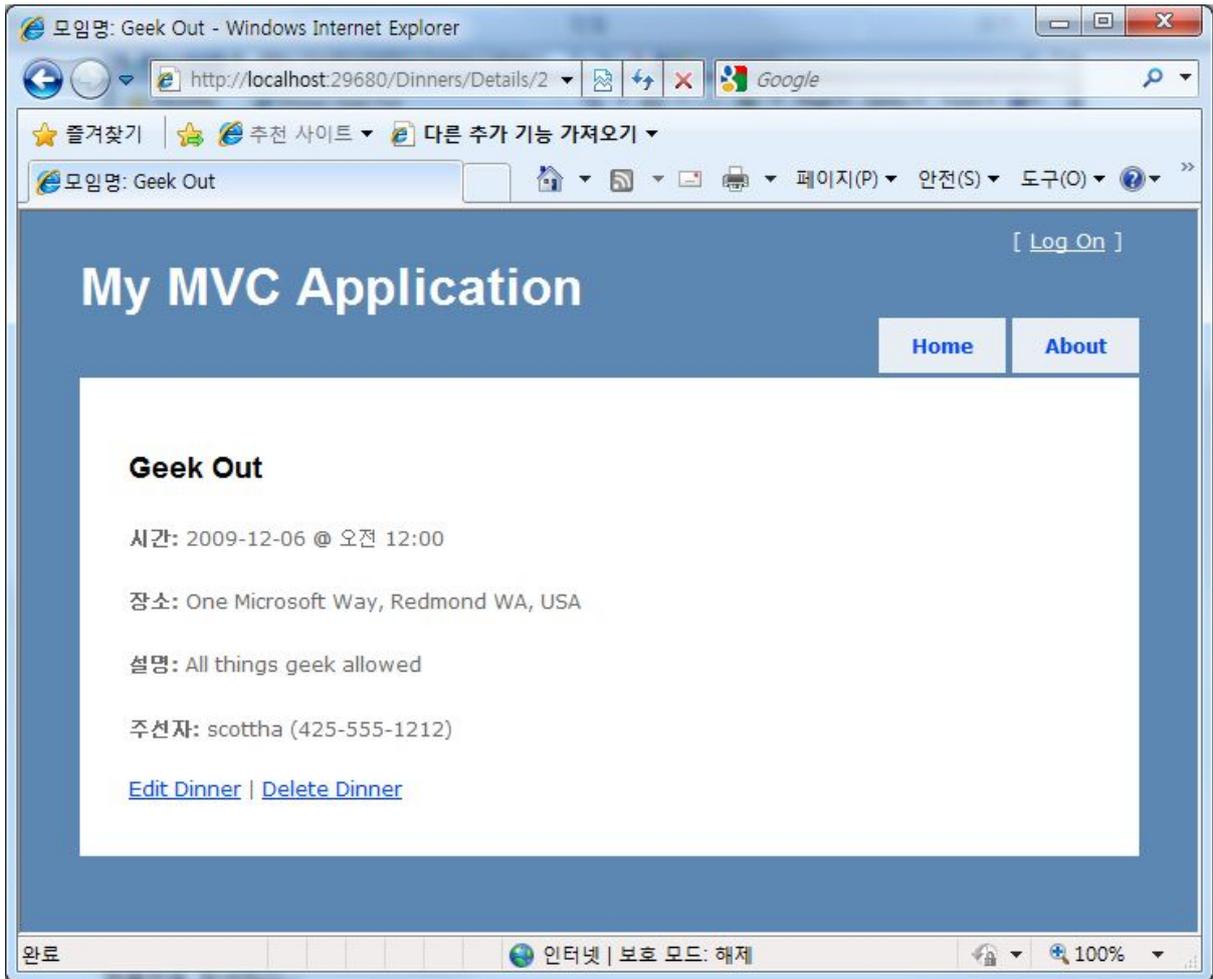


그림 1-76

Views 디렉터리의 구조와 뷰의 명명 규칙

기본적으로 ASP.NET MVC 애플리케이션은 뷰 템플릿의 이름을 해석할 때 특별한 명명 규칙을 사용한다. 따라서 개발자는 컨트롤러 클래스에서 뷰 템플릿을 호출할 때 뷰 템플릿의 전체 경로를 사용하지 않아도 된다. 기본적으로 ASP.NET MVC는 `Views` 디렉터리에서 뷰 템플릿을 탐색한다.

예를 들어 우리는 "Index", "Details" 그리고 "NotFound" 등 세 개의 뷰를 사용하는 `DinnersController` 클래스를 사용하고 있다. ASP.NET MVC는 이 세 개의 뷰를 탐색할 때 애플리케이션의 루트 아래에 있는 `Views/Dinners` 디렉터리에서 탐색한다.

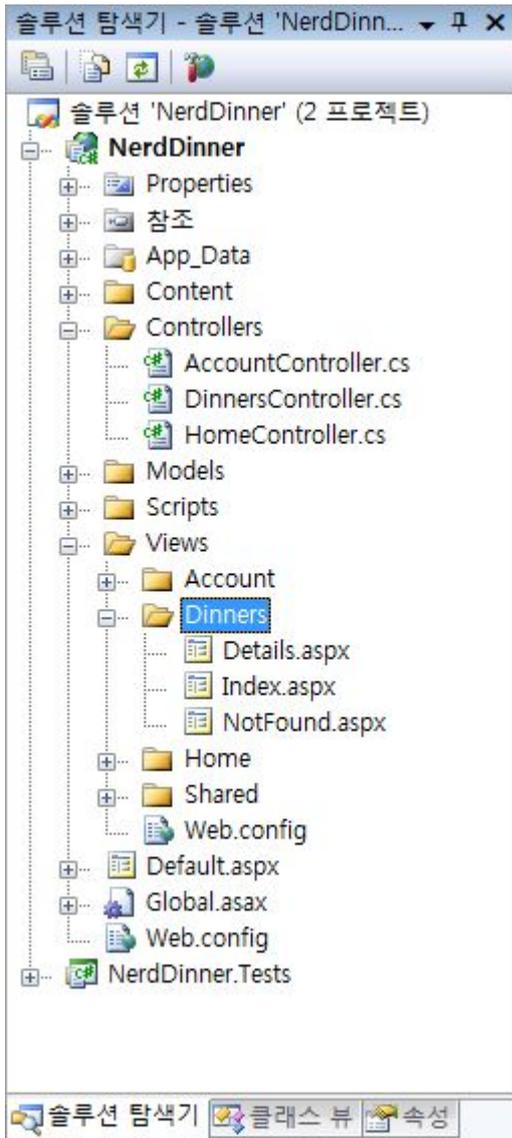


그림 1-77

위의 그림에서 보듯이 애플리케이션이 세 개의 컨트롤러 (DinnersController, HomeController, AccountController 등)이며 그 중 두 개는 프로젝트가 생성될 때 기본으로 생성된 것들이다) 클래스를 사용하고 있기 때문에 `Views` 디렉터리에도 각각의 컨트롤러 클래스를 위한 세 개의 하위 디렉터리가 생성되어 있음을 알 수 있다.

HomeController 컨트롤러 클래스와 AccountController 컨트롤러 클래스가 참조하는 뷰들은 각각 `Views\Home` 디렉터리와 `Views\Account` 디렉터리에서 뷰의 이름을 해석한다. `Views\Shared` 디렉터리는 애플리케이션 내의 여러 컨트롤러가 공유할 뷰 템플릿을 위해 사용된다. ASP.NET MVC가 뷰 템플릿의 이름을 해석할 때는 가장 먼저 `Views\{컨트롤러 이름}` 폴더를 탐색하고 만일 적절한 뷰 템플릿을 찾지 못하면 `Views\Shared` 디렉터를 탐색한다.

각각의 뷰 템플릿의 이름을 지정할 때 권장되는 가이드라인은 뷰 템플릿의 이름으로 자신을 렌더

링할 컨트롤러의 액션 메서드 이름과 동일한 이름을 사용하는 것이다. 예를 들어 "Index" 액션 메서드는 "Index" 뷰를 사용하여 결과를 렌더링하며 "Details" 액션 메서드는 "Details" 뷰를 사용하여 결과를 렌더링한다. 이렇게 함으로써 어떤 뷰 템플릿이 어떤 액션 메서드와 관련되어 있는지 쉽게 파악할 수 있다.

뷰 템플릿의 이름이 액션 메서드의 이름과 동일한 경우에는 개발자가 명시적으로 뷰 템플릿의 이름을 지정할 필요가 없다. 대신 View() 메서드를 호출할 때 (뷰 템플릿의 이름은 생략하고) 모델 객체만 전달할 수 있다. 그러면 ASP.NET MVC는 자동으로 `Views[컨트롤러 이름][액션 메서드 이름]` 뷰 템플릿을 탐색하여 렌더링한다.

덕분에 컨트롤러의 코드를 조금 더 깔끔하게 유지할 수 있으며 코드 내에서 이름이 중복 사용되는 것을 방지할 수 있다.

```
public class DinnersController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
    // GET: /Dinners/
    public ActionResult Index() {
        var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        return View(dinners);
    }
    //
    // GET: /Dinners/Details/2
    public ActionResult Details(int id) {
        Dinner dinner = dinnerRepository.GetDinner(id);
        if (dinner == null)
            return View("NotFound");
        else
            return View(dinner);
    }
}
```

지금까지의 코드 만으로도 Dinners 객체의 목록과 상세 보기 기능을 훌륭하게 구현할 수 있다.

데이터의 생성, 수정 및 삭제 양식을 위한 시나리오

지금까지 컨트롤러와 뷰에 대해 설명하고 데이터의 나열 및 상세 보기 기능을 구현하기 위해 이들을 어떻게 활용할 수 있는지를 보여주었다. 다음 단계는 DinnersController 클래스의 기능을 확장하여 Dinners 객체를 생성, 수정 혹은 삭제할 수 있는 기능을 추가하는 것이다.

DinnersController에 의해 처리되는 URL들

앞서 우리는 /Dinners와 /Dinners/Details/[id] 등 두 가지 URL을 지원하기 위한 액션 메서드를 DinnersController 클래스에 추가했었다.

| URL | HTTP 동사 | 설명 |
|-----------------------|---------|-------------------------|
| /Dinners | GET | 예정된 모임 데이터의 목록을 나열한다. |
| /Dinners/Details/[id] | GET | 특정 모임 데이터의 상세 정보를 표시한다. |

이번에는 /Dinners/Edit/[id]와 /Dinners/Create 그리고 /Dinners/Delete/[id] 등 세 개의 추가적인 URL을 제공하기 위한 액션 메서드를 추가해보자. 이 URL들은 이미 등록된 모임 데이터의 수정과 새로운 모임 데이터의 추가, 그리고 삭제 기능을 제공하기 위한 것이다.

새로 추가될 URL들은 HTTP GET 방식과 HTTP POST 방식을 모두 지원하게 될 것이다. 세 URL들을 HTTP GET 방식으로 요청하면 데이터를 위한 기본적인 HTML(데이터의 수정을 위해 Dinners 객체의 데이터가 채워진 "수정" 페이지와 "생성"을 위한 빈 페이지, 그리고 "삭제"를 위한 삭제 확인 페이지 등)을 표시하게 된다. 세 URL을 HTTP POST 방식으로 요청하면 DinnerRepository 클래스를 이용하여 각각 데이터를 수정/생성/삭제 하는 동작을 수행하게 된다.

| URL | HTTP 동사 | 설명 |
|----------------------|---------|---|
| /Dinners/Edit/[id] | GET | Dinners 객체의 데이터를 수정하는 양식 페이지를 보여준다. |
| | POST | 사용자가 변경한 데이터가 적용된 Dinners 객체를 데이터베이스에 저장한다. |
| /Dinners/Create | GET | 사용자가 새로운 데이터를 추가할 수 있는 빈 양식 페이지를 보여준다. |
| | POST | 새로운 Dinners 객체를 생성하고 이를 데이터베이스에 저장한다. |
| /Dinners/Delete/[id] | GET | 지정된 모임 데이터를 삭제할 것인지를 확인하는 페이지를 보여준다. |
| | POST | 지정된 모임 데이터를 데이터베이스에서 삭제한다. |

그러면 우선 수정 시나리오부터 구현을 시작해보자.

HTTP-GET 방식을 위한 Edit 액션 메서드 구현하기

우선은 HTTP "GET" 방식을 지원하는 Edit 액션 메서드를 구현해보자. 이 메서드는 /Dinners/Edit/[id] 형식의 URL이 요청되었을 때 호출된다. 이 메서드를 구현하는 코드는 다음과 같다.

```
//
// GET: /Dinners/Edit/2
public ActionResult Edit(int id) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    return View(dinner);
}
```

위의 코드는 DinnerRepository 클래스를 이용하여 Dinners 객체를 조회하고 뷰 템플릿을 이용하여 검색된 Dinners 객체를 렌더링한다. View() 메서드에 뷰 템플릿의 이름을 명시적으로 지정하지 않았기 때문에 이 메서드는 뷰 템플릿 명명 규칙에 따라 /Views/Dinners/Edit.aspx 경로의 뷰 템플릿 파일을 사용하게 된다.

이 뷰 템플릿 파일을 생성하려면 다음 그림과 같이 Edit 액션 메서드를 마우스 오른쪽 버튼으로 클릭하고 "Add View" 메뉴를 선택한다.

```
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id)
{
    Dinners dinner = dinnerRepository.GetDinner(id);
    |
    return View(dinner);
}
}
```



그림 1-78

다음 그림과 같이 "Add View" 대화 상자가 나타나면 뷰 템플릿 파일이 Dinners 객체를 전달받아 사용하도록 선택하고 코드를 자동으로 생성하기 위해 "Edit" 템플릿을 선택한다.

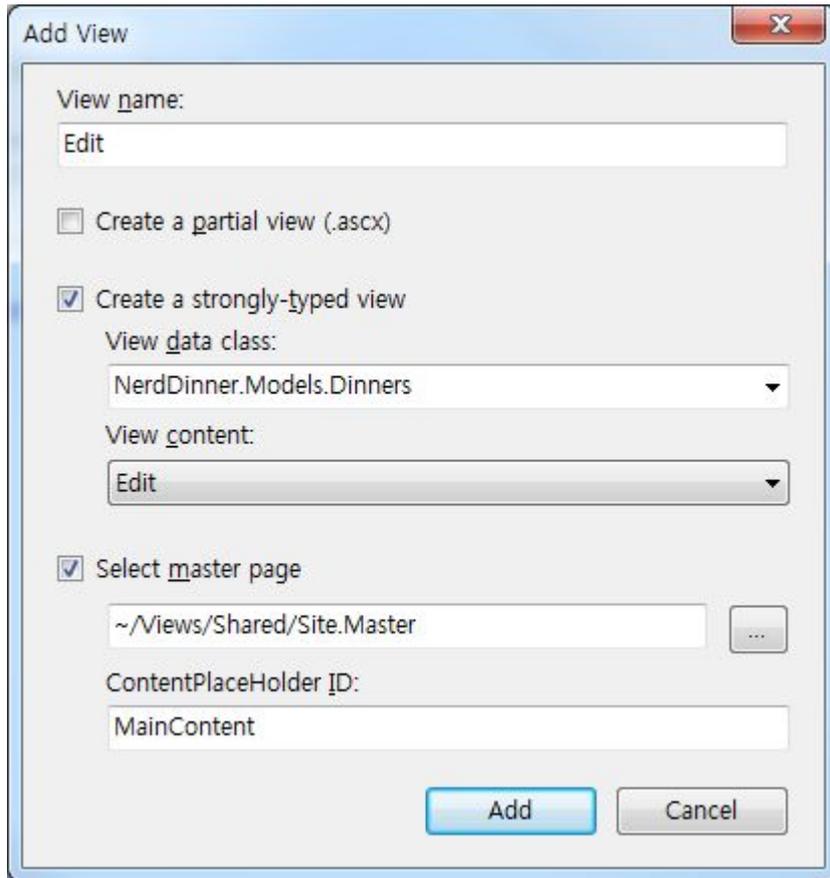


그림 1-79

"Add" 버튼을 클릭하면 Visual Studio는 "Edit.aspx" 뷰 템플릿을 파일을 "WViewsWDinnersW" 디렉터리에 추가하고 코드 편집기에 "Edit.aspx" 뷰 템플릿 파일을 로드한다. 이 파일에는 다음과 같은 코드가 미리 작성되어 있다.

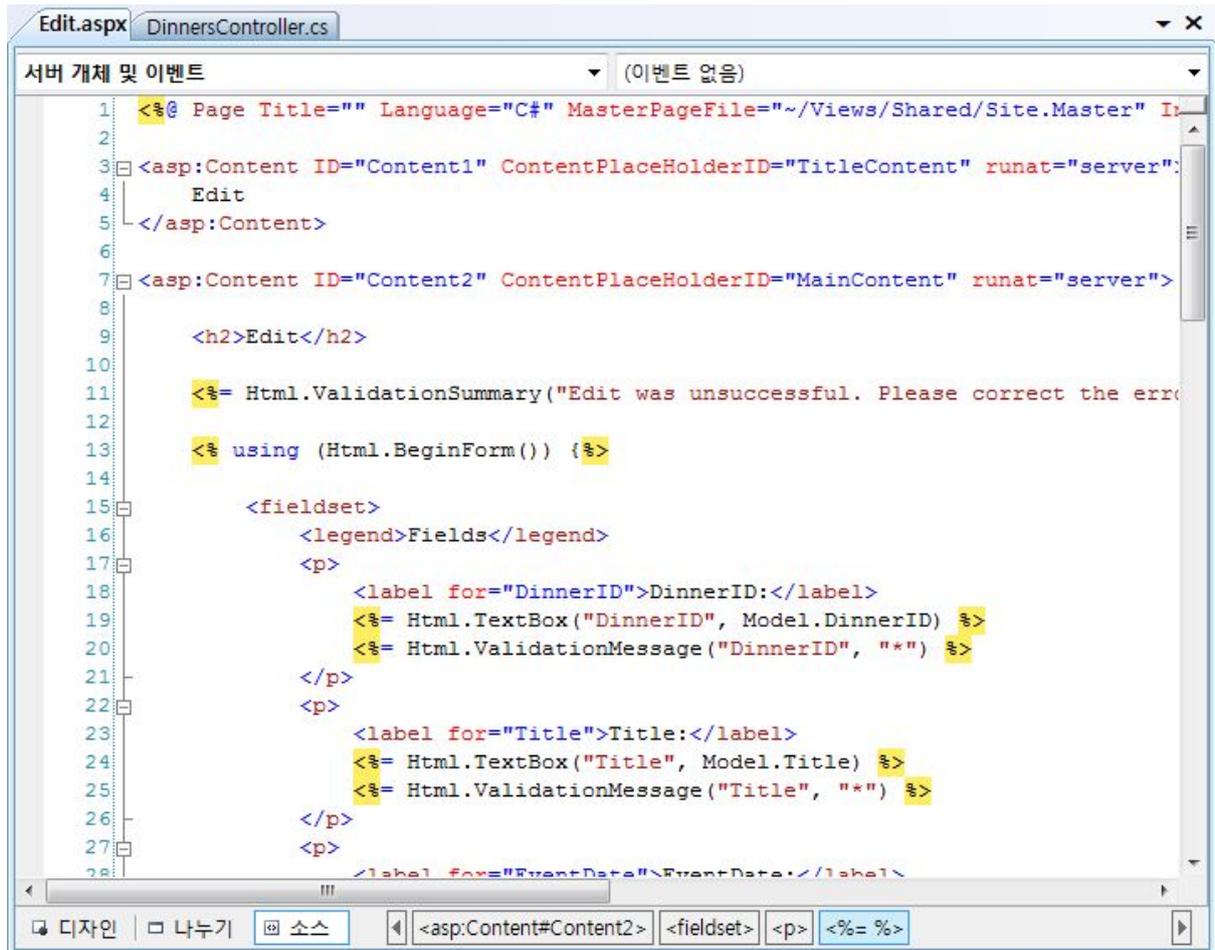


그림 1-80

그러면 자동 코드 생성 기능이 생성해 준 코드를 살짝 변경하여 이 페이지가 아래와 같은 콘텐츠를 갖도록 수정한다 (노출하고 싶지 않은 일부 속성을 제거하였다).

```

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
Edit: <%=Html.Encode(Model.Title) %>
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>Edit Dinner</h2>
<%= Html.ValidationSummary("Please correct the errors and try again.") %>
<% using (Html.BeginForm()) { %>
<fieldset>
<p>
<label for="Title">Dinner Title:</label>
<%= Html.TextBox("Title") %>
<%= Html.ValidationMessage("Title", "**") %>
</p>
<p>
<label for="EventDate">Event Date:</label>
<%= Html.TextBox("EventDate", String.Format("{0:g}",
Model.EventDate)) %>
<%= Html.ValidationMessage("EventDate", "**") %>
</p>
<p>
<label for="Description">Description:</label>

```

```

<%= Html.TextArea("Description") %>
<%= Html.ValidationMessage("Description", "*") %>
</p>
<p>
<label for="Address">Address:</label>
<%= Html.TextBox("Address") %>
<%= Html.ValidationMessage("Address", "*") %>
</p>
<p>
<label for="Country">Country:</label>
<%= Html.TextBox("Country") %>
<%= Html.ValidationMessage("Country", "*") %>
</p>
<p>
<label for="ContactPhone">Contact Phone #:</label>
<%= Html.TextBox("ContactPhone") %>
<%= Html.ValidationMessage("ContactPhone", "*") %>
</p>
<p>
<label for="Latitude">Latitude:</label>
<%= Html.TextBox("Latitude") %>
<%= Html.ValidationMessage("Latitude", "*") %>
</p>
<p>
<label for="Longitude">Longitude:</label>
<%= Html.TextBox("Longitude") %>
<%= Html.ValidationMessage("Longitude", "*") %>
</p>
<p>
<input type="submit" value="Save" />
</p>
</fieldset>
<% } %>
</asp:Content>

```

애플리케이션을 실행하고 `/Dinners/Edit/1` URL을 요청하면 다음 그림과 같은 페이지를 볼 수 있다.

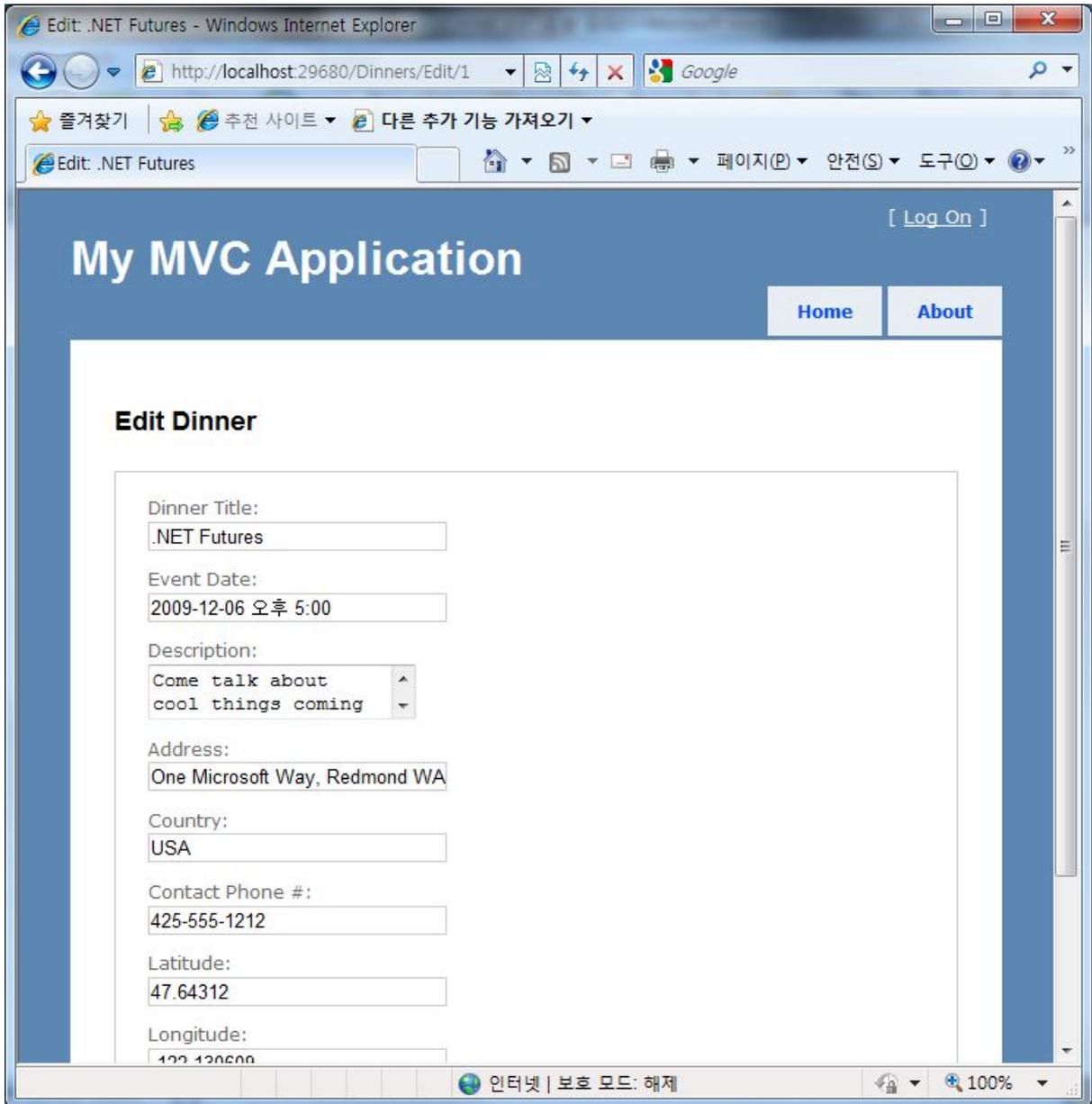


그림 1-81

이 뷰 템플릿에 의해 생성된 HTML 코드는 아래와 같다. 생성된 HTML 코드는 표준 HTML 코드이며 “저장” 버튼인 `<input type="submit" />` 버튼을 클릭하면 `/Dinners/Edit/1` URL에 HTTP POST 요청을 보내는 `<FORM>` 태그가 생성되어 있다. `<input type="text">` 요소들은 각각 수정할 수 있는 속성들을 표현하기 위해 사용된다.

```

<form action="/Dinners/Edit/1" method="post">
  <fieldset>
    <p>
      <label for="Title">Dinner Title:</label>
      <input id="Text1" name="Title" type="text" value=".NET Futures" />
    </p>
    <p>
      <label for="EventDate">Event Date:</label>
      <input id="EventDate" name="EventDate" type="text" value="2009-12-06 오후 5:00" />
    </p>
    <!-- 일부 코드 생략 -->
    <p>
      <input type="submit" value="Save" />
    </p>
  </fieldset>
</form>

```

그림 1-82

Html.BeginForm() 메서드와 Html.EndForm() 메서드

“Edit.aspx” 뷰 템플릿 파일은 Html.ValidationSummary(), Html.BeginForm(), Html.EndForm(), Html.TextBox(), Html.ValidationMessage() 등의 HTML 관련 메서드들을 사용한다. 이와 같은 메서드들은 HTML 요소를 렌더링할 뿐만 아니라 유효성 검사와 관련된 기능도 지원한다.

Html.BeginForm() 메서드

Html.BeginForm() 메서드는 HTML <FORM> 태그를 렌더링하는 메서드이다. Edit.aspx 뷰 템플릿 파일을 살펴보면 이 메서드를 사용할 때 C#의 “using” 구문을 사용하고 있음을 알 수 있을 것이다. 여는 중괄호는 <FORM> 콘텐츠의 시작을 의미하며 닫는 중괄호는 </FORM> 요소의 끝을 의미한다.

```

<% using (Html.BeginForm()) { %>
<fieldset>
<!-- 코드의 단순화를 위해 필드를 생략하였음 -->
<p>
<input type="submit" value="Save" />
</p>
</fieldset>
<% } %>

```

그러나 “using” 구문을 사용하는 방법이 적합하지 못한 경우에는 다음과 같이 Html.BeginForm() 메서드와 Html.EndForm() 메서드를 조합하여 사용할 수 있다 (물론 결과는 동일하다).

```

<% Html.BeginForm(); %>
<fieldset>
<!-- 코드의 단순화를 위해 필드를 생략하였음 -->
<p>
<input type="submit" value="Save" />

```

```
</p>
</fieldset>
<% Html.EndForm(); %>
```

매개 변수를 사용하지 않는 `Html.BeginForm()` 메서드를 호출하면 현재의 URL에 HTTP POST 요청을 보내는 `<FORM>` 태그가 렌더링되기 때문에 Edit 뷰 템플릿에는 `<form action="/Dinners/Edit/1" method="post">`와 같은 태그가 렌더링되었다. 만일 다른 URL로 HTTP POST 요청을 보내고 싶다면 `Html.BeginForm()` 메서드에 해당 URL을 명시적으로 지정하면 된다.

Html.TextBox() 메서드

Edit 뷰 템플릿은 `<input type="text" />` 요소를 렌더링하기 위해 다음과 같이 `Html.TextBox()` 메서드를 사용하였다.

```
<%= Html.TextBox("Title") %>
```

위에서 사용된 `Html.TextBox()` 메서드는 하나의 매개 변수가 사용되었는데 이 매개 변수는 `<input type="text" />` 요소의 `id`와 `name` 특성에 사용되는 동시에 `value` 특성에 사용될 모델 객체의 속성 이름으로도 사용된다. 예를 들어 Edit 뷰 템플릿에 "Title" 속성 값이 ".NET Futures"인 `Dinners` 객체를 전달하고 뷰 템플릿에서 `Html.TextBox("Title")` 메서드를 호출하면 `<input type="text" id="Title" name="Title" value="차세대 .NET" />`과 같은 결과가 렌더링된다.

또한 `Html.TextBox()` 메서드의 첫 번째 매개 변수에 전달된 값은 `id`와 `name` 특성에만 사용하도록 하고 `value` 특성에 사용할 값을 다음과 같이 명시적으로 전달할 수도 있다.

```
<%= Html.TextBox("Title", Model.Title) %>
```

간혹 요소의 값을 다른 형식으로 출력하고자 하는 경우도 있을 수 있다. 이런 경우에는 .NET이 제공하는 `String.Format()` 메서드를 유용하게 활용할 수 있다. Edit 뷰 템플릿에서는 다음과 같이 `String.Format()` 메서드를 이용하여 (DateTime 타입을 사용하는) `Dinners` 객체의 `EventDate` 속성 값을 출력할 때 초 단위 값을 출력하지 않도록 하고 있다.

```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

`Html.TextBox()` 메서드의 세 번째 매개 변수는 요소에 HTML 특성을 지정하기 위해 선택적으로 사용되는 매개 변수이다. 다음의 코드는 추가로 `size="30"`과 `class="mycssclass"` 특성을 렌더링하는 방법을 보여준다. 특히 "class"가 C#의 키워드이기 때문에 "@" 문자를 이용하여 class 특성과의 이름 충돌 문제를 해결하고 있음을 알 수 있다.

```
<%= Html.TextBox("Title", Model.Title, new { size=30, @class="myclass" } ) %>
```

HTTP-POST 방식을 위한 Edit 액션 메서드 구현하기

이제 HTTP GET 방식을 위한 Edit 액션 메서드를 구현했다. 브라우저를 통해 /Dinners/Edit/1 URL을 요청하면 다음과 같은 페이지가 나타난다.

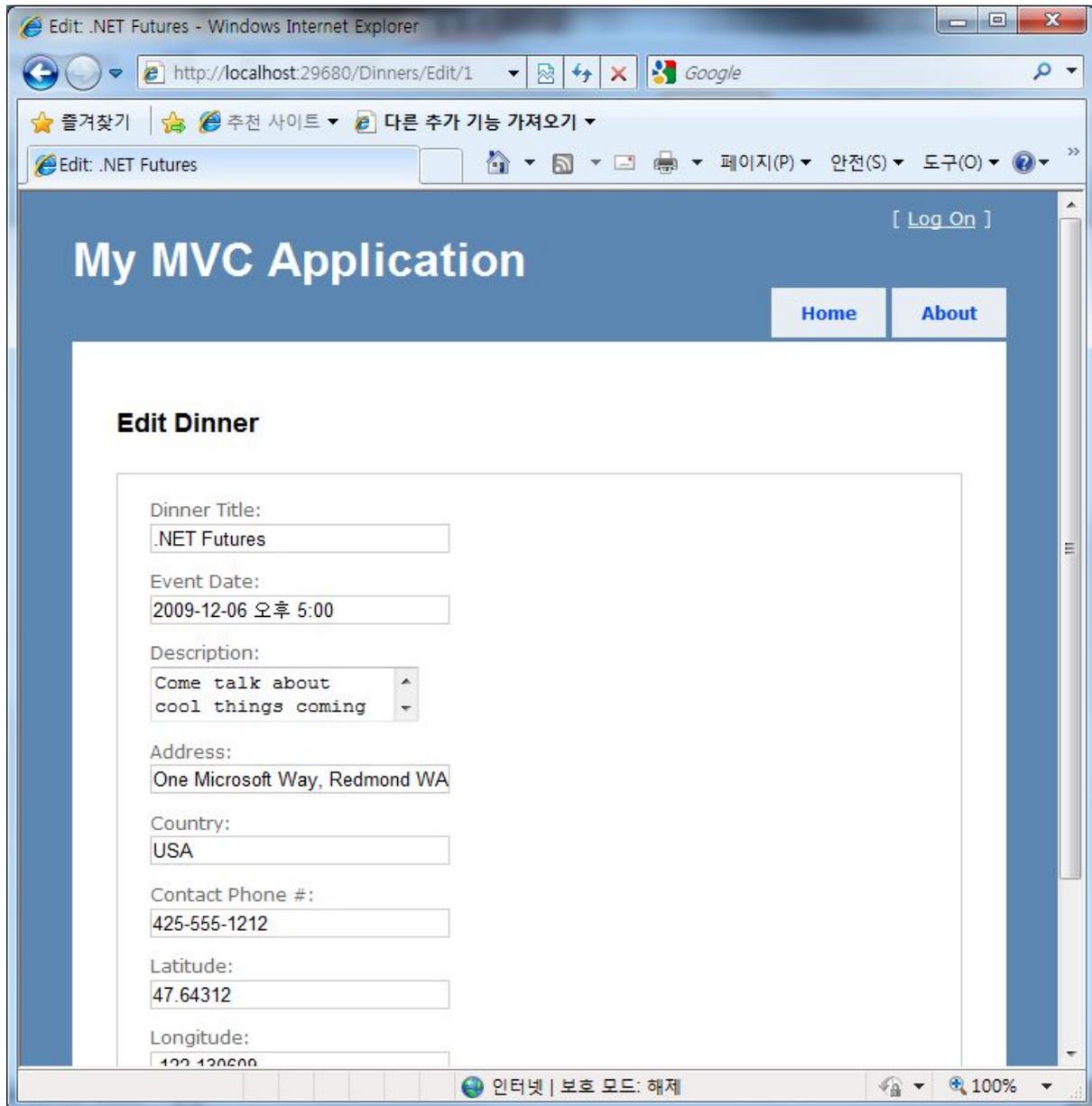


그림 1-83

“저장” 버튼을 클릭하면 사용자가 <INPUT> 요소들에 입력한 데이터는 /Dinners/Edit/1 URL에 POST 방식으로 전달된다. 그러면 이제 Dinners 객체를 데이터베이스에 저장하는 HTTP POST 동작을 수행하는 액션 메서드를 구현해보자.

코드 편집기에서 DinnersController 클래스의 "Edit" 액션 메서드를 재정의한 후 HTTP POST 방식을 지원하기 위해 재정의한 메서드에 다음과 같이 "AcceptVerbs" 특성을 추가한다.

```
//
// POST: /Dinners/Edit/2
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    ...
}
```

재정의된 액션 메서드에 [AcceptVerbs] 특성을 적용하면 ASPNET MVC는 현재 요청에 사용된 HTTP 동사에 따라 자동적으로 적당한 메서드를 선택하여 호출한다. 따라서 /Dinners/Edit/[id] URL을 HTTP POST 방식으로 요청하면 방금 추가한 Edit 액션 메서드가 실행되며 그 외의 다른 HTTP 동사를 이용한 요청은 앞서 구현했던 ([AcceptVerbs] 특성이 지정되지 않은) Edit 액션 메서드를 호출하게 된다.

=====

왜 하필이면 HTTP 동사로 구분하는걸까?

어쩌면 여러분은 "왜 동일한 URL의 동작을 HTTP 동사로 구분해야 하는걸까? 그냥 데이터를 로드하거나 저장하기 위한 URL을 별도로 구현해도 되잖아?"라고 생각할 수도 있다. 예를 들어 /Dinners/Edit/[id] URL은 데이터를 수정하기 위한 양식 페이지를 보여주고 /Dinners/Save/[id] URL을 이용해서 데이터를 저장해도 될텐데 말이다.

그러나 두 개의 URL을 사용하는 것은 우리가 /Dinners/Save/2 URL로 HTTP POST 요청을 보낼 때 사용자의 입력에 오류가 있어서 HTML 양식을 다시 렌더링해야 하는 경우나 사용자가 /Dinners/Save/2 URL을 브라우저의 주소 표시줄에 직접 입력하여 액세스를 시도하는 경우에 오히려 단점으로 작용할 수 있다. 만일 사용자가 이러한 URL을 브라우저의 즐겨찾기에 추가하거나 혹은 복사하여 친구에게 메일로 전달한다면 해당 URL은 (HTTP POST 방식으로 전달된 데이터를 필요로 하기 때문에) 올바르게 동작하지 않을 것이다.

/Dinners/Edit/[id]와 같은 하나의 URL을 HTTP 동사로 구분하여 처리하면 사용자가 URL을 즐겨찾기에 추가하거나 친구에게 URL을 전달하더라도 안전하게 페이지를 제공할 수 있게 된다.

=====

전송된 값의 조회

"Edit" 액션 메서드에서 HTTP POST 방식으로 전달된 매개 변수들은 여러 가지 방법으로 조회할 수 있다. 가장 간단한 방법은 다음과 같이 Controller 클래스가 제공하는 Request 속성을 이용하여 폼 데이터 컬렉션에 액세스하여 필요한 값을 얻어오는 방법이다.

```
//
```

```

// POST: /Dinners/Edit/2
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    // Dinners 객체를 조회한다.
    Dinners dinner = dinnerRepository.GetDinner(id);
    // 전송된 값들을 이용하여 Dinners 객체를 업데이트한다.
    dinner.Title = Request.Form["Title"];
    dinner.Description = Request.Form["Description"];
    dinner.EventDate = DateTime.Parse(Request.Form["EventDate"]);
    dinner.Address = Request.Form["Address"];
    dinner.Country = Request.Form["Country"];
    dinner.ContactPhone = Request.Form["ContactPhone"];
    // 변경 사항을 데이터베이스에 저장한다.
    dinnerRepository.Save();
    // 저장된 Dinners 객체의 상세 보기 페이지로 이동한다.
    return RedirectToAction("Details", new { id = dinner.DinnerID });
}

```

이 방법은 간단하기는 하지만 작성해야 하는 코드의 양이 비교적 많은 편이며 특히 에러 처리 로직을 추가해야 하는 경우에는 더욱 그렇다.

조금 더 나은 방법은 Controller 클래스가 제공하는 UpdateModel() 메서드를 사용하는 것이다. 이 메서드는 우리가 전달한 객체의 속성들을 HTTP POST 방식으로 전달된 값들로 업데이트한다. 이때 .NET Reflection API들을 이용하여 값을 업데이트할 객체의 속성들을 결정하며 클라이언트로부터 전달된 값들을 자동으로 변환하여 속성에 대입해 준다.

다음의 코드는 UpdateModel() 메서드를 이용하여 Edit 액션 메서드를 수정한 코드이다.

```

//
// POST: /Dinners/Edit/2
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    Dinners dinner = dinnerRepository.GetDinner(id);
    UpdateModel(dinner);
    dinnerRepository.Save();
    return RedirectToAction("Details", new { id = dinner.DinnerID });
}

```

이제 /Dinners/Edit/1 URL에 접근하여 다음과 같이 제목을 변경해보자.

Edit Dinner

Dinner Title:

Event Date:

그림 1-84

“저장” 버튼을 클릭하면 Edit 액션 메서드에 HTTP POST 요청을 보내게 되며 수정된 값들이 데이터베이스에 저장된다. 그런 후 (새로 변경된 값들을 보여주는) 상세 보기 페이지로 이동하게 된다.

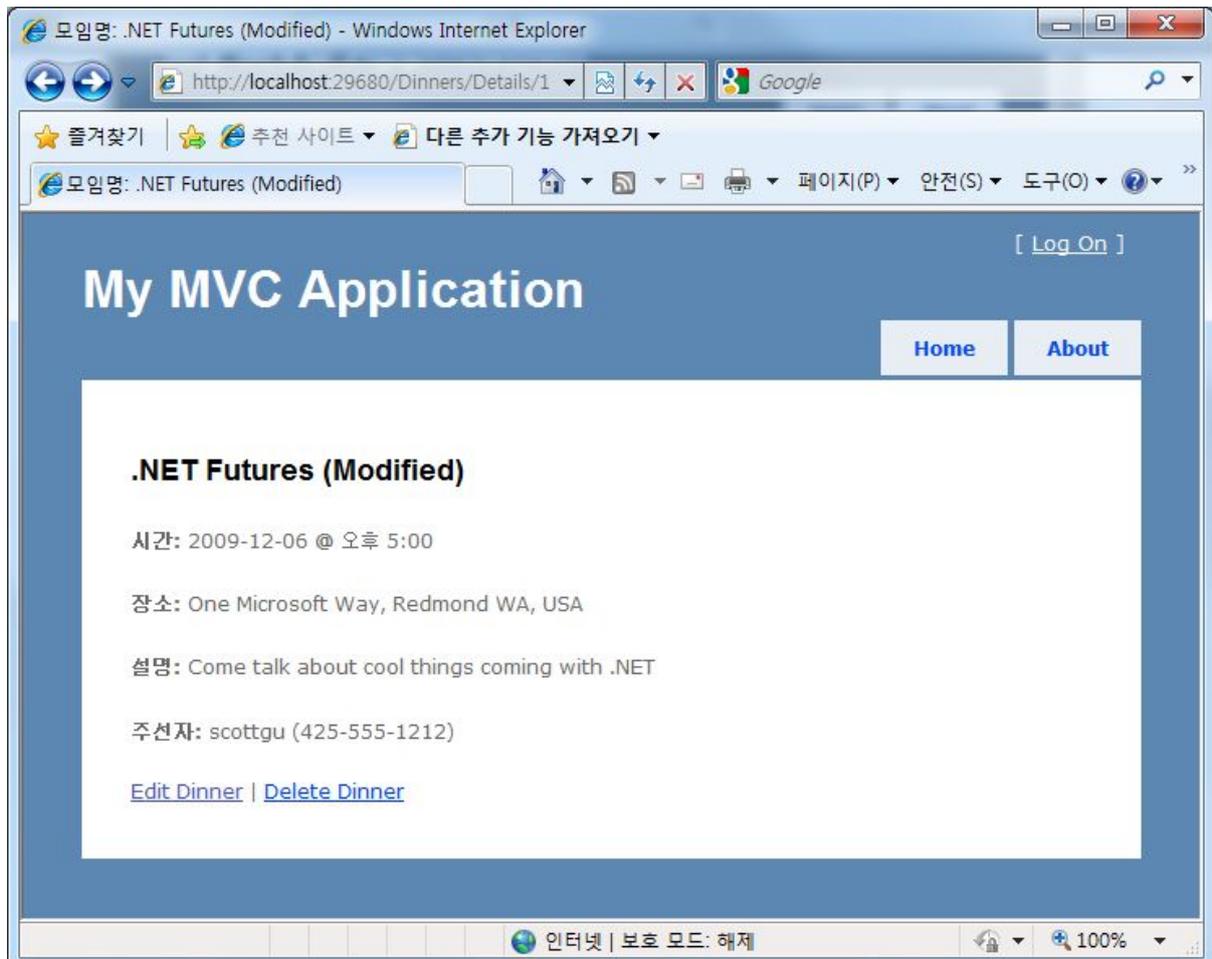


그림 1-85

오류의 처리

조금 전에 수정한 HTTP POST 방식을 위한 Edit 액션 메서드는 제대로 동작하기는 하지만 이는 어떠한 오류도 발생하지 않은 경우에 한해서이다.

만일 사용자가 값을 수정하는 도중 실수로 잘못된 값을 입력했다면 사용자가 이를 정정하는데 도움이 될만한 에러 메시지를 표시하도록 페이지를 다시 렌더링해야 한다. 물론 사용자가 올바르게 값을 입력한 경우 (예를 들면 잘못된 날짜 형식 문자열을 입력한 경우)나 올바르게 비즈니스 규칙에 어긋나는 값을 입력한 경우에 대해서도 마찬가지이다. 오류가 발생한 경우에는 사용자가 이전에 입력한 값들을 유지해서 사용자가 같은 데이터를 다시 입력하지 않도록 해 주어야 한다. 이와 같은 작업은 우리가 수행하고자 하는 작업이 성공적으로 완료되기 까지 몇 번이고 반복 될 수 있다.

ASPNET MVC는 오류를 처리하고 폼 데이터를 유지하는데 필요한 아주 훌륭한 기능들을 내장하고 있다. 이 기능들이 어떻게 동작하는지 확인하기 위해 다음과 같이 Edit 액션 메서드를 수정해 보자.

```
//
// POST: /Dinners/Edit/2
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    Dinners dinner = dinnerRepository.GetDinner(id);
    try {
        UpdateModel(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        foreach (var issue in dinner.GetRuleViolations()) {
            ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
        }
        return View(dinner);
    }
}
```

위의 코드는 앞서 작성한 코드와 유사하지만 전체 코드를 try ~ catch 블록으로 둘러싸고 있다. 만일 UpdateModel() 메서드를 호출했을 때 혹은 (Dinners 객체가 비즈니스 규칙에 어긋나면 예외가 발생하는) DinnerRepository 클래스의 Save 메서드를 호출할 때 예외가 발생하면 catch 블록이 실행된다. catch 블록에서는 Dinners 객체가 어떤 비즈니스 규칙을 만족하지 못했는지 조회하여 이들을 ModelState 객체 (잠시 후에 설명하겠다.)에 추가하고 뷰를 다시 렌더링한다.

그러면 위의 코드가 어떻게 동작하는지 살펴보기 위해 애플리케이션을 다시 실행하고 Dinners 객체의 데이터를 수정하는 페이지로 이동하여 제목에는 빈 문자열을 입력하고 날짜에는 "BUGUS"를 입력한 후 국가를 "대한민국"으로 선택하고 전화 번호 항목에 미국식 전화 번호를 입력한다. 이제 "저장" 버튼을 클릭하면 HTTP POST 방식을 지원하는 Edit 액션 메서드가 실행되지만 (입력 값에 오류가 있기 때문에) Dinners 객체가 올바르게 저장되지 못하고 페이지가 다시 렌더링 될 것이다.

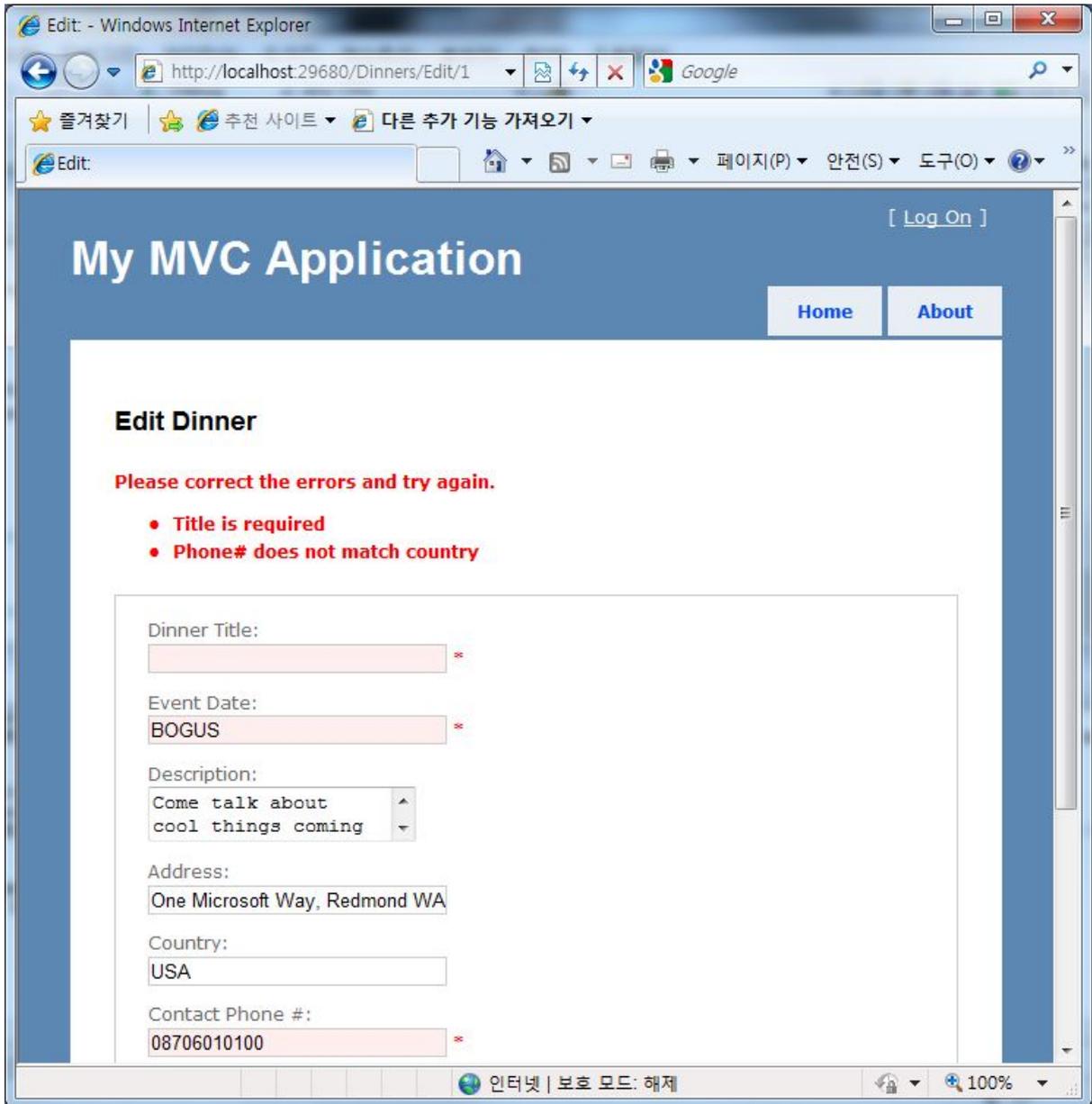


그림 1-86

그림에서 보듯이 애플리케이션은 제법 쓸만한 오류 처리 기능을 제공한다. 잘못된 값이 입력된 텍스트 요소들은 붉은 색으로 표시되며 유효성 검사 오류 메시지가 사용자에게 표시된다. 또한 전체 페이지에는 앞서 사용자가 입력했던 값들이 그대로 유지되어 있어 사용자들에게 편의를 제공하고 있다.

자, 그렇다면 이와 같은 일들이 어떻게 가능한 것일까? 제목과 날짜, 전화 번호 입력 필드들은 어떻게 스스로를 붉은 색으로 표시하며 어떻게 이전에 사용자가 입력했던 값들을 다시 보여줄 수 있을까? 그리고 어떻게 에러 메시지들이 페이지 상단에 나타나는 것일까? 이러한 일들이 가능한 이유는 사용자 입력에 대한 유효성 검사 및 오류 처리를 손쉽게 구현할 수 있도록 ASP.NET MVC가 구현하고 있는 기능들을 활용하여 페이지를 구현하였기 때문이다.

유효성 검사에 관련된 메서드들과 ModelState 속성

Controller 클래스는 뷰가 처리하고자 하는 모델 객체에 오류가 있음을 표시하기 위해 "ModelState"라는 컬렉션 속성을 제공한다. ModelState 속성에 저장된 오류 항목들은 (앞서 예제에서 "Title", "EventDate" 혹은 "ContactPhone" 속성과 같이) 문제가 된 모델 객체의 속성의 이름이며 ("제목을 입력하세요"와 같이) 사용자가 이해하기 쉬운 에러 메시지를 지정할 수 있다

UpdateModel() 메서드는 폼 데이터로 전송된 값들을 모델 객체의 속성에 대입할 때 오류가 발생하면 ModelState 컬렉션을 자동으로 조작한다. 예를 들어 Dinners 객체의 EventDate 속성은 DateTime 타입을 사용하기 때문에 UpdateModel() 메서드는 사용자가 입력했던 "BUGUS"라는 값을 대입할 수 없게 되고 이 경우 ModelState 컬렉션에 EventDate 속성에 값을 대입할 수 없음을 알리는 오류 항목을 추가한다.

또한 앞서 코드의 catch 블록에서처럼 개발자들이 명시적으로 오류 항목을 ModelState 컬렉션에 추가할 수도 있다. 앞서의 코드를 다시 살펴보면 catch 블록에서는 Dinners 객체에서 발생한 비즈니스 규칙 오류를 ModelState 컬렉션에 추가하였다.

```
[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    Dinners dinner = dinnerRepository.GetDinner(id);
    try {
        UpdateModel(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        foreach (var issue in dinner.GetRuleViolations()) {
            ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
        }
        return View(dinner);
    }
}
```

ModelState 속성과 HTML 헬퍼 메서드들

Html.TextBox()와 같은 HTML 헬퍼 메서드들은 결과를 렌더링하는 과정에서 ModelState 컬렉션을 검사한다. 만일 렌더링할 요소와 관련된 오류가 존재한다면 사용자가 입력했던 값을 출력하고 오류가 발생한 항목에 적용될 CSS 스타일을 함께 지정한다.

예를 들어 "Edit" 뷰에서 우리가 EventDate 속성을 위해 사용한 Html.TextBox() 메서드는 다음과 같다.

```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

오류가 발생한 상황에서 뷰가 렌더링 될 때 `Html.TextBox()` 메서드는 `ModelState` 컬렉션에 `Dinners` 객체의 "EventDate" 속성과 관련된 오류가 존재하는지 검사한다. 만일 사용자가 입력한 값(예제의 경우 "BUGUS")에 문제가 있다고 판단되면 다음과 같이 사용자가 입력한 값을 표시하며 CSS 스타일이 추가된 `<input type="text" />` 태그를 렌더링한다.

```
<input class="input-validation-error" id="EventDate" name="EventDate" type="text" value="BUGUS" />
```

오류가 발생한 항목에 적용될 CSS 스타일은 우리가 마음대로 재정의할 수 있다. 기본적으로 사용되는 CSS 오류 클래스인 "input-validation-error" 스타일은 `WContentWsite.css` 파일에 다음과 같이 정의되어 있다.

```
.input-validation-error
{
border: 1px solid #ff0000;
background-color: #ffebee;
}
```

이 스타일이 적용된 입력 필드의 모습은 다음 그림과 같다.



그림 1-87

Html.ValidationMessage() 메서드

`Html.ValidationMessage()` 메서드는 `ModelState` 컬렉션에 추가된 에러 메시지를 출력하기 위해 사용할 수 있다.

```
<%= Html.ValidationMessage("EventDate") %>
```

위의 코드는 다음과 같은 HTML 코드를 렌더링한다.

```
<span class="field-validation-error"> The value '웹지니' is invalid</span>
```

`Html.ValidationMessage()` 메서드의 두 번째 매개 변수는 페이지에 표시될 에러 메시지를 개발자가 직접 지정하기 위한 매개 변수이다.

```
<%= Html.ValidationMessage("EventDate", "*") %>
```

위의 코드는 다음과 같은 HTML 코드를 렌더링한다.

`*`

렌더링된 코드에서 알 수 있듯이 EventDate 속성에 대해 기본적으로 제공되는 오류 메시지 대신 개발자가 지정한 메시지가 출력되었다.

Html.ValidationSummary() 메서드

Html.ValidationSummary() 메서드는 ModelState 컬렉션에 추가된 모든 오류 메시지를 요약하여 `` 코드를 이용하여 출력하는 기능을 제공한다.

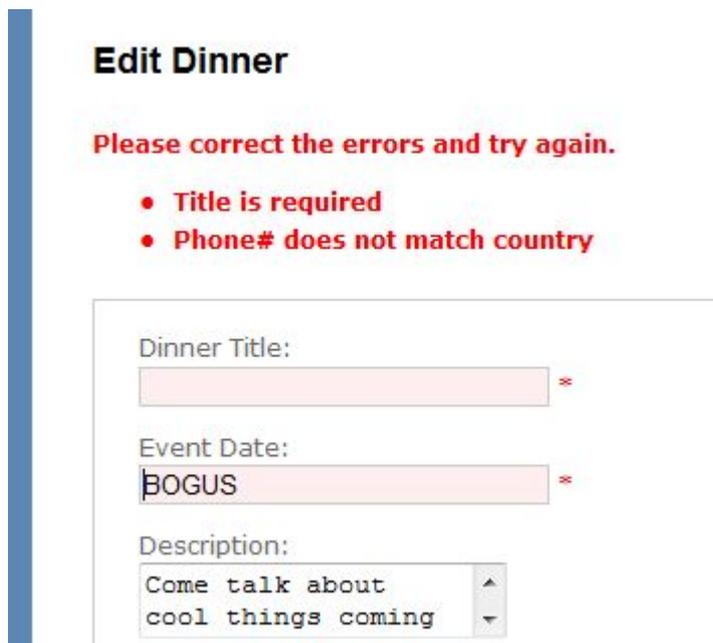


그림 1-88

Html.ValidationSummary() 메서드는 선택적으로 문자열 매개 변수를 사용하는데 이 문자열 매개 변수는 모든 오류 메시지의 상단에 출력될 제목 문자열을 정의할 때 사용한다.

```
<%= Html.ValidationSummary("다음과 같은 오류가 발생했습니다.") %>
```

물론 오류 메시지에 적용될 CSS 스타일도 재정의가 가능하다.

AddRuleViolations 메서드 활용하기

앞서 구현했던 HTTP POST 방식을 위한 Edit 액션 메서드에서 우리는 다음과 같이 Dinners 객체에서 발생한 모든 규칙 오류를 검사하여 컨트롤러의 ModelState 컬렉션에 추가하는 catch 블록을 작성하였다.

```
catch {
```

```

foreach (var issue in dinner.GetRuleViolations()) {
ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
}
return View(dinner);
}

```

동일한 기능을 조금 더 간단하게 구현하기 위해 NerdDinner 프로젝트에 "ControllerHelpers" 클래스를 추가하고 ASP.NET MVC의 ModelStateDictionary 클래스에 적용될 "AddRuleValidation" 확장 메서드를 구현해보자. 이 확장 메서드는 RuleValidation 클래스들이 표현하는 오류를 ModelStateDictionary 클래스에 추가하는 코드를 캡슐화한다.

```

public static class ControllerHelpers {
public static void AddRuleViolation(this ModelStateDictionary modelState,
IEnumerable<RuleViolation> errors) {
foreach (RuleViolation issue in errors) {
ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
}
}
}

```

메서드를 구현했다면 HTTP POST 방식을 위한 Edit 액션 메서드가 이 메서드를 사용하여 Dinners 객체의 규칙 오류를 ModelState 컬렉션에 추가하도록 수정할 수 있다.

완벽하게 구현된 Edit 액션 메서드

다음의 코드는 Dinners 객체를 수정하기 위해 구현된 Edit 액션 메서드의 전체 코드이다.

```

//
// GET: /Dinners/Edit/2
public ActionResult Edit(int id) {
Dinner dinner = dinnerRepository.GetDinner(id);
return View(dinner);
}
//
// POST: /Dinners/Edit/2
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
Dinner dinner = dinnerRepository.GetDinner(id);
try {
UpdateModel(dinner);
dinnerRepository.Save();
return RedirectToAction("Details", new { id=dinner.DinnerID });
}
catch {
ModelState.AddModelErrors(dinner.GetRuleViolations());
return View(dinner);
}
}

```

이 코드의 장점은 컨트롤러 클래스와 뷰 템플릿 모두 Dinner 모델 객체가 수행하는 유효성 검사나 비즈니스 규칙에 대해 알 필요가 없다는 것이다. 나중에 모델 객체에 새로운 규칙을 추가하더

라도 컨트롤러나 뷰 템플릿의 코드는 전혀 변경될 필요가 없다. 이와 같은 구현은 향후 애플리케이션에 추가적인 요구사항이 발생하더라도 코드의 변경을 최소화할 수 있는 유연함을 제공한다.

HTTP GET 방식을 위한 Create 액션 메서드 구현하기

이것으로 DinnersController 클래스의 "Edit" 동작을 모두 구현하였으므로 이제 새로운 데이터를 추가하는 "생성" 기능을 구현해보자.

우선은 HTTP "GET" 방식에 대한 동작을 구현할 것이다. 이 메서드는 /Dinners/Create URL이 요청되면 호출되며 다음과 같이 구현한다.

```
//  
// GET: /Dinners/Create  
public ActionResult Create() {  
    Dinners dinner = new Dinners() {  
        EventDate = DateTime.Now.AddDays(7)  
    };  
    return View(dinner);  
}
```

위의 코드는 새로운 Dinners 객체를 생성하고 EventDate 속성에 1주일 뒤의 날짜를 대입한다. 그런 후 새로운 Dinners 객체를 표시할 뷰를 렌더링한다. View() 메서드에 렌더링될 뷰 이름을 명시하지 않았기 때문에 이름 명명 규칙에 의해 /Views/Dinners/Create.aspx 뷰 템플릿을 기본적으로 사용하게 될 것이다.

그러면 이 뷰 템플릿을 생성해보자. Create 액션 메서드를 마우스 오른쪽 버튼으로 클릭하고 "Add View" 메뉴를 선택한다. "Add View" 대화 상자에서는 Dinners 객체를 뷰 템플릿에 전달하도록 선택하고 "Create" 템플릿을 선택하여 코드 자동 생성 기능을 활용하도록 한다.

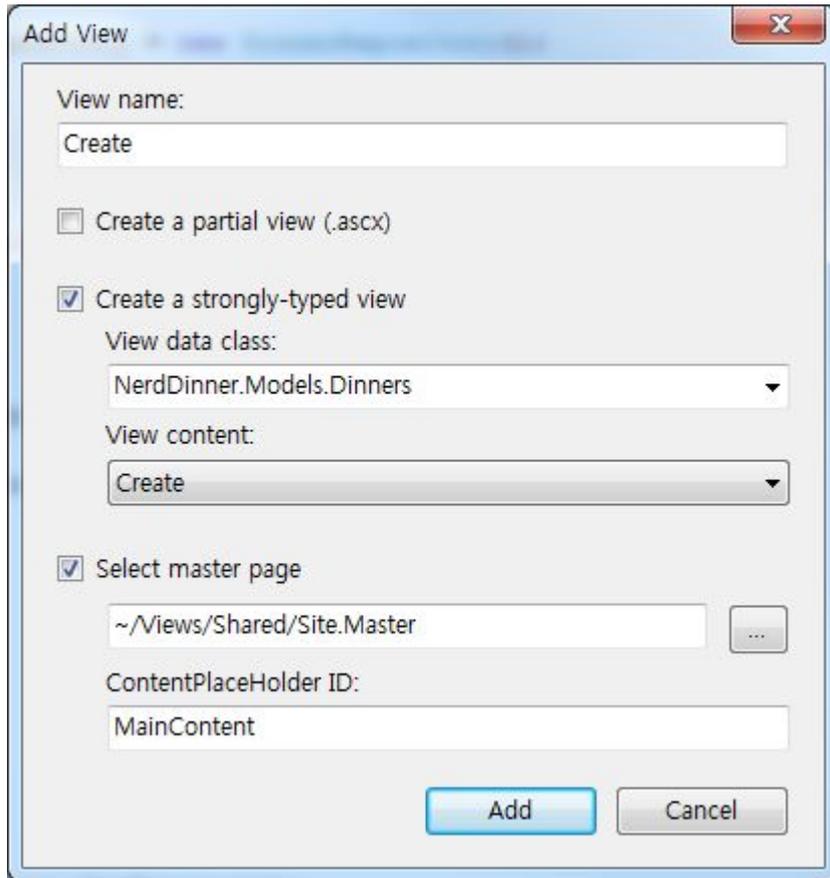


그림 1-89

“Add” 버튼을 클릭하면 Visual Studio는 “\Views\Dinners\” 디렉터리에 “Create.aspx” 뷰 템플릿을 추가하고 이 파일을 IDE에 로드한다.

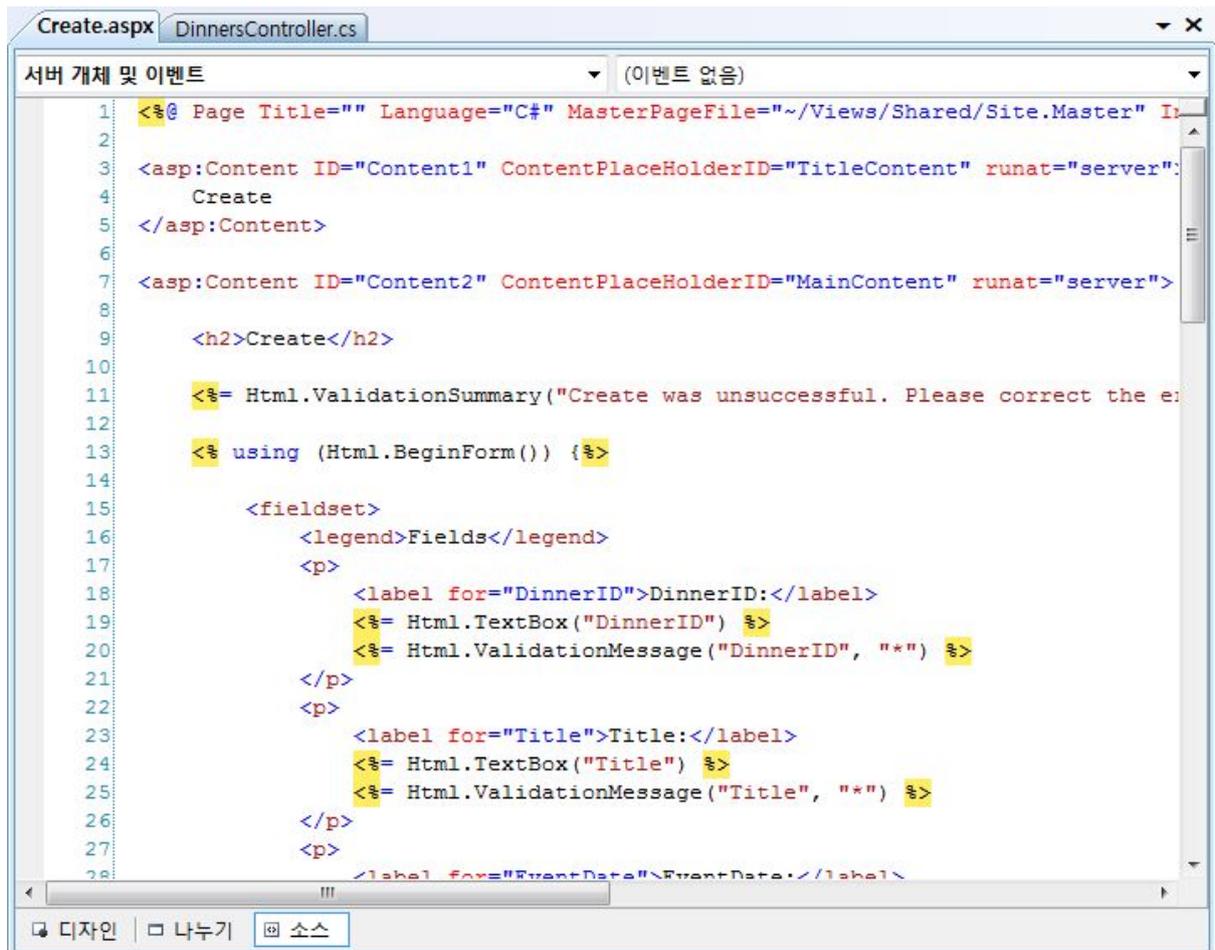


그림 1-90

이제 기본적으로 생성된 코드를 다음과 같이 수정한다.

```

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
Host a Dinner
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>Host a Dinner</h2>
<%= Html.ValidationSummary("Please correct the errors and try again.") %>
<%= using (Html.BeginForm()) {%>
<fieldset>
<p>
<label for="Title">Title:</label>
<%= Html.TextBox("Title") %>
<%= Html.ValidationMessage("Title", "**") %>
</p>
<p>
<label for="EventDate">Event Date:</label>
<%= Html.TextBox("EventDate") %>
<%= Html.ValidationMessage("EventDate", "**") %>
</p>
<p>
<label for="Description">Description:</label>
<%= Html.TextArea("Description") %>
<%= Html.ValidationMessage("Description", "**") %>

```

```

</p>
<p>
<label for="Address">Address:</label>
<%= Html.TextBox("Address") %>
<%= Html.ValidationMessage("Address", "*") %>
</p>
<p>
<label for="Country">Country:</label>
<%= Html.TextBox("Country") %>
<%= Html.ValidationMessage("Country", "*") %>
</p>
<p>
<label for="ContactPhone">ContactPhone:</label>
<%= Html.TextBox("ContactPhone") %>
<%= Html.ValidationMessage("ContactPhone", "*") %>
</p>
<p>
<label for="Latitude">Latitude:</label>
<%= Html.TextBox("Latitude") %>
<%= Html.ValidationMessage("Latitude", "*") %>
</p>
<p>
<label for="Longitude">Longitude:</label>
<%= Html.TextBox("Longitude") %>
<%= Html.ValidationMessage("Longitude", "*") %>
</p>
<p>
<input type="submit" value="Save" />
</p>
</fieldset>
<% } %>
</asp:Content>

```

이제 애플리케이션을 실행하고 "/Dinners/Create" URL을 요청해 보자. 그러면 브라우저는 아래와 같은 페이지를 보여줄 것이다.

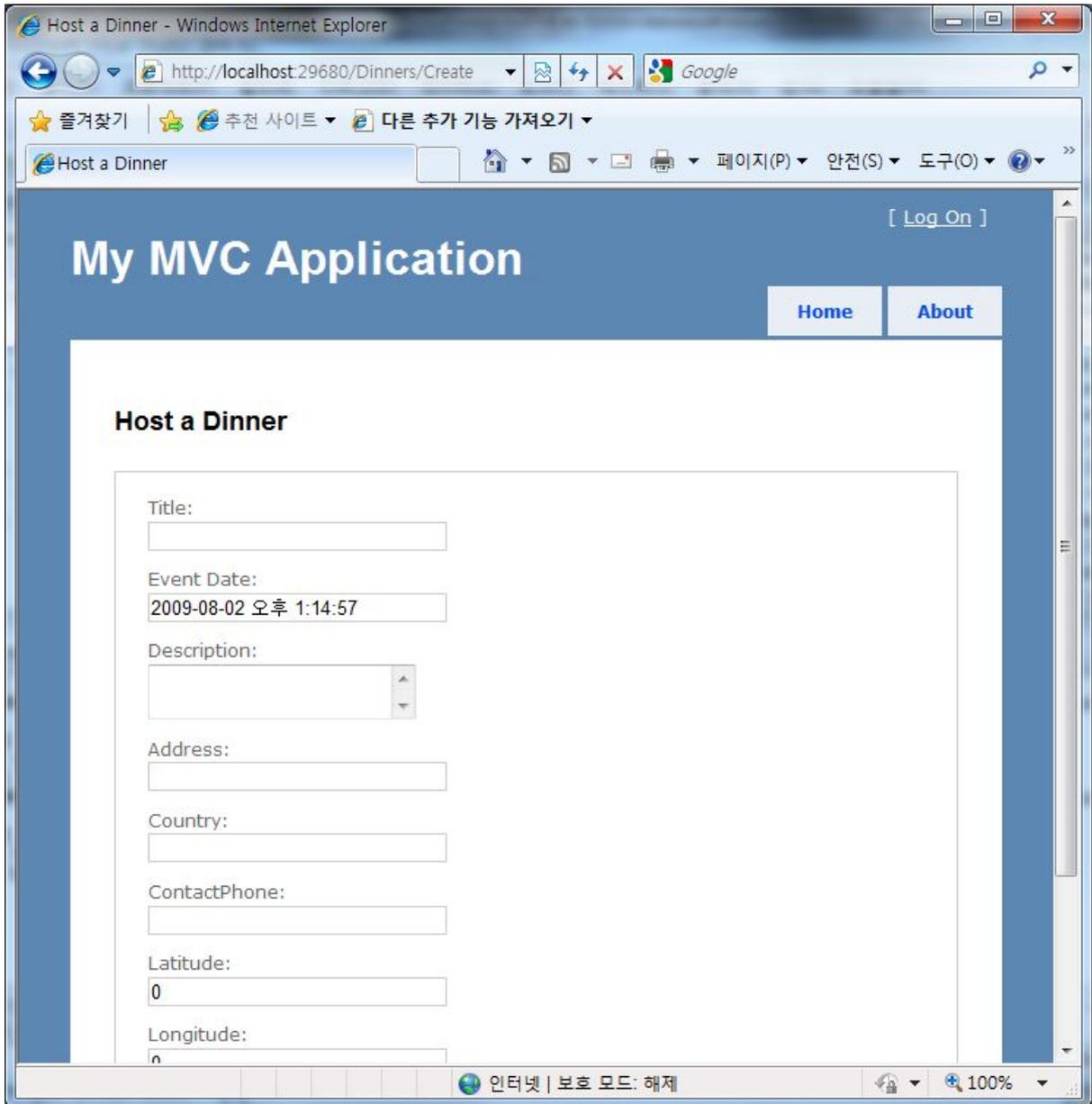


그림 1-91

HTTP POST 방식을 위한 Create 액션 메서드 구현하기

조금 전 우리는 HTTP GET 방식을 지원하기 위한 Create 액션 메서드를 구현하였다. 이 페이지에서 사용자가 “저장” 버튼을 클릭하면 사용자가 페이지에 입력한 데이터들은 HTTP POST 방식으로 /Dinners/Create URL에 전달된다.

그러면 생성 기능을 구현하기 위해 HTTP POST 방식의 Create 액션 메서드를 구현해보자. 아래와 같이 DinnersController 클래스에 “Create” 액션 메서드를 재정의하고 “AcceptVerbs” 특성을 적용하여 HTTP POST 방식의 요청을 처리할 수 있도록 한다.

//

```
// POST: /Dinners/Create
[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Create() {
    ...
}
```

HTTP POST 방식을 지원하는 "Create" 액션 메서드 내에서 폼 데이터를 액세스하는 방법은 여러 가지가 존재한다.

첫 번째 방법은 다음의 코드와 같이 새로운 Dinners 객체를 생성하고 (앞서 수정 기능을 구현할 때 사용했던) UpdateModel() 메서드를 호출하여 폼 데이터를 적용하는 방법이다. 그런 후 값이 적용된 Dinners 객체를 DinnerRepository 클래스를 이용하여 데이터베이스에 저장하고 상세 보기 페이지로 이동하도록 한다.

```
//
// POST: /Dinners/Create
[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Create(FormCollection formValues) {
    Dinners dinner = new Dinners();
    try {
        UpdateModel(dinner);
        dinnerRepository.Add(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new {id=dinner.DinnerID});
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());
        return View(dinner);
    }
}
```

또 다른 방법은 Create() 액션 메서드가 Dinners 객체를 메서드 매개 변수로 전달받도록 구현하는 방법이다. 그러면 ASP.NET MVC는 자동으로 새로운 Dinners 객체의 인스턴스를 생성하고 사용자가 입력한 값들을 Dinners 객체의 속성 값으로 대입한 후 액션 메서드에 전달한다.

```
//
// POST: /Dinners/Create
[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Create(Dinner dinner) {
    if (ModelState.IsValid) {
        try {
            dinner.HostedBy = "SomeUser";
            dinnerRepository.Add(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details", new {id = dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
        }
    }
    return View(dinner);
}
```

위의 액션 메서드는 ModelState.IsValid 속성을 이용하여 사용자가 입력한 폼 데이터가 적용된 Dinners 객체가 유효한 상태인지를 검사한다. ModelState.IsValid 속성은 (EventDate 속성에 “웹지니”와 같은 문자열을 대입한 것처럼) 모델 객체에 오류가 있으면 false를 리턴하며 액션 메서드는 무언가 문제가 있으면 페이지를 다시 표시한다.

사용자가 입력한 값에 문제가 없으면 액션 메서드는 새로운 Dinners 객체를 DinnerRepository 클래스에 추가하고 데이터베이스에 저장한다. 이 작업은 try / catch 블록으로 싸여있어 (dinnerRepository.Save() 메서드를 호출했을 때 예외가 발생하는 등의) 비즈니스 규칙 위반이 발생하면 페이지를 다시 표시하도록 구현되었다.

이와 같은 오류 처리 기능이 동작하는지 확인하려면 /Dinners/Create URL을 요청하여 페이지 양식에 새로운 Dinners 객체를 위한 데이터를 채운다. 유효하지 않은 값을 입력했다면 다음 그림과 같이 양식 페이지가 다시 보여질 것이다.

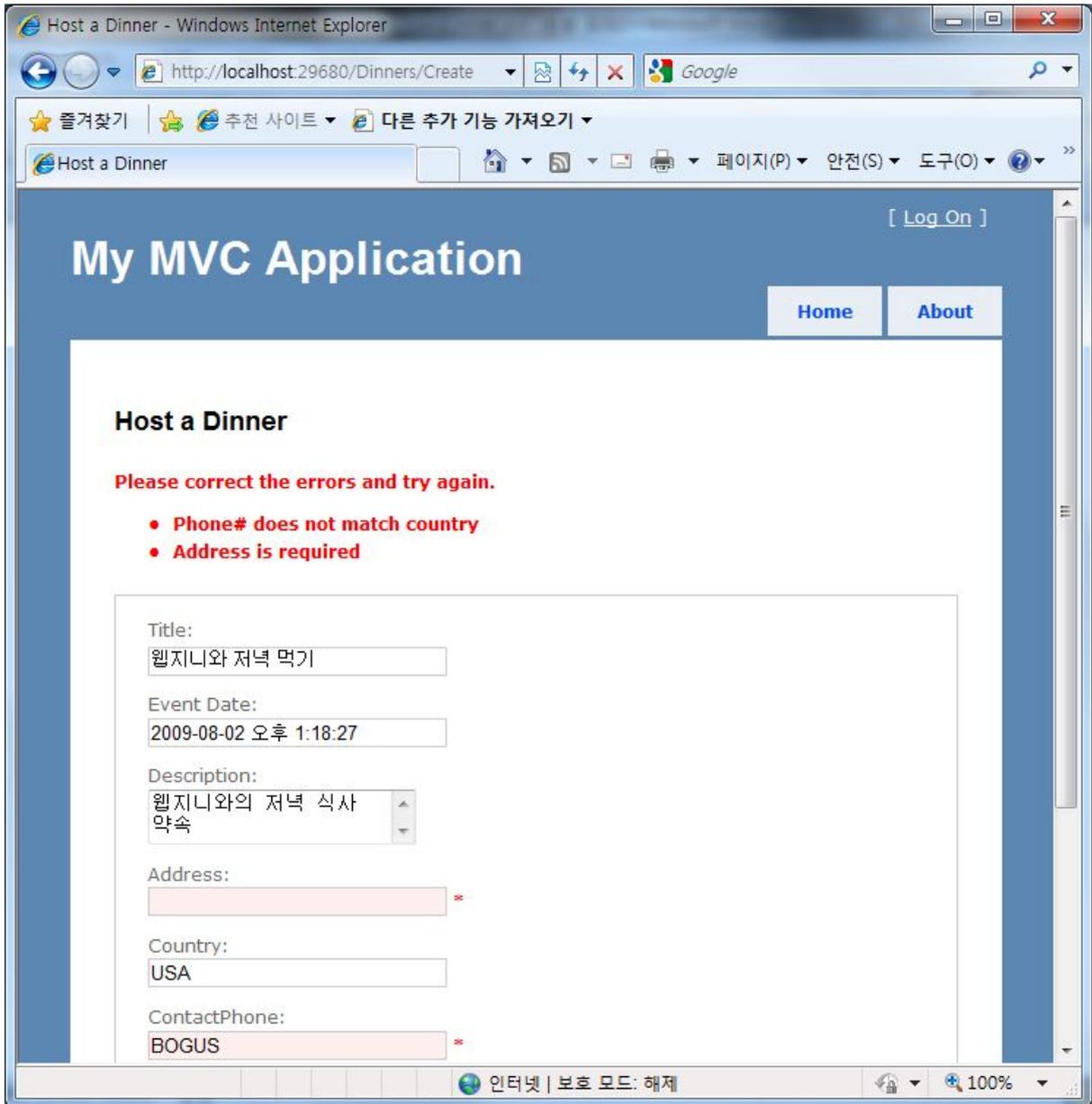


그림 1-92

그림에서 알 수 있듯이 생성 페이지는 수정 페이지와 동일한 유효성 검사 및 비즈니스 규칙 검사 로직을 사용한다. 이는 유효성 검사와 비즈니스 규칙 검사를 실행하는 코드가 컨트롤러나 뷰가 아닌 모델 객체에 정의되어 있기 때문이다. 이는 유효성 검사 혹은 비즈니스 규칙 검사 로직을 한 곳에서 관리할 수 있으며 애플리케이션 전체에 걸쳐 적용되도록 할 수 있음을 의미한다. 따라서 새로운 규칙을 추가하거나 기존의 규칙을 수정하더라도 Create나 Edit 액션 메서드를 수정할 필요가 없다.

생성 페이지에서 올바른 값들만을 입력하고 “저장” 버튼을 클릭하면 DinnerRepository 클래스에 성공적으로 Dinners 객체가 추가되고 이 객체의 데이터는 데이터베이스에 저장된다. 그런 후 /Dinners/Details/[id] URL로 리다이렉트 하여 새로 추가된 Dinners 객체의 상세 정보를 표시한다.

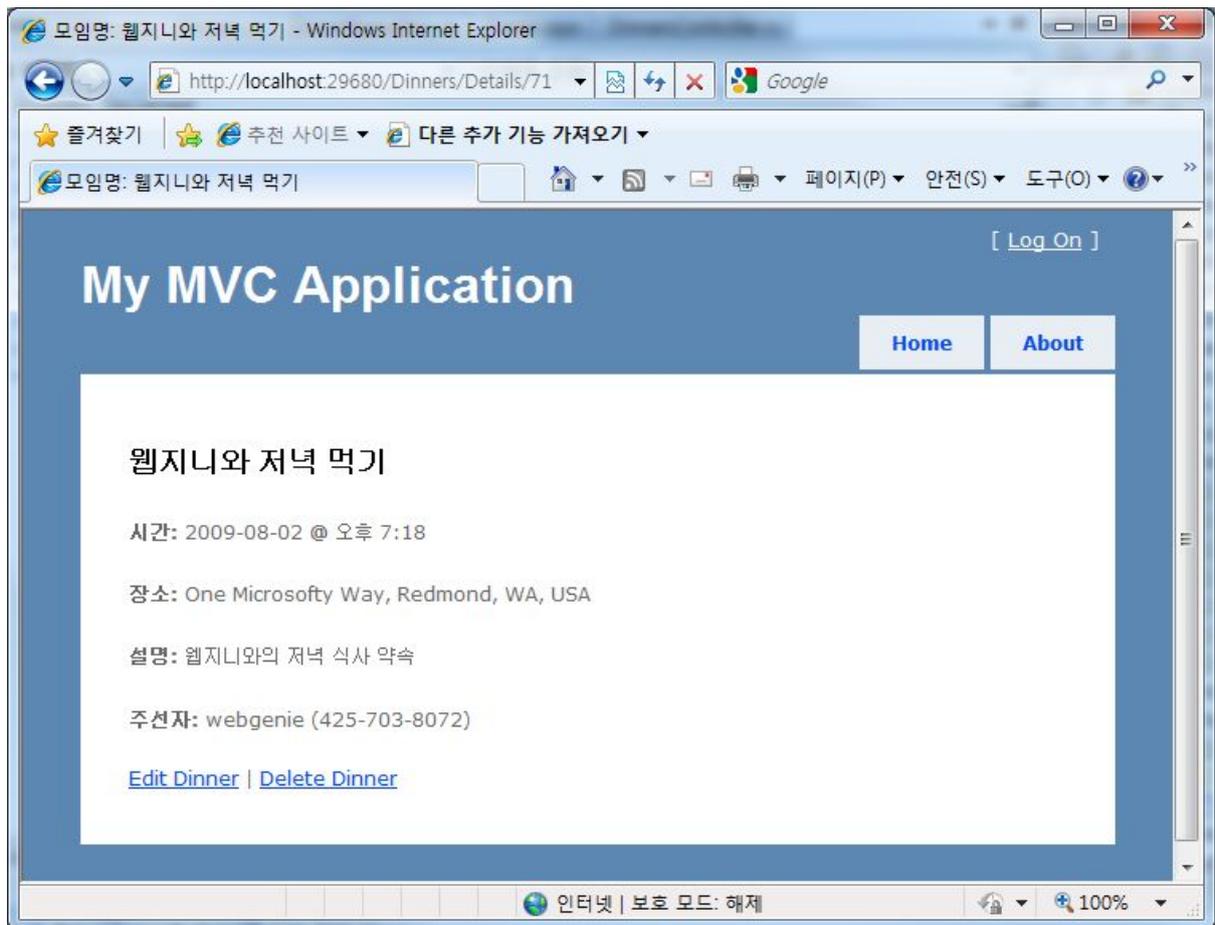


그림 1-93

HTTP GET 방식을 위한 Delete 액션 메서드 구현하기

이제 DinnersController 클래스에 "삭제" 기능을 추가해보자.

우선은 HTTP GET 방식의 요청을 처리하는 액션 메서드부터 구현하자. 이 메서드는 /Dinners/Delete/[id] URL에 대한 요청이 들어올 때 호출되며 코드는 다음과 같다.

```
//
// HTTP GET: /Dinners/Delete/1
public ActionResult Delete(int id) {
    Dinners dinner = dinnerRepository.GetDinner(id);
    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
```

이 액션 메서드는 먼저 삭제할 Dinners 객체를 조회한다. Dinners 객체가 검색되면 이 Dinners 객체를 이용하여 뷰를 렌더링하고 객체를 검색하지 못하면 (혹은 이미 삭제되었으면) 앞서 "Details"

액션 메서드의 구현에서 사용했던 "NotFound" 뷰 템플릿을 보여준다.

이제 코드 편집기에서 Delete 액션 메서드를 마우스 오른쪽 버튼으로 클릭하고 "Add View" 메뉴를 선택하여 "Delete" 뷰 템플릿을 추가해보자. "Add View" 대화 상자에서는 뷰 템플릿이 사용할 모델 객체를 Dinners 객체로 지정하고 "Empty" 템플릿을 사용하도록 설정한다.

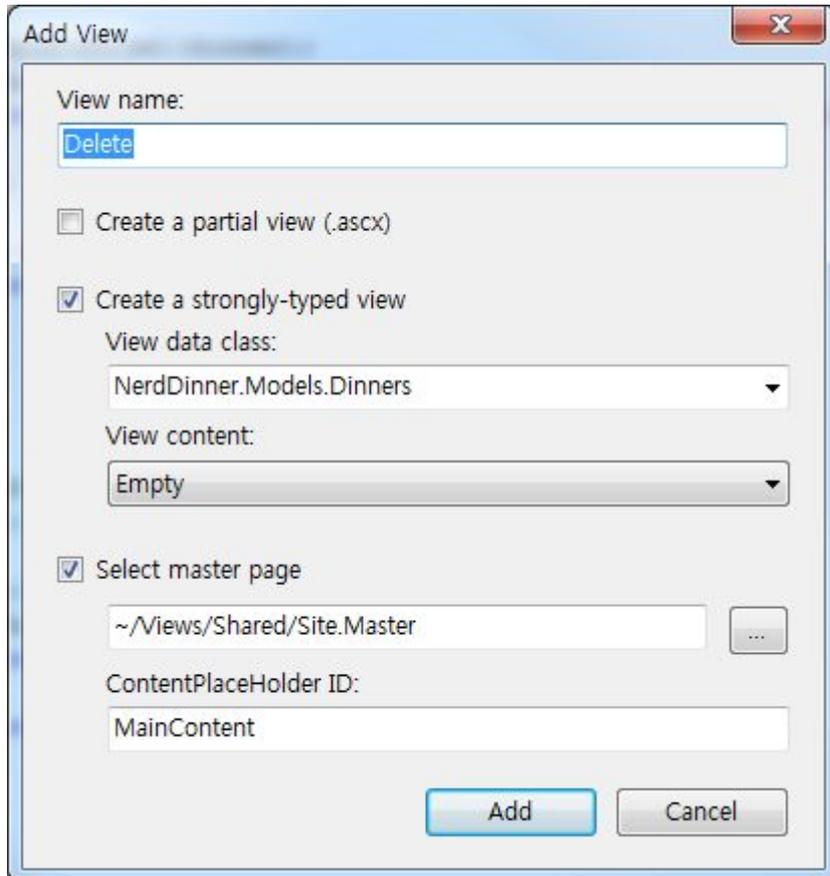


그림 1-94

"Add" 버튼을 클릭하면 Visual Studio는 "Views\Dinners" 디렉터리에 "Delete.aspx" 뷰 템플릿을 추가한다. 이제 다음과 같이 약간의 HTML과 코드를 추가하여 삭제 확인 페이지를 구성해보자.

```
<asp:Content ID="Title" ContentPlaceHolderID="head" runat="server">
Delete Confirmation: <%=Html.Encode(Model.Title) %>
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>
Delete Confirmation
</h2>
<div>
<p>Please confirm you want to cancel the dinner titled:
<i> <%=Html.Encode(Model.Title) %>? </i> </p>
</div>
<% using (Html.BeginForm()) { %>
<input name="confirmButton" type="submit" value="Delete" />
<% } %>
```

</asp:Content>

이 코드는 삭제될 Dinners 객체의 Title 속성 값을 출력하고 사용자가 “삭제” 버튼을 클릭하면 /Dinners/Delete/[id] URL로 HTTP POST 요청을 보내는 <FORM> 요소를 렌더링한다.

애플리케이션을 실행하고 “/Dinners/Delete/[id]” URL을 요청하면 유효한 Dinners 객체가 검색된 경우 다음 그림과 같은 페이지를 볼 수 있다.

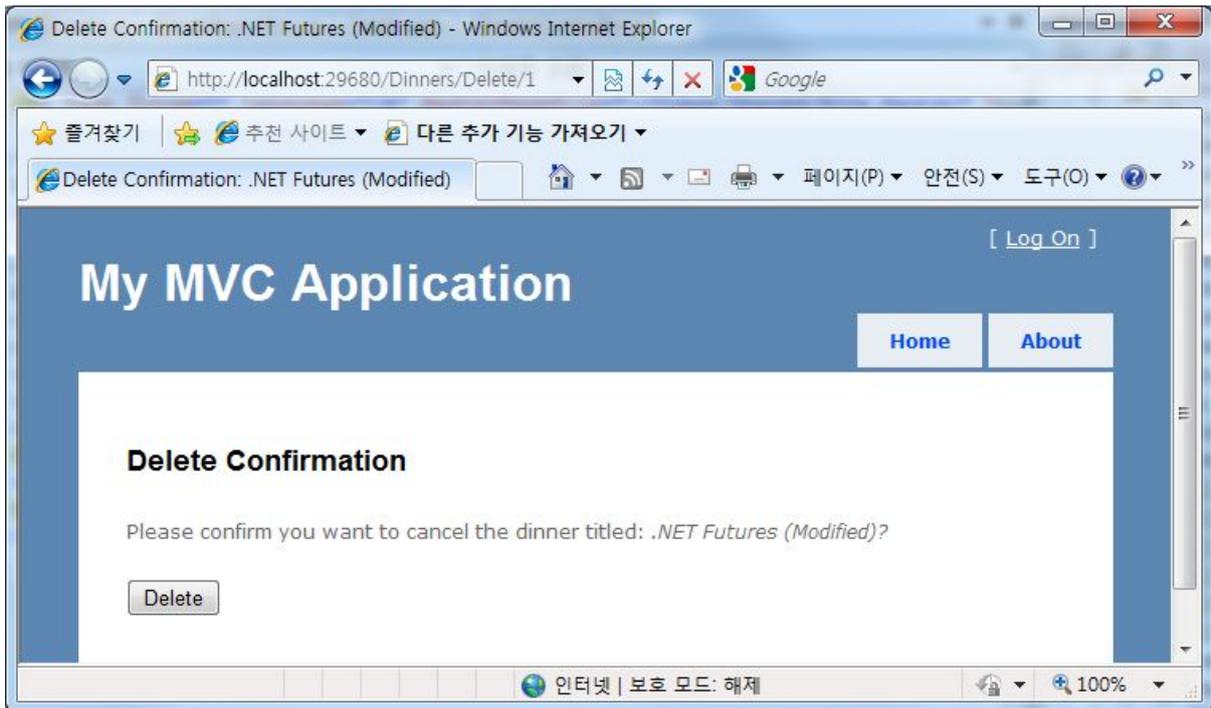


그림 1-95

=====

왜 우리는 POST 방식을 사용하는걸까?

“대체 왜 삭제 확인 페이지에 <FORM> 태그를 만드는 쓸데없는 노력을 해야 하는거지? 그냥 표준 하이퍼링크를 이용해서 삭제 동작을 수행하는 액션 메서드로 링크를 설정해도 될텐데!” 라고 생각하는 독자들이 있을지도 모르겠다.

POST 방식을 사용하는 이유는 웹 수집기와 검색 엔진들이 삭제 URL을 수집하여 다른 사용자들에게 노출하고 사용자들이 이 URL을 클릭하게 됨으로써 원치 않게 데이터가 삭제되는 상황을 방지하기 위해서이다. HTTP GET 방식의 URL은 웹 수집기나 검색 엔진들이 가져가도 비교적 안전하며 이렇게 수집된 URL을 통해 사용자들이 해당 URL에 접근한다 하더라도 HTTP POST 방식의 URL에는 접근하지 못한다고 생각할 수 있다.

가장 좋은 규칙은 데이터에 변경을 가하는 작업은 항상 HTTP POST 방식을 사용하는 것이다.

HTTP POST 방식을 위한 Delete 액션 메서드 구현하기

앞서 완성한 HTTP GET 방식의 액션 메서드가 보여주는 삭제 확인 페이지는 사용자가 “삭제” 버튼을 클릭하면 데이터를 삭제하기 위해 /Dinners/Delete/[id] URL로 HTTP POST 요청을 전달한다.

그러면 이 HTTP POST 요청을 처리하여 데이터를 삭제하는 액션 메서드를 다음과 같이 구현해보자.

```
//  
// HTTP POST: /Dinners/Delete/1  
[AcceptVerbs(HttpVerbs.Post)]  
public ActionResult Delete(int id, string confirmButton) {  
    Dinner dinner = dinnerRepository.GetDinner(id);  
    if (dinner == null)  
        return View("NotFound");  
    dinnerRepository.Delete(dinner);  
    dinnerRepository.Save();  
    return View("Deleted");  
}
```

HTTP POST 방식의 Delete 액션 메서드는 먼저 삭제할 Dinners 객체를 검색한다. 만일 (이미 삭제되었거나 기타 다른 이유로) 원하는 객체를 검색하지 못하면 “NotFound” 뷰 템플릿을 렌더링하며 원하는 객체를 찾았다면 DinnerRepository 클래스를 이용하여 객체를 삭제하고 “Deleted” 템플릿을 렌더링한다.

“Deleted” 뷰 템플릿은 코드 편집기에서 액션 메서드를 마우스 오른쪽 버튼으로 클릭하고 “Add View” 메뉴를 선택하여 생성할 수 있다. “Add View” 대화 상자에서는 생성할 뷰 템플릿의 이름을 “Deleted”로 변경하고 (모델 타입은 지정할 필요 없이) 앞서와 마찬가지로 “Empty” 템플릿을 선택한 후 다음과 같이 HTML 코드를 작성한다.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">  
Dinner Deleted  
</asp:Content>  
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">  
<h2>Dinner Deleted.</h2>  
<div>  
<p>Your dinner was successfully deleted.</p>  
</div>  
<div>  
<p><a href="/dinners">Click for upcoming dinners</a></p>  
</div>  
</asp:Content>
```

이제 애플리케이션을 실행하고 “/Dinners/Delete/[id]” URL에 유효한 Dinners 객체의 ID를 전달하여 요청하면 다음과 같이 삭제 확인 페이지가 나타난다.

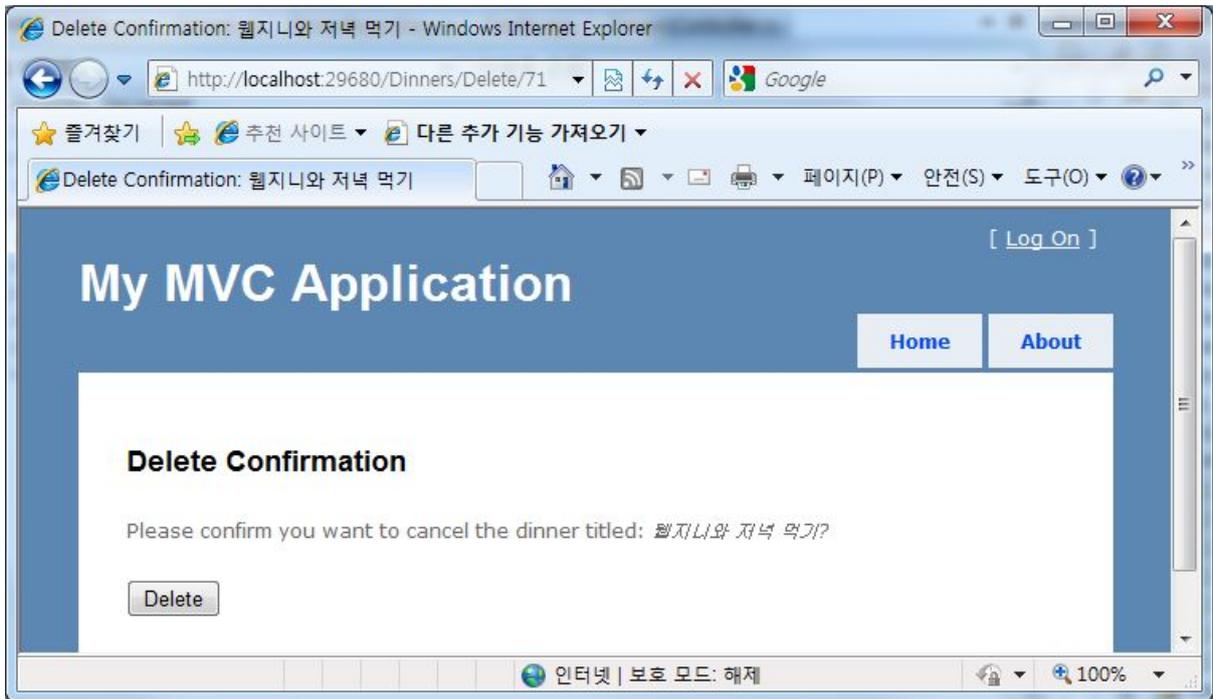


그림 1-96

여기서 "삭제" 버튼을 클릭하면 /Dinners/Delete/[id] URL로 HTTP POST 요청이 전달되어 데이터가 데이터베이스에서 삭제되고 "Deleted" 뷰 템플릿이 렌더링된다.

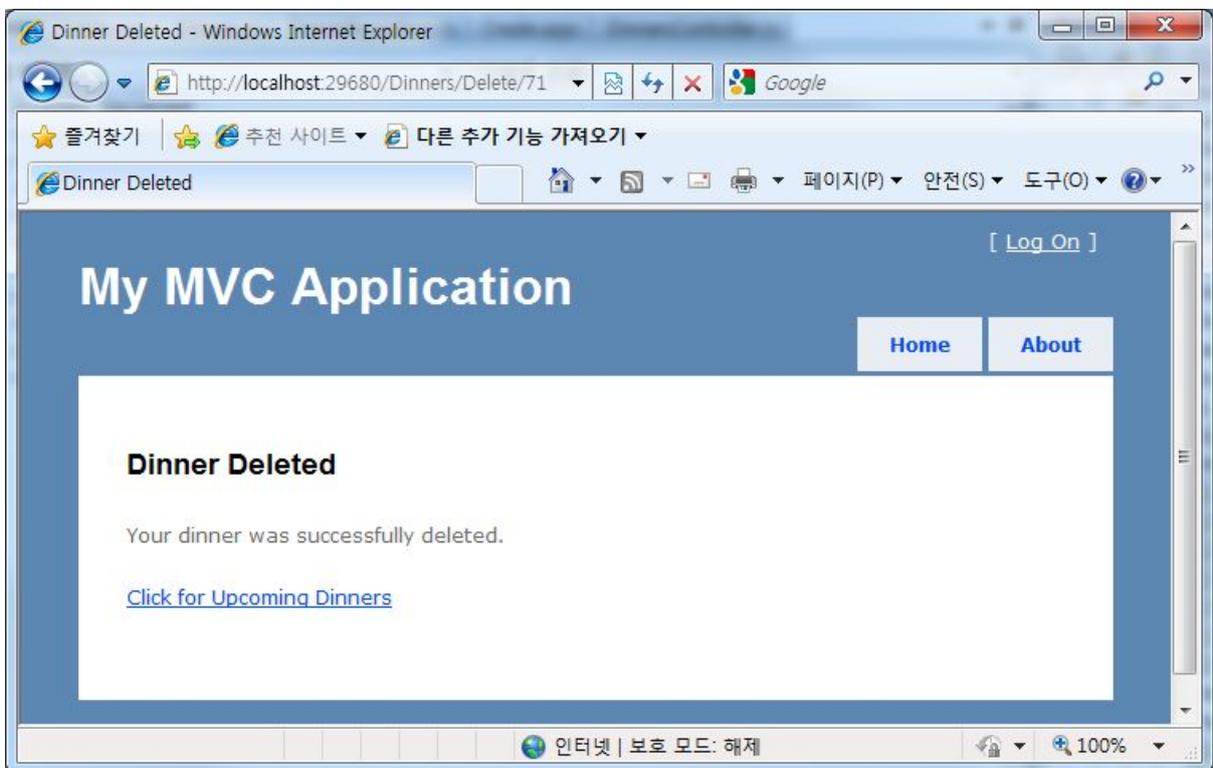


그림 1-97

모델 바인딩시의 보안에 대한 고려

지금까지 우리는 ASP.NET MVC가 제공하는 두 가지 모델 바인딩(Model-binding) 기능에 대해 살펴보았다. 첫 번째 방법은 UpdateModel() 메서드를 호출하여 이미 인스턴스가 생성된 모델 객체의 속성 값을 변경하는 방법이며 두 번째 방법은 ASP.NET MVC가 액션 메서드의 매개 변수로 사용될 모델 객체를 자동으로 생성하도록 하는 방법이다. 두 가지 모두 강력하며 매우 쓸만하다.

그러나 이와 같은 강력함을 활용하기 위해서는 여러분의 책임이 필요하다. 즉 사용자가 입력한 값을 가져올 때는 반드시 이 값이 유효한지를 꼼꼼히 살펴야 하며 사용자의 데이터를 모델에 바인딩할 때도 마찬가지이다. 사용자가 입력한 값들에 대해서는 HTML과 자바스크립트 인젝션 공격, 그리고 SQL 인젝션 공격(이 애플리케이션은 이와 같은 공격들을 자동으로 방지하는 LINQ to SQL을 사용하기 때문에 걱정하지 않아도 되지만)을 대비하여 항상 인코딩을 수행해야 한다. 사용자가 입력한 값들에 대해서는 클라이언트 측 유효성 검사 코드만을 사용해서는 안되며 반드시 서버 측 코드에서도 유효성 검사를 수행해야 한다.

여러분이 ASP.NET MVC의 바인딩 기능을 사용할 때 고려해야 할 또 다른 보안적인 이슈는 바인딩하고자 하는 객체의 범위이다. 특히 여러분이 속성에 대해 바인딩을 수행할 때 발생할 수 있는 잠재적인 보안 문제를 인식하고 반드시 사용자에게 의해 업데이트되어야 하는 속성들에 대해서만 바인딩을 수행해야 한다.

기본적으로 UpdateModel() 메서드는 요청에 포함된 변수들과 동일한 이름을 가진 모델 객체의 모든 속성 값들을 업데이트하려고 시도한다. 마찬가지로 액션 메서드에 매개 변수로 전달되는 객체 역시 기본적으로 모든 폼 매개 변수들을 속성에 대입한다.

사용 방식 기반의 바인딩 제한

모델 바인딩 기능을 이용할 때 접근이 허용되는 속성의 목록을 명시적으로 정의함으로써 바인딩 정책을 제한할 수 있다. 접근을 허용할 속성의 목록은 다음과 같이 문자열 배열 형태로 UpdateModel() 메서드에 전달하면 된다.

```
string[] allowedProperties = new[]{ "Title", "Description",  
"ContactPhone", "Address",  
"EventDate", "Latitude",  
"Longitude"};  
UpdateModel(dinner, allowedProperties);
```

액션 메서드의 매개 변수로 전달되는 객체인 경우에는 다음과 같이 [Bind] 특성을 이용하여 바인딩을 허용할 속성의 목록을 지정할 수 있다.

```
// POST: /Dinners/Create
```

```
[AcceptVerbs (HttpVerbs.Post)]
public ActionResult Create( [Bind(Include="Title,Address")] Dinner dinner )
{
    ...
}
```

타입 기반의 바인딩 제한

또 다른 방법은 타입을 이용하여 바인딩 규칙을 제한하는 방법이다. 이렇게 하면 특정 바인딩 규칙을 한 번만 정의하여 (UpdateModel() 메서드 방법과 액션 메서드의 매개 변수 방법에 관계 없이) 여러 컨트롤러와 액션 메서드에서 재활용할 수 있다.

타입 기반 바인딩 제한을 수행하려면 타입에 [Bind] 특성을 추가하거나 Global.ascx 파일의 애플리케이션에 타입 기반의 바인딩 규칙을 등록(다른 개발자가 구현한 타입을 사용하는 경우에 유용하다)하면 된다. 그런 후 Bind 특성의 Include 혹은 Exclude 속성에 바인딩을 허용하는 클래스나 인터페이스를 지정하면 된다.

NerdDinner 애플리케이션은 이 방법을 이용하여 다음과 같이 바인딩 가능한 속성을 제한하기 위한 [Bind] 특성을 사용하였다.

```
[Bind(Include="Title,Description,EventDate,Address,Country,ContactPhone,Longitude,Longitude")]
public partial class Dinner {
}
```

위의 코드에서 알 수 있듯이 RSVPs 컬렉션 속성과 DinnerID 속성, HostedBy 속성에는 바인딩을 허용하지 않았다. 보안상 바인딩을 허용하지 않은 속성들은 액션 메서드 내에서 코드를 통해 명시적으로 값을 변경해 주어야 한다.

CRUD 기능에 대한 정리

ASP.NET MVC는 폼 데이터가 전송되었을 때의 처리를 위해 제법 많은 수의 내장 기능들을 제공하며 우리는 이들 중 일부를 사용하여 DinnerRepository 클래스를 이용한 CRUD UI를 구현했다. (역자 주: CRUD란 Create, Retrieve, Update, Delete의 약자로 데이터베이스에 데이터를 추가, 조회, 수정 및 삭제 하는 기능들을 의미한다.)

우리는 모델에 초점을 맞추어 애플리케이션의 기능을 구현하였다. 이는 우리가 구현한 모든 유효성 검사 및 비즈니스 규칙 검사 로직이 컨트롤러나 뷰가 아닌 모델 계층에 구현되어 있다는 것을 의미한다. 따라서 컨트롤러와 뷰 템플릿은 Dinner 모델 클래스에 정의된 비즈니스 규칙에 대해서는 전혀 알지 못한다.

애플리케이션을 이와 같이 구현하면 테스트를 수행하기가 보다 편리해진다. 뿐만 아니라 새로운

비즈니스 규칙을 추가하더라도 컨트롤러나 뷰 템플릿의 코드를 변경할 필요가 없다. 따라서 향후에 애플리케이션을 빠르게 확장하거나 변경할 수 있다.

이제 DinnersController 클래스는 데이터를 나열하거나 상세 데이터를 보여줄 수 있을 뿐 아니라 새로운 데이터를 추가하거나 수정 혹은 삭제하는 기능도 제공할 수 있게 되었다. 다음은 이 클래스의 전체 소스 코드이다.

```
public class DinnersController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
    // GET: /Dinners/
    public ActionResult Index() {
        var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        return View(dinners);
    }
    //
    // GET: /Dinners/Details/2
    public ActionResult Details(int id) {
        Dinner dinner = dinnerRepository.GetDinner(id);
        if (dinner == null)
            return View("NotFound");
        else
            return View(dinner);
    }
    //
    // GET: /Dinners/Edit/2
    public ActionResult Edit(int id) {
        Dinner dinner = dinnerRepository.GetDinner(id);
        return View(dinner);
    }
    //
    // POST: /Dinners/Edit/2
    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(int id, FormCollection formValues) {
        Dinner dinner = dinnerRepository.GetDinner(id);
        try {
            UpdateModel(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details", new { id = dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
            return View(dinner);
        }
    }
    //
    // GET: /Dinners/Create
    public ActionResult Create() {
        Dinner dinner = new Dinner() {
            EventDate = DateTime.Now.AddDays(7)
        };
        return View(dinner);
    }
    //
    // POST: /Dinners/Create
    [AcceptVerbs(HttpVerbs.Post)]
```

```

public ActionResult Create(Dinner dinner) {
    if (ModelState.IsValid) {
        try {
            dinner.HostedBy = "SomeUser";
            dinnerRepository.Add(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details", new{id=dinner.DinnerID});
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
        }
    }
    return View(dinner);
}
//
// HTTP GET: /Dinners/Delete/1
public ActionResult Delete(int id) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
// HTTP POST: /Dinners/Delete/1
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(int id, string confirmButton) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (dinner == null)
        return View("NotFound");
    dinnerRepository.Delete(dinner);
    dinnerRepository.Save();
    return View("Deleted");
}
}
}

```

ViewData와 ViewModel

지금까지 우리는 데이터를 포스팅하는 몇 가지 시나리오를 검토하고 생성과 수정, 삭제 기능을 어떻게 구현할 수 있는지에 대해 살펴보았다. 이제 DinnersController 클래스를 더욱 확장하여 보다 풍부한 폼 편집 시나리오를 지원할 수 있도록 구현해보자. 이 과정에서 우리는 컨트롤러로부터 뷰로 데이터를 전달하는 두 가지 방법을 살펴보게 될 것이다.

컨트롤러에서 뷰 템플릿으로 데이터 전달하기

MVC 패턴의 특징 중 하나는 애플리케이션을 구성하는 서로 다른 컴포넌트들 사이의 "역할의 분리(Separation of Concerns)"이다. 모델과 컨트롤러 그리고 뷰에는 각자의 역할과 책임이 정해져 있으며 잘 정리된 방법으로 서로 커뮤니케이션을 한다. 이와 같은 방법은 테스트와 코드 재사용성 면에서 많은 도움이 된다.

Controller 클래스는 HTML 응답을 렌더링할 뷰를 결정할 때 반드시 뷰 템플릿이 필요로 하는 데이터들을 전달해 주어야 한다. 뷰 템플릿은 데이터의 조회나 애플리케이션과 관련된 로직을 절대

로 실행해서는 안되며 오로지 컨트롤러가 전달해 준 데이터나 모델을 렌더링하는 것에만 집중해야 한다.

현재 DinnersController 클래스가 뷰 템플릿에 데이터를 전달하는 방법은 매우 간단하며 직관적이다. Index() 액션 메서드의 경우에는 Dinners 객체의 컬렉션을 전달하며 Details(), Edit(), Delete() 액션 메서드의 경우에는 하나의 Dinners 객체를 전달한다. 애플리케이션에 보다 많은 UI 관련 기능이 추가될수록 뷰 템플릿이 HTML 응답을 렌더링하기 위해 필요로 하는 데이터 역시 늘어날 것이다. 예를 들어 Edit 뷰나 Create 뷰에서 사용하는 “국가” 필드를 텍스트 상자가 아닌 드롭다운 리스트로 변경한다고 가정해보자. 아마도 드롭다운 목록에 전체 국가의 목록을 일일이 작성하는 것보다는 동적으로 국가의 목록을 가져와 나열하는 것이 더 좋을 것이다. 그러려면 컨트롤러에서 뷰 템플릿에 Dinners 객체 외에 사용 가능한 국가의 목록도 함께 전달해 줄 방법이 필요하다.

그렇다면 이와 같은 문제를 해결할 수 있는 두 가지 방법에 대해 살펴보도록 하자.

ViewData 속성의 활용

Controller 클래스는 컨트롤러가 뷰에 추가적인 데이터를 전달할 수 있도록 “ViewData” 사전 속성 (Dictionary Property)을 제공한다. (역자 주: 사전 속성이라는 다소 어색한 번역은 해당 속성이 Dictionary 타입을 사용하고 있음을 표현하기 위해 사용하였다. ViewData 속성은 ViewDataDictionary 타입을 사용하고 있다.)

예를 들어 “Edit” 뷰에서 “국가”를 입력하는 텍스트 상자를 드롭다운 목록으로 변경하고자 하는 경우 Edit() 액션 메서드를 다음과 같이 수정하여 (Dinners 객체 외에도) SelectList 객체를 뷰 템플릿에 전달할 수 있도록 해야 한다.

```
//  
// GET: /Dinners/Edit/5  
[Authorize]  
public ActionResult Edit(int id) {  
    Dinner dinner = dinnerRepository.GetDinner(id);  
    ViewData["Countries"] = new SelectList(PhoneValidator.Countries,  
    dinner.Country);  
    return View(dinner);  
}
```

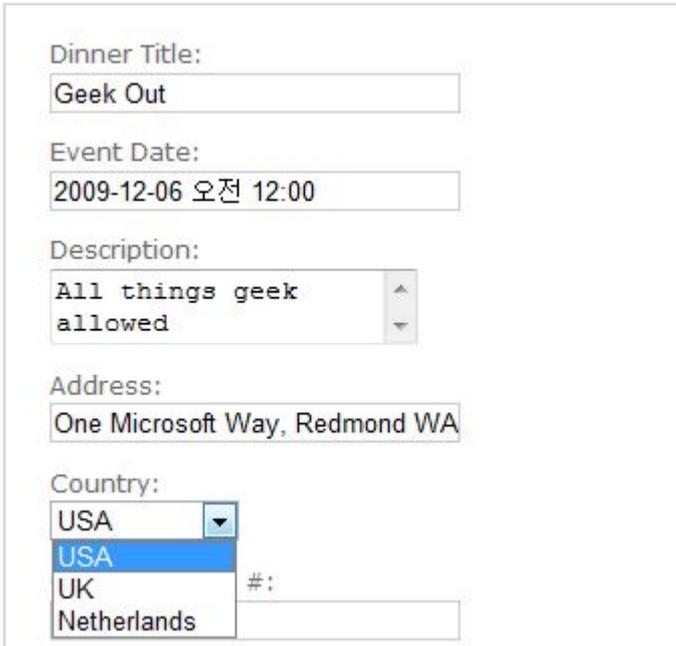
위의 코드에서 사용된 SelectList 객체의 생성자 메서드는 드롭다운 목록을 구성할 때 사용할 국가의 목록과 목록에서 현재 선택될 항목의 값을 전달받는다.

다음으로 Edit.aspx 뷰 템플릿을 수정하여 Html.TextBox() 메서드 대신 Html.DropDownList() 메서드를 사용하도록 다음과 같이 코드를 수정한다.

```
<%= Html.DropDownList("Country", ViewData["Countries"] as SelectList) %>
```

Html.DropDownList() 메서드는 두 개의 매개 변수를 필요로 한다. 첫 번째 매개 변수는 렌더링될 HTML 요소의 name 특성에 지정될 이름이며 두 번째 매개 변수는 ViewData 사전을 통해 전달되는 "SelectList" 타입의 모델 객체이다. 이 때 사전 객체를 통해 전달된 객체를 SelectList 타입으로 변환하기 위해서 C#의 "as" 연산자를 사용하였다.

이제 애플리케이션을 실행하고 브라우저를 통해 /Dinners/Edit/1 URL을 요청하면 다음 그림과 같이 데이터 수정을 위한 UI가 업데이트 되어 텍스트 상자 대신 드롭다운 목록이 나타나는 것을 볼 수 있다.



The screenshot shows a web form with the following fields:

- Dinner Title:
- Event Date:
- Description:
- Address:
- Country: (dropdown menu with options: USA, UK, Netherlands)
- #:

그림 1-98

Edit 뷰 템플릿은 HTTP POST 요청을 처리하는 과정에서 오류가 발생했을 때에도 사용되기 때문에 이 메서드 역시 ViewData 속성을 이용하여 뷰 템플릿에 SelectList 객체를 전달할 수 있도록 다음과 같이 수정해야 한다.

```
//  
// POST: /Dinners/Edit/5  
[AcceptVerbs(HttpVerbs.Post)]  
public ActionResult Edit(int id, FormCollection collection) {  
    Dinner dinner = dinnerRepository.GetDinner(id);  
    try {  
        UpdateModel(dinner);  
        dinnerRepository.Save();  
        return RedirectToAction("Details", new { id=dinner.DinnerID });  
    }  
    catch {  
        ModelState.AddModelErrors(dinner.GetRuleViolations());  
        ViewData["countries"] = new SelectList(PhoneValidator.Countries,  
        dinner.Country);  
    }  
}
```

```
return View(dinner);
}
}
```

이제 DinnersController 클래스는 데이터 수정 기능에서 드롭다운 목록을 지원할 수 있게 되었다.

ViewModel 패턴의 활용

ViewData 사전을 이용하면 애플리케이션을 쉽고 빠르게 구현할 수 있다. 그러나 일부 개발자들은 문자열 기반의 사전 객체를 사용할 경우 오타로 인해 컴파일 시점에 알아낼 수 없는 오류가 발생할 우려가 있다는 이유로 문자열 기반의 사전 객체를 사용하는 것을 좋아하지 않는다. 또한 타입이 명확하지 않은 ViewData 사전을 사용하면 C#처럼 강력한 타입 언어를 뷰 템플릿에서 사용하는 경우에는 "as" 연산자나 형 변환 작업이 추가로 필요하기도 하다.

또 다른 방법은 "ViewMode" 패턴이라 불리는 방법을 사용하는 것이다. 이 패턴을 이용한다는 것은 특정 뷰를 위해 강력한 타입의 클래스들을 생성하여 뷰 템플릿이 필요로 하는 동적인 값이나 콘텐츠를 위한 속성들을 정의하는 방법을 사용한다는 것이다. 이 방법을 이용하면 컨트롤러 클래스들은 뷰 템플릿에 뷰에 최적화된 클래스들을 생성하여 전달할 수 있게 된다. 또한 타입에 안전하며 컴파일 시점에서의 타입 검사 및 뷰 템플릿 코드에서의 인텔리센스 지원 등이 가능해진다.

예를 들어 새로운 데이터를 추가하는 경우를 위해 다음과 같이 Dinners 객체와 SelectList 객체를 위한 속성을 제공하는 "DinnersFormViewModel" 클래스를 정의할 수 있다.

```
public class DinnerFormViewModel {
    // Properties
    public Dinner Dinner { get; private set; }
    public SelectList Countries { get; private set; }
    // Constructor
    public DinnerFormViewModel(Dinner dinner) {
        Dinner = dinner;
        Countries = new SelectList(PhoneValidator.Countries,
            dinner.Country);
    }
}
```

그런 후 다음과 같이 Edit() 액션 메서드를 수정하여 DinnerFormViewModel 객체를 생성하고 저장소에서 검색된 Dinners 객체를 뷰 템플릿에 전달하도록 구현할 수 있다.

```
//
// GET: /Dinners/Edit/5
[Authorize]
public ActionResult Edit(int id) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    return View(new DinnerFormViewModel(dinner));
}
```

다음으로 뷰 템플릿을 수정하여 "Dinner" 객체 대신 "DinnerFormViewModel" 객체를 사용하도록 "Edit.aspx" 페이지의 Inherits 특성을 다음과 같이 변경한다.

```
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerFormViewMode  
1>
```

그러면 뷰 템플릿의 "Model" 속성은 우리가 전달한 DinnerFormViewModel 객체를 참조하게 되어 다음 그림과 같이 인텔리센스가 동작하게 된다.

```
<%= Html.TextBox("Title", Model. %>
```



그림 1-99

```
<%= Html.TextBox("Title", Model.Dinner. %>
```

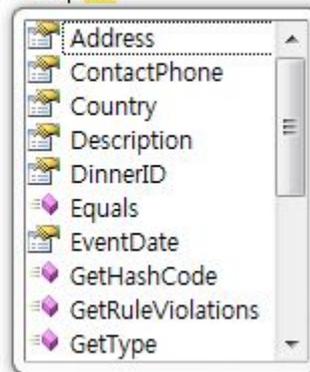


그림 1-100

이제는 뷰 코드를 손 볼 차례이다. 다음의 코드에서 알 수 있듯이 렌더링 될 요소들의 이름은 변경하지 않았지만 HTML 헬퍼 메서드들이 DinnerFormViewModel 클래스로부터 값을 가져오도록 수정하였다.

```
<p>  
<label for="Title">Dinner Title:</label>  
<%= Html.TextBox("Title", Model.Dinner.Title) %>  
<%= Html.ValidationMessage("Title", "*") %>  
</p>  
<p>  
<label for="Country">Country:</label>  
<%= Html.DropDownList("Country", Model.Countries) %>
```

```
<%= Html.ValidationMessage("Country", "*") %>
</p>
```

또한 HTTP POST 요청을 처리할 Edit 액션 메서드 역시 DinnerFormViewModel 클래스를 사용하여 다음과 같이 수정한다.

```
//
// POST: /Dinners/Edit/5
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection collection) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    try {
        UpdateModel(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());
        return View(new DinnerFormViewModel(dinner));
    }
}
```

마찬가지로 Create() 액션 메서드 역시 DinnerFormViewModel 클래스를 사용하여 국가 목록을 드롭다운 리스트로 변경할 수 있다. 다음의 코드는 HTTP GET 방식을 위한 Create() 액션 메서드를 수정한 코드이다.

```
//
// GET: /Dinners/Create
public ActionResult Create() {
    Dinner dinner = new Dinner() {
        EventDate = DateTime.Now.AddDays(7)
    };
    return View(new DinnerFormViewModel(dinner));
}
```

HTTP POST 방식을 위한 Create() 액션 메서드는 다음과 같이 수정한다.

```
//
// POST: /Dinners/Create
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Dinner dinner) {
    if (ModelState.IsValid) {
        try {
            dinner.HostedBy = "SomeUser";
            dinnerRepository.Add(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details", new { id=dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinnerToCreate.GetRuleViolations());
        }
    }
    return View(new DinnerFormViewModel(dinnerToCreate));
}
```

이제 Edit과 Create 액션 메서드는 국가의 목록을 드롭다운 목록을 통해 보여줄 수 있게 되었다.

맞춤형 ViewModel 클래스들

앞서 Dinners 객체의 수정과 생성 시나리오를 위해 구현한 DinnerFormViewModel 객체는 Dinners 객체를 속성으로 직접 노출하며 SelectList 모델 속성도 제공한다. 이와 같은 방법은 우리가 구현하려는 뷰 템플릿이 렌더링할 HTML UI가 우리가 정의한 모델 객체와 매우 유사한 경우에는 큰 문제 없이 동작한다.

그러나 뷰 템플릿이 렌더링할 UI가 모델 객체와 유사하지 않은 경우에는 뷰가 필요로 하는 데이터에 보다 최적화된 객체 모델을 제공하는 ViewModel 클래스를 정의해야 하며 이들은 애플리케이션에 정의된 모델 객체와 완전히 다른 형태가 될 수도 있다. 예를 들어 완전히 다른 이름의 속성을 사용하거나 혹은 여러 모델 객체들의 속성을 조합한 이름의 속성을 사용할 수도 있다.

이처럼 맞춤형 ViewModel 클래스들은 컨트롤러가 뷰로 데이터를 전달할 때 사용할 수 있을 뿐 아니라 컨트롤러의 액션 메서드에 의해 전달된 폼 데이터를 처리할 때도 유용하게 사용될 수 있다. 이와 같은 시나리오를 지원하기 위해서 액션 메서드가 포스트된 데이터를 이용하여 ViewModel 객체를 업데이트하고 실제 객체 모델을 조회하기 위해 맞춤형 ViewModel 객체의 인스턴스를 사용하도록 구현할 수 있다.

맞춤형 ViewModel 클래스들을 사용하면 보다 나은 유연성을 제공할 수 있으며 뷰 템플릿 코드 내에서 렌더링 코드를 찾는 시간이나 액션 메서드에서 포스트된 데이터를 처리하는 코드가 복잡해 지는 것을 방지할 수도 있다. 이는 대체로 여러분이 정의한 모델 클래스가 UI에 완벽하게 대응하지 못한다는 것을 뜻하며 이런 경우 중간에 맞춤형 ViewModel 클래스를 적용하는 것이 도움이 될 수 있다.

부분 뷰와 마스터 페이지

ASP.NET MVC의 디자인 철학 중 하나는 “반복 되는 코드를 줄이자(Do Not Repeat Yourself, “DRY”라고 부르기도 한다)” 원칙이다. DRY 디자인은 중복되는 코드와 로직을 제거하여 궁극적으로 애플리케이션을 빠르게 구현하고 손쉽게 유지하기 위한 것이다.

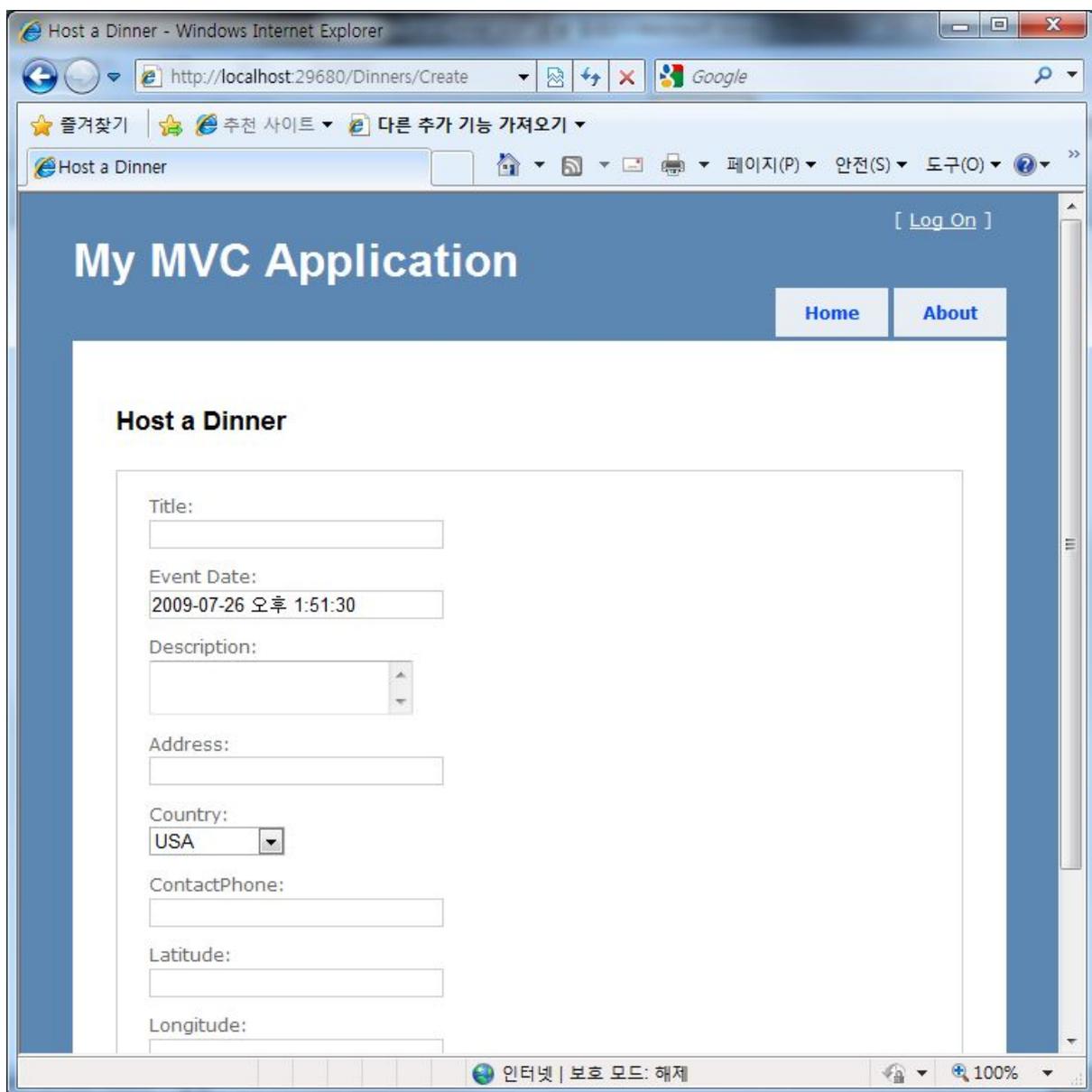
지금까지 구현했던 NerdDinner 애플리케이션에서도 이와 같은 DRY 원칙이 적용된 부분을 손쉽게 찾을 수 있다. 몇 가지 예를 들자면 유효성 검사 로직을 모델 계층에 구현하여 컨트롤러에서 수정과 생성 기능을 구현할 때 재활용한 점이나 “NotFound” 뷰 템플릿을 Edit()과 Details(), Delete() 액션 메서드에서 공유하는 점, View() 메서드를 호출할 때 뷰 이름을 명시적으로 지정하지 않아도 되도록 뷰 템플릿을 명명 규칙에 따라 생성한 점, Edit()과 Create() 액션 메서드에서 DinnerFormViewModel 클래스를 공유하여 사용하는 점 등이 바로 이런 원칙이 적용된 부분이라

할 수 있다.

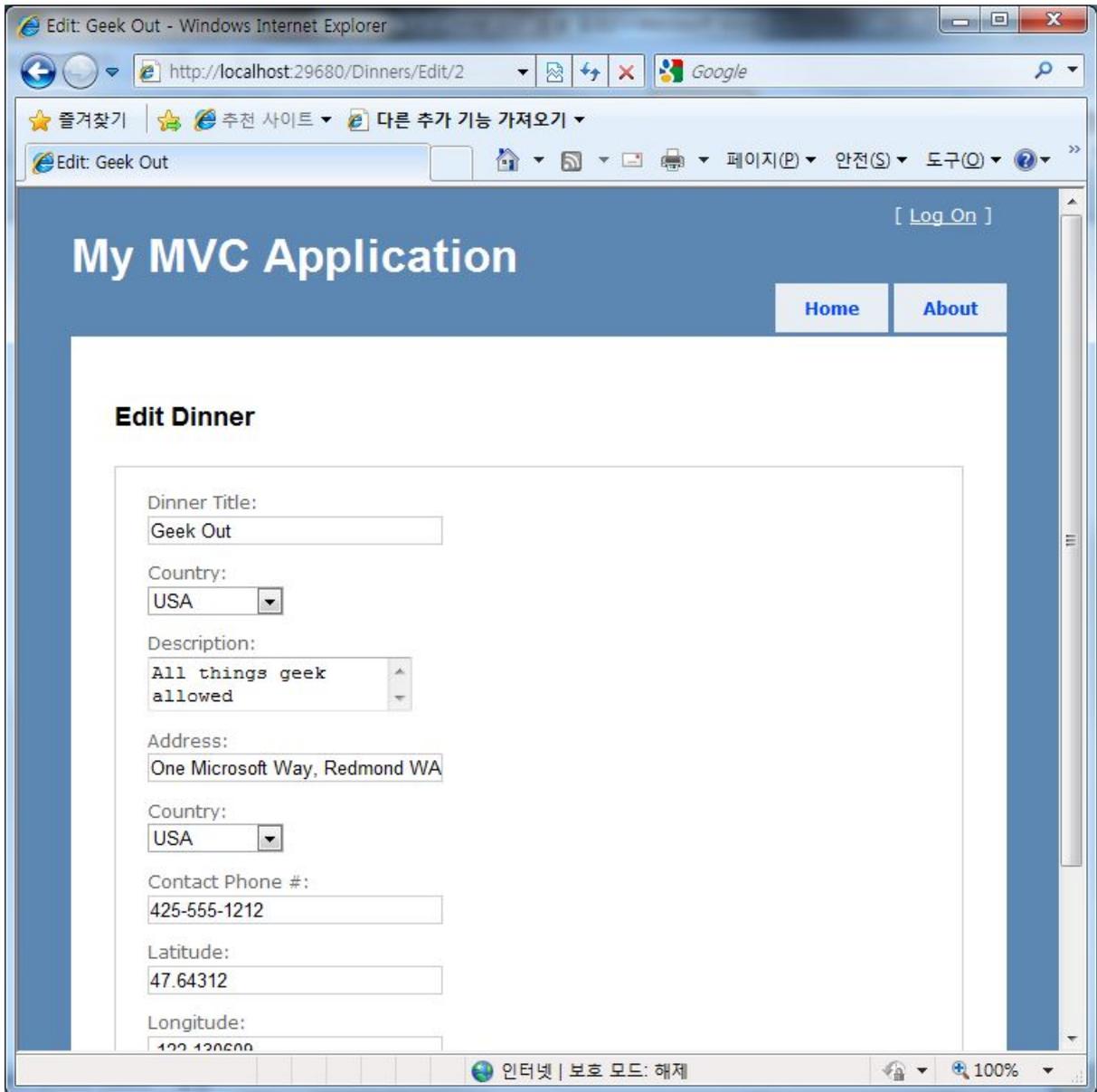
그러면 이와 같은 "DRY 원칙"을 적용하여 중복되는 코드를 제거할 수 있는 뷰 템플릿이 또 있는지 살펴보도록 하자.

Edit과 Create 뷰 템플릿 다시 보기

우리는 Dinners 객체를 위한 양식 UI를 위해 "Edit.aspx"와 "Create.aspx" 등 두 개의 뷰 템플릿을 사용하고 있다. 이 둘이 얼마나 비슷한지 시각적으로 비교해 보자. 다음 그림은 새로운 Dinners 객체를 생성하는 페이지의 모습이다.



그리고 아래의 그림은 이미 존재하는 Dinners 객체의 데이터를 수정하기 위한 페이지이다.



크게 다른 점이 눈에 띄는가? 브라우저의 제목과 헤더 텍스트를 제외하고 페이지의 레이아웃이나 입력 컨트롤들은 모두 동일하다.

“Edit.aspx”와 “Create.aspx” 뷰 템플릿을 열어 보면 이 둘이 동일한 폼 레이아웃과 입력 컨트롤 코드를 사용하고 있음을 볼 수 있다. 이와 같은 중복은 Dinners 클래스에 새로운 속성을 추가하거나 수정할 때마다 똑 같은 코드를 두 번 수정해야 한다는 것을 의미한다. 아마도 그다지 내키지 않을 것이다.

부분 뷰 템플릿의 활용

ASP.NET MVC에서는 “부분 뷰 (Partial View)”를 정의하여 페이지의 일부를 렌더링하는 로직을 캡

솔화할 수 있도록 지원하고 있다. 부분 뷰는 렌더링 로직을 한 번만 구현하고 이를 애플리케이션에서 필요한 곳마다 재활용할 수 있는 매우 유용한 개념이다.

Edit.aspx와 Create.aspx 뷰 템플릿에서 중복된 코드를 제거하기 위해서 "DinnerForm.ascx"라는 이름의 부분 뷰를 생성하고 공통적으로 사용되는 폼 레이아웃과 입력 요소들을 캡슐화할 수 있다. 그러려면 /Views/Dinners 디렉터리를 마우스 오른쪽 버튼으로 클릭하고 "추가 > View" 메뉴를 선택한다.

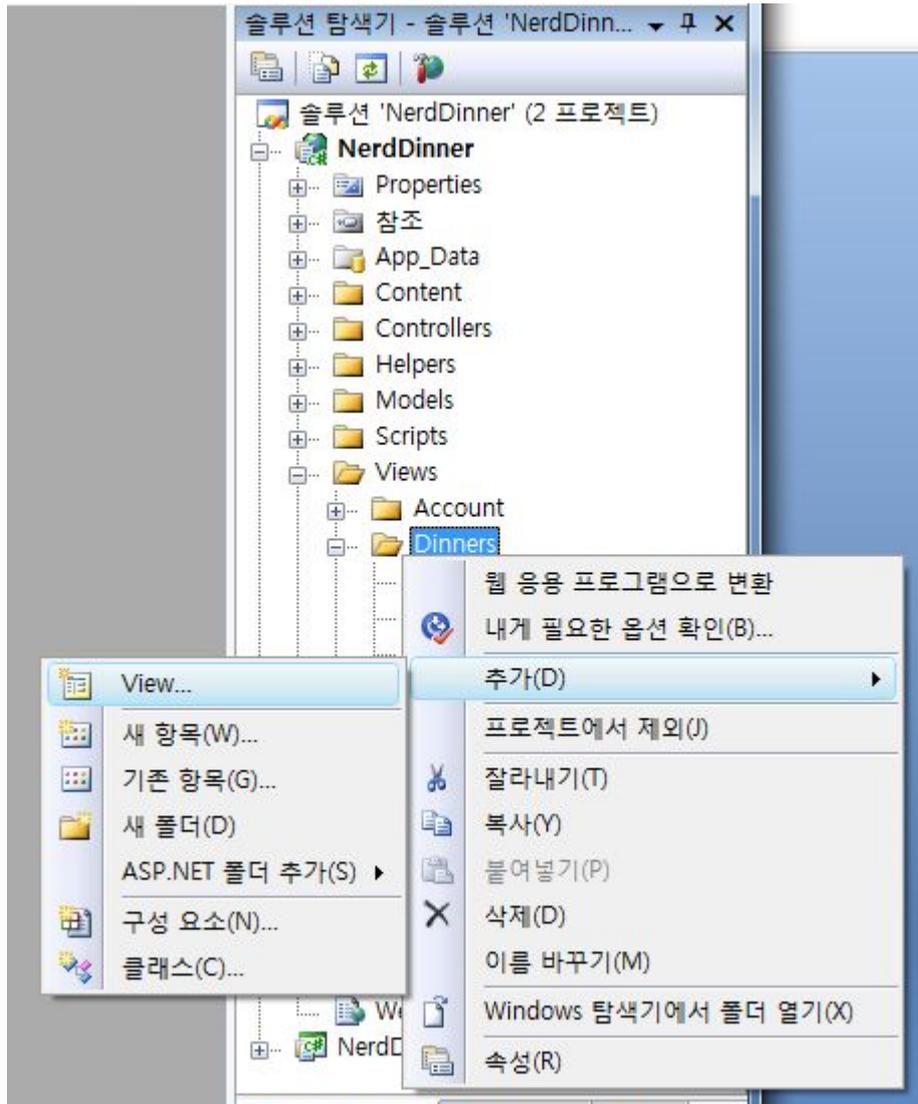


그림 1-103

"Add View" 대화 상자가 나타나면 뷰의 이름을 "DinnerForm"이라고 입력하고 "Create a partial view" 체크 박스를 선택한 후 DinnerFormViewModel 클래스를 사용하도록 설정한다.

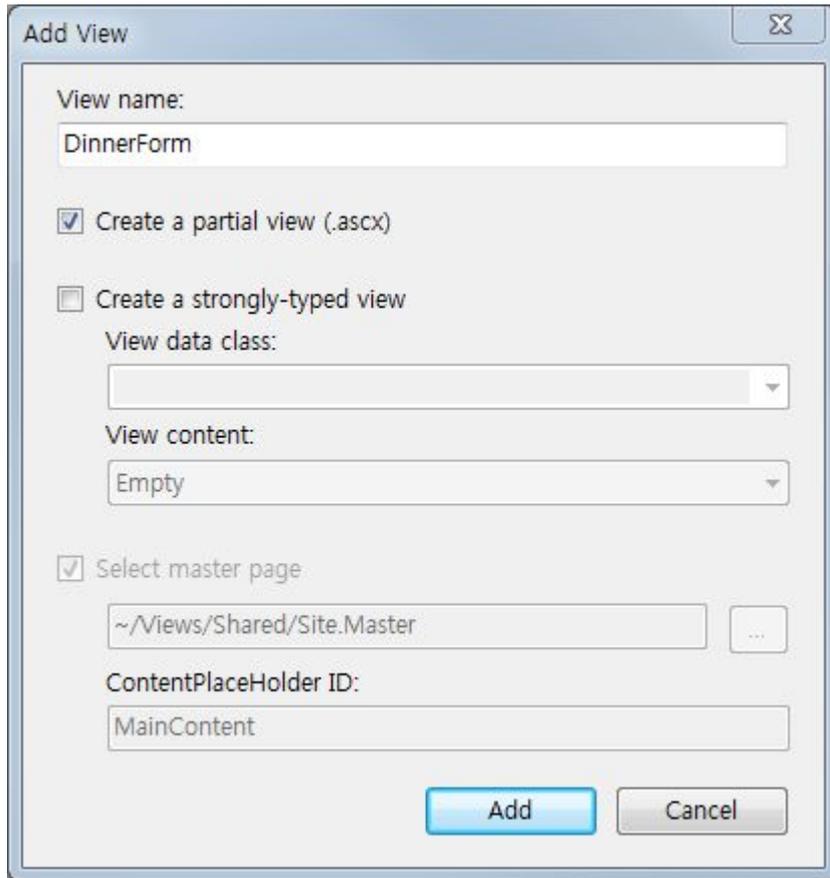


그림 1-104

“Add” 버튼을 클릭하면 Visual Studio는 “\Views\Dinners” 디렉터리에 “DinnerForm.aspx” 뷰 템플릿 파일을 새로 추가한다.

이제 Edit.aspx나 Create.aspx 뷰 템플릿에서 폼 레이아웃과 입력 컨트롤 코드를 복사하여 “DinnerForm.ascx” 부분 뷰 템플릿에 붙여넣자.

```
<%= Html.ValidationSummary("Please correct the errors and try again.") %>
<% using (Html.BeginForm()) { %>
<fieldset>
<p>
<label for="Title">Dinner Title:</label>
<%= Html.TextBox("Title", Model.Dinner.Title) %>
<%= Html.ValidationMessage("Title", "*") %>
</p>
<p>
<label for="EventDate">Event Date:</label>
<%= Html.TextBox("EventDate", Model.Dinner.EventDate) %>
<%= Html.ValidationMessage("EventDate", "*") %>
</p>
<p>
<label for="Description">Description:</label>
<%= Html.TextArea("Description", Model.Dinner.Description) %>
<%= Html.ValidationMessage("Description", "*") %>
</p>
<p>
```

```

<label for="Address">Address:</label>
<%= Html.TextBox("Address", Model.Dinner.Address) %>
<%= Html.ValidationMessage("Address", "*") %>
</p>
<p>
<label for="Country">Country:</label>
<%= Html.DropDownList("Country", Model.Countries) %>
<%= Html.ValidationMessage("Country", "*") %>
</p>
<p>
<label for="ContactPhone">Contact Phone #:</label>
<%= Html.TextBox("ContactPhone", Model.Dinner.ContactPhone) %>
<%= Html.ValidationMessage("ContactPhone", "*") %>
</p>
<p>
<input type="submit" value="Save" />
</p>
</fieldset>
<% } %>

```

그런 후 Edit과 Create 뷰 템플릿의 코드를 수정하여 중복된 부분을 제거하고 DinnerForm 부분 뷰 템플릿을 사용하도록 해야 한다. 그러려면 다음과 같이 Html.RenderPartial("DinnerForm") 메서드를 호출하면 된다.

Create.aspx

```

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
Host a Dinner
</asp:Content>
<asp:Content ID="Create" ContentPlaceHolderID="MainContent" runat="server">
<h2>Host a Dinner</h2>
<% Html.RenderPartial("DinnerForm"); %>
</asp:Content>

```

Edit.aspx

```

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
Edit: <%=Html.Encode(Model.Dinner.Title) %>
</asp:Content>
<asp:Content ID="Edit" ContentPlaceHolderID="MainContent" runat="server">
<h2>Edit Dinner</h2>
<% Html.RenderPartial("DinnerForm"); %>
</asp:Content>

```

Html.RenderPartial 메서드를 호출할 때는 부분 뷰 템플릿의 이름을 명시적으로(예를 들어 ~/Views/Dinners/DinnerForm.aspx와 같이)지정해야 한다. 위의 코드에서는 ASP.NET MVC의 이름 명명 규칙을 사용하였기 때문에 단지 "DinnerForm"이라는 이름만 지정하여도 부분 뷰를 렌더링할 수 있다. 부분 뷰를 렌더링할 때 ASP.NET MVC는 우선 명명 규칙에 해당하는 뷰 디렉터리(DinnersController 컨트롤러의 경우 /Views/Dinners 디렉터리)에서 탐색한다. 만일 지정된 부분 뷰 템플릿을 찾지 못하면 /Views/Shared 디렉터리에서 탐색을 계속한다.

Html.RenderPartial() 메서드에 부분 뷰의 이름만 지정하면 ASP.NET MVC는 부분 뷰를 호출하는 뷰 템플릿에 전달된 것과 동일한 모델 객체와 ViewData 사전 객체를 전달한다. 반면 재정의된

Html.RenderPartial() 메서드를 사용하면 부분 뷰 템플릿에 다른 모델 객체나 ViewData 사전 객체를 전달할 수 있다. 이와 같은 기능은 부분 뷰 템플릿에 모델 객체나 ViewData 객체의 일부만 전달해야 할 때 유용하게 사용될 수 있다.

=====

왜 <%= %> 대신 <% %>를 사용할까?

위의 코드에서 알 수 있는 한 가지 미묘한 차이는 Html.RenderPartial() 메서드를 호출할 때는 <%= %> 블록 대신 <% %> 블록을 사용한다는 점이다.

ASP.NET에서 <%= %> 블록은 개발자가 어떤 값을 렌더링하고 싶어한다는 것을 의미한다 (예를 들어 <%= "Hello" %> 구문은 "Hello"라는 문자열을 렌더링한다). <% %> 블록은 개발자가 코드를 실행하고 싶어 하며 그 내부의 렌더링 로직이 명시적으로 실행되어야 함을 의미한다 (예를 들면 <% Response.Write("Hello"); %>).

Html.RenderPartial() 메서드를 호출할 때 <% %> 블록을 사용한 이유는 Html.RenderPartial() 메서드가 콘텐츠를 문자열로 리턴하는 대신 뷰 템플릿의 응답 스트림에 직접 출력하기 때문이다. 이와 같이 처리하는 이유는 성능의 효율성 때문이며 이렇게 함으로써 출력할 문자열을 임시로 저장하기 위해 거대한 문자열 객체를 만들 필요가 없어진다. 따라서 메모리의 사용을 줄이고 전체적인 애플리케이션의 성능을 향상시킬 수 있다.

Html.RenderPartial() 메서드를 호출할 때 자주 저지를 수 있는 실수는 <% %> 블록의 마지막에 세미콜론(;)을 추가하는 것을 잊어버리는 것이다. 예를 들어 다음의 코드는 컴파일 오류를 발생시킨다.

```
<% Html.RenderPartial("DinnerForm") %>
```

따라서 다음과 같이 세미콜론을 반드시 사용해야 한다.

```
<% Html.RenderPartial("DinnerForm"); %>
```

이는 <% %> 블록이 코드 구문을 가지고 있으며 C# 코드 구문은 반드시 세미 콜론으로 끝을 맺어야 하기 때문이다.

=====

부분 뷰 템플릿으로 깔끔한 코드 유지하기

앞서 우리는 여러 곳에서 동일한 코드를 중복적으로 사용하는 것을 방지하기 위해 "DinnerForm" 부분 뷰 템플릿을 구현하며 대부분의 경우 이런 목적으로 부분 뷰 템플릿을 생성한다.

그러나 어느 한 곳에서만 사용되는 부분 뷰를 만들어야 할 때도 있다. 뷰 템플릿의 코드가 매우 복잡할 때는 해당 뷰 템플릿을 하나 이상의 부분 뷰 템플릿으로 나누면 가독성을 향상시킬 수 있다.

예를 들어 우리의 프로젝트에 포함된 Site.master 파일에 사용된 다음의 코드를 살펴보자 (마스터 페이지에 대해서는 잠시 후에 다시 설명하겠다). 이 파일에 사용된 코드는 상대적으로 직관적이어서 읽기가 쉬운데 그 이유는 우측 상단의 로그인/로그아웃 링크를 생성하는 코드가 "LogOnUserControl" 부분 뷰로 캡슐화 되어 있기 때문이다.

```
<div id="header">
<div id="title">
<h1>My MVC Application</h1>
</div>
<div id="logindisplay">
<% Html.RenderPartial("LogOnUserControl"); %>
</div>
<div id="menucontainer">
<ul id="menu">
<li><%= Html.ActionLink("Home", "Index", "Home") %></li>
<li><%= Html.ActionLink("About", "About", "Home") %></li>
</ul>
</div>
</div>
```

만일 뷰 템플릿에 뒤섞인 코드와 HTML 마크업이 읽기에 혼란스럽다고 느낀다면 코드의 일부를 부분 뷰로 분리할 경우 코드가 보다 명확해질 수 있는지 고려해 볼 필요가 있다.

마스터 페이지

ASP.NET MVC는 앞서 설명한 부분 뷰 뿐만 아니라 사이트의 일반적인 레이아웃과 최상위 수준의 HTML 마크업을 정의하는데 사용되는 "마스터 페이지" 템플릿도 지원한다. 마스터 페이지에는 ContentPlaceHolder 컨트롤을 추가하여 뷰에 의해 "채워지는" 영역을 재정의할 수도 있다. 마스터 페이지를 이용하면 사이트 전체에 일관적인 레이아웃을 매우 효율적으로 (그리고 DRY틱하게) 적용할 수 있다.

기본적으로 새로 생성된 ASP.NET MVC 프로젝트에는 마스터 페이지 템플릿이 자동적으로 추가된다. 이 마스터 페이지의 이름은 "Site.master"이며 #Views#Shared# 폴더에 존재한다.

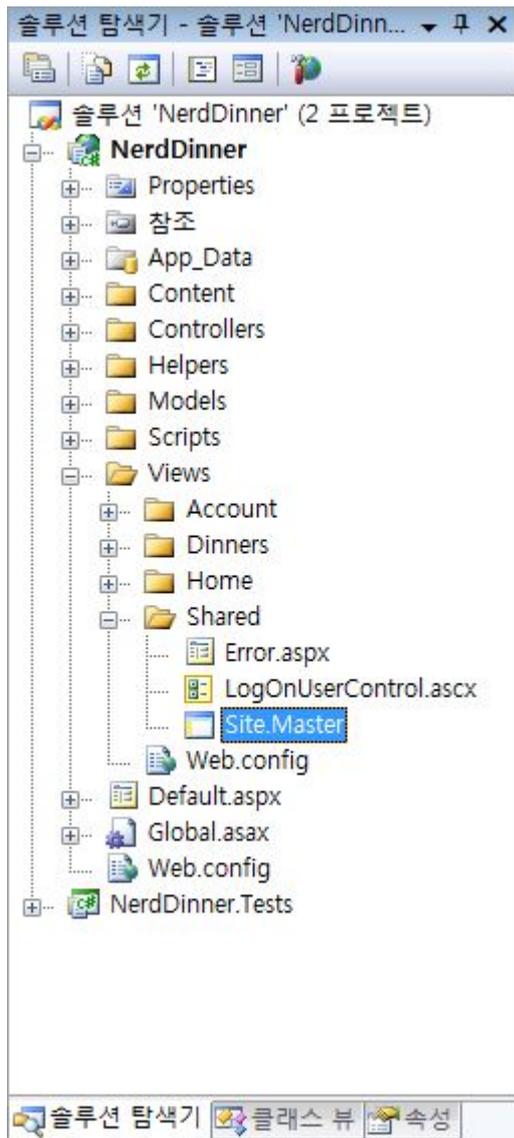


그림 1-105

기본적으로 추가된 Site.master 파일에 사용된 코드는 아래와 같다. 이 파일은 사이트의 콘텐츠를 둘러싸는 HTML 코드를 정의하고 있으며 상단에 페이지 이동을 위한 메뉴도 제공한다. 또한 제목이 표시될 영역과 페이지의 콘텐츠가 표시될 영역을 표시하는 두 개의 ContentPlaceHolder 컨트롤을 가지고 있다.

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
<link href="../../../Content/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div class="page">
<div id="header">
```

```

<div id="title">
<h1>My MVC Application</h1>
</div>
<div id="logindisplay">
<% Html.RenderPartial("LogOnUserControl"); %>
</div>
<div id="menucontainer">
<ul id="menu">
<li><%= Html.ActionLink("Home", "Index", "Home") %></li>
<li><%= Html.ActionLink("About", "About", "Home") %></li>
</ul>
</div>
</div>
<div id="main">
<asp:ContentPlaceHolder ID="MainContent" runat="server" />
</div>
</div>
</body>
</html>

```

우리가 NerdDinner 애플리케이션을 위해 생성한 모든 뷰 템플릿("List", "Details", "Edit", "Create", "NotFound" 등)들은 이 Site.master 템플릿을 사용한다. "Add View" 대화 상자에서 뷰 템플릿을 생성할 때 각 뷰 템플릿의 상단에 위치한 <% @Page %> 지시문의 "MasterPageFile" 특성에는 기본적으로 이 파일이 지정된다.

```

<%@ Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerViewModel>"
MasterPageFile="~/Views/Shared/Site.Master" %>

```

따라서 Site.master 페이지의 콘텐츠는 우리가 임의로 변경할 수 있으며 우리가 만든 뷰 템플릿이 렌더링될 때 콘텐츠가 변경되고 자동으로 적용된다.

그러면 Site.master 페이지의 제목 부분을 변경하여 "My MVC Application" 대신 우리가 구현하고 있는 예제 애플리케이션의 제목인 "NerdDinner" 문자열이 나타나도록 변경해보자. 또한 페이지 이동을 위한 메뉴들 중 첫 번째 탭은 "Find Dinner" (HomeController의 Index() 메서드에 의해 처리된다.)로 변경하고 "Host Dinner" (DinnersController의 Create() 메서드에 의해 처리된다.) 탭을 새로 추가하자.

```

<div id="header">
<div id="title">
<h1>NerdDinner</h1>
</div>
<div id="logindisplay">
<% Html.RenderPartial("LogOnUserControl"); %>
</div>
<div id="menucontainer">
<ul id="menu">
<li><%= Html.ActionLink("Find Dinner", "Index", "Home") %></li>
<li><%= Html.ActionLink("Host Dinner", "Create", "Dinners") %></li>
<li><%= Html.ActionLink("About", "About", "Home") %></li>
</ul>
</div>

```

</div>

Site.master 파일을 저장하고 브라우저를 새로 고침해보면 수정된 제목과 페이지 이동 탭이 애플리케이션 전체에 적용된다. 예를 들어 /Dinners/ URL은 아래와 같은 화면을 보여준다.

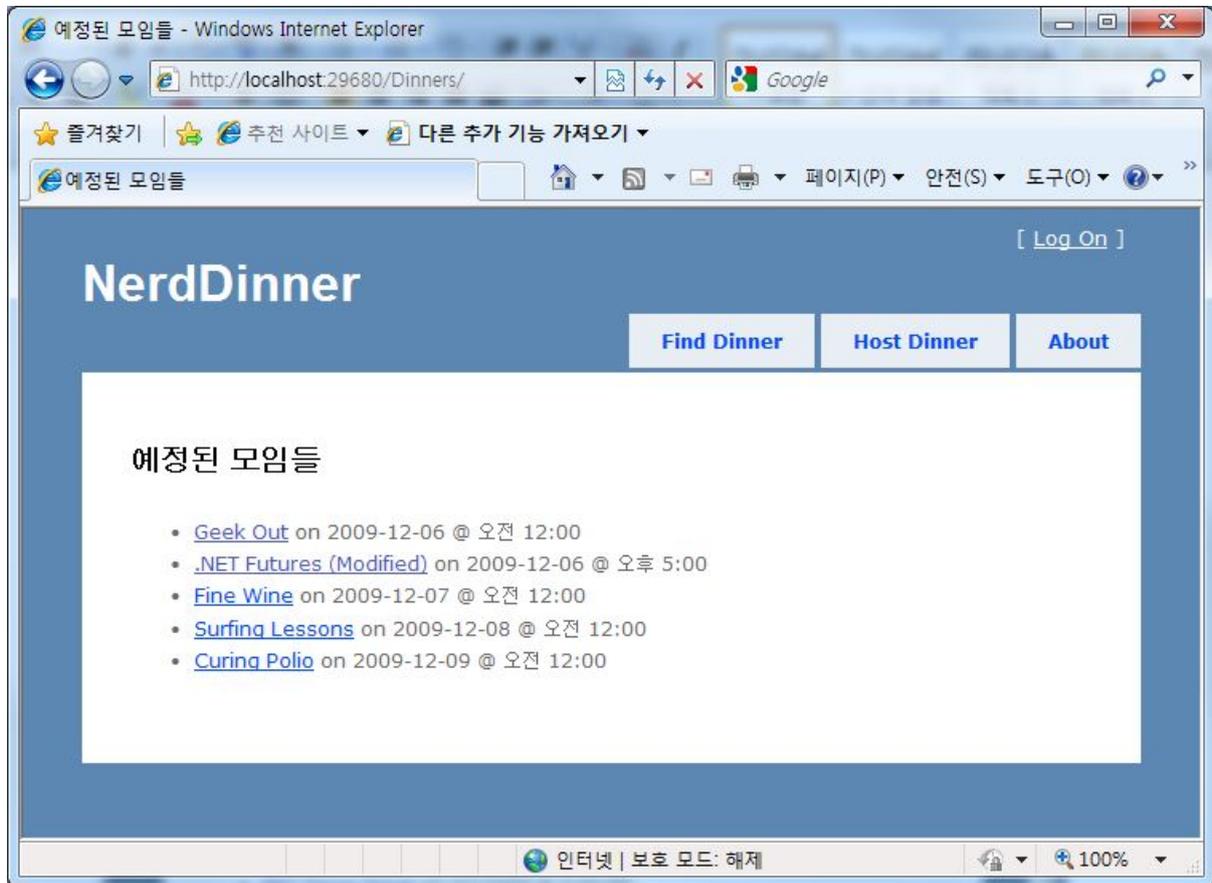


그림 1-106

/Dinners/Edit/[id] URL의 경우에는 다음과 같은 화면을 보여준다.

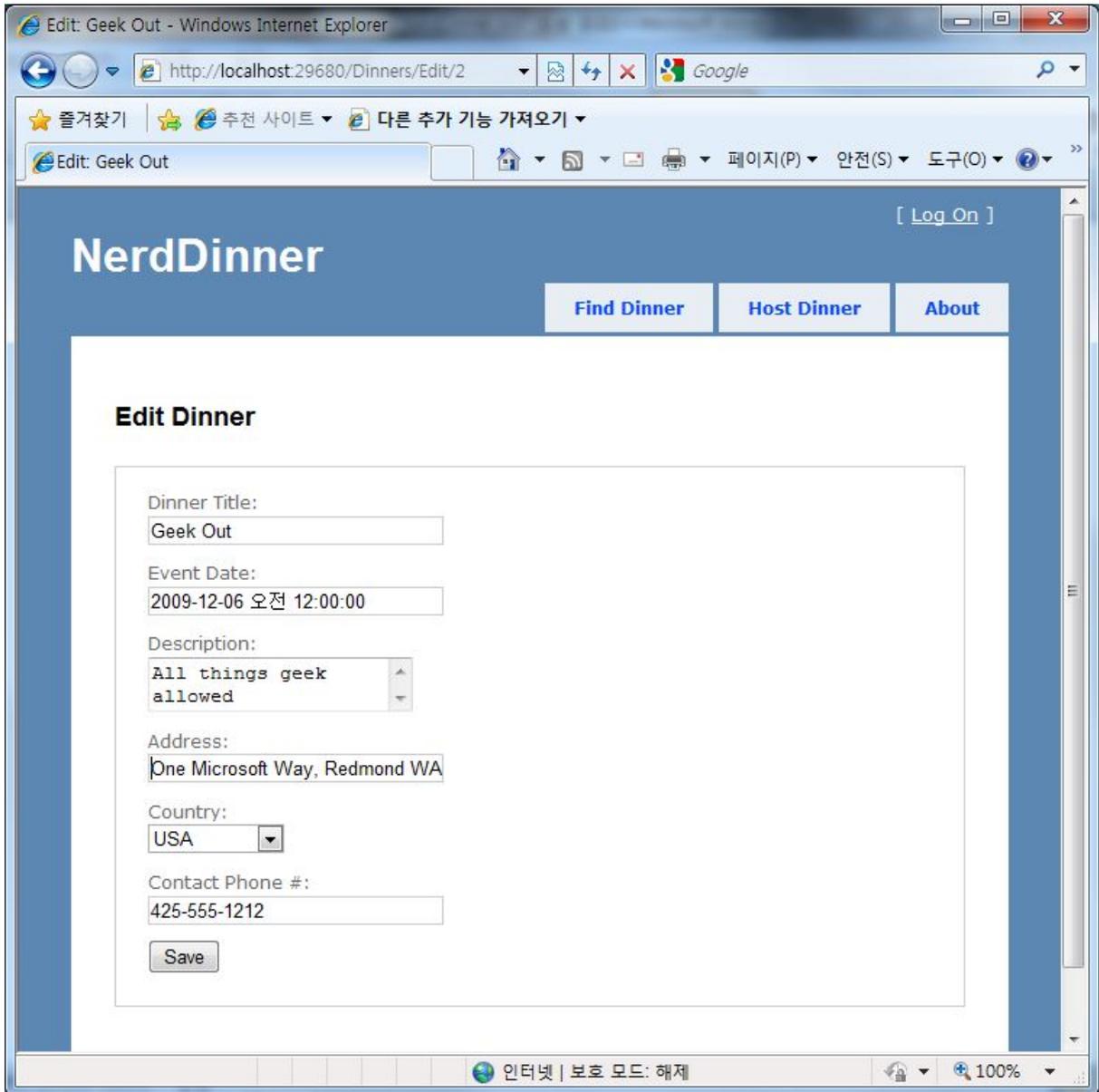


그림 1-107

이처럼 부분 뷰와 마스터 페이지는 뷰를 명확하게 정리할 수 있는 매우 유연한 옵션이다. 뷰의 콘텐츠와 코드에 대한 중복을 방지할 수 있으면서 보다 읽기 쉬운 뷰 템플릿을 편리하게 유지 보수할 수 있다.

페이징 기능 구현하기

만일 이 사이트가 성공적으로 서비스된다면 수천개의 예정된 모임 데이터가 쌓일 것이다. 따라서 이렇게 축적되는 데이터를 제대로 표시하며 사용자가 데이터를 탐색할 수 있도록 UI를 구성해야 한다. 그러기 위해서는 /Dinners URL에서 한 번에 1천개의 데이터를 보여주는 대신 페이징 기능을 추가하여 한 번에 10개의 데이터만을 보여주도록 하고 사용자가 [이전] 혹은 [다음] 링크를 클릭하여 SEO에 최적화된 방법으로 전체 리스트를 탐색할 수 있도록 구현할 것이다.

Index() 액션 메서드 다시 살펴보기

우리가 구현한 DinnersController 클래스의 Index() 액션 메서드는 현재 다음과 같은 코드로 구현되어 있다.

```
//  
// GET: /Dinners/  
public ActionResult Index() {  
    var dinners = dinnerRepository.FindUpcomingDinners().ToList();  
    return View(dinners);  
}
```

/Dinners URL에 대한 요청이 들어오면 이 액션 메서드는 모든 예정된 모임 데이터를 조회하여 한꺼번에 출력한다.

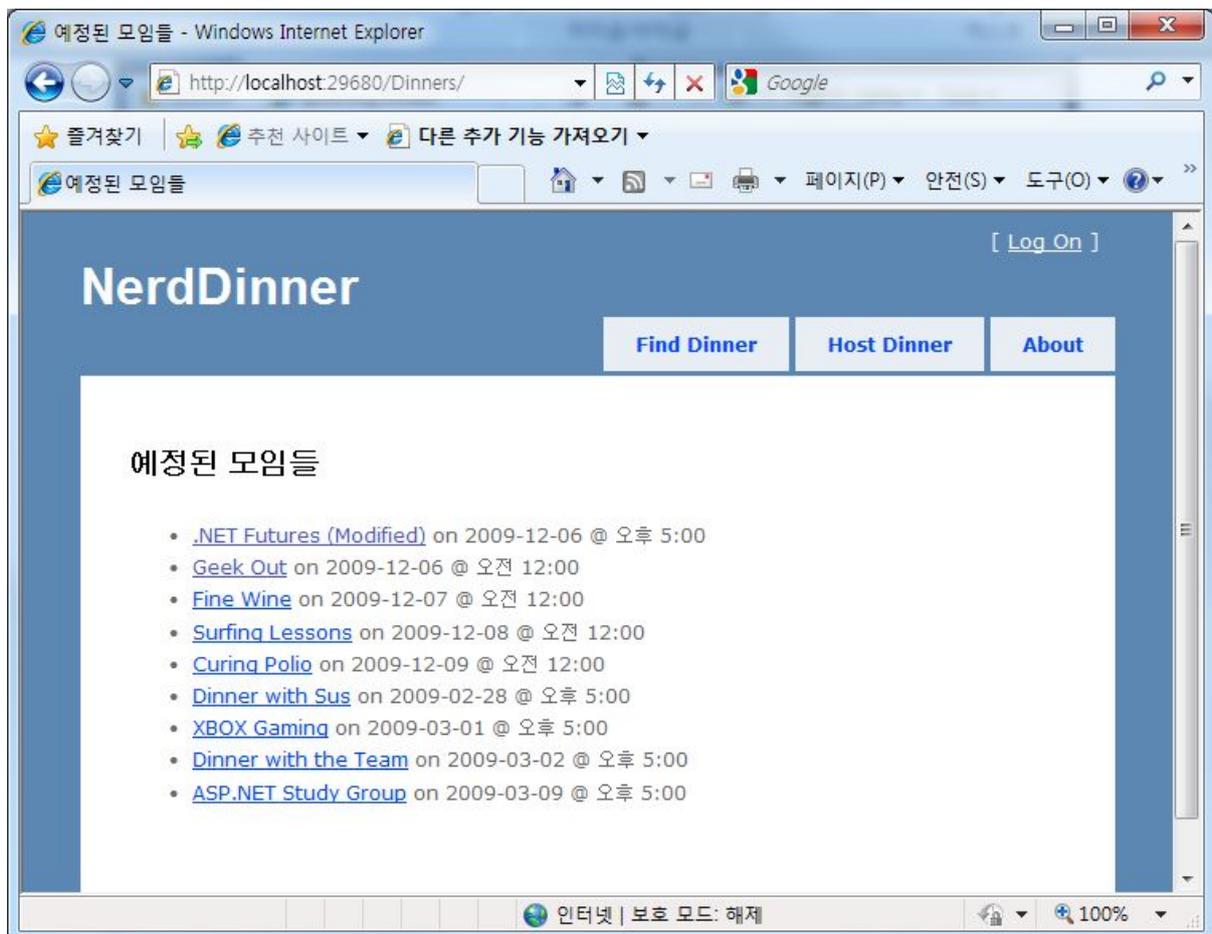


그림 1-08

IQueryable<T> 인터페이스 이해하기

IQueryable<T> 인터페이스는 .NET 3.5에 포함된 LINQ를 위해 추가된 인터페이스이다. 이 인터페

이스는 “지연된 실행(Deferred Execution)” 시나리오를 지원하기 위한 강력한 기능을 제공하며 우리가 페이징 기능을 구현하는데 활용될 수 있다.

DinnerRepository 클래스의 FindUpcomingDinners() 메서드는 다음과 같이 IQueryable<Dinner> 객체들을 리턴하도록 구현되어 있다.

```
public class DinnerRepository {
    private NerdDinnerDataContext db = new NerdDinnerDataContext();
    //
    // Query Methods
    public IQueryable<Dinner> FindUpcomingDinners() {
        return from dinner in db.Dinners
            where dinner.EventDate > DateTime.Now
            orderby dinner.EventDate
            select dinner;
    }
}
```

FindUpcomingDinners() 메서드가 리턴하는 IQueryable<Dinner> 객체는 LINQ to SQL을 이용하여 Dinners 객체를 데이터베이스에서 조회하는 쿼리를 캡슐화하고 있다. 중요한 것은 우리가 실제로 쿼리 내의 데이터에 액세스하거나 혹은 ToList() 메서드를 호출하기 전까지는 데이터베이스에 실제로 쿼리를 실행하지 않는다는 점이다. FindUpcomingDinners() 메서드를 호출하는 코드는 실제 쿼리가 데이터베이스에서 실행되기 전에 IQueryable<Dinner> 객체에 “연결된(Chained)” 작업이나 필터를 선택적으로 적용할 수 있다. LINQ to SQL은 충분히 뚝뚝해서 조합된 쿼리를 데이터베이스에 실행해서 필요한 데이터만을 추출해낸다.

페이징 로직을 구현하기 위해서는 Index() 액션 메서드를 다음과 같이 수정해서 ToList() 메서드를 호출하기 전에 “Skip”과 “Take” 연산자를 적용하여 IQueryable<Dinner> 객체의 리스트를 얻어와야 한다.

```
//
// GET: /Dinners/
public ActionResult Index() {
    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
    var paginatedDinners = upcomingDinners.Skip(10).Take(20).ToList();
    return View(paginatedDinners);
}
```

위의 코드는 데이터베이스에 저장된 데이터 중 처음 10개의 데이터를 건너뛰고 그 이후의 20개의 데이터를 가져온다. 똑똑한 LINQ to SQL은 SQL 불필요한 데이터를 건너뛰도록 최적화된 쿼리를 구성하여 웹서버가 아닌 데이터베이스에서 실행한다. 즉 우리가 수십만개의 데이터가 축적된 데이터베이스에 대해 작업을 수행하더라도 우리가 원하는 10개의 데이터만 조회된다는 뜻이다 (따라서 매우 효율적이며 확장 가능하다).

URL에 “page” 변수 추가하기

페이지의 범위를 코드를 이용하여 지정하는 것보다는 아무래도 “Page” 변수를 URL에 지정하여 사용자가 요구하는 범위에 해당하는 Dinners 객체들을 보여주는 것이 훨씬 좋을 것이다.

쿼리 문자열 활용하기

다음의 코드는 /Dinners?pag=2와 같은 형식의 URL을 지원할 수 있도록 Index() 액션 메서드를 수정한 코드이다.

```
//  
// GET: /Dinners/  
// /Dinners?page=2  
public ActionResult Index(int? page) {  
    const int pageSize = 10;  
    var upcomingDinners = dinnerRepository.FindUpcomingDinners();  
    var paginatedDinners = upcomingDinners.Skip((page ?? 0) * pageSize)  
        .Take(pageSize)  
        .ToList();  
    return View(paginatedDinners);  
}
```

위의 수정된 Index() 액션 메서드는 “page”라는 이름의 매개 변수를 사용하며 이 매개 변수는 int 타입의 Nullable 타입을 사용하고 있다. 따라서 /Dinners?page=2와 같은 URL이 요청되면 이 매개 변수에 “2”라는 값이 전달되며 /Dinners 처럼 page 변수 값이 지정되지 않은 경우에는 page 변수에 null이 전달된다.

page 변수의 값에 페이지의 크기 (Page Size, 예제의 경우에는 10개)를 곱하면 한 페이지에 보여 줄 Dinners 객체의 개수를 결정할 수 있다. Nullable 타입을 다룰 때는 C#의 널 검사 연산자 (?? 연산자)를 사용하면 편리하다. 위의 코드에서는 page 변수의 값이 null이면 기본 값으로 0을 사용하도록 구현되어 있다.

URL에 포함된 값 활용하기

앞서와 같이 쿼리 문자열을 사용하는 방법 외에도 /Dinners/Page/2 나 /Dinners/2와 같이 실제 URL에 변수의 값을 포함하여 전달할 수도 있다. ASP.NET MVC에 포함된 강력한 URL 라우팅 엔진은 이와 같은 URL을 손쉽게 만들어 낼 수 있다.

사용자 정의 라우팅 규칙을 추가로 등록하면 요청되는 URL이나 URL의 형식에 따라 적절한 컨트롤러 클래스의 액션 메서드를 호출할 수 있다. 그러면 새로운 라우팅 규칙을 등록하기 위해서 Global.asax 파일을 열어보자.

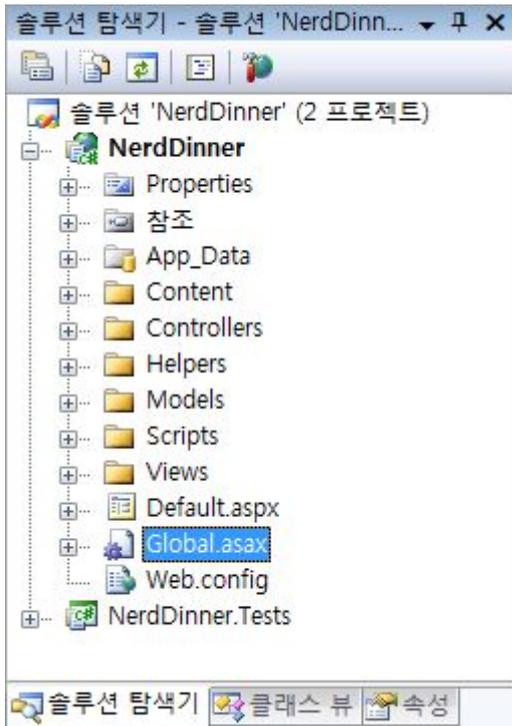


그림 1-109

Global.asax 파일에는 이미 route.MapRoute() 메서드를 이용하여 라우팅 규칙을 등록하는 코드가 작성되어 있다. MapRoute() 메서드를 이용하여 다음과 같이 새로운 라우팅 규칙을 등록하자.

```
public void RegisterRoutes(RouteCollection routes) {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "UpcomingDinners",
        "Dinners/Page/{page}",
        new { controller = "Dinners", action = "Index" }
    );
    routes.MapRoute(
        "Default", // 라우팅 규칙의 이름
        "{controller}/{action}/{id}", // URL 형식
        new { controller="Home", action="Index", id="" } // 매개 변수의 기본 값
    );
}
void Application_Start() {
    RegisterRoutes(RouteTable.Routes);
}
```

위의 코드에서는 "UpcomingDinners"라는 이름의 새로운 라우팅 규칙을 등록하고 있다. 이 라우팅 규칙은 "/Dinners/Page/{page}"와 같은 형식을 사용하며 여기서 {page} 키워드가 바로 URL에 포함될 page 변수 값이 위치할 곳을 의미한다. MapRoute() 메서드의 세 번째 매개 변수는 이 형식과 일치하는 URL 요청을 DinnersController 컨트롤러의 Index() 액션 메서드로 처리하도록 지정하고 있다.

이 경우 앞서 쿼리 문자열을 사용할 때와 동일한 Index() 메서드를 사용하면 되며 page 매개 변수의 값은 쿼리 문자열이 아니라 URL에서 얻어오게 된다.

```
//
// GET: /Dinners/
// /Dinners/Page/2
public ActionResult Index(int? page) {
    const int pageSize = 10;
    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
    var paginatedDinners = upcomingDinners.Skip((page ?? 0) * pageSize)
        .Take(pageSize)
        .ToList();
    return View(paginatedDinners);
}
```

이제 애플리케이션을 실행하고 /Dinners URL을 요청하면 다음 그림과 같이 10개의 데이터를 보게 된다.

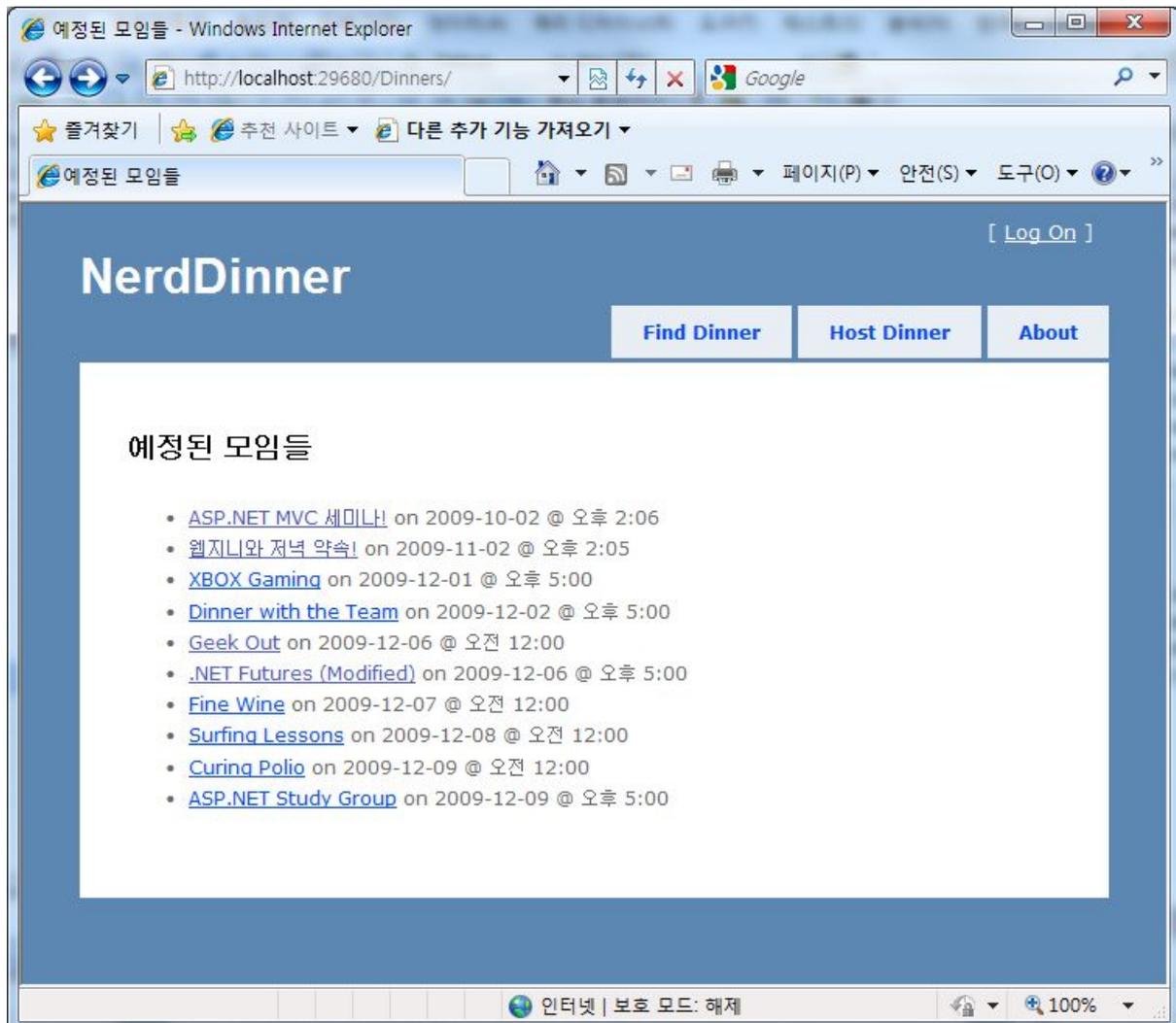


그림 1-110

그리고 /Dinners/Page/1 URL을 요청하면 다음 그림과 같이 다음 페이지의 데이터를 보게 될 것이다.

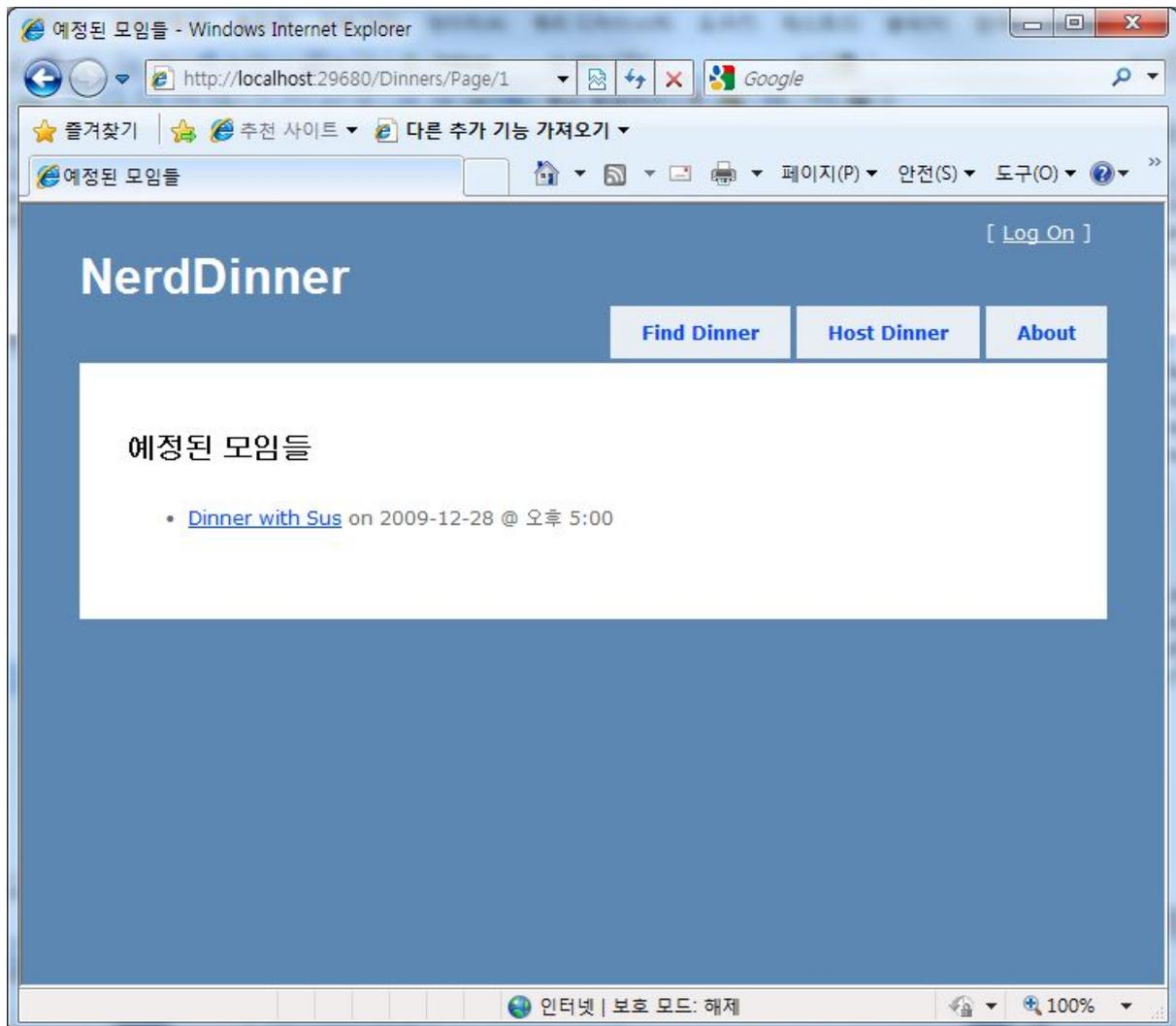


그림 1-111

페이지 이동을 위한 UI 추가하기

페이징 기능을 추가하기 위한 마지막 단계는 뷰 템플릿에 “이전” 페이지로 이동하기 위한 버튼과 “다음” 페이지로 이동하기 위한 버튼을 추가하여 사용자가 페이지를 마음대로 이동할 수 있도록 하는 것이다.

이 기능을 올바르게 구현하려면 현재 데이터베이스에 몇 개의 데이터가 존재하는지와 한 페이지에 몇 개의 데이터를 보여줄 것인지를 우선 결정해야 한다. 그런 후에는 현재 보여지는 페이지의 데이터가 첫 페이지의 데이터이거나 마지막 페이지의 데이터인지를 확인하여 그에 따라 “이전” 버튼이나 “다음” 버튼을 보이거나 보이지 않도록 해야 한다. 이 로직을 구현하기 위한 코드 역시 `Index()` 메서드에 구현하면 되지만 이 로직을 구현하는 재사용 가능한 새로운 클래스를 프로젝트

에 추가하는 방법도 나쁘지 않다.

다음의 코드는 .NET 프레임워크에 내장된 List<T> 클래스를 상속하여 "PaginatedList" 클래스를 간단하게 구현해 본 코드이며 IQueryable 인터페이스가 가진 데이터들을 페이징 할 수 있는 재사용 가능한 컬렉션 클래스를 구현하고 있다. NerdDinner 애플리케이션의 경우 IQueryable<Dinner> 타입을 주로 사용하지만 PaginatedList 클래스는 IQueryable<Product>나 IQueryable<Customer>와 같은 타입에도 얼마든지 활용이 가능하다.

```
public class PaginatedList<T> : List<T> {
    public int PageIndex { get; private set; }
    public int PageSize { get; private set; }
    public int TotalCount { get; private set; }
    public int TotalPages { get; private set; }
    public PaginatedList(IQueryable<T> source, int pageIndex, int pageSize) {
        PageIndex = pageIndex;
        PageSize = pageSize;
        TotalCount = source.Count();
        TotalPages = (int) Math.Ceiling(TotalCount / (double) PageSize);
        this.AddRange(source.Skip(PageIndex * PageSize).Take(PageSize));
    }
    public bool HasPreviousPage {
        get {
            return (PageIndex > 0);
        }
    }
    public bool HasNextPage {
        get {
            return (PageIndex+1 < TotalPages);
        }
    }
}
```

위의 코드에서 알 수 있듯이 PaginatedList 클래스는 "PageIndex"나 "PageSize", "TotalCount", "TotalPages" 등의 속성 값을 계산해 주며 현재 페이지가 이전 혹은 다음 페이지로 이동할 수 있는지 여부를 표시하기 위한 "HasPreviousPage"와 "HasNextPage"와 같은 속성들도 제공한다. 위의 코드는 두 개의 SQL 쿼리를 사용하는데 첫 번째 쿼리는 전체 Dinners 객체의 개수를 얻어오기 위한 것이며 (이 쿼리는 SELECT COUNT 구문을 사용하기 때문에 객체들을 리턴하는 대신 전체 데이터의 개수를 정수형으로 리턴한다.) 두 번째 쿼리는 데이터베이스에 저장된 객체들 중 우리가 필요로 하는 데이터만을 골라낸다.

이제 DinnersController.Index() 액션 메서드를 수정해서 DinnerRepository.FindUpcomingDinners() 메서드가 리턴하는 결과로부터 PaginatedList<Dinner> 컬렉션을 생성하여 뷰 템플릿에 전달하도록 다음과 같이 수정해야 한다.

```
// GET: /Dinners/
// /Dinners/Page/2
public ActionResult Index(int? page) {
    const int pageSize = 10;
    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
```

```

var paginatedDinners = new PaginatedList<Dinner>(upcomingDinners,
page ?? 0,
pageSize);
return View(paginatedDinners);
}

```

다음으로 `Views\Dinners\Index.aspx` 뷰 템플릿을 수정하여 이 템플릿이 `ViewPage<IEnumerable<Dinner>>` 타입 대신 `ViewPage<NerdDinner.Helpers.PaginatedList<Dinner>>` 타입을 사용하도록 수정한 후 뷰 템플릿의 하단에 이전 및 다음 버튼을 보여주도록 다음과 같이 수정한다.

```

<% if (Model.HasPreviousPage) { %>
<%= Html.RouteLink("<<<",
"UpcomingDinners",
new { page=(Model.PageIndex-1) }) %>
<% } %>
<% if (Model.HasNextPage) { %>
<%= Html.RouteLink(">>>",
"UpcomingDinners",
new { page = (Model.PageIndex + 1) }) %>
<% } %>

```

위의 예제 코드를 보면 `Html.RouteLink()` 메서드를 이용해서 하이퍼링크를 생성하는 방법을 알 수 있다. 이 메서드는 우리가 지금까지 사용해왔던 `Html.ActionLink()` 메서드와 유사하지만 한 가지 차이점은 우리가 `Global.asax` 파일에서 정의했던 "UpcomingDinners"라는 이름의 라우팅 규칙을 이용하여 링크를 생성한다는 점이다. 이는 `/Dinners/Page/{page}` 형식의 URL을 이용하여 `Index()` 액션 메서드를 호출하겠다는 것을 의미한다. 이 때 `{page}` 키워드의 값은 우리가 `PageIndex` 속성을 이용해 제공한 값이 지정된다.

이제 애플리케이션을 다시 실행하면 다음 그림과 같이 10개의 데이터가 출력되는 것을 볼 수 있다.

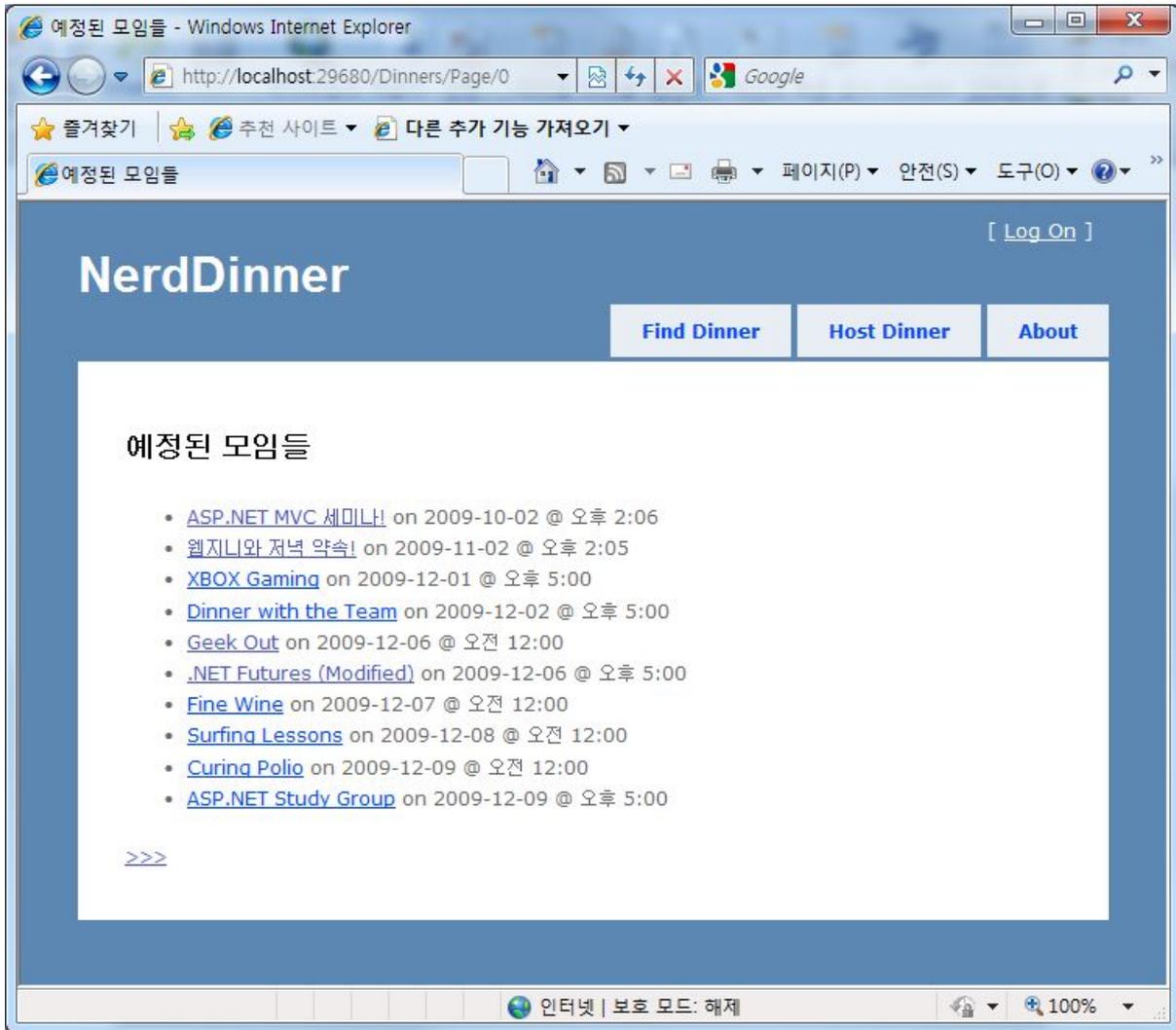


그림 1-112

위 그림에서 알 수 있듯이 페이지의 하단에는 이전 혹은 다음 페이지로 이동하기 위한 <<< 혹은 >>> 로 표시되는 페이징 UI가 존재하며 검색 엔진에 최적화된 형태의 URL을 이용하여 이전 혹은 다음 페이지로 이동할 수 있는 링크를 제공한다.

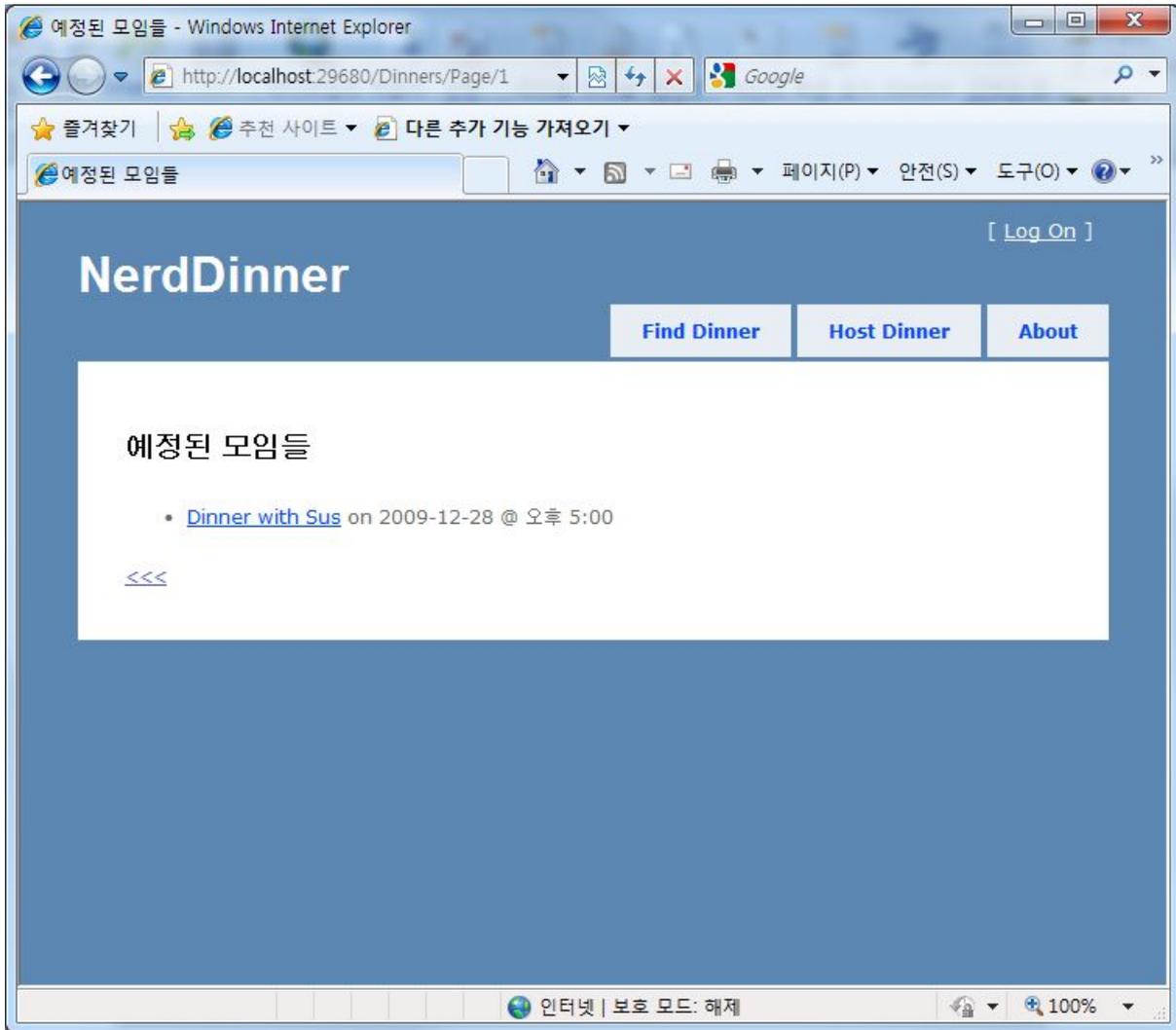


그림 1-113

=====

IQueryable<T> 인터페이스의 구현에 대하여

IQueryable<T>는 (페이징이나 쿼리의 조합 등) 다양한 형태의 지연된 실행 (Deferred Execution) 을 구현한 매우 강력한 기능이다. 그러나 다른 강력한 기능들과 마찬가지로 IQueryable<T> 인터페이스를 사용할 때는 오용하지 않도록 주의를 기울여야 한다.

IQueryable<T> 인터페이스를 구현한 객체를 리턴한다는 것은 리턴된 객체에 다른 추가적인 작업을 덧붙일 수 있으며 그것이 최종적인 쿼리로 변환되어 저장소에서 실행된다는 것을 반드시 인지하고 있어야 한다. 만일 다른 코드를 통해 추가적인 작업을 덧붙일 수 없도록 하려면 이미 실행된 결과 집합만을 가지고 있는 IList<T>나 List<T> 타입 혹은 IEnumerable<T> 타입을 리턴해야 한다.

페이징 기능의 경우에는 저장소의 메서드를 호출할 때 추가적인 작업 없이 데이터의 페이징 로직

을 실행해야 한다. 그러기 위해서는 FindUpcomingDinners() 메서드가 PaginatedList 타입을 리턴하도록 다음과 같이 수정해야 한다.

```
PaginatedList<Dinner> FindUpcomingDinners(int pageIndex, int pageSize) { }
```

또는 다음과 같이 IList<Dinner> 타입을 리턴하도록 해야 하며 "totalCount" 값은 out 매개 변수로 지정하여 데이터의 총 개수를 리턴할 수 있어야 한다.

```
IList<Dinner> FindUpcomingDinners(int pageIndex, int pageSize, out int totalCount) { }
```

=====

인증과 권한 설정

이제 NerdDinner 애플리케이션에 방문한 사람은 누구라도 모임 데이터를 생성하고 상세 정보를 수정할 수 있게 되었다. 이제는 새로운 데이터를 생성하기 위해서는 회원 가입을 해야 하며 데이터의 수정은 데이터를 생성한 사용자만 가능하도록 애플리케이션을 수정해보자.

이와 같이 애플리케이션의 보안을 강화하려면 인증과 권한을 적용해야 한다.

인증과 권한 부여

인증(Authentication)이란 애플리케이션에 액세스하는 사용자의 신원을 인식하고 확인하는 과정을 말한다. 더 간단히 말하자면 사용자가 웹사이트에 접근할 때 "누구"인지를 확인하는 작업인 것이다.

ASP.NET은 브라우저 사용자를 확인하는 여러 가지 방법을 제공한다. 인터넷 웹 애플리케이션의 경우 가장 보편적으로 사용되는 방법은 "폼 인증(Forms Authentication)" 방법이다. 폼 인증이란 사용자가 로그인할 수 있는 HTML 페이지를 제공하고 사용자가 입력한 사용자의 ID와 비밀번호를 데이터베이스나 혹은 다른 비밀번호 저장소를 이용하여 확인하는 방법이다. 사용자의 ID와 비밀번호가 확인된다면 이후의 요청에 대해 사용자를 인식할 수 있도록 암호화된 HTTP 쿠키를 생성할 수 있다. NerdDinner 애플리케이션에서는 바로 이 폼 인증 방식을 이용하여 인증을 구현한다.

권한 부여(Authorization)란 인증된 사용자가 특정 URL이나 리소스에 대한 접근 혹은 어떤 동작을 실행할 수 있는지 여부를 판단하는 과정을 말한다. 예를 들어 NerdDinner 애플리케이션은 반드시 인증을 받은 사용자만이 /Dinners/Create URL에 접근하여 새로운 모임을 주최할 수 있으며 모임을 주최한 사람만이 모임의 상세 정보를 수정할 수 있도록 구현할 예정이다.

폼 인증과 AccountController 클래스

Visual Studio에 추가된 ASP.NET MVC 프로젝트 템플릿은 새로운 ASP.NET MVC 프로젝트를 생성할 때 폼 인증을 미리 구현해 주며 로그인 페이지까지 자동으로 생성해 주어 사이트에 손쉽게 보안을 적용할 수 있다.

기본적으로 사용되는 Site.master 페이지는 인증되지 않은 사용자가 방문한 경우 우측 상단에 "Log On" 링크를 제공한다.

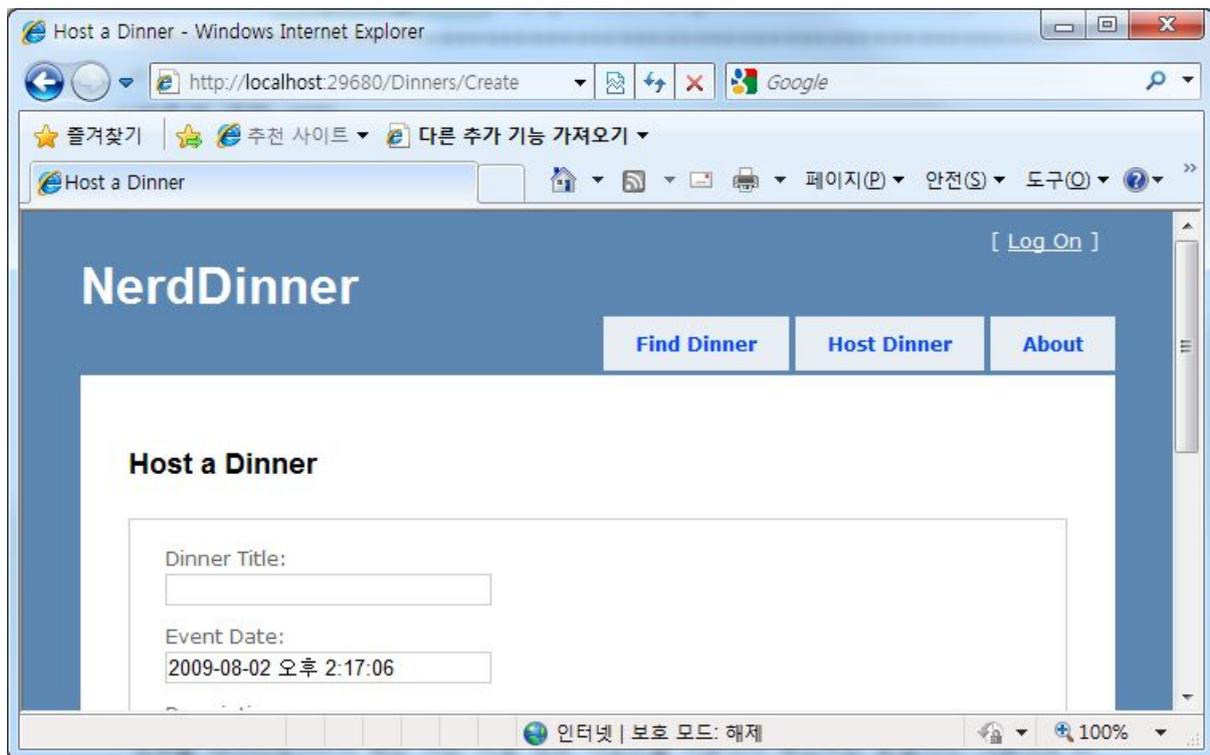


그림 1-114

"Log On" 링크를 클릭하면 사용자는 /Account/LogOn URL로 이동하게 된다.

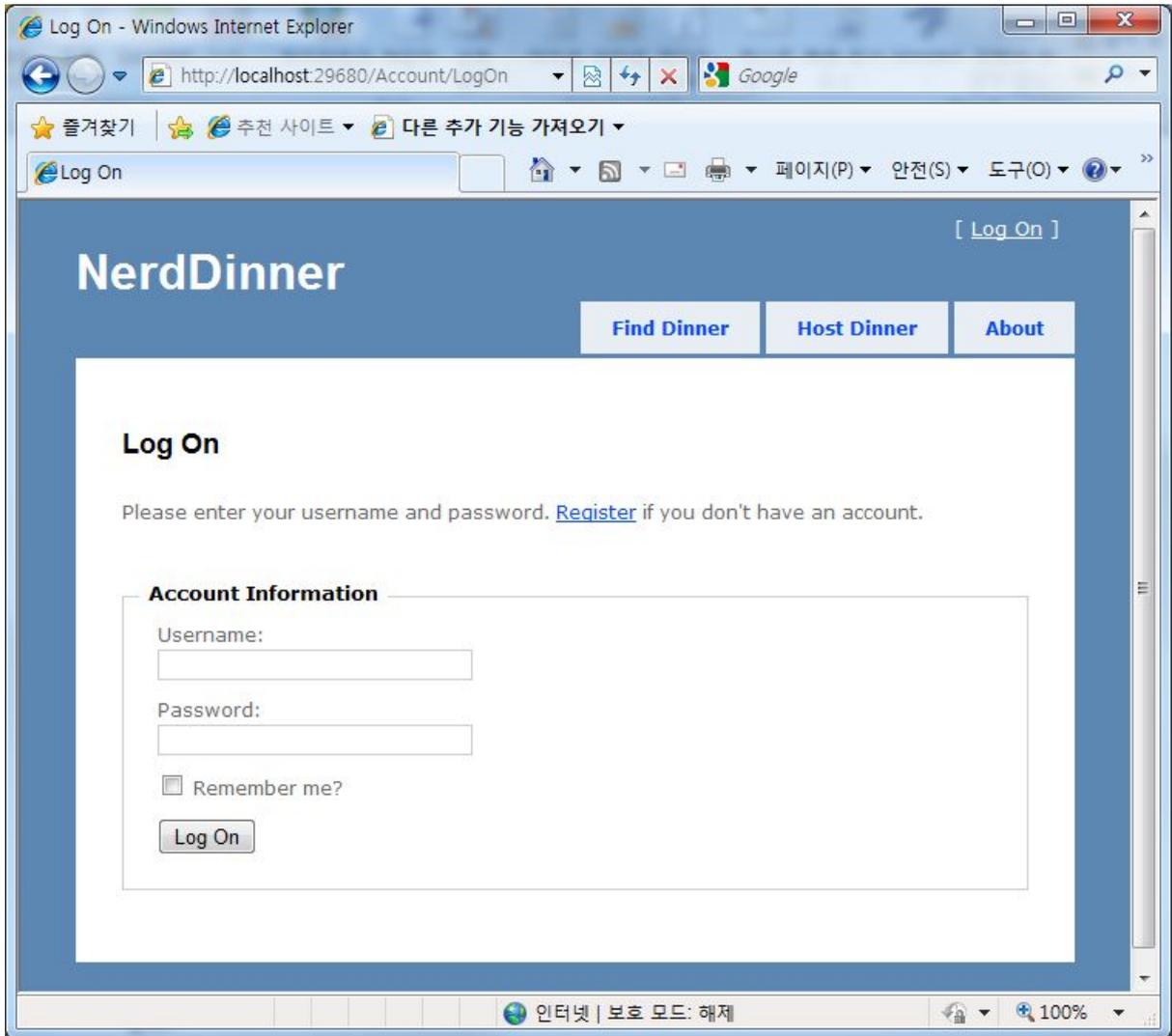


그림 1-115

아직 사이트에 등록되지 않은 사용자라면 "Register" 링크를 클릭하여 새로운 계정을 등록할 수 있다. 이 링크를 클릭한 사용자는 계정 정보를 입력할 수 있는 /Account/Register URL로 이동하게 된다.

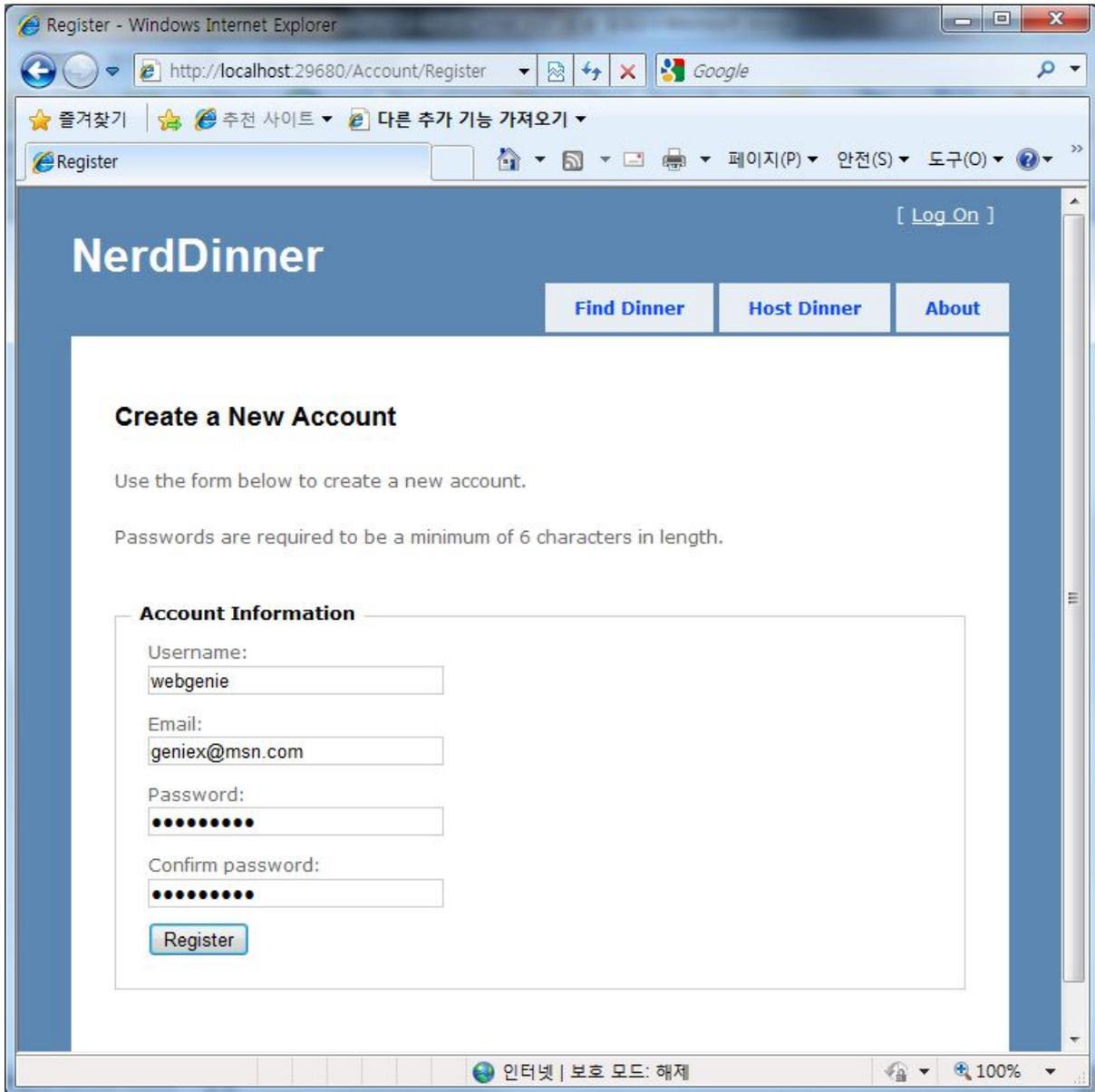


그림 1-116

“Register” 버튼을 클릭하면 새로운 사용자가 ASPNET 멤버십 시스템(Membership System)에 등록되며 사용자는 폼 인증 방식을 이용하여 웹사이트에 로그인 한 상태가 된다.

사용자가 로그인하면 Site.master 페이지의 우측 상단에는 “Welcome [사용자이름]!” 메시지가 나타나며 “Log On” 링크 대신 “Log Off” 링크가 나타나게 된다. 이 “Log Off” 링크를 클릭하면 사용자가 로그아웃 하게 된다.



그림 1-117

지금까지 살펴 본 로그인, 로그아웃 그리고 계정 생성 등의 기능은 Visual Studio가 프로젝트를 생성할 때 추가한 AccountController 클래스에 구현되어 있으며 AccountController 클래스가 사용하는 UI는 `Views\Account` 디렉터리의 뷰 템플릿들을 이용하여 구현되어 있다.

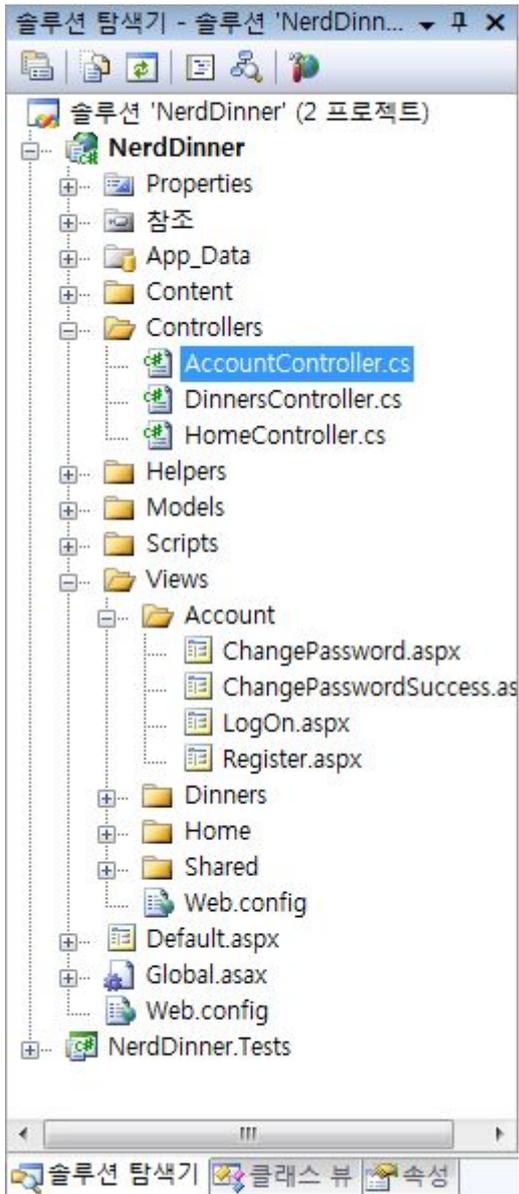


그림 1-118

AccountController 클래스는 ASP.NET의 폼 인증 시스템을 이용하여 인증 쿠키를 발급하며 ASP.NET의 멤버십 API(Membership API)를 사용하여 사용자 계정을 저장하거나 사용자의 ID와 비밀번호를 확인하는 등의 작업을 수행한다. ASP.NET의 멤버십 API는 확장이 가능하여 모든 종류의 비밀번호 저장소를 활용할 수 있다. ASP.NET은 사용자 계정 정보를 SQL 데이터베이스나 액티브 디렉터리 (Active Directory)에 저장할 수 있는 멤버십 제공자(Membership Provider)를 제공하고 있다.

프로젝트의 루트에 위치한 “web.config” 파일을 열어 <membership> 섹션을 이용하면 NerdDinner 애플리케이션이 사용할 멤버십 제공자의 구성을 변경할 수 있다. 프로젝트에 기본적으로 생성되는 web.config 파일에는 “ApplicationServices”라는 이름의 연결 문자열(Connection String)을 사용하는 SQL 멤버십 제공자가 구성되어 있다.

기본적으로 사용되는 "ApplicationServices" 연결 문자열(web.config 파일의 "<connectionStrings>" 섹션에 정의되어 있다)은 SQL Express 제품을 사용하도록 구성되어 있으며 "App_Data" 디렉터리의 "aspnetdb.mdf" 파일을 사용하도록 지정되어 있다. 만일 이 데이터베이스가 존재하지 않는다면 멤버십 API를 처음 사용할 때 ASP.NET이 자동으로 필요한 데이터베이스를 생성한다.

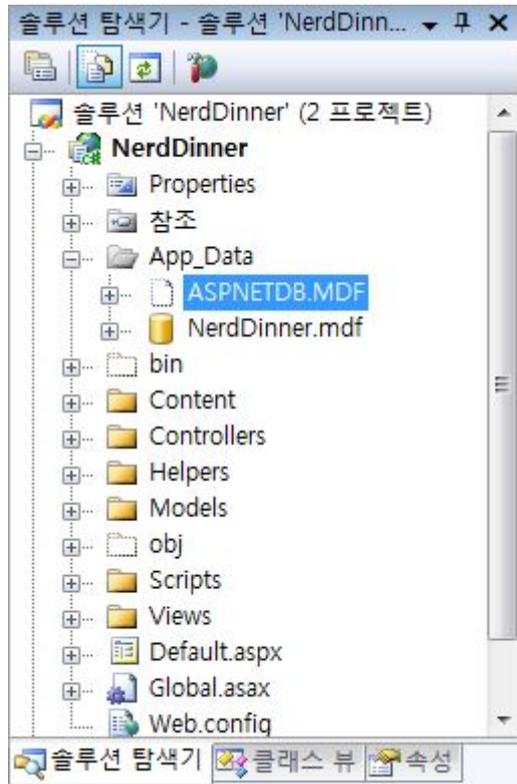


그림 1-119

SQL Express 제품대신 SQL 서버 (Microsoft SQL Server) 제품을 사용하고자 한다면 (혹은 원격지의 데이터베이스 서버에 연결하고자 한다면) "ApplicationServices" 연결 문자열을 수정하고 이 연결 문자열에 의해 연결될 데이터베이스 서버에 멤버십 데이터베이스가 생성되어 있는지 확인만 하면 된다. 만일 멤버십 데이터베이스가 생성되어 있지 않다면 멤버십 API를 비롯한 ASP.NET의 다른 애플리케이션 서비스들(역자 주: 멤버십 서비스 외에도 프로파일(Profile) 서비스와 역할(Role) 서비스가 제공된다)이 사용하는 데이터베이스 스키마를 자동으로 생성해주는 "aspnet_regsql.exe" 도구를 이용하여 데이터베이스를 생성할 수 있다. 이 도구는 #Windows#Microsoft.NET#Framework#v2.0.50727 디렉터리에 존재한다.

[Authorize] 필터로 /Dinners/Create URL에 인증 구현하기

앞서 살펴본 NerdDinner 애플리케이션의 인증 및 사용자 계정 관리 기능을 구현하기 위해 우리가 작성한 코드는 전혀 없음에도 불구하고 사용자들은 애플리케이션에 사용자 계정을 등록하고 로그인 이나 로그아웃을 실행할 수 있다. 이제 권한 부여 기능을 추가하여 사용자의 인증 상태와

ID에 따라 사용자가 어떤 기능을 실행할 수 있는지 없는지를 제어할 것이다.

우선 `DinnersController` 클래스의 `Create()` 액션 메서드에 권한 부여 로직을 추가해보자. 특히 `/Dinners/Create` URL에 접근하기 위해서는 먼저 로그인을 하도록 구현할 것이다. 만일 로그인하지 않은 사용자가 접근하면 로그인 페이지로 이동하도록 하여 사용자에게 먼저 로그인 할 것을 요구하게 된다.

이와 같은 기능을 구현하는 로직은 매우 간단하다. 우선은 아래 코드와 같이 `Create()` 액션 메서드에 `[Authorize]` 액션 필터를 추가한다.

```
//  
// GET: /Dinners/Create  
[Authorize]  
public ActionResult Create() {  
    ...  
}  
//  
// POST: /Dinners/Create  
[AcceptVerbs(HttpVerbs.Post), Authorize]  
public ActionResult Create(Dinner dinnerToCreate) {  
    ...  
}
```

ASP.NET MVC는 재사용 가능한 로직을 액션 메서드에 선언적으로 적용할 수 있는 “액션 필터 (Action Filters)”를 제공한다. `[Authorize]` 액션 필터는 ASP.NET MVC가 기본적으로 제공하는 액션 필터 중 하나로 이 액션 필터를 이용하면 컨트롤러 클래스와 액션 메서드에 권한 부여 로직을 선언적으로 적용할 수 있다.

예제에서와 같이 `[Authorize]` 액션 필터에 아무런 매개 변수를 지정하지 않으면 반드시 로그인한 사용자만 해당 액션 메서드를 실행할 수 있도록 설정할 수 있으며 이 경우 로그인 하지 않은 사용자가 접근하면 자동적으로 로그인 페이지로 이동하게 된다. 로그인 페이지로 이동할 때 사용자가 원래 요청했던 URL은 로그인 페이지의 쿼리 문자열 매개 변수로 전달되며 (예를 들면 `/Account/LogOn?ReturnUrl=%2fDinners%2fCreate`와 같이 전달된다) 사용자가 로그인을 성공적으로 수행하면 `AccountController`는 해당 URL로 사용자를 되돌려 보낸다.

`[Authorize]` 액션 필터는 사용자가 반드시 로그인을 해야 함은 물론 액션 필터가 지정된 액션 메서드를 호출할 수 있는 사용자의 목록이나 사용자가 속한 역할 그룹의 목록을 나열할 수 있는 “Users” 속성과 “Roles” 속성을 제공한다. 예를 들어 다음의 코드는 `/Dinners/Create` URL은 “webgenie”와 “scottgu” 두 명의 사용자만 사용이 가능하도록 지정한 코드이다.

```
[Authorize (Users="scottgu,webgenie")]  
public ActionResult Create() {  
    ...  
}
```

코드 내에 사용자의 ID를 직접 지정하는 것은 유지보수 측면에서 그다지 좋은 방법은 아니다. 더 좋은 방법은 코드가 검사할 "역할 그룹(Role)"을 정의하고 데이터베이스나 액티브 디렉터리를 이용하여 사용자와 역할 그룹을 매핑하는 방법이다 (이렇게 하면 실제 사용자의 매핑 정보를 코드에 포함시키지 않을 수 있다). ASP.NET은 사용자와 역할 그룹을 손쉽게 매핑할 수 있도록 역할 기반 보안 API(Role Management API)를 제공할 뿐만 아니라 역할 보안 제공자 (Role Provider) 역시 제공하고 있다. 이와 같은 기능을 이용하면 다음과 같은 방법으로 "Admin" 역할 그룹에 속하는 사용자만 /Dinners/Create URL에 접근할 수 있도록 지정할 수 있다.

```
[Authorize(Roles="admin")]
public ActionResult Create() {
    ...
}
```

새로운 모임 데이터를 추가할 때 User.Identity.Name 속성 활용하기

Controller 기반 클래스가 제공하는 User.Identity.Name 속성을 이용하면 현재 로그인 한 사용자의 ID를 손쉽게 얻어올 수 있다.

앞서 HTTP POST 방식을 위한 Create() 액션 메서드를 구현할 때 우리는 "HostedBy" 속성에 대입될 사용자 ID를 하드 코딩 형태로 지정했었다. 이제 이 코드를 수정하여 User.Identity.Name 속성을 이용하여 사용자의 ID를 대입하고 저녁 모임을 주최하는 사용자가 자동으로 참여자 목록에 추가되도록 구현해 보자.

```
//
// POST: /Dinners/Create
[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Create(Dinners dinner) {
    if (ModelState.IsValid) {
        try {
            dinner.HostedBy = User.Identity.Name;
            RSVP rsvp = new RSVP();
            rsvp.AttendeeName = User.Identity.Name;
            dinner.RSVP.Add(rsvp);
            dinnerRepository.Add(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details", new { id=dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
        }
    }
    return View(new DinnerFormViewModel(dinner));
}
```

Create() 액션 메서드에는 [Authorize] 액션 필터가 적용되어 있기 때문에 ASP.NET MVC는 /Dinners/Create URL에 접근한 사용자가 로그인 한 사용자인 경우에만 실행된다. 따라서 User.Identity.Name 속성은 항상 현재 로그인한 사용자의 이름을 리턴하게 된다.

기존의 모임 데이터를 수정할 때 User.Identity.Name 속성 활용하기

이제 약간의 로직을 추가하여 현재 모임을 주최한 사람만 모임 데이터를 수정할 수 있도록 애플리케이션의 기능을 제한해보자.

이 기능을 구현하려면 Dinners 클래스(앞서 Dinners.cs 파일에 부분 클래스로 정의했었다)에 "IsHostedBy(사용자ID)"와 같은 메서드를 추가해야 한다. 이 메서드는 지정된 사용자 ID가 HostedBy 속성에 보관된 사용자 ID와 일치하는지 여부에 따라 true 혹은 false를 리턴하며 대소문자를 구분하지 않도록 구현되어 있다.

```
public partial class Dinner {
    public bool IsHostedBy(string userName) {
        return HostedBy.Equals(userName,
            StringComparison.InvariantCultureIgnoreCase);
    }
}
```

그런 후 DinnersController 클래스의 Edit() 액션 메서드에 [Authorize] 액션 필터를 적용하여 로그인 한 사용자만 /Dinners/Edit/[id] URL에 접근할 수 있도록 한다.

이제 IsHostedBy(사용자ID) 메서드를 사용하여 현재 사용자가 조회된 Dinners 객체의 HostedBy 속성과 일치하는지를 판단하는 코드를 Edit() 액션 메서드에 다음과 같이 추가해보자. 만일 현재 로그인 한 사용자가 모임을 주최한 사용자가 아니라면 "InvalidOwner" 뷰 템플릿을 보여주게 된다.

```
//
// GET: /Dinners/Edit/5
[Authorize]
public ActionResult Edit(int id) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");
    return View(new DinnerFormViewModel(dinner));
}
//
// POST: /Dinners/Edit/5
[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Edit(int id, FormCollection collection) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");
    try {
        UpdateModel(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new {id = dinner.DinnerID});
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());
    }
}
```

```
return View(new DinnerFormViewModel(dinner));
}
}
```

이제 Visual Studio의 솔루션 탐색기에서 `Views\Dinners` 디렉토리를 마우스 오른쪽 버튼으로 클릭하고 [추가 > View] 메뉴를 선택하여 "InvalidOwner" 뷰를 생성한 후 다음과 같이 오류 메시지를 표시하는 HTML 코드를 추가한다.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
<title>데이터 액세스 오류</title>
</asp:Content>
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2>데이터 액세스 오류</h2>
<p>이 데이터를 수정할 권한이 없습니다.</p>
</asp:Content>
```

이제 자신이 직접 생성하지 않은 모임 데이터를 수정하려고 하면 다음 그림과 같은 페이지를 보게 된다.

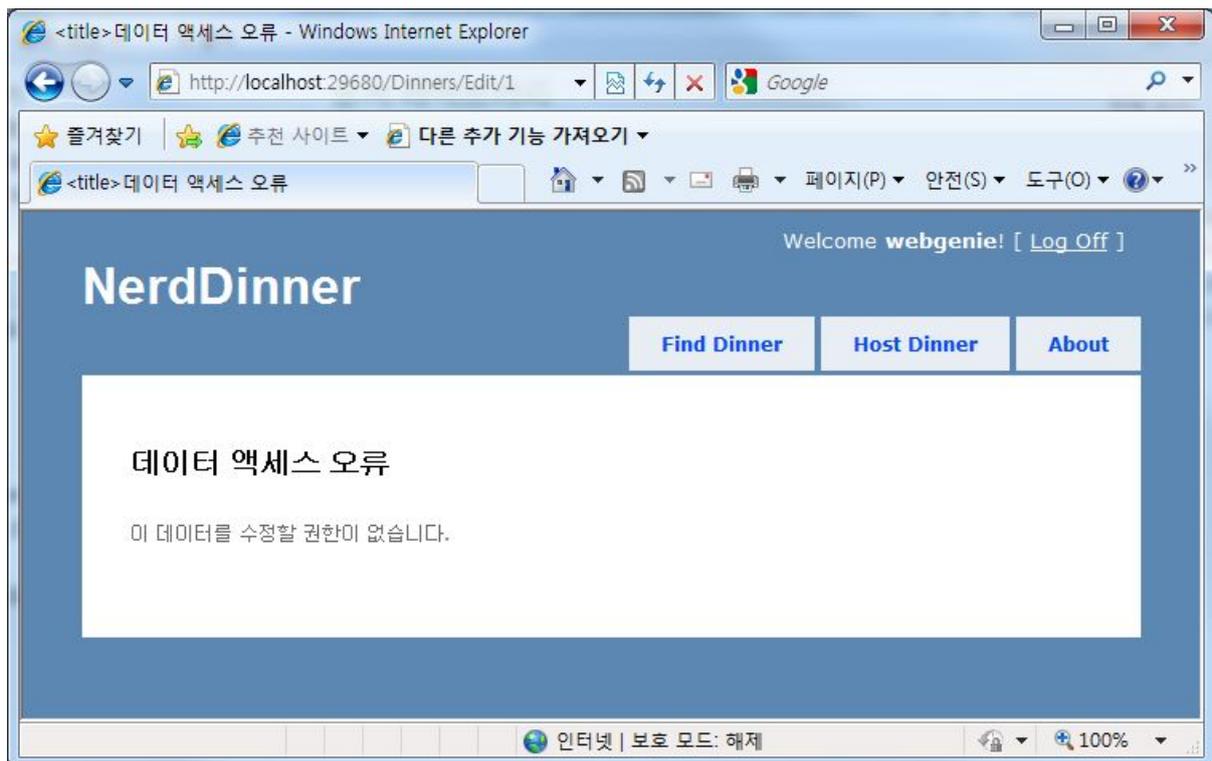


그림 1-120

모임 데이터를 삭제하는 `Delete()` 액션 메서드에도 지금까지와 동일한 방법을 적용하여 모임 데이터를 생성한 사용자만 데이터를 삭제할 수 있도록 애플리케이션을 수정할 수 있다.

수정/삭제 링크를 보이거나 숨기기

데이터 상세 보기 페이지에는 다음 그림과 같이 DinnersController 클래스의 Edit()과 Delete() 액션 메서드를 호출하기 위한 링크가 제공되고 있다.

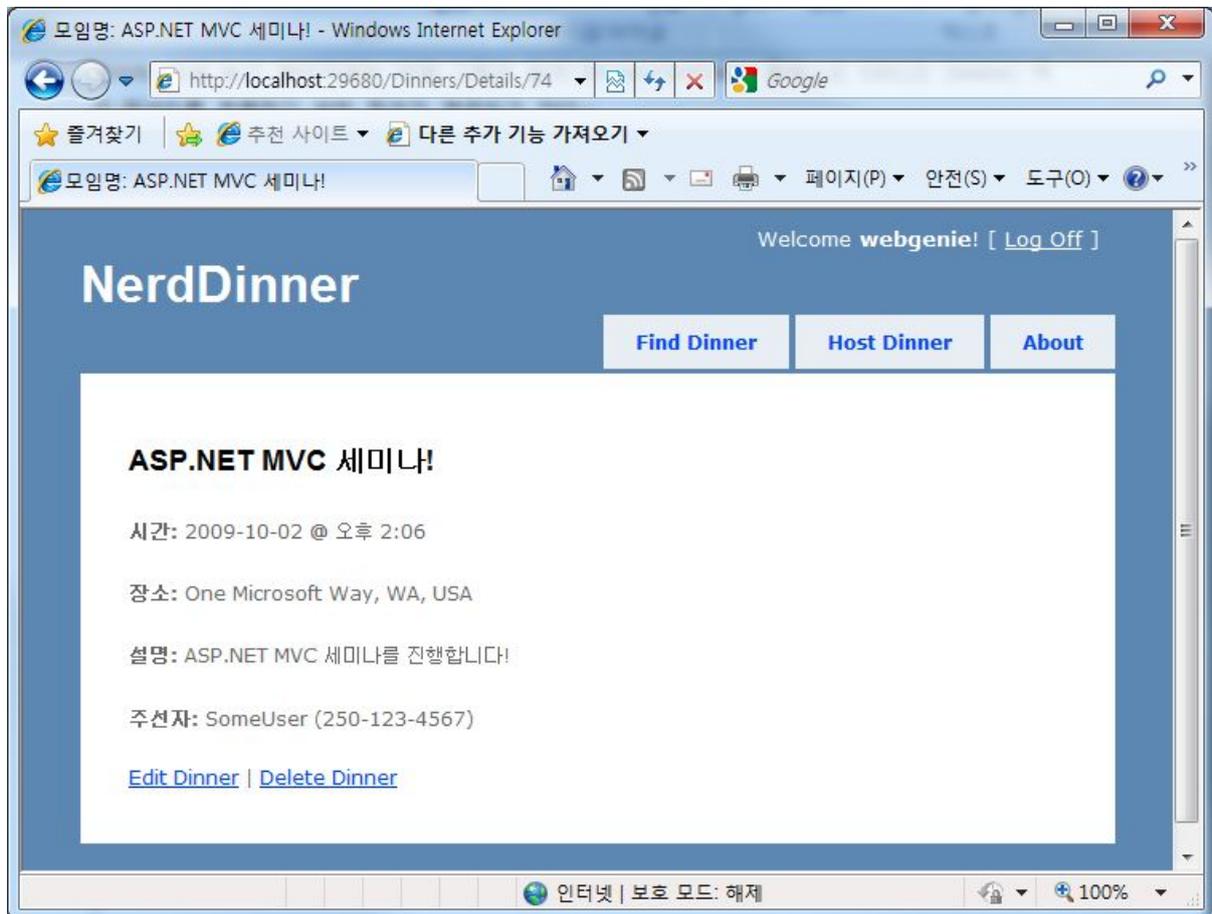


그림 1-121

현재 애플리케이션은 사용자가 해당 데이터를 생성한 사용자인지 여부를 판단하지 않고 두 개의 링크를 무조건 보여주고 있다. 이 부분을 수정하여 현재 보여지는 모임 데이터를 생성한 사용자에게만 수정 및 삭제 링크가 보여지도록 해보자.

상세 보기 페이지를 구현한 Details() 액션 메서드는 다음과 같이 지정된 Dinners 객체를 검색하여 이를 뷰 템플릿에 전달하고 있다.

```
//  
// GET: /Dinners/Details/5  
public ActionResult Details(int id) {  
    Dinner dinner = dinnerRepository.GetDinner(id);  
    if (dinner == null)  
        return View("NotFound");  
    return View(dinner);  
}
```

따라서 뷰 템플릿에 앞서 Dinners 클래스에 추가한 IsHostedBy(사용자ID) 메서드를 이용하여 현재 사용자가 누구인지에 따라 수정 및 삭제 링크를 보이거나 보이지 않도록 다음과 같이 코드를 수정할 수 있다.

```
<% if (Model.IsHostedBy(Context.User.Identity.Name)) { %>
<%= Html.ActionLink("Edit Dinner", "Edit", new { id=Model.DinnerID })%> |
<%= Html.ActionLink("Delete Dinner", "Delete", new { id=Model.DinnerID})%>
<% } %>
```

모임 참여 기능에 AJAX 적용하기

이제 로그인 한 사용자가 관심있는 저녁 모임에 참여할 수 있는 기능을 구현해보자. 이 기능은 저녁 모임 상세 보기 페이지에 AJAX 방식을 이용하여 구현할 것이다.

사용자가 이미 참여 중인지 확인하기

사용자가 /Dinners/Details/[id] URL에 접근하면 특정 저녁 모임의 상세 정보를 확인할 수 있다.

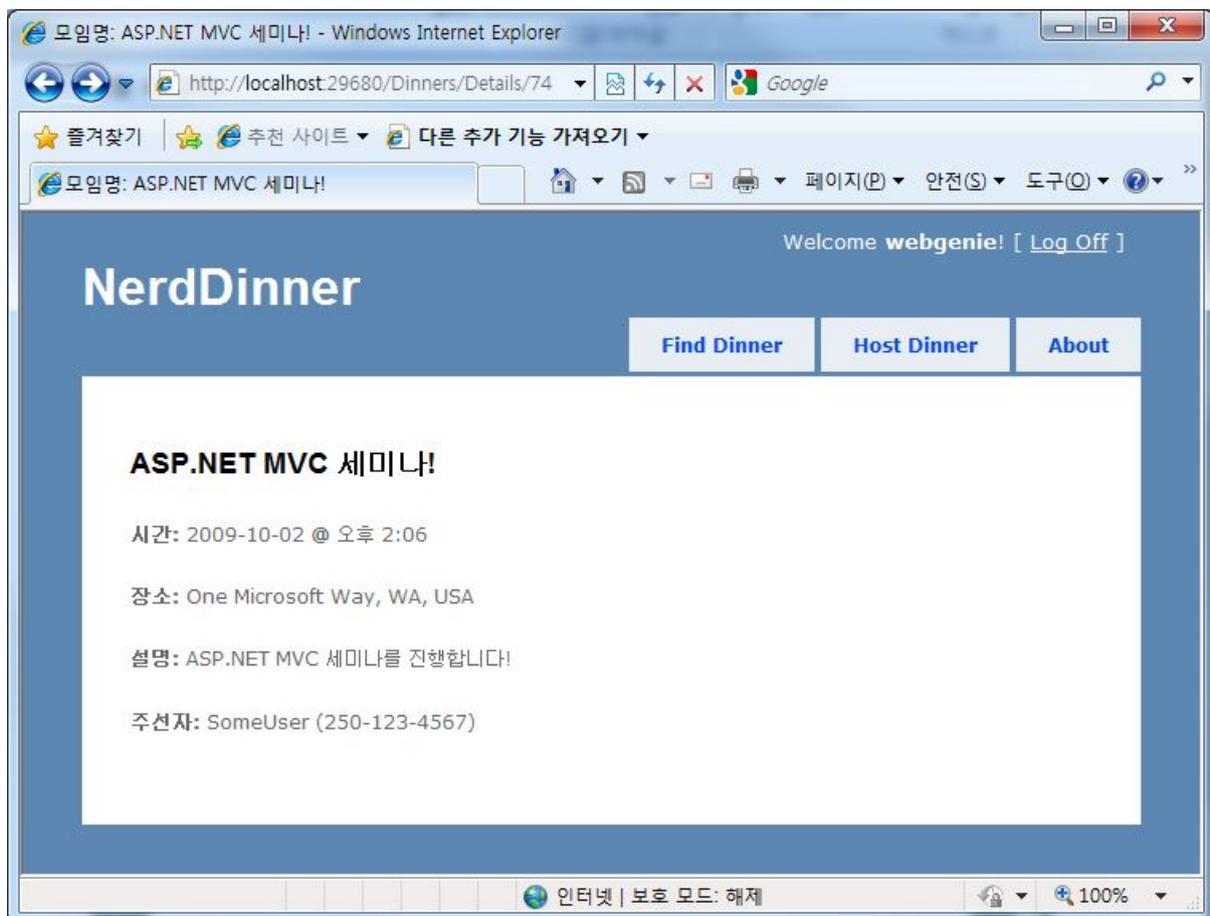


그림 1-122

이 페이지를 구현한 Details() 메서드는 다음과 같은 코드로 구현되어 있다.

```
//
// GET: /Dinners/Details/2
public ActionResult Details(int id) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
```

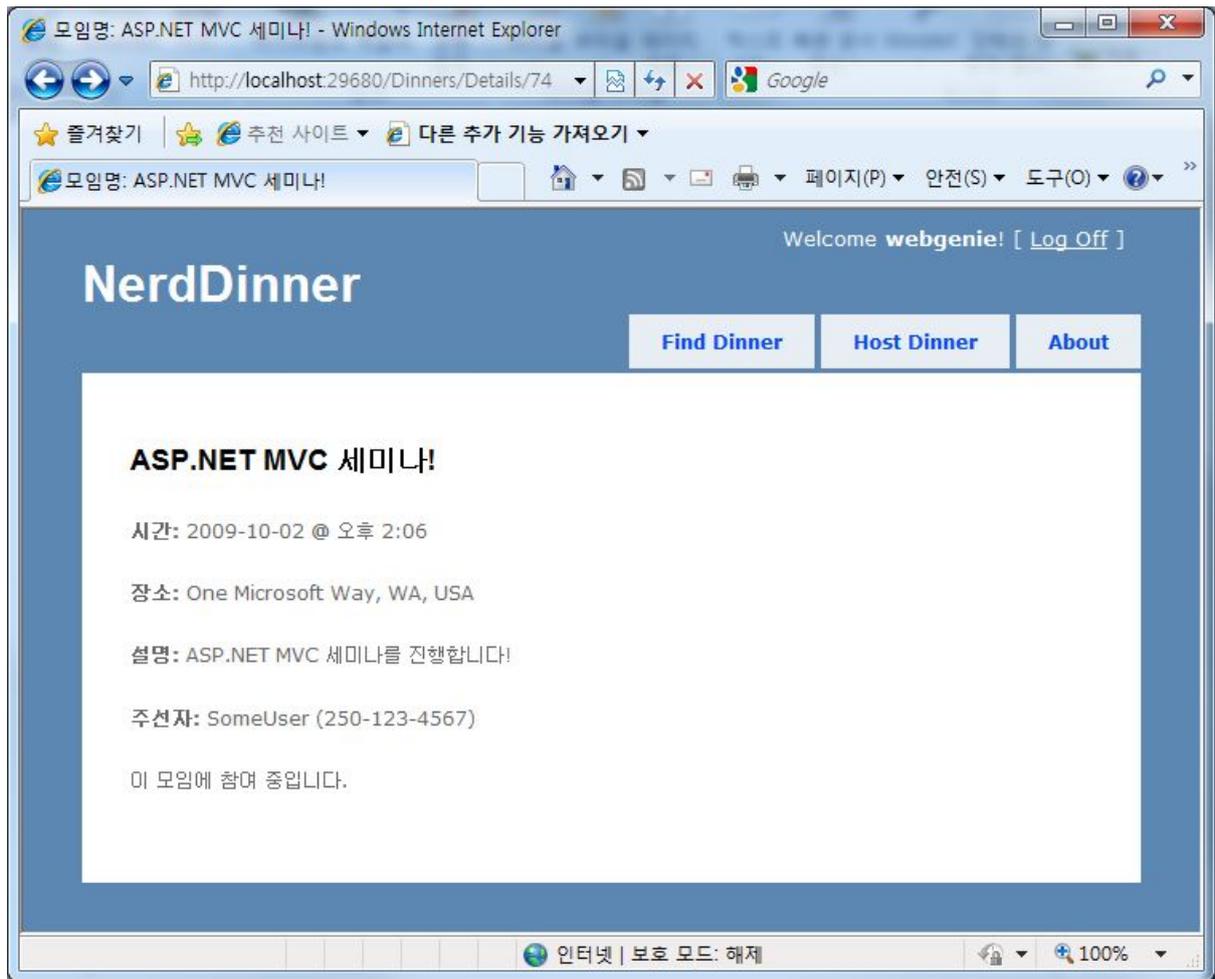
참여 기능을 구현하기 위한 첫 번째 단계는 Dinners 객체에 IsUserRegistered(사용자ID) 메서드를 추가하는 것이다 (앞서 추가했던 Dinner.cs 부분 클래스 파일에 추가한다). 이 메서드는 지정된 사용자가 해당 모임에 이미 참여하고 있는지 여부를 판단하여 true 혹은 false를 리턴한다.

```
public partial class Dinner {
    public bool IsUserRegistered(string userName) {
        return RSVPs.Any(r => r.AttendeeName.Equals(userName,
            StringComparison.InvariantCultureIgnoreCase));
    }
}
```

그런 후 Details.aspx 뷰 템플릿에 다음과 같은 코드를 추가하여 사용자의 모임 참여 여부를 보여 주는 메시지를 출력하도록 한다.

```
<% if (Request.IsAuthenticated) { %>
<% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>
<p>이 모임에 참여 중입니다.</p>
<% } else { %>
<p>이 모임에 참여하고 있지 않습니다.</p>
<% } %>
<% } else { %>
이 모임에 참여하려면 <a href="/Account/Logon">로그인</a> 하세요.
<% } %>
```

이제 사용자가 참여 중인 모임에 대한 상세 보기 페이지를 방문하면 다음 그림과 같은 페이지를 보게 된다.



또한 아직 참여 중이지 않은 모임에 대한 상세 보기 페이지는 다음 그림과 같다.



그림 1-124

Register 액션 메서드 구현하기

이제 사용자가 모임 상세 보기 페이지를 통해서 모임에 참여할 수 있는 기능을 추가해보자.

이 기능을 구현하려면 우선 `WControlls` 디렉토리를 마우스 오른쪽 버튼으로 클릭하고 `Add > Controller` 메뉴를 선택하여 "RSVPController"라는 새로운 컨트롤러 클래스를 추가해야 한다.

RSVPController 컨트롤러를 추가했으면 `Dinners` 객체의 ID를 인수로 전달받아 해당 `Dinners` 객체를 조회하고 현재 로그인 한 사용자가 해당 모임에 참여 중인지 여부를 판단한 후 아직 참여 중이지 않다면 RSVP 객체를 추가하여 현재 로그인 한 사용자를 해당 모임에 참여하도록 하는 "Register" 메서드를 다음과 같이 구현한다.

```
public class RSVPController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
    // AJAX: /Dinners/Register/1
    [Authorize, AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Register(int id) {
        Dinner dinner = dinnerRepository.GetDinner(id);
```

```

if (!dinner.IsUserRegistered(User.Identity.Name)) {
    RSVP rsvp = new RSVP();
    rsvp.AttendeeName = User.Identity.Name;
    dinner.RSVP.Add(rsvp);
    dinnerRepository.Save();
}
return Content("모임에 참여해 주셔서 감사합니다.");
}
}

```

코드에서 보듯이 이 메서드는 액션 메서드의 실행 결과로 텍스트를 리턴한다. 물론 이 텍스트는 뷰 템플릿을 이용해서 보여줄 수도 있지만 사용자에게 보여질 콘텐츠의 크기가 얼마되지 않으므로 Controller 기반 클래스가 제공하는 Content() 메서드를 이용하여 사용자에게 보여질 콘텐츠를 리턴한다.

AJAX 방식으로 Register 액션 메서드 호출하기

모임 상세 보기 페이지는 AJAX 방식으로 Register 메서드를 호출한다. AJAX를 이용한 메서드 호출 기능은 매우 간단하게 구현할 수 있다. 우선 다음과 같이 두 개의 스크립트 라이브러리를 참조한다.

```

<script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
<script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>

```

첫 번째 라이브러리는 ASP.NET AJAX의 클라이언트 측 스크립트 라이브러리를 참조한다. 이 파일의 크기는 (압축한 경우) 24K 정도이며 클라이언트 측 AJAX의 핵심 기능을 구현하고 있다. 두 번째 라이브러리는 (우리가 사용할) ASP.NET MVC가 제공하는 AJAX 메서드와 통합되는 유틸리티 함수들을 구현하고 있다.

이제 앞서 “이 모임에 참여하고 있지 않습니다.”라는 문자열을 보여주던 뷰 템플릿을 수정하여 문자열 대신 RSVPController 클래스의 Register() 액션 메서드를 AJAX 방식으로 호출하는 링크를 추가해보자.

```

<div id="rsvpmsg">
<% if (Request.IsAuthenticated) { %>
<% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>
<p>이 모임에 참여 중입니다.</p>
<% } else { %>
<%= Ajax.ActionLink( "이 모임에 참여하기",
"Register", "RSVP",
new { id=Model.DinnerID },
new AjaxOptions { UpdateTargetId="rsvpmsg" }) %>
<% } %>
<% } else { %>
이 모임에 참여하려면 <a href="/Account/Logon">로그인</a> 하세요.
<% } %>

```

</div>

위의 코드에서 사용된 Ajax.ActionLink() 메서드는 ASP.NET MVC에 포함된 메서드로 Html.ActionLink() 메서드와 유사하지만 액션 메서드를 실행하는 URL로 이동하는 대신 AJAX 방식을 액션 메서드를 호출하는 링크를 생성한다. 이 코드에서는 "RSVPController" 컨트롤러의 "Register" 액션 메서드에 "DinnerID" 속성 값을 "id" 매개 변수로 전달하고 있다. 마지막으로 전달한 AjaxOptions 매개 변수는 액션 메서드가 리턴하는 텍스트를 가져와 id가 "rsvpmsg"인 HTML <DIV> 요소에 표시하겠다는 것을 의미한다.

이제 사용자가 아직 참여 중이지 않은 모임에 대한 상세 보기 페이지에 접근하면 다음과 같이 모임에 참여할 수 있는 링크가 보여지게 된다.

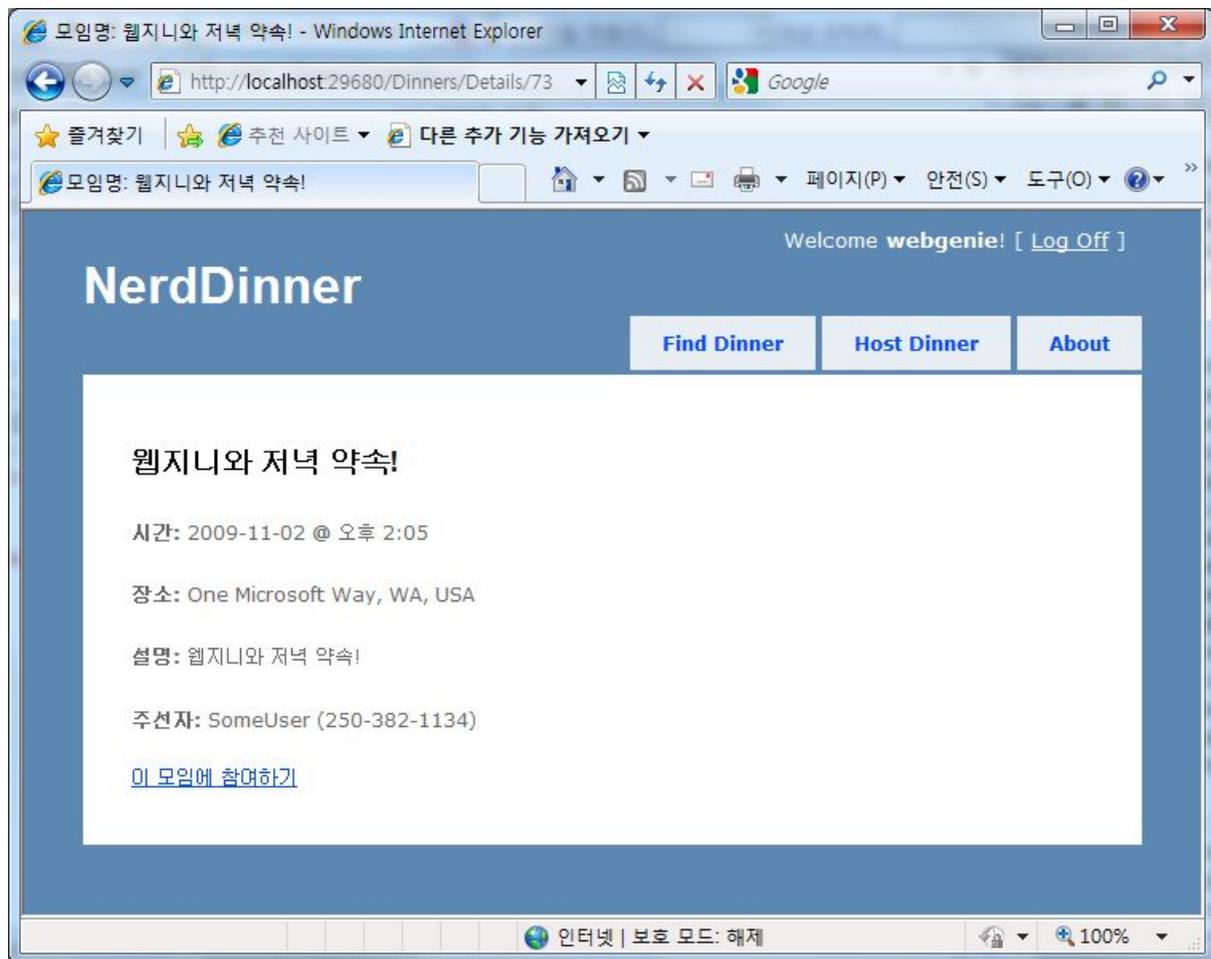


그림 1-125

사용자가 "이 모임에 참여하기" 링크를 클릭하면 RSVPController 클래스의 Register() 액션 메서드가 AJAX 방식으로 호출되며 액션 메서드가 성공적으로 실행되면 다음과 같이 메시지가 업데이트 된다.

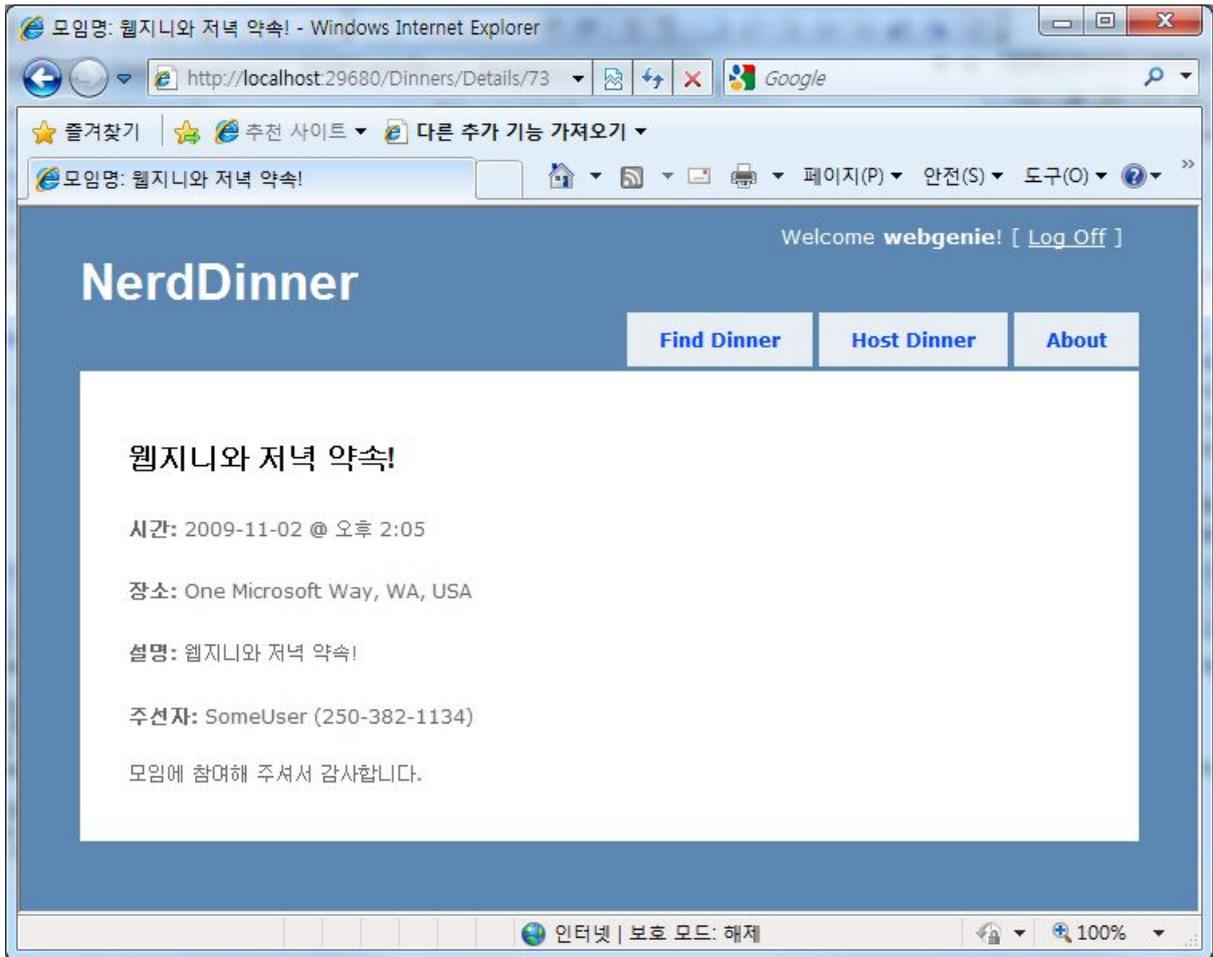


그림 1-126

AJAX 호출에 필요한 네트워크 대역폭과 트래픽은 매우 가볍다. 사용자가 “이 모임에 참여하기” 링크를 클릭하면 매우 작은 크기의 HTTP POST 요청이 /Dinners/Register/1 URL에 다음과 같이 전달된다.

```
POST /Dinners/Register/49 HTTP/1.1
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Referer: http://localhost:8080/Dinners/Details/49
```

Register() 액션 메서드의 응답 콘텐츠 역시 매우 간단하다.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 29
이 모임에 참여 중입니다.
```

이처럼 AJAX 호출은 매우 가벼워서 상대적으로 느린 네트워크 환경에서도 빠르게 잘 동작한다.

jQuery를 이용한 애니메이션 추가하기

앞서 구현했던 AJAX 호출은 빠르게 잘 동작한다. 때로는 너무 빠르게 동작해서 사용자는 모임 참여 링크가 텍스트로 변경되었는지 미처 알아채지 못하는 경우도 있다. 이 결과를 보다 명확하게 하기 위해 메시지가 수정될 때 사용자의 주의를 끌 수 있도록 간단한 애니메이션을 구현해보자.

ASP.NET MVC의 프로젝트 템플릿은 마이크로소프트가 후원하는 걸출한 (그리고 매우 보편적으로 사용되는) jQuery라는 이름의 오픈 소스 자바스크립트 라이브러리를 포함하고 있다. jQuery는 멋진 HTML DOM 선택기와 다양한 효과 등 여러 가지 기능을 제공한다.

jQuery를 사용하려면 우선 스크립트 참조를 추가해야 한다. jQuery는 애플리케이션 내의 여러 곳에서 활용할 예정이므로 모든 페이지가 jQuery 스크립트를 사용할 수 있도록 Site.master 파일에 다음과 같이 스크립트를 참조하는 코드를 추가한다.

```
<script src="/Scripts/jquery-1.3.2.js" type="text/javascript"></script>
```

=====
Tip: 자바스크립트 코드를 작성할 때 인텔리센스를 지원해 주는 "Visual Studio 2008 SP1용 자바스크립트 인텔리센스 핫픽스를 설치하면 더욱 편리하게 (jQuery를 포함하여) 자바스크립트 코드를 작성할 수 있다. 이 핫픽스는 <http://tinyurl.com/vs2008javascripthotfix>에서 다운로드할 수 있다.
=====

jQuery를 이용해서 작성된 코드는 대부분의 경우 CSS 선택자를 이용하여 HTML 요소를 탐색하는 "\$()" 자바스크립트 메서드를 사용한다. 예를 들어 \$("#rsvpmsg") 메서드는 id 특성 값이 "rsvpmsg"인 HTML 요소를 찾아내며 \$(".something")은 CSS 클래스 이름이 "something"인 HTML 요소들을 모두 찾아낸다. "현재 선택된 모든 라디오 버튼"처럼 복잡한 HTML 요소들을 탐색하는 것도 \$("input[@type=radio][@checked]")와 같이 간단하게 찾아낼 수 있다.

일단 요소들을 찾아내면 이후의 동작을 수행하기 위한 메서드를 호출할 수 있다. 예를 들어 발견된 요소들을 보이지 않도록 하려면 \$("#rsvpmsg").hide()와 같이 코드를 작성하면 된다.

모임 참여 기능의 경우에는 id 특성 값이 "rsvpmsg"인 <DIV> 요소를 찾아 요소 내의 텍스트의 크기가 확대되는 애니메이션을 수행하는 "AnimateRSVPMessage"라는 이름의 자바스크립트 함수를 구현할 것이다. 다음의 코드는 작은 크기의 텍스트를 0.4초에 걸쳐 점차 크게 만드는 애니메이션을 구현한다.

```
<script type="text/javascript">  
function AnimateRSVPMessage() {  
$("#rsvpmsg").animate({fontSize: "1.5em"}, 400);  
}  
</script>
```

그런 후 Ajax.ActionLink() 메서드의 AJAX 요청이 성공적으로 수행되었을 때 호출될 자바스크립트 함수에 방금 구현한 자바스크립트 함수를 지정한다 (AjaxOptions 객체의 "OnSuccess" 이벤트 속성에 지정하면 된다).

```
<%= Ajax.ActionLink( "이 모임에 참여하기",  
"Register", "RSVP",  
new { id=Model.DinnerID },  
new AjaxOptions { UpdateTargetId="rsvpmsg",  
OnSuccess="AnimateRSVPMessage" }) %>
```

이제 "이 모임에 참여하기" 링크를 클릭하여 AJAX 요청이 성공적으로 실행되면 메시지가 애니메이션을 통해 점차 커지는 효과를 볼 수 있다.

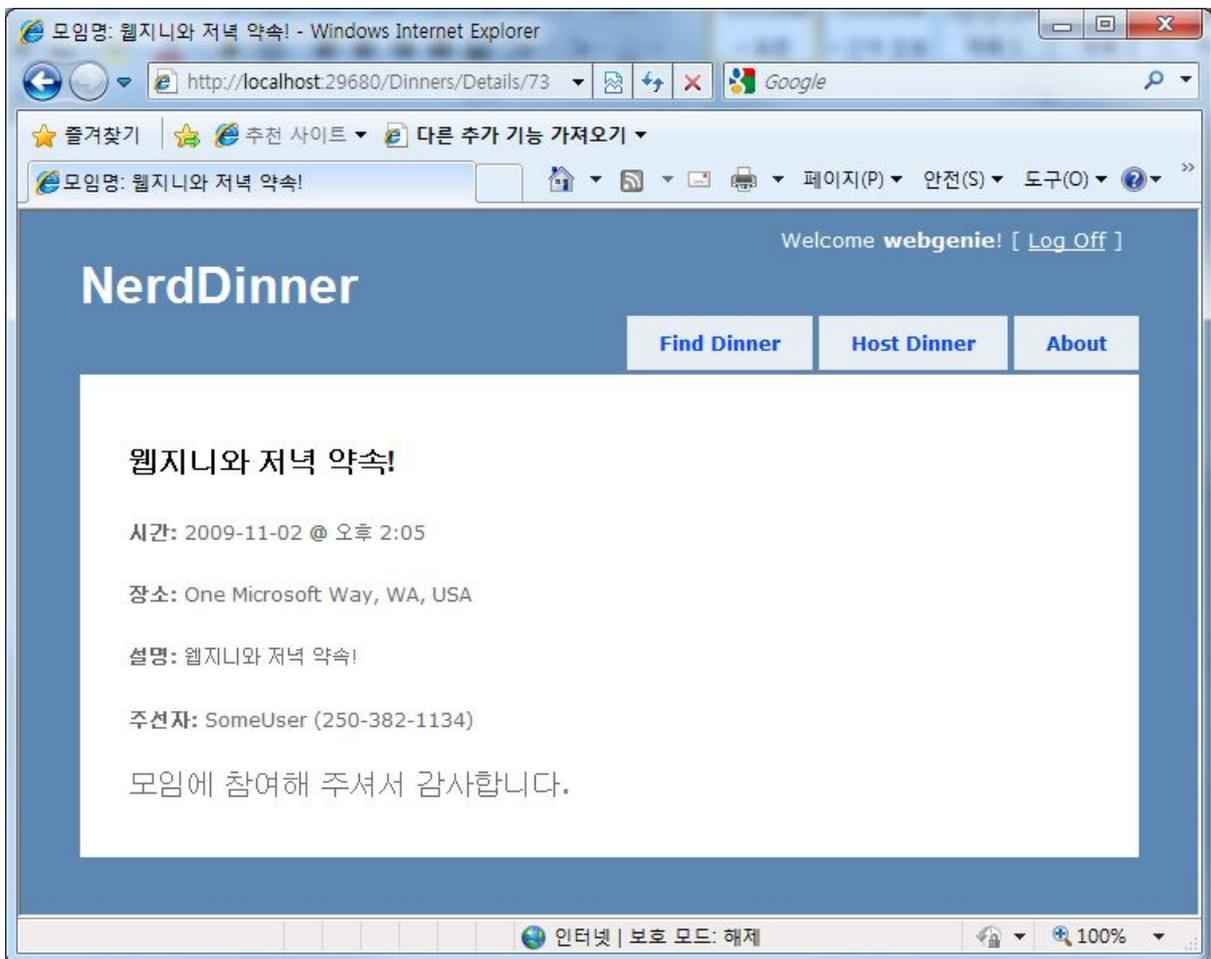


그림 1-127

AjaxOptions 객체는 "OnSuccess" 이벤트 외에도 OnBegin, OnFailure, OnComplete 등 (다양한 속성과 유용한 옵션들을 제공하는) 이벤트들을 추가로 제공한다.

코드 정리 - RSVP 부분 뷰 템플릿 리팩토링하기

지금까지의 과정을 거쳐오면서 모임 상세 보기 페이지의 코드는 점차 길어지기 시작하여 이제는 이해하기가 다소 어려운 수준에 이르렀다. 코드의 가독성을 높이기 위해 모임의 참여 여부를 표시하는 부분의 코드를 RSVPStatus.ascx라는 이름의 부분 뷰 템플릿으로 분리해보자.

먼저 `Views\Dinners` 디렉터리를 마우스 오른쪽 버튼으로 클릭하고 추가 > View 메뉴를 선택한다. Add View 대화 상자가 나타나면 Dinners 객체를 모델 객체로 사용하도록 지정하고 Details.aspx 페이지에서 모임 참여 여부를 보여주는 부분의 코드를 복사해온다.

이번에는 수정 및 삭제 링크를 보여주는 코드를 위해 EditAndDeleteLinks.ascx라는 이름의 부분 뷰 템플릿을 새로 추가하자. 이 부분 뷰 템플릿 역시 Dinners 객체를 모델 객체로 사용하도록 지정하고 Details.aspx 뷰 템플릿에서 코드를 복사해오면 된다.

이제 Details.aspx 뷰 템플릿은 다음과 같이 두 개의 `Html.RenderPartial()` 메서드를 사용하도록 수정할 수 있다.

```
<% Html.RenderPartial("RSVPStatus"); %>
<% Html.RenderPartial("EditAndDeleteLinks"); %>
```

이렇게 함으로써 보다 깔끔하고 유지보수하기 쉬운 코드로 뷰 템플릿을 구현할 수 있다.

AJAX를 이용하여 지도 통합하기

이번에는 애플리케이션에 AJAX 기능을 이용한 지도를 통합함으로써 시각적으로 보다 뛰어난 사용자 경험을 제공할 수 있도록 구현해보자.

지도를 위한 부분 뷰 템플릿 생성하기

지도는 애플리케이션의 여러 곳에서 재사용할 예정이므로 코드를 깔끔하게 유지하기 위해서는 지도 부분을 하나의 뷰 템플릿으로 만들어 여러 컨트롤러와 뷰 템플릿이 재사용 가능한 형태로 구현해야 한다. 지도를 위한 부분 뷰 템플릿은 `Views\Dinners` 디렉터리에 "map.ascx"라는 이름으로 추가하자.

`Views\Dinners` 디렉터리를 마우스 오른쪽 버튼으로 클릭하고 추가 > View 메뉴를 선택한 후 이름을 "Map.ascx"로 입력하고 부분 뷰 템플릿으로 만들도록 선택한 후 "Dinner" 객체를 모델 객체로 사용하도록 선택한다.

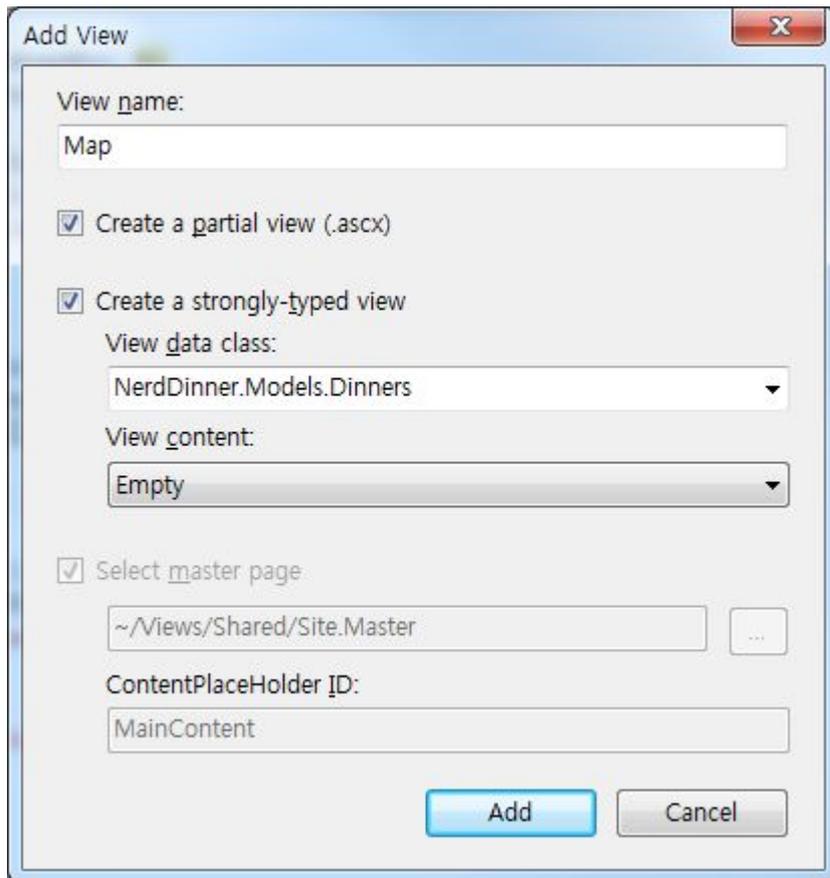


그림 1-128

"Add" 버튼을 클릭하여 새로운 뷰 템플릿을 추가한 후 다음의 코드를 작성한다.

```
<script src="http://dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.2"
type="text/javascript"></script>
<script src="/Scripts/Map.js" type="text/javascript"></script>
<div id="theMap">
</div>
<script type="text/javascript">
$(document).ready(function() {
var latitude = <%=Model.Latitude %>;
var longitude = <%=Model.Longitude %>;
if ((latitude == 0) || (longitude == 0))
LoadMap();
else
LoadMap(latitude, longitude, mapLoaded);
});
function mapLoaded() {
var title = "<%= Html.Encode(Model.Title) %>";
var address = "<%= Html.Encode(Model.Address) %>";
LoadPin(center, title, address);
map.SetZoomLevel(14);
}
</script>
```

첫 번째 <script> 태그는 마이크로소프트의 Virtual Earth 6.2 지도 라이브러리를 가져오는 코드가

며 두 번째 `<script>` 태그는 우리가 자바스크립트로 구현할 지도 관련 함수들이 구현되어 있는 `map.js` 파일을 가져오는 코드이다. `<DIV id="theMap">` 요소는 Virtual Earth를 통해 가져온 지도를 보여줄 컨테이너로서의 역할을 담당한다.

우리가 `<script>` 태그를 이용해서 가져오는 자바스크립트 파일에는 지도 뷰 템플릿을 위한 두 개의 자바스크립트 함수가 구현되어 있다. 첫 번째 함수는 jQuery를 이용하여 페이지가 자바스크립트 코드를 실행할 준비가 완료되었을 때 실행되는 함수로 Virtual Earth 지도 컨트롤을 로드하기 위해서 `Map.js` 파일에 구현된 `LoadMap()` 함수를 호출한다. 두 번째 함수는 콜백 함수(Callback Function)로 사용자가 지도의 특정 위치에 핀을 꽂을 때 호출될 함수이다.

위의 코드를 살펴보면 클라이언트 스크립트 블록 내에서 특정 모임 장소의 위치를 의미하는 위도(Latitude)와 경도(Longitude) 값을 사용하기 위해 서버 측 코드 블록(`<%= %>`)을 사용하고 있음을 알 수 있다. 이와 같은 방법은 클라이언트 스크립트에 동적인 값을 적용할 수 있는 보편적인 방법이다 (별도의 AJAX 요청을 사용하지 않기 때문에 더 빠르게 동작할 수 있다). `<%= %>` 코드 블록은 뷰가 서버 측에서 렌더링될 때 실행되기 때문에 결과적으로 렌더링된 HTML 코드에는 자바스크립트에서 사용할 수 있는 값이 출력되게 된다 (예를 들면 `var latitude = 42.64312;`)와 같이 렌더링된다).

Map.js 유틸리티 라이브러리 구현하기

이제 지도 뷰 템플릿에서 사용할 수 있는 자바스크립트 함수들만을 구현할 `Map.js` 파일을 생성해 보자 (앞서 언급했던 `LoadMap` 메서드와 `LoadPin` 메서드를 구현할 것이다). Visual Studio의 솔루션 탐색기에서 `WScript` 디렉토리를 마우스 오른쪽 버튼으로 클릭하고 "추가 > 새 항목" 메뉴를 클릭한 후 `JScript` 파일 템플릿을 선택하고 "Map.js"라는 이름을 지정한다.

그런 후 다음과 같이 Virtual Earth 지도를 표시하고 모임 장소를 표시하기 위한 핀을 표시하는 자바스크립트 코드를 `Map.js` 파일에 작성한다.

```
var map = null;
var points = [];
var shapes = [];
var center = null;
function LoadMap(latitude, longitude, onMapLoaded) {
    map = new VEMap('theMap');
    options = new VEMapOptions();
    options.EnableBirdseye = false;
    // 컨트롤 막대를 조금 더 보기 좋게 만든다.
    map.SetDashboardSize(VEDashboardSize.Small);
    if (onMapLoaded != null)
        map.onLoadMap = onMapLoaded;
    if (latitude != null && longitude != null) {
        center = new VELatLong(latitude, longitude);
    }
    map.LoadMap(center, null, null, null, null, null, null, options);
}
```

```

function LoadPin(LL, name, description) {
var shape = new VEShape(VEShapeType.Pushpin, LL);
//푸시 핀에 제목과 설명을 표시한다.
shape.SetTitle("<span class=\"pinTitle\"> " + escape(name) + "</span>");
if (description !== undefined) {
shape.SetDescription("<p class=\"pinDetails\">" +
escape(description) + "</p>");
}
map.AddShape(shape);
points.push(LL);
shapes.push(shape);
}
function FindAddressOnMap(wher) {
var numberOfResults = 20;
var setBestMapView = true;
var showResults = true;
map.Find("", wher, null, null, null,
numberOfResults, showResults, true, true,
setBestMapView, callbackForLocation);
}
function callbackForLocation(layer, resultsArray, places,
hasMore, VErrorMessage) {
clearMap();
if (places == null)
return;
//검색된 위치에 대한 푸시 핀을 생성한다.
$.each(places, function(i, item) {
var description = "";
if (item.Description !== undefined) {
description = item.Description;
}
var LL = new VELatLong(item.LatLong.Latitude,
item.LatLong.Longitude);
LoadPin(LL, item.Name, description);
});
//모든 푸시 핀이 화면에 나타나도록 한다.
if (points.length > 1) {
map.SetMapView(points);
}
//정확한 장소를 찾으면 해당 주소를 표시한다.
if (points.length === 1) {
$("#Latitude").val(points[0].Latitude);
$("#Longitude").val(points[0].Longitude);
}
}
function clearMap() {
map.Clear();
points = [];
shapes = [];
}

```

모임 생성 및 수정 페이지에 지도 기능을 추가하기

이제 새 모임을 생성하거나 기존 모임 데이터를 수정하는 페이지에 지도 기능을 추가해보자. 다 행히 지도를 추가하는 것은 그다지 어렵지 않으며 컨트롤러 클래스의 코드를 수정할 필요도 없다.

게다가 Create 뷰 템플릿과 Edit 뷰 템플릿은 동일한 양식을 표시하기 위해 "DinnerForm" 이라는 부분 뷰를 공유하여 만들어졌으므로 여기에 지도를 추가하면 한 번의 작업으로 모임 생성 페이지와 수정 페이지에 모두 지도를 적용할 수 있다.

그러면 `Views\Dinners\DinnerForm.ascx` 부분 뷰 템플릿을 열고 새로 구현한 지도 부분 뷰 템플릿을 추가해보자. 다음의 코드는 지도 뷰 템플릿을 위한 코드가 추가된 DinnerForm 부분 뷰 템플릿의 코드이다 (코드를 단순화 하기 위해 HTML 양식들을 구현한 코드는 생략하였다).

```
<%= Html.ValidationSummary() %>
<% using (Html.BeginForm()) { %>
<fieldset>
<div id="dinnerDiv">
<p>
HTML 폼 요소들은 생략
</p>
<p>
<input type="submit" value="Save" />
</p>
</div>
<div id="mapDiv">
<% Html.RenderPartial("Map", Model.Dinner); %>
</div>
</fieldset>
<script type="text/javascript">
$(document).ready(function() {
$("#Address").blur(function(evt) {
$("#Latitude").val("");
$("#Longitude").val("");
var address = jQuery.trim($("#Address").val());
if (address.length < 1)
return;
FindAddressOnMap(address);
});
});
</script>
<% } %>
```

위와 같이 구현한 DinnerForm 부분 뷰 템플릿은 "DinnerFormViewModel" 타입을 모델 객체로 사용한다 (왜냐하면 이 뷰는 Dinners 객체뿐 아니라 국가 목록을 표시하는 드롭다운 목록을 위해 SelectList 객체들도 사용하기 때문이다). 그러나 우리가 구현한 지도 뷰 템플릿은 Dinners 객체만 사용하므로 부분 뷰 템플릿을 렌더링할 때 다음과 같이 DinnerFormViewModel 타입의 Dinner 속성만 전달해도 된다.

```
<% Html.RenderPartial("Map", Model.Dinner); %>
```

우리가 부분 뷰에 추가한 자바스크립트 코드는 id 특성 값이 "Address"인 HTML 텍스트 상자의 "blur" 이벤트를 이용한다. 여러분은 사용자가 텍스트 상자를 클릭하거나 혹은 탭 키를 이용하여 텍스트 상자에 포커스를 이동한 경우 "focus" 이벤트가 발생한다는 것을 알고 있을 것이다. 반대로 "blur" 이벤트는 사용자가 다른 컨트롤로 포커스를 이동하여 텍스트 상자가 포커스를 잃는 경

우에 발생하는 이벤트이다. 위의 코드는 blur 이벤트가 발생했을 때 위도와 경도 텍스트 상자의 값을 초기화하고 새로운 위치의 지도를 보여준다. 그러면 map.js 파일에 정의한 콜백 함수는 우리가 Virtual Earth로 전달했던 위치에 해당하는 새로운 위도와 경도를 텍스트 상자에 출력한다.

이제 애플리케이션을 실행하고 "모임 생성" 탭을 클릭하면 다음 그림과 같이 기본 위치를 보여주는 지도를 볼 수 있다.

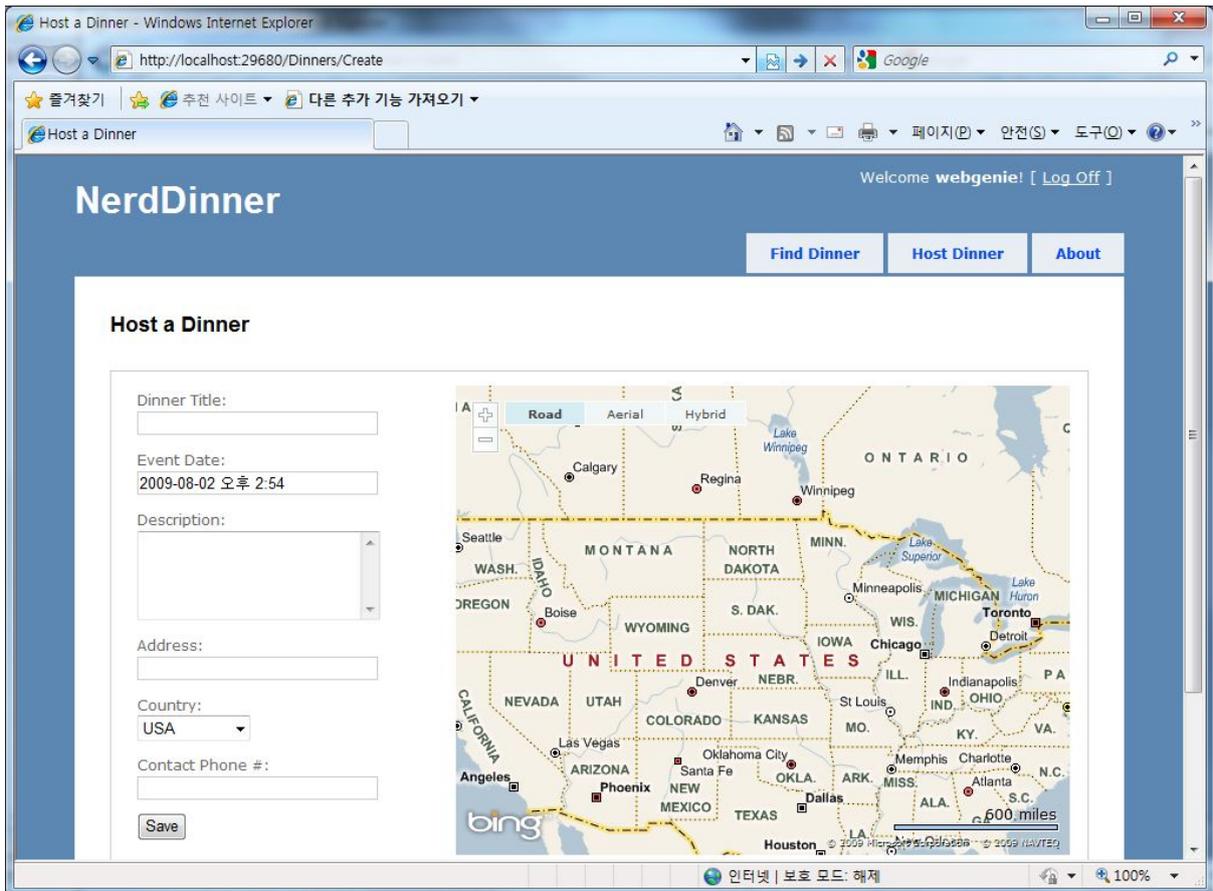
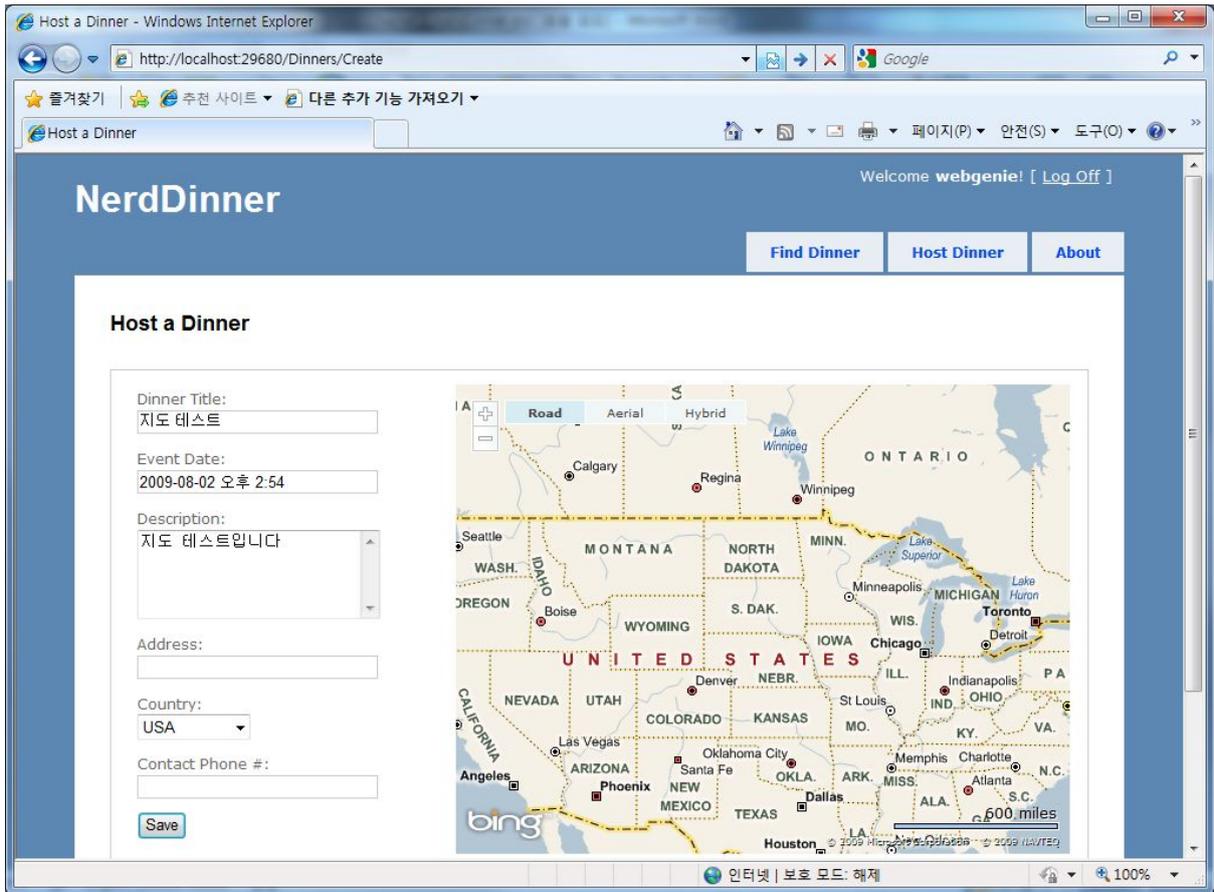


그림 1-129

모임 장소의 주소를 입력하고 탭 키를 눌러보면 지도가 자동으로 해당 위치를 보여주게 되며 우리가 구현했던 이벤트 핸들러가 지도의 위치에 해당하는 위도와 경도를 텍스트 상자에 입력해준다.



이제 새로운 모임 데이터를 저장하고 새로 추가한 모임 데이터를 수정하는 페이지로 이동해보면 다음 그림과 같이 페이지가 로드될 때 지도가 나타나는 것을 볼 수 있다.

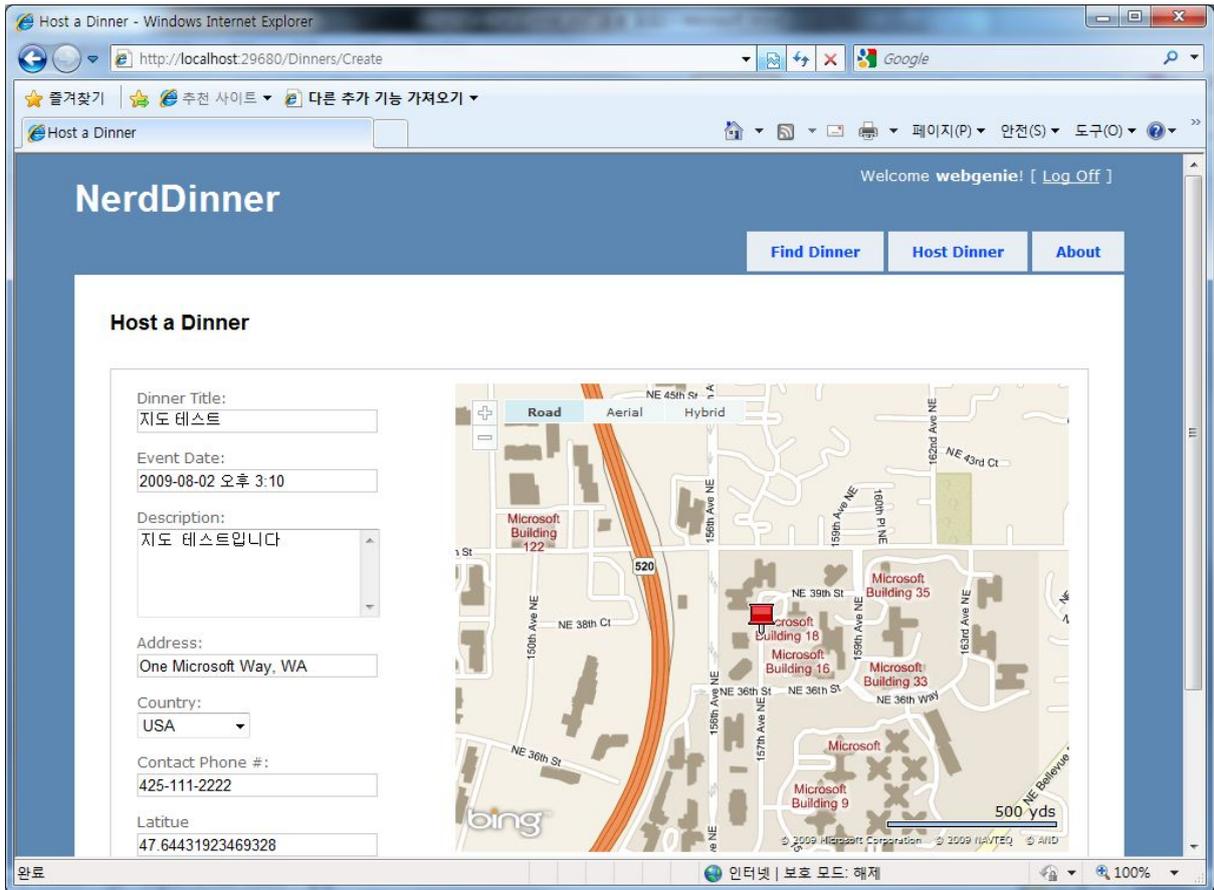


그림 1-130

주소를 입력하는 텍스트 상자의 값이 변경될 때마다 지도와 위도/경도는 계속해서 자동으로 업데이트된다.

이제 모임 장소를 지도가 보여주고 있으므로 위도와 경도 필드는 (주소가 변경될 때마다 지도가 이 값을 자동으로 업데이트하므로) 굳이 화면에 보여질 필요가 없어졌다. 다음과 같이 `Html.TextBox()` 메서드를 `Html.Hidden()` 메서드로 변경하여 위도와 경도가 페이지에서 보이지 않도록 수정해보자.

```
<p>
<%= Html.Hidden("Latitude", Model.Dinner.Latitude) %>
<%= Html.Hidden("Longitude", Model.Dinner.Longitude) %>
</p>
```

이제 새 모임 생성과 모임 수정 페이지는 보다 사용자가 사용하기 쉬워졌으며 더 이상 일반 사용자가 입력하기 힘든 위도와 경도를 요구하지 않게 되었다 (물론 이 값들은 여전히 데이터베이스에 저장된다).

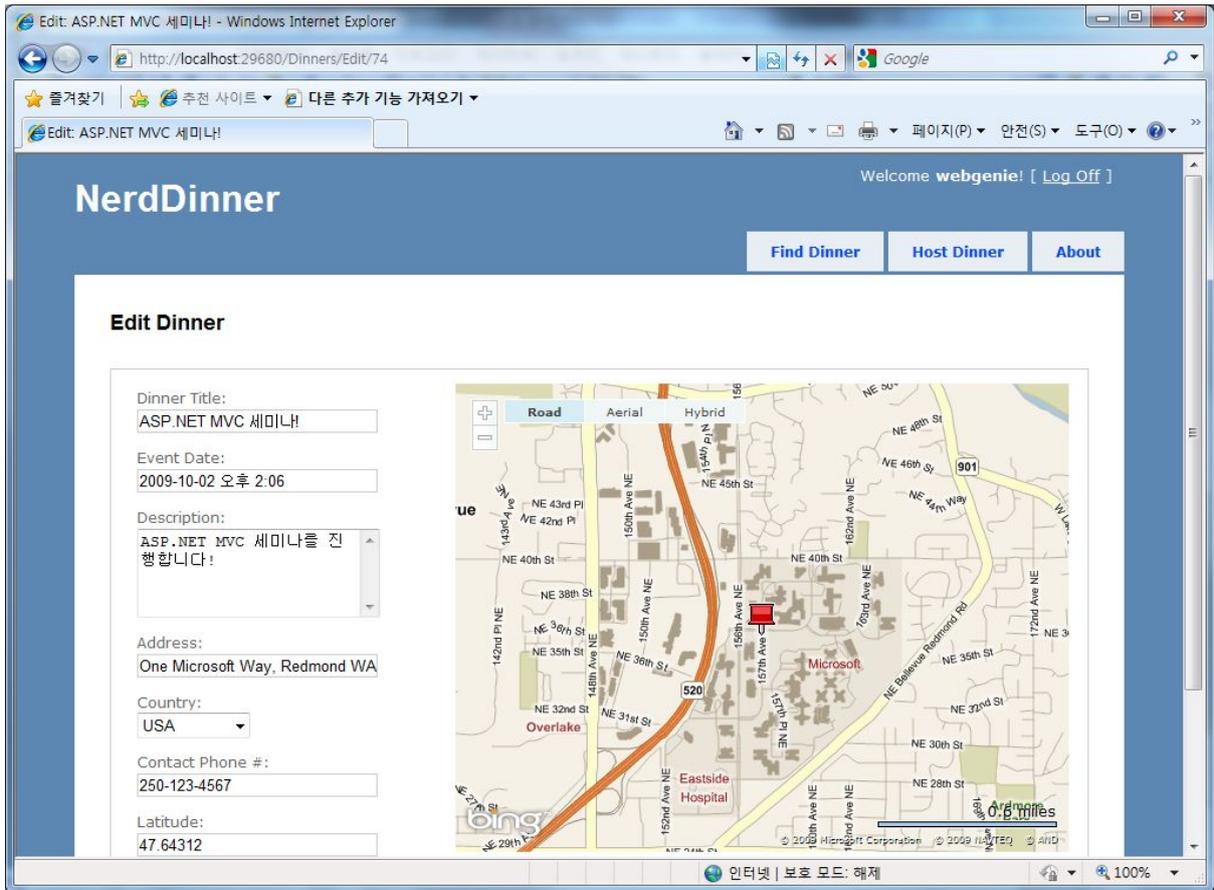


그림 1-131

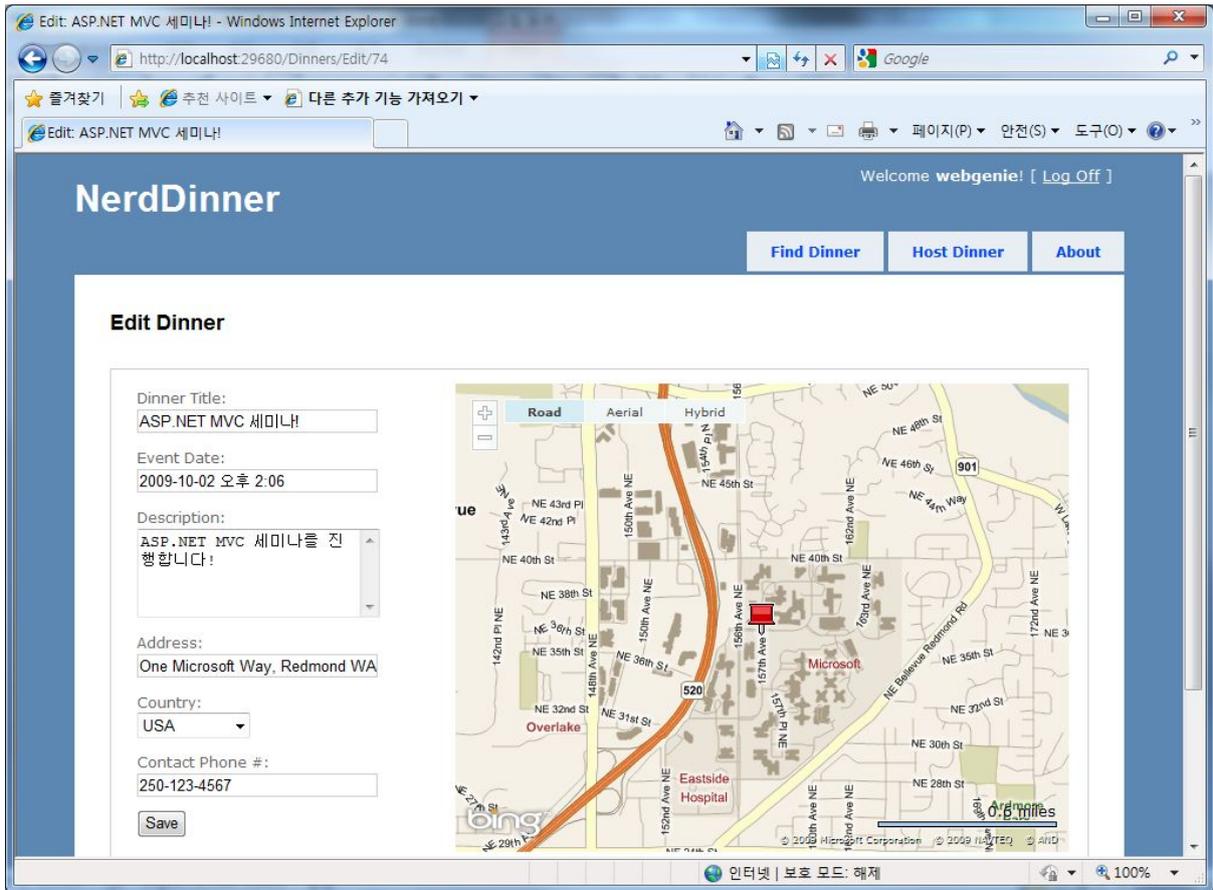


그림 1-132

모임 정보 상세 보기 페이지에 지도 추가하기

앞서 모임 생성 페이지와 모임 수정 페이지에 지도를 추가하였으므로 이번에는 모임 상세 보기 페이지에 지도를 추가해보자. 우리가 해야 할 일은 Details 뷰 템플릿에 `<% Html.RenderPartial("map");` 코드를 추가하는 것 뿐이다.

지도가 추가된 Details 뷰 템플릿의 소스 코드는 다음과 같다.

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
<%= Html.Encode(Model.Title) %>
</asp:Content>
<asp:Content ID="details" ContentPlaceHolderID="MainContent"
runat="server">
<div id="dinnerDiv">
<h2><%= Html.Encode(Model.Title) %></h2>
<p>
<strong>시간:</strong>
<%= Model.EventDate.ToShortDateString() %>
<strong>@</strong>
<%= Model.EventDate.ToShortTimeString() %>
</p>
<p>
```

```

<strong>장소:</strong>
<%= Html.Encode (Model.Address) %>,
<%= Html.Encode (Model.Country) %>
</p>
<p>
<strong>설명:</strong>
<%= Html.Encode (Model.Description) %>
</p>
<p>
<strong>주선자:</strong>
<%= Html.Encode (Model.HostedBy) %>
(<%= Html.Encode (Model.ContactPhone) %>)
</p>
<% Html.RenderPartial ("RSVPStatus"); %>
<% Html.RenderPartial ("EditAndDeleteLinks"); %>
</div>
<div id="mapDiv">
<% Html.RenderPartial ("map"); %>
</div>
</asp:Content>

```

이제 사용자가 /Dinners/Details/[id] URL을 방문하면 모임에 대한 상세 정보와 모임 장소가 표시된 지도를 보게 된다 (이 지도에는 정확한 위치를 표시하기 위한 핀이 표시되며 이 핀에 마우스를 가져가면 모임의 제목과 위치가 표시된다). 또한 모임에 참여하기 위한 링크 역시 보여진다.

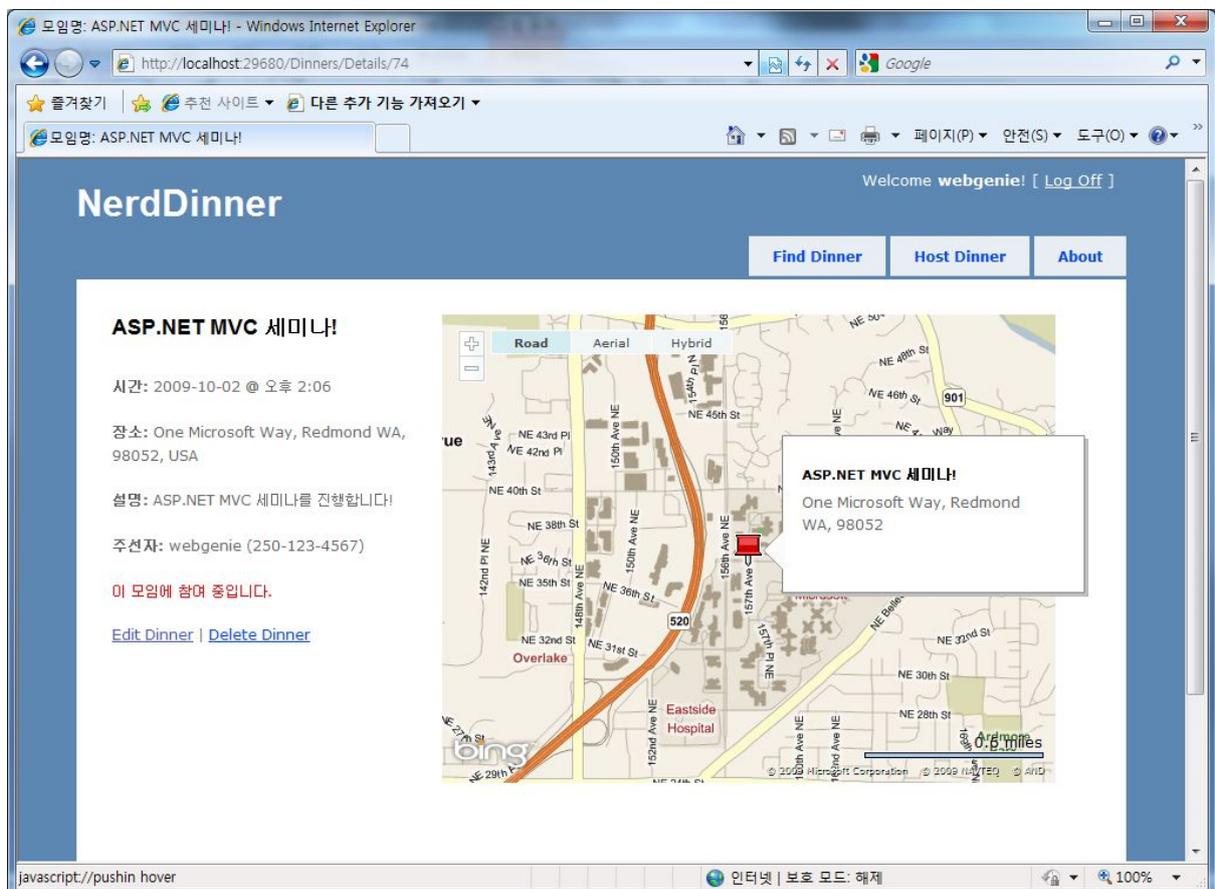


그림 1-133

데이터베이스와 저장소 클래스에서 위치를 검색하기

AJAX와 관련된 마지막 기능은 애플리케이션의 첫 페이지에 지도를 추가하고 현재 사용자와 가장 근접한 곳에서 주최될 모임들을 그래픽적으로 검색하는 기능이다.

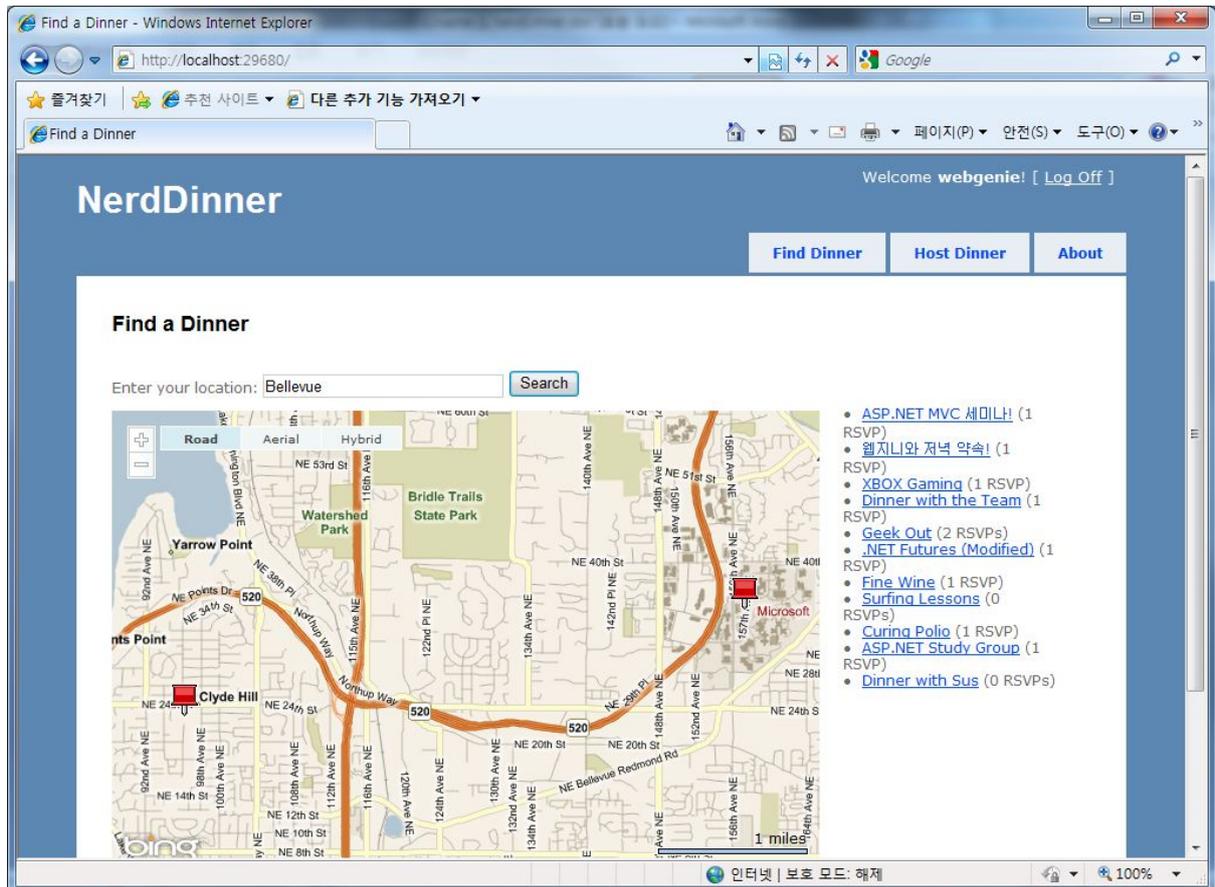


그림 1-134

이 기능을 구현하려면 우선 데이터베이스와 저장소 계층에서 위치를 기반으로 반경 내 검색을 효율적으로 수행하는 기능을 구현해야 한다. 이 기능은 SQL 서버 2008에 새롭게 추가된 공간 정보 관리 기능(<http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx>)을 이용하거나 혹은 게리 드라이든(Gary Dryden)이 SQL 서버의 기능을 이용하여 작성한 온라인 게시물(<http://www.codeproject.com/KB/cs/distancebetweenlocations.aspx>) 혹은 롭 코너리(Rob Conery)가 LINQ to SQL을 이용하여 작성한 게시글 (<http://blog.wekeroad.com/2007/08/30/linq-andgeocoding/>)을 참고하면 된다.

이 기능을 구현하기 위해서 우선 Visual Studio의 "서버 탐색기"를 열고 NerdDinner 데이터베이스를 선택한 후 "함수" 노드를 마우스 오른쪽 버튼으로 클릭하고 "새로 추가 > 스칼라 반환 함수" 메뉴를 선택한다.

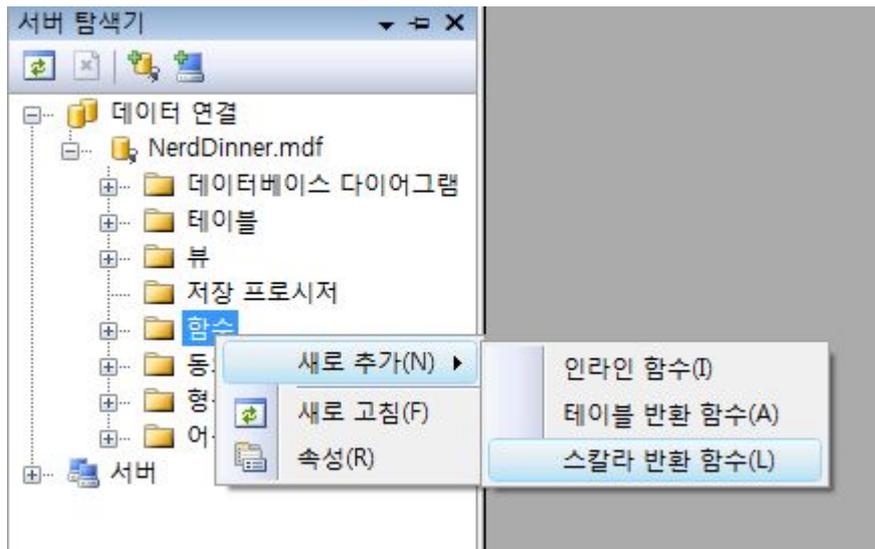


그림 1-135

그런 후 아래의 DistanceBetween 함수의 소스 코드를 붙여넣는다.

```
CREATE FUNCTION [dbo].[DistanceBetween] (@Lat1 as real,
@Long1 as real, @Lat2 as real, @Long2 as real)
RETURNS real
AS
BEGIN
DECLARE @dLat1InRad as float(53);
SET @dLat1InRad = @Lat1 * (PI()/180.0);
DECLARE @dLong1InRad as float(53);
SET @dLong1InRad = @Long1 * (PI()/180.0);
DECLARE @dLat2InRad as float(53);
SET @dLat2InRad = @Lat2 * (PI()/180.0);
DECLARE @dLong2InRad as float(53);
SET @dLong2InRad = @Long2 * (PI()/180.0);
DECLARE @dLongitude as float(53);
SET @dLongitude = @dLong2InRad - @dLong1InRad;
DECLARE @dLatitude as float(53);
SET @dLatitude = @dLat2InRad - @dLat1InRad;
/* Intermediate result a. */
DECLARE @a as float(53);
SET @a = SQUARE (SIN (@dLatitude / 2.0)) + COS (@dLat1InRad)
* COS (@dLat2InRad)
* SQUARE(SIN (@dLongitude / 2.0));
/* Intermediate result c (great circle distance in Radians). */
DECLARE @c as real;
SET @c = 2.0 * ATN2 (SQRT (@a), SQRT (1.0 - @a));
DECLARE @kEarthRadius as real;
/* SET kEarthRadius = 3956.0 miles */
SET @kEarthRadius = 6376.5; /* kms */
DECLARE @dDistance as real;
SET @dDistance = @kEarthRadius * @c;
return (@dDistance);
END
```

다음으로 "NearestDinners"라는 이름으로 테이블 반환 함수를 새로 추가한다.

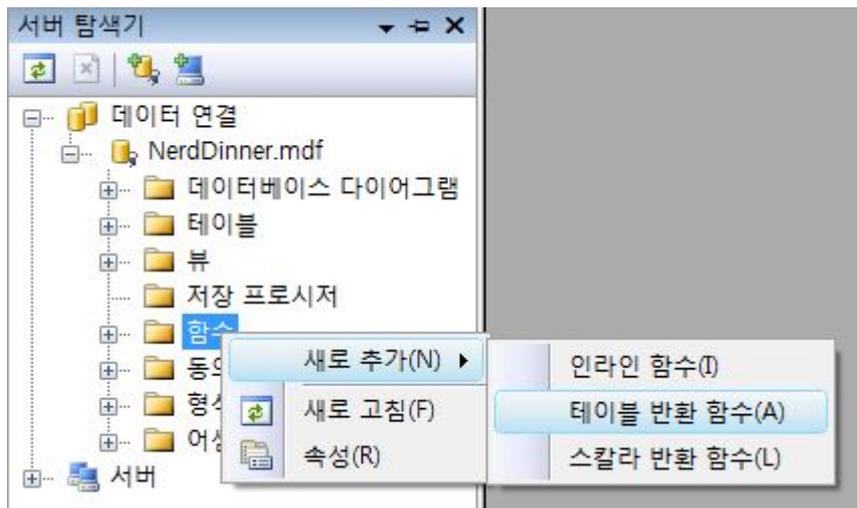


그림 1-136

“NearestDinners” 테이블 반환 함수는 DistanceBetween 함수를 사용하여 현재 위도와 경도를 기준으로 100마일 이내의 지역에서 발생하는 모임 데이터를 가져온다.

```
CREATE FUNCTION [dbo].[NearestDinners]
(
    @lat real,
    @long real
)
RETURNS TABLE
AS
RETURN
SELECT Dinners.DinnerID
FROM Dinners
WHERE dbo.DistanceBetween(@lat, @long, Latitude, Longitude) <100
```

이 함수를 호출하려면 Visual Studio의 솔루션 탐색기에서 #Models 디렉터리의 NerdDinner.dbml 파일을 더블 클릭하여 LINQ to SQL 디자이너를 연다.

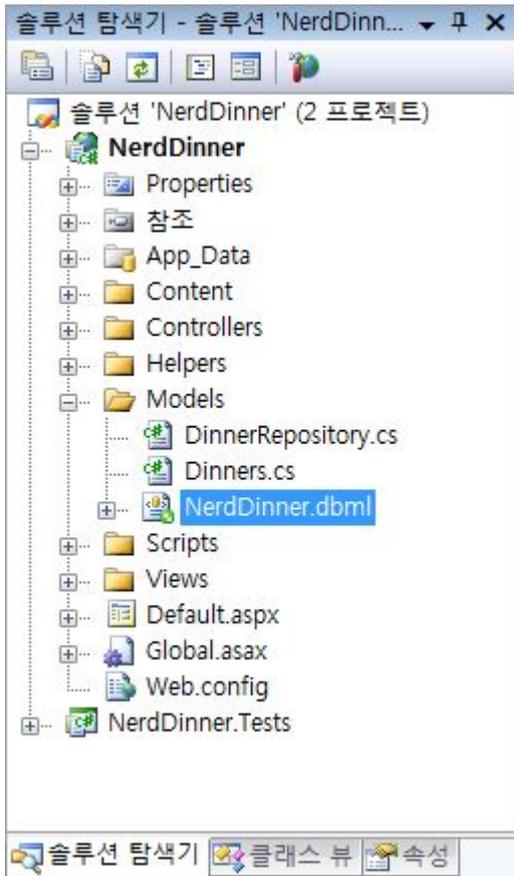


그림 1-137

그런 후 NearestDinners 함수와 DistanceBetween 함수를 LINQ to SQL 디자이너로 끌어다놓으면 이 두 함수가 NerdDinnerDataContext 클래스에 메서드로 추가된다.

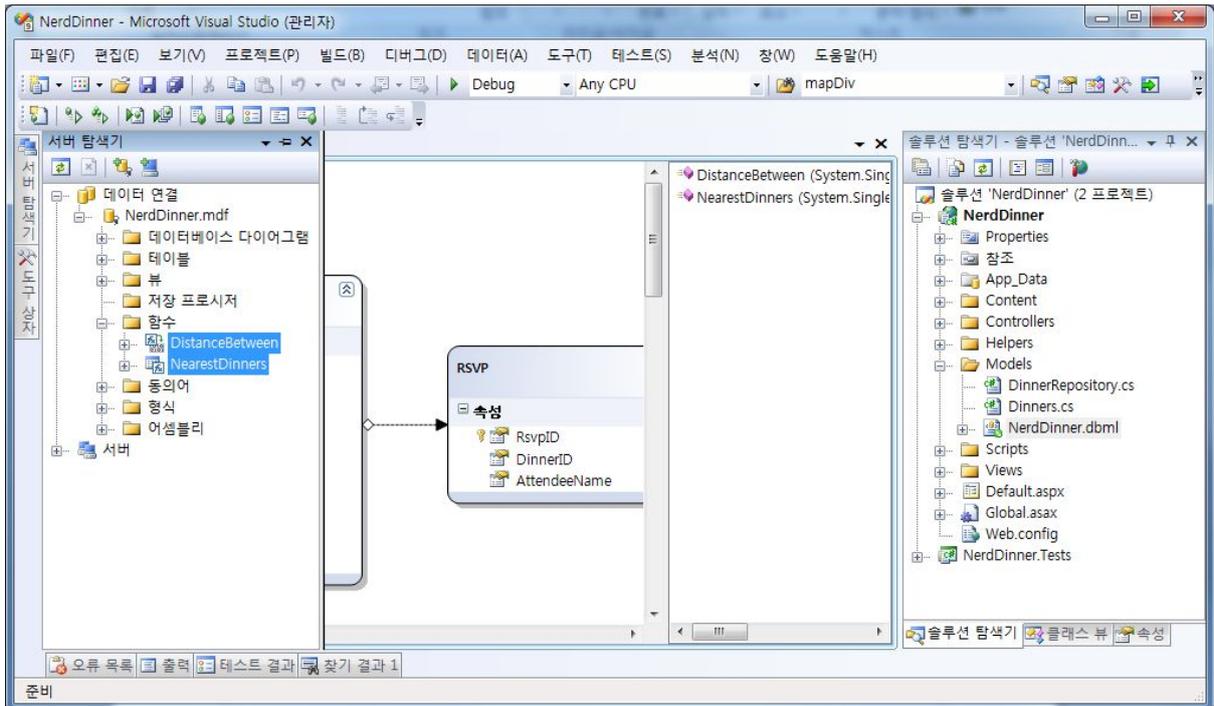


그림 1-138

그런 후 지정된 위치에서 100마일 내에서 발생하는 모임 데이터를 리턴하는 “FindByLocation” 메서드를 DinnerRepository 클래스에 다음과 같이 추가한다.

```
public IQueryable<Dinner> FindByLocation(float latitude, float longitude) {
    var dinners = from dinner in FindUpcomingDinners()
    join i in db.NearestDinners(latitude, longitude)
    on dinner.DinnerID equals i.DinnerID
    select dinner;
    return dinners;
}
```

JSON을 이용한 AJAX 검색 메서드 구현하기

이제 조금 전에 DinnerRepository 클래스에 추가한 FindByLocation() 메서드를 호출하여 지도에 데이터를 표시하는 액션 메서드를 컨트롤러 클래스에 구현해보자. 이 메서드는 검색된 Dinners 객체들을 JSON (JavaScript Object Notation, 역자 주: JSON에 대한 자세한 내용은 <http://json.org> 를 참고하기 바란다) 형식으로 리턴하여 클라이언트 스크립트에서 손쉽게 조작할 수 있도록 구현할 것이다.

먼저 #Controllers 디렉토리를 마우스 오른쪽 버튼을 클릭하여 “추가 > Controller” 메뉴를 선택한 후 “SearchController”라는 이름의 컨트롤러 클래스를 추가한다. 그런 후 새로 추가한 컨트롤러 클래스에 “SearchByLocation”라는 이름의 메서드를 다음과 같이 작성한다.

```
public class JsonDinner {
    public int DinnerID { get; set; }
    public string Title { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public string Description { get; set; }
    public int RSVPCount { get; set; }
}

public class SearchController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
    // AJAX: /Search/SearchByLocation
    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult SearchByLocation(float longitude, float latitude) {
        var dinners = dinnerRepository.FindByLocation(latitude, longitude);
        var jsonDinners = from dinner in dinners
        select new JsonDinner {
            DinnerID = dinner.DinnerID,
            Latitude = dinner.Latitude,
            Longitude = dinner.Longitude,
            Title = dinner.Title,
            Description = dinner.Description,
            RSVPCount = dinner.RSVPs.Count
        };
        return Json(jsonDinners.ToList());
    }
}
```

```
}  
}
```

SearchController의 SearchByLocation 메서드는 내부적으로 DinnerRepository 클래스의 FindByLocation 메서드를 호출하여 가까운 곳의 모임 데이터를 가져온다. 이 때 Dinners 객체를 클라이언트에 직접 리턴하는 대신 JsonDinners 객체를 리턴한다. JsonDinners 클래스는 Dinners 클래스의 속성 중 일부만 노출하고 있다(예를 들자면 보안 상의 이유로 현재 모임에 참여 중인 사람들의 이름은 노출하지 않는다). 또한 Dinners 객체에는 존재하지 않는 RSVPCount 속성을 제공하는데 이 속성은 현재 모임과 관련된 참여자 정보를 가져와 동적으로 참여자의 수를 계산하여 리턴한다.

Controller 기반 클래스의 Json() 메서드는 모임 정보의 컬렉션을 JSON 형식으로 리턴할 수 있다. JSON 형식은 간단한 데이터 구조를 표현하기 위한 문자열 형식이다. 앞서 구현한 액션 메서드에 의해 리턴된 JsonDinners 객체를 JSON 형식으로 표현하면 다음과 같다.

```
[{"DinnerID":53,"Title":"Dinner with the Family","Latitude":47.64312,"Longitude":-122.130609,"Description":"Fun dinner","RSVPCount":2}, {"DinnerID":54,"Title":"Another Dinner","Latitude":47.632546,"Longitude":-122.21201,"Description":"Dinner with Friends","RSVPCount":3}]
```

jQuery를 이용하여 JSON 기반의 AJAX 메서드 호출하기

이제 NerdDinner 애플리케이션의 첫 페이지에서 SearchController 클래스의 SearchByLocation 메서드를 호출할 준비가 완료되었다. 이 메서드를 호출하려면 우선 /Views/Home/Index.aspx 뷰 템플릿을 열고 텍스트 상자와 검색 버튼, 지도와 함께 id가 "dinnerList"인 <DIV> 태그 등을 다음과 같이 추가한다.

```
<h2>Find a Dinner</h2>  
<div id="mapDivLeft">  
<div id="searchBox">  
Enter your location: <%= Html.TextBox("Location") %>  
<input id="search" type="submit" value="Search" />  
</div>  
<div id="theMap">  
</div>  
</div>  
<div id="mapDivRight">  
<div id="dinnerList"></div>  
</div>
```

그런 후 페이지에 다음과 같이 두 개의 자바스크립트 함수를 추가한다.

```
<script type="text/javascript">  
$(document).ready(function () {  
LoadMap();  
});  
$("#search").click(function (evt) {
```

```

var where = jQuery.trim($("#Location").val());
if (where.length < 1)
return;
FindDinnersGivenLocation(where);
});
</script>

```

첫 번째 자바스크립트 함수는 페이지가 처음 로드될 때 지도를 로드하며 두 번째 자바스크립트 함수는 검색 버튼에 자바스크립트의 click 이벤트를 바인딩한다. 이 버튼을 클릭하면 map.js 파일에 구현한 FindDinnersGivenLocation() 자바스크립트 함수를 호출한다.

```

function FindDinnersGivenLocation(where) {
map.Find("", where, null, null, null, null, null, false,
null, null, callbackUpdateMapDinners);
}

```

이 FindDinnersGivenLocation 자바스크립트 함수는 Virtual Earth 컨트롤의 map.Find() 메서드를 호출하여 검색된 위치를 지도의 중앙에 표시한다. Virtual Earth의 지도 서비스가 리턴되면 map.Find() 메서드는 마지막 인수로 전달된 callbackUpdateMapDinners 콜백 함수를 호출한다.

이 callbackUpdateMapDinners() 메서드가 바로 실제로 작업이 일어나는 부분이다. 이 메서드는 jQuery의 \$.post() 메서드를 이용하여 SearchController 클래스의 SearchByLocation() 메서드를 AJAX 방식으로 호출하며 새로 그려질 위치를 표현하는 위도와 경도를 매개 변수로 전달한다. 또한 \$.post() 메서드가 완료되면 호출될 인라인 함수를 정의하고 있으며 SearchByLocation() 메서드에 의해 리턴된 JSON 형식의 모임 데이터가 "dinners" 매개 변수에 전달된다. 그런 후 각각의 모임 데이터에 대해 foreach 구문을 수행하면서 각 모임의 위도와 경도 및 기타 다른 속성들을 이용하여 지도에 핀을 표시한다. 또한 지도의 오른쪽에 HTML 코드를 이용하여 모임의 목록을 만들어낸다. 그런 후 지도의 핀과 모임 목록에 이벤트를 바인딩하여 사용자가 핀이나 모임 제목에 마우스를 가져가면 상세 정보가 표시되도록 한다.

```

function callbackUpdateMapDinners(layer, resultsArray,
places, hasMore, VEErrorMessage) {
$("#dinnerList").empty();
clearMap();
var center = map.GetCenter();
$.post("/Search/SearchByLocation", { latitude: center.Latitude,
longitude: center.Longitude },
function(dinners) {
$.each(dinners, function(i, dinner) {
var LL = new VELatLong(dinner.Latitude,
dinner.Longitude, 0, null);
var RsvpMessage = "";
if (dinner.RSVPCount == 1)
RsvpMessage = "" + dinner.RSVPCount + " RSVP";
else
RsvpMessage = "" + dinner.RSVPCount + " RSVPs";
// 지도에 핀을 추가한다.
LoadPin(LL, '<a href="/Dinners/Details/' + dinner.DinnerID + '>'
+ dinner.Title + '</a>',

```

```

"<p>" + dinner.Description + "</p>" + RsvpMessage);
//지도의 오른쪽에 <ul> 목록을 추가한다.
$('#dinnerList').append($('- 

```

이제 애플리케이션을 실행하고 첫 페이지를 방문해보면 지도가 보여질 것이다. 검색 텍스트 상자에 지역의 이름을 입력하면 그 지역과 가까운 곳에서 주최되는 모임 데이터가 나타난다.

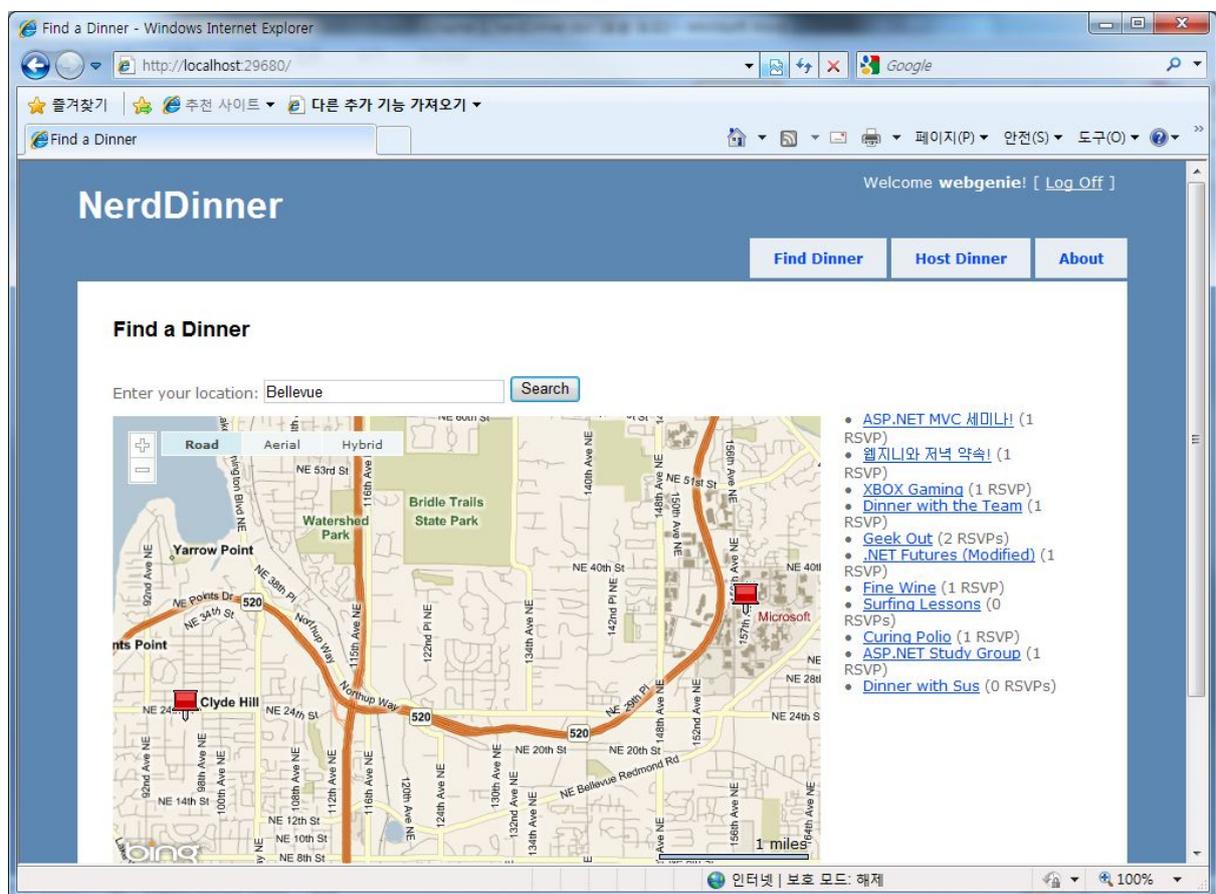


그림 1-139

지도에 나타난 모임 지역에 마우스 포인터를 가져가면 다음 그림과 같이 상세 정보가 나타난다.

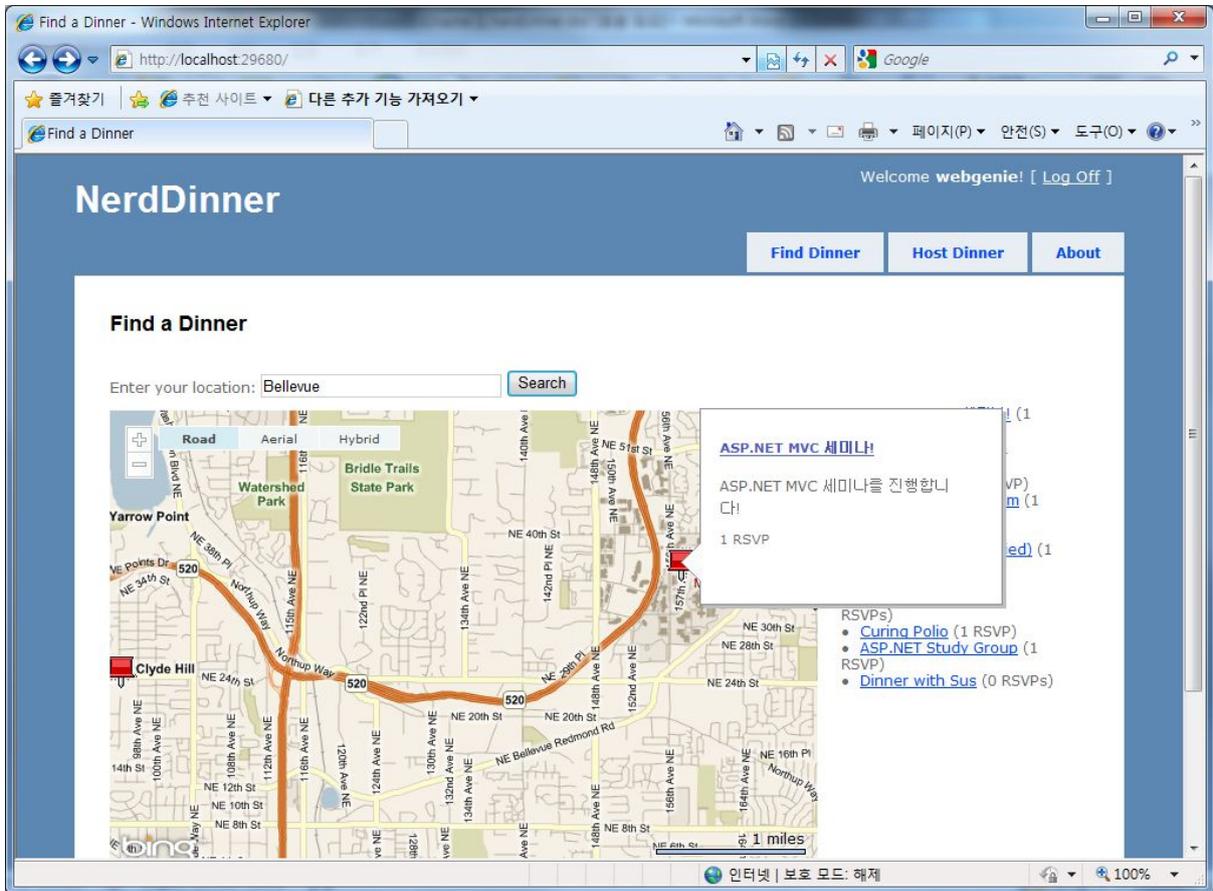


그림 1-140

상세 보기 대화 상자나 지도 오른쪽의 모임 제목을 클릭하면 해당 모임에 대한 상세 보기 페이지로 이동하게 된다.

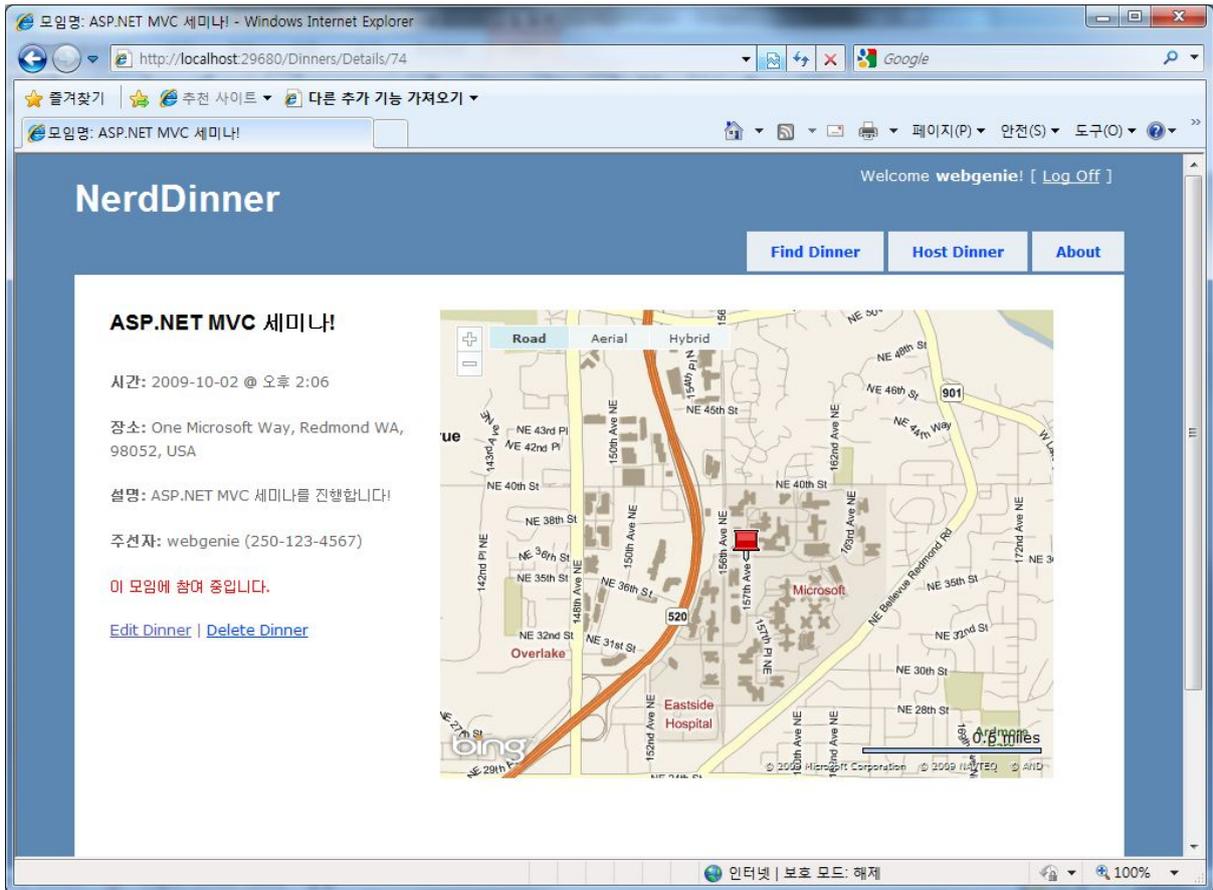


그림 1-141

단위 테스트 수행하기

지금부터는 NerdDinner 애플리케이션의 기능을 검증하고 향후에 애플리케이션을 수정하거나 발전시키는데 있어 안정성을 확보하기 위한 자동화된 단위 테스트를 구현해보자.

단위 테스트가 필요한 이유

어느날 아침 출근길에 지금 구현 중인 애플리케이션에 적절한 방법이 불현듯 생각났다고 가정해보자. 당신은 이 방법으로 애플리케이션을 지금 보다 훨씬 더 개선할 수 있음을 깨달았다. 그 방법이란 코드를 깔끔하게 정리할 수 있는 리팩토링 방법일 수도 있고 새로운 기능을 추가하는 방법일 수도 있으며 현존하는 버그를 수정하는 방법일 수도 있다.

여러분이 사무실에 도착하여 컴퓨터 앞에 앉았을 때 직면할 수 있는 과제는 "이 방법을 도입하는 것이 과연 안전할까?"라는 의문이다. 애플리케이션을 변경했을 때 뭔가 다른 역효과가 발생하지 않을까? 애플리케이션을 수정하는 것은 단 몇 분의 작업으로 완료될만큼 간단하지만 애플리케이션의 동작을 테스트하는데는 몇 시간이 걸린다면 어떻게 해야 할까? 애플리케이션의 전체 시나리오를 테스트 하지 못해서 실 서버에서 동작하는 애플리케이션이 다운되는 일은 없을까? 내가 지

금 수정하는 작업이 정말 효과적인 것일까?

자동화된 단위 테스트(Unit Test)는 여러분이 작성한 코드가 올바르게 동작할 것인지에 대한 염려를 날려버리고 지속적으로 애플리케이션의 기능을 개선할 수 있는 방법을 제공한다. 애플리케이션의 기능을 빠르게 테스트할 수 있는 단위 테스트를 도입하면 코드를 안정적으로 작성할 수 있으며 편안한 마음으로 애플리케이션의 기능을 개선할 수 있다. 또한 유지보수 하기 쉬우며 투자 대비 효율이 극대화되어 보다 오래 애플리케이션 내에서 살아남을 수 있는 코드를 작성하기 위한 솔루션을 제공한다.

ASP.NET MVC 프레임워크는 애플리케이션의 기능에 대한 단위 테스트를 쉽고 자연스럽게 수행할 수 있는 기능을 제공한다. 또한 테스트를 우선으로 애플리케이션을 개발하는 테스트 주도 개발(TDD: Test Driven Development) 방법론을 적용하는데 큰 도움이 된다.

NerdDinner.Tests 프로젝트

처음 NerdDinner 애플리케이션을 개발할 때 우리는 다음 그림과 같이 애플리케이션 개발을 위한 단위 테스트 프로젝트를 함께 생성할 것인지를 묻는 대화 상자를 본 적이 있다.

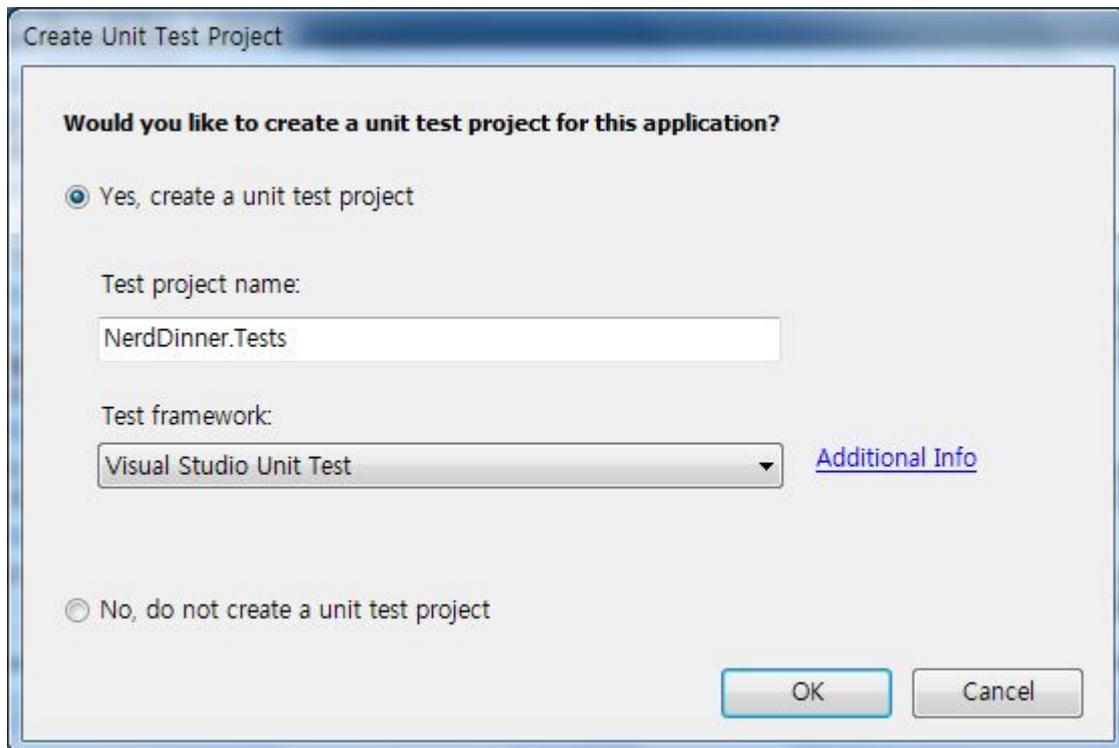


그림 1-142

그 때 당시 우리는 “Yes, create a unit test project” 옵션을 선택했기 때문에 솔루션에는 “NerdDinner.Tests” 프로젝트가 생성되어 있다.

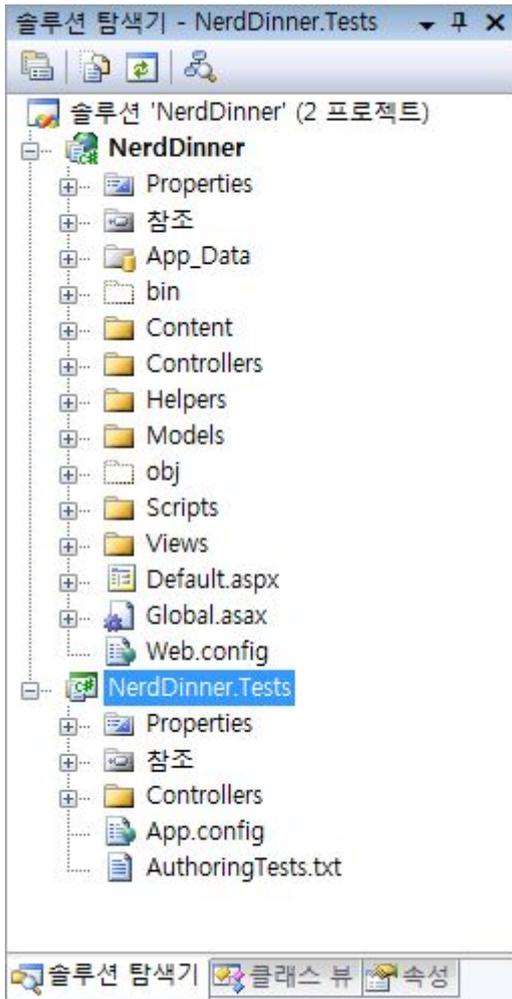


그림 1-143

NerdDinner.Tests 프로젝트는 NerdDinner 애플리케이션 프로젝트의 어셈블리를 참조하고 있어 애플리케이션의 기능을 검증할 수 있는 자동화된 테스트를 손쉽게 추가할 수 있다.

Dinner 모델 클래스에 대한 단위 테스트 생성하기

그러면 애플리케이션의 모델 계층을 구현할 때 추가했던 Dinners 클래스를 검증하기 위한 일련의 테스트를 NerdDinner.Tests 프로젝트에 추가해보자.

우선 모델과 관련된 테스트를 생성할 것이므로 “Models”라는 새 폴더를 NerdDinner.Tests 프로젝트에 생성한다. 그런 후 이 폴더를 마우스 오른쪽 버튼으로 클릭하고 추가 > 새 테스트 메뉴를 선택하면 다음 그림과 같이 “새 테스트 추가” 대화 상자가 나타난다.

이 대화 상자에서 “단위 테스트” 항목을 선택하고 이름은 “DinnerTest.cs”라고 입력한다.

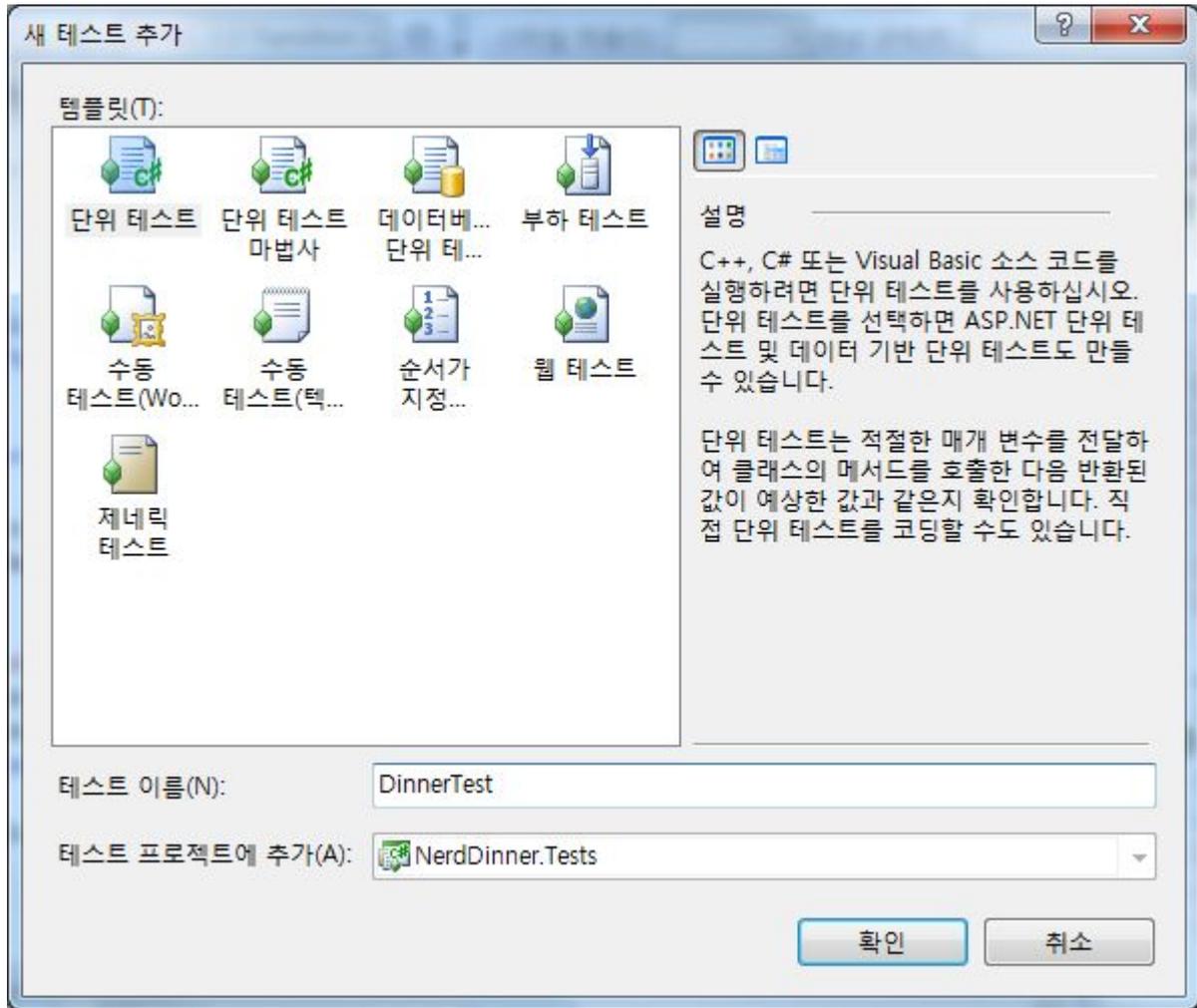


그림 1-144

[확인] 버튼을 클릭하면 Visual Studio는 DinnerTest.cs 파일을 프로젝트에 추가하고 코드 편집기를 통해 파일의 코드를 보여준다.

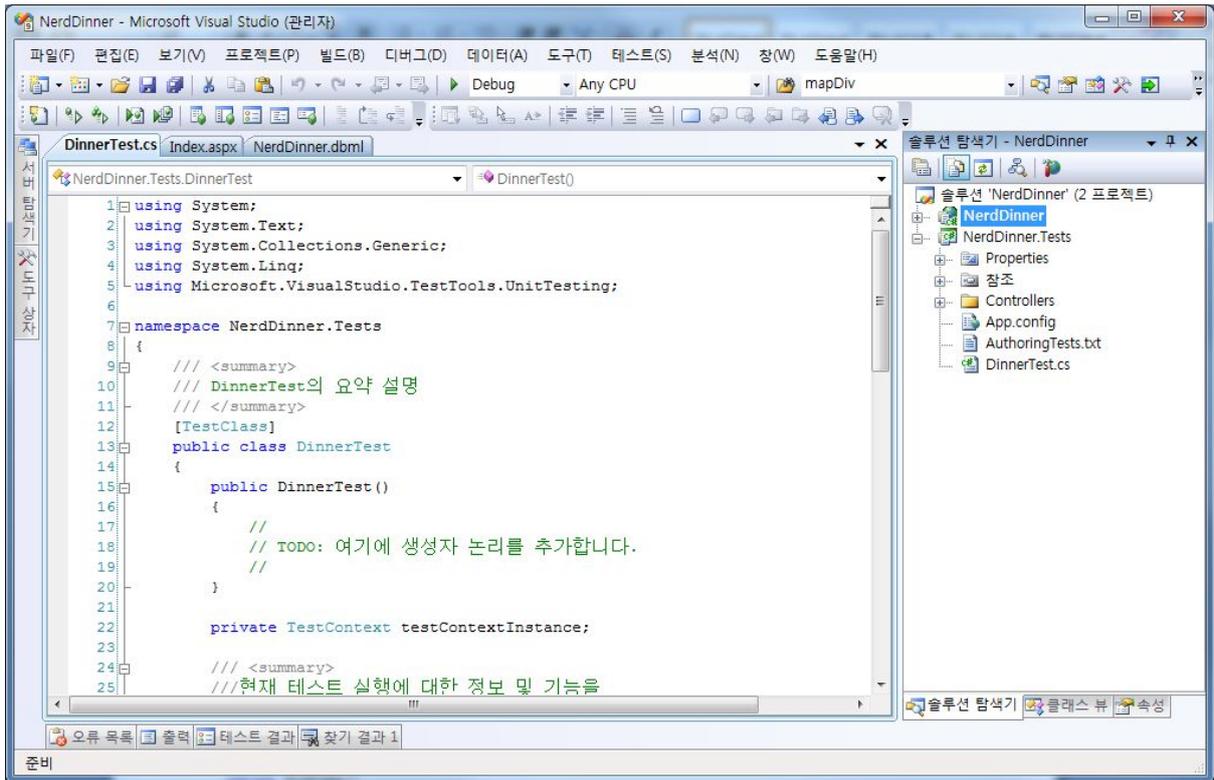


그림 1-145

Visual Studio의 기본 단위 테스트 템플릿은 약간은 복잡해 보이는 기초적인 뼈대 코드를 제공한다. 우선 이 코드를 다음과 같이 정리해보자.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using NerdDinner.Models;
namespace NerdDinner.Tests.Models {
[TestClass]
public class DinnerTest {
}
}
```

DinnerTest 클래스에 적용한 [TestClass] 특성은 이 클래스가 테스트 코드를 가지고 있으며 경우에 따라 테스트를 초기화거나 단계적으로 나누는 등의 코드를 가지고 있음을 표시한다. 테스트 메서드는 이 클래스에 public 메서드로 구현되며 [TestMethod] 특성을 이용하여 표시한다.

다음의 코드는 Dinners 클래스를 검증하기 위해 추가할 두 개의 테스트 메서드를 보여준다. 첫 번째 테스트 메서드는 새 Dinners 클래스의 인스턴스를 생성했을 때 모든 속성 값이 올바르게 설정되지 않은 경우 해당 Dinners 객체가 유효한 객체로 인식되지 않는지를 검증한다. 이와 반대로 두 번째 테스트 메서드는 모든 속성들이 유효한 값을 가졌을 경우 Dinners 객체가 유효한 객체임으로 인식되는지를 검증한다.

```

[TestClass]
public class DinnerTest {
[TestMethod]
public void Dinner_Should_Not_Be_Valid_When_Some_Properties_Incorrect() {
//Arrange
Dinner dinner = new Dinner() {
Title = "Test title",
Country = "USA",
ContactPhone = "BOGUS"
};
// Act
bool isValid = dinner.IsValid;
//Assert
Assert.IsFalse(isValid);
}
[TestMethod]
public void Dinner_Should_Be_Valid_When_All_Properties_Correct() {
//Arrange
Dinner dinner = new Dinner {
Title = "Test title",
Description = "Some description",
EventDate = DateTime.Now,
HostedBy = "ScottGu",
Address = "One Microsoft Way",
Country = "USA",
ContactPhone = "425-703-8072",
Latitude = 93,
Longitude = -92,
};
// Act
bool isValid = dinner.IsValid;
//Assert
Assert.IsTrue(isValid);
}
}
}

```

위의 코드에서 보듯이 테스트 메서드의 이름이 매우 명확하다 (그리고 제법 길다). 이렇게 하는 이유는 우리가 수백에서 수천개의 아주 작은 테스트 메서드를 가질 수 있으며 그런 경우 어떤 테스트 메서드들이 어떤 역할을 하는지 빠르게 판단할 수 있어야 하기 때문이다 (특히 테스트를 실행한 이후 실패한 테스트 메서드의 목록에서 무언가를 찾아야 한다면 더욱 그래야 한다). 테스트 메서드의 이름은 항상 메서드가 테스트하는 기능을 포함해야 한다. 위에서 우리는 “명사_Should_동사” 형태의 이름 규칙을 사용하였다.

테스트 메서드를 구현할 때는 “AAA (Arrange, Act, Assert: 준비, 동작, 확인)” 형식을 사용하고 있다.

준비(Arrange): 테스트할 단위를 설정한다.

실행(Act): 테스트 단위를 실행하고 그 결과를 수집한다.

확인(Assert): 테스트 단위가 올바르게 동작했는지 확인한다.

테스트 메서드를 작성할 때 하나의 테스트 메서드가 너무 많은 작업을 수행하도록 구현하는 것은

좋지 않다. 대신 각각의 테스트 메서드가 한 가지 기능만을 검증하도록 구현해야 한다 (그렇게 하면 테스트에 실패했을 때 어느 지점에서 실패했는지 손쉽게 알 수 있다). 가장 좋은 방법은 각각의 테스트 메서드에서 Assert 클래스를 이용한 결과의 검증을 한 가지만 수행하는 것이다. 만일 테스트 메서드가 여러 개의 결과를 확인하고 있다면 이 결과들이 모두 하나의 기능을 위해 검증할 필요가 있는 것인지 확인해야 한다. 만일 그렇지 않다면 별도의 테스트 메서드를 구현할 것을 권한다.

테스트 실행하기

Visual Studio 2008 Professional(혹은 그 상위 버전)은 IDE 내에서 Visual Studio의 단위 테스트 프로젝트를 실행할 수 있는 테스트 실행기를 내장하고 있다. 테스트 > 실행 > 솔루션의 모든 테스트 메뉴를 선택하면 (혹은 단축키 Ctrl + R, A를 차례로 누르면) 솔루션 내에 구현한 모든 단위 테스트를 실행할 수 있다. 혹은 코드 편집기 내에서 실행하고자 하는 테스트 클래스나 테스트 메서드로 커서를 이동한 후 테스트 > 실행 > 현재 컨텍스트의 테스트 메뉴를 선택하면 (혹은 단축키 Ctrl + R, T를 차례로 누르면) 단위 테스트의 일부를 실행할 수 있다.

그러면 DinnerTest 클래스로 커서를 이동하고 "Ctrl + R, T" 단축키를 눌러 앞서 작성한 테스트를 실행해보자. 그러면 다음 그림과 같이 Visual Studio의 "테스트 결과" 창이 나타나 테스트의 실행 결과를 보여준다.

| 결과 | 테스트 이름 | 프로젝트 | 오류 메시지 |
|--|---|------------------|--------|
| <input checked="" type="checkbox"/> 성공 | Dinner_Should_Not_Be_Valid_When_Some_Properties_Incorrect | NerdDinner.Tests | |
| <input checked="" type="checkbox"/> 성공 | Dinner_Should_Validate_When_All_Properties_Correct | NerdDinner.Tests | |

그림 1-146

=====
 Note: Visual Studio의 테스트 결과 창은 기본적으로 클래스 이름 열을 보여주지 않는다. 테스트 결과 창에 새로운 열을 추가하려면 테스트 결과 창을 마우스 오른쪽 버튼으로 클릭하고 열 추가/제거 메뉴를 선택하면 된다.
 =====

테스트는 단 몇 초면 끝나며 그림에서와 같이 두 개의 테스트가 모두 성공했음을 볼 수 있다. 이제 계속해서 특정 규칙을 검사하는 테스트 메서드를 추가하여 테스트 클래스를 확장해 나갈 수 있으며 Dinners 클래스에 추가한 두 개의 메서드 - IsUserHost() 메서드와 IsUserRegistered() 메서드 - 를 위한 테스트 메서드도 추가할 수 있다. Dinners 클래스를 위한 테스트 메서드를 하나의 테스트 클래스에 구현하면 향후에 비즈니스 규칙이나 유효성 검사가 추가되었을 때 쉽고 빠르게 그에 대한 테스트 로직을 구현할 수 있다. 단위 테스트를 이용하면 Dinners 클래스에 새로운 로직을 추가하고 이 새로운 로직이 기존의 로직과 충돌이 일어나지 않는지를 단 몇 초만에 확인할

수 있다.

테스트 메서드의 이름을 구체적으로 정하면 이 테스트 메서드가 어떤 사항들을 확인하는지 손쉽게 확인할 수 있다. 필자의 경우 도구 > 옵션 메뉴에서 테스트 도구 > 테스트 실행 노드를 열고 실패했거나 결과가 불충분한 단위 테스트 결과를 두 번 클릭하여 테스트의 실패 지점 표시 옵션을 활성화한다. 이렇게 하면 테스트 결과 창에 실패한 테스트를 더블 클릭하여 곧바로 테스트가 실패한 지점으로 이동할 수 있게 된다.

DinnersController 클래스를 위한 단위 테스트 생성하기

이번에는 DinnersController 클래스의 기능을 확인하는 단위 테스트를 구현해보자. 테스트 프로젝트의 "Controllers" 폴더를 마우스 오른쪽 버튼으로 클릭하고 추가 > 새 테스트 메뉴를 선택한 후 "DinnersControllerTest.cs"라는 이름의 "단위 테스트"를 추가한다.

테스트 클래스를 생성했으면 Details() 액션 메서드를 확인하기 위해 다음과 같이 두 개의 테스트 메서드를 추가해보자. 우선 첫 번째 테스트 메서드는 데이터베이스에 존재하는 모임 데이터를 요청한 경우 뷰가 올바르게 렌더링되는지를 확인하며 두 번째 테스트 메서드는 존재하지 않는 모임 데이터를 요청한 경우 "NotFound" 뷰가 올바르게 렌더링되는지 확인한다.

```
[TestClass]
public class DinnersControllerTest {
    [TestMethod]
    public void DetailsAction_Should_Return_View_For_ExistingDinner() {
        // 1단계 준비
        var controller = new DinnersController();
        // 2단계 실행
        var result = controller.Details(1) as ViewResult;
        // 3단계 확인
        Assert.IsNotNull(result, "Expected View");
    }
    [TestMethod]
    public void DetailsAction_Should_Return_NotFoundView_For_BogusDinner() {
        // 1단계 준비
        var controller = new DinnersController();
        // 2단계 실행
        var result = controller.Details(999) as ViewResult;
        // 3단계 확인
        Assert.AreEqual("NotFound", result.ViewName);
    }
}
```

위의 코드는 빌드 시점에는 문제가 없지만 테스트를 실행해 보면 다음 그림과 같이 테스트에 실패하게 된다.

| 결과 | 테스트 이름 | 프로젝트 | 오류 메시지 |
|----|--|------------------|--------------------|
| 실패 | DetailsAction_Should_Return_View_For_ExistingDinner | NerdDinner.Tests | 테스트 메서드 NerdDinner |
| 실패 | DetailsAction_Should_Return_NotFoundView_For_BogusDinner | NerdDinner.Tests | 테스트 메서드 NerdDinner |

그림 1-147

오류 메시지를 읽어보면 DinnerRepository 클래스가 데이터베이스 연결에 실패했기 때문에 오류가 발생했음을 알 수 있다. NerdDinner 애플리케이션은 WApp_Data 폴더에 저장된 SQL Express 데이터베이스 파일에 연결하기 위한 연결 문자열을 사용하고 있는데 NerdDinner.Tests 프로젝트는 애플리케이션 프로젝트와는 다른 디렉터리에서 실행되기 때문에 연결 문자열에 지정된 경로를 사용할 수 없기 때문이다.

이 경우 SQL Express 데이터베이스 파일을 테스트 프로젝트에 복사하고 관련된 연결 문자열을 App.config 파일에 추가하여 문제를 해결할 수도 있다. 이렇게 하면 테스트 프로젝트는 무사히 실행될 것이다.

그러나 단위 테스트 프로젝트가 실제로 데이터베이스에 연결되는 것은 다음과 같은 면에서 단점으로 작용한다.

- 단위 테스트가 빠르게 실행되지 못한다. 단위 테스트에 시간이 오래 걸린다면 아마도 여러분은 단위 테스트를 자주 수행하고 싶지 않을 것이다. 단위 테스트가 수 초 내로 끝나서 마치 빌드하는 것처럼 테스트도 자연스럽게 실행할 수 있다면 이상적일 것이다.
- 테스트의 준비나 제거 로직을 복잡하게 만들 수 있다. 여러분은 각각의 테스트가 서로 독립적으로 (부작용이나 의존성 없이) 동작하기를 바랄 것이다. 테스트가 실제 데이터베이스를 대상으로 동작한다면 테스트를 실행할 때마다 상태를 점검하고 다시 초기화하는 작업이 필요하다.

그러면 이와 같은 문제를 우회하여 테스트 시에 실제 데이터베이스가 필요치 않도록 하기 위한 “의존성 주입 (Dependency Injection, 이하 DI)” 패턴에 대해 알아보자.

의존성 주입 (Dependency Injection) 패턴

현재 DinnersController 클래스는 DinnerRepository 클래스와 강하게 “연결되어” (Tightly-Coupled) 있다. 여기서 말하는 “연결”이란 어떤 클래스가 작업을 수행하기 위해 다른 클래스에 의존하고 있음을 의미한다.

```
public class DinnersController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    //
```

```
// GET: /Dinners/Details/5
public ActionResult Details(int id) {
    Dinner dinner = dinnerRepository.FindDinner(id);
    if (dinner == null)
        return View("NotFound");
    return View(dinner);
}
```

DinnersController 클래스는 DinnerRepository 클래스에 매우 의존적이며 DinnerRepository 클래스는 데이터베이스 연결을 필요로 하기 때문에 DinnersController 클래스의 액션 메서드들을 테스트 하기 위해서도 데이터베이스 연결이 필요하다.

“DI” 패턴을 이용하면 이와 같은 문제를 피할 수 있다. “의존성 주입” 패턴은 (데이터베이스 액세스를 제공하는 저장소 클래스같은) 클래스에 대한 의존성을 다른 클래스 내에서 묵시적으로 생성하지 않는다. 대신 이 의존성은 해당 클래스를 사용하는 클래스의 생성자 매개 변수를 이용하여 명시적으로 전달된다. 만일 이 의존성이 인터페이스를 이용해 정의되었다면 우리는 단위 테스트를 위해 “가상의” 의존 객체를 전달할 수 있는 유연함을 얻을 수 있다. 따라서 실제로 데이터베이스에 대한 연결이 필요없는 테스트만을 위한 의존 클래스를 구현할 수 있다.

그러면 DI 패턴을 실제로 적용하기 위해 DinnersController 클래스 내에 DI 패턴을 구현해보자.

IDinnerRepository 인터페이스 추출하기

DI 패턴을 구현하기 위해 가장 먼저 해야 할 일은 컨트롤러 클래스들이 모임 데이터를 조회하거나 수정하기 위해 필요한 기능을 캡슐화하는 IDinnerRepository 인터페이스를 생성하는 일이다.

₩Models 디렉토리를 마우스 오른쪽 버튼을 클릭하고 [추가 > 새 항목] 메뉴를 선택하여 IDinnerRepository.cs 라는 이름의 인터페이스 파일을 새로 추가하여 인터페이스를 직접 정의할 수도 있다.

그러나 Visual Studio Professional (혹은 그 상위 버전)에 내장된 리팩터링 도구를 이용하면 이미 구현된 DinnerRepository 클래스로부터 인터페이스를 자동으로 추출할 수도 있다. Visual Studio를 이용해서 인터페이스를 추출하려면 코드 편집기에 DinnerRepository 클래스 파일을 열고 마우스 오른쪽 버튼으로 클릭한 후 리팩터링 > 인터페이스 추출 메뉴를 선택한다.

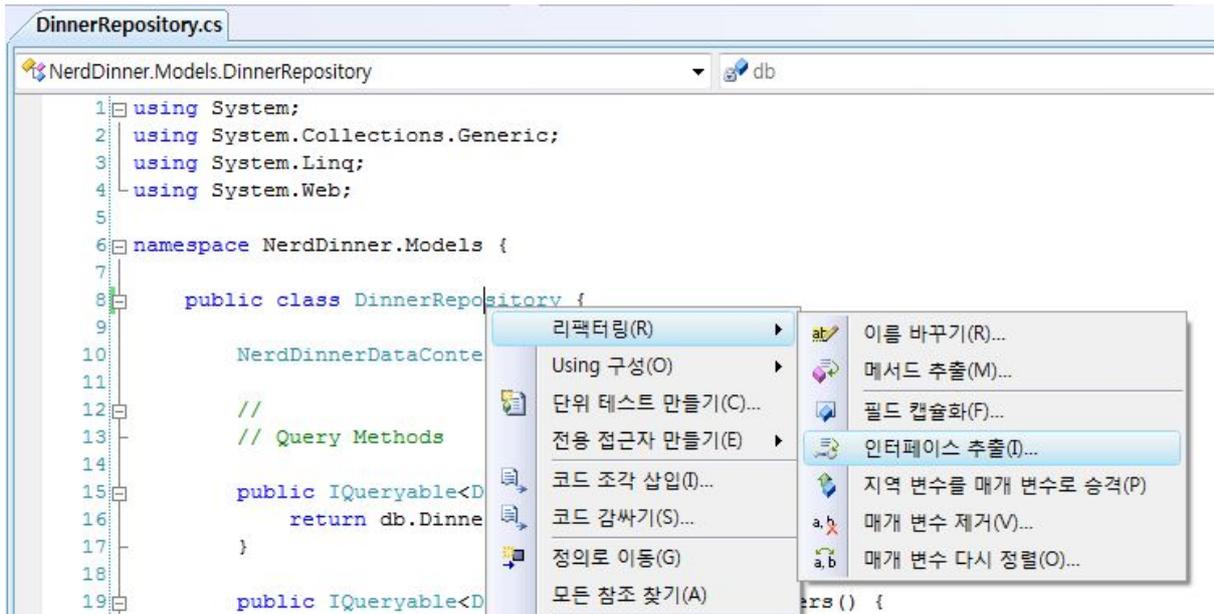


그림 1-148

그러면 “인터페이스 추출” 대화 상자가 나타나 생성할 인터페이스의 이름을 묻게 된다. 이 대화 상자는 기본적으로 IDinnerRepository라는 이름을 제공하며 DinnerRepository 클래스의 모든 public 메서드를 인터페이스 멤버로 선택해준다.

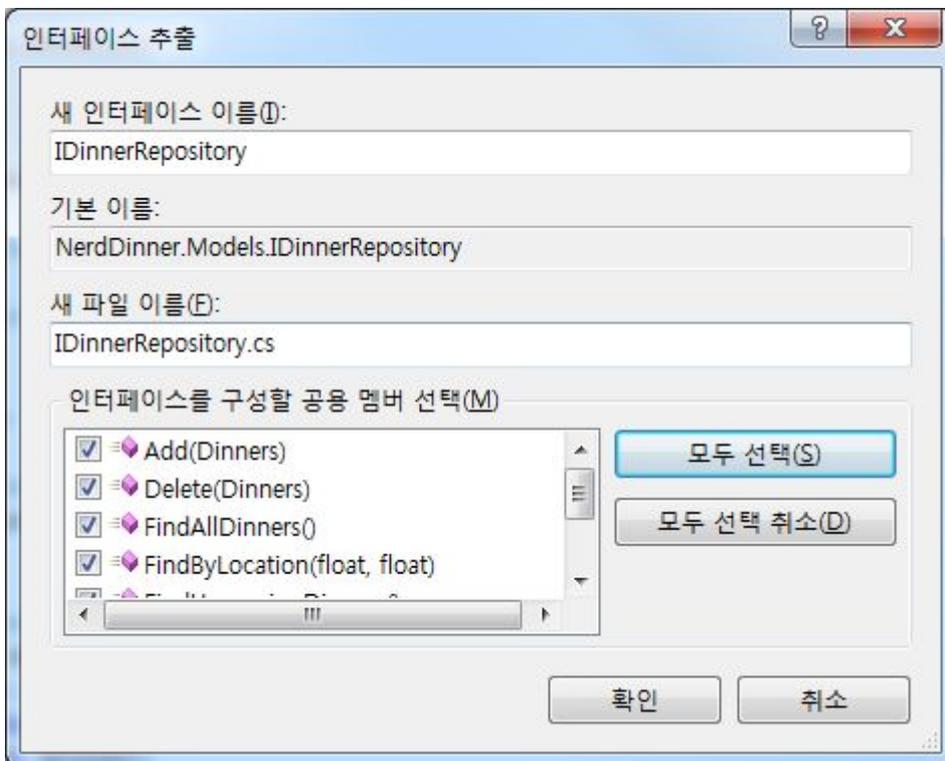


그림 1-149

“확인” 버튼을 클릭하면 Visual Studio는 다음과 같은 IDinnerRepository 인터페이스를 애플리케이션에 추가해준다.

```
public interface IDinnerRepository {
    IQueryable<Dinner> FindAllDinners();
    IQueryable<Dinner> FindByLocation(float latitude, float longitude);
    IQueryable<Dinner> FindUpcomingDinners();
    Dinner GetDinner(int id);
    void Add(Dinner dinner);
    void Delete(Dinner dinner);
    void Save();
}
```

또한 DinnerRepository 클래스는 다음과 같이 인터페이스를 구현하는 클래스로 수정된다.

```
public class DinnerRepository : IDinnerRepository {
    ...
}
```

DinnersController 클래스의 생성자를 이용한 DI 패턴 구현하기

이제 DinnersController 클래스가 앞서 새로 추가한 인터페이스를 사용하도록 수정해보자.

현재 DinnersController 클래스는 항상 DinnerRepository 클래스의 인스턴스를 저장하는 “dinnerRepository” 필드를 사용하도록 구현되어 있다.

```
public class DinnersController : Controller {
    DinnerRepository dinnerRepository = new DinnerRepository();
    ...
}
```

이제 “dinnerRepository” 클래스를 DinnerRepository 클래스 타입 대신 IDinnerRepository 인터페이스 타입을 사용하도록 수정한다. 그런 후 두 개의 DinnersController 클래스 생성자 메서드를 추가할 것이다. 이 중 하나는 IDinnerRepository 인터페이스를 매개 변수로 사용한다. 다른 하나는 기본 생성자로 기존의 DinnerRepository 클래스 인스턴스를 그대로 사용한다.

```
public class DinnersController : Controller {
    IDinnerRepository dinnerRepository;
    public DinnersController()
    : this(new DinnerRepository()) {
    }
    public DinnersController(IDinnerRepository repository) {
        dinnerRepository = repository;
    }
    ...
}
```

기본적으로 ASP.NET MVC는 기본 생성자를 이용하여 컨트롤러 클래스의 인스턴스를 생성하기 때

문에 DinnersController 클래스는 런타임에 항상 DinnerRepository 클래스를 사용하게 된다.

이제 단위 테스트 프로젝트를 수정해서 DinnersController 클래스에 추가한 생성자 메서드를 통해 “가상의” 저장소를 전달하도록 한다. 이 가상의 저장소는 실제로 데이터베이스에 연결될 필요가 없으며 메모리 내의 예제 데이터를 사용한다.

FakeDinnerRepository 클래스 구현하기

이제 FakeDinnerRepository 클래스를 구현해보자.

우선은 NerdDinner.Tests 프로젝트에 “Fakes” 디렉터리를 추가하고 (마우스 오른쪽 버튼을 클릭하여 추가 > 클래스 메뉴를 선택하여) FakeDinnerRepository 클래스를 추가하자.

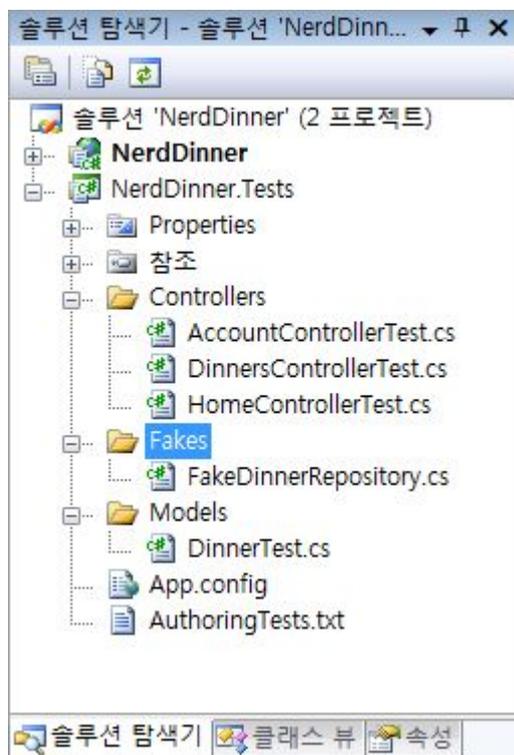


그림 1-150

FakeDinnerRepository 클래스가 IDinnerRepository 인터페이스를 구현하도록 코드를 수정한 후 다음 그림과 같이 “IDinnerRepository 인터페이스 구현” 메뉴를 선택한다.

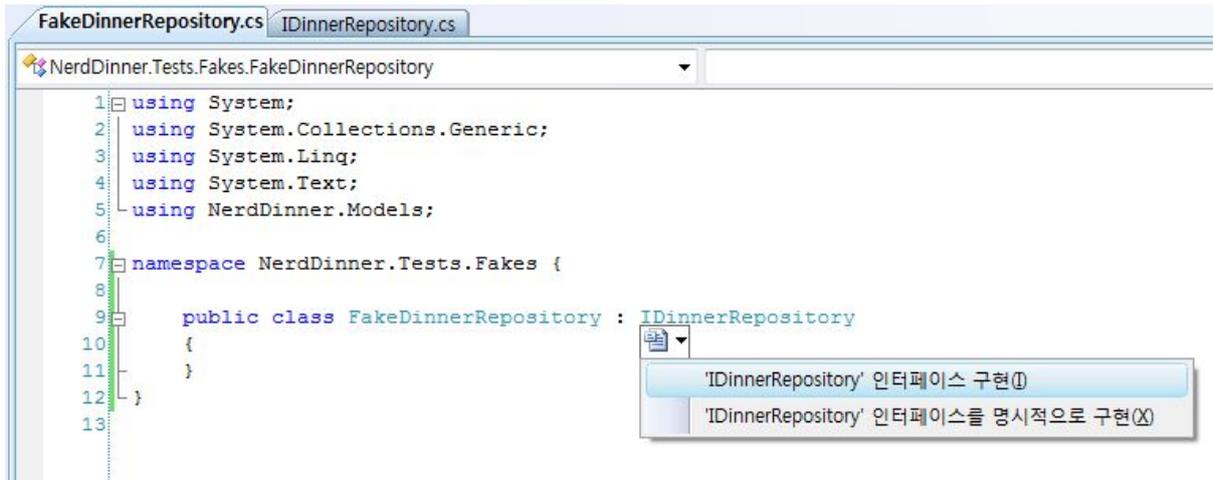


그림 1-151

그러면 Visual Studio가 다음과 같이 IDinnerRepository 인터페이스에 정의된 멤버들을 FakeDinnerRepository 클래스에 추가한다.

```
public class FakeDinnerRepository : IDinnerRepository {
public IQueryable<Dinner> FindAllDinners () {
throw new NotImplementedException ();
}
public IQueryable<Dinner> FindByLocation(float lat, float lon){
throw new NotImplementedException ();
}
public IQueryable<Dinner> FindUpcomingDinners () {
throw new NotImplementedException ();
}
public Dinner GetDinner(int id) {
throw new NotImplementedException ();
}
public void Add(Dinner dinner) {
throw new NotImplementedException ();
}
public void Delete(Dinner dinner) {
throw new NotImplementedException ();
}
public void Save () {
throw new NotImplementedException ();
}
}
```

이제 FakeDinnerRepository 클래스가 메모리 상의 List<Dinner> 컬렉션을 이용하여 작업을 수행 하도록 구현하고 생성자 메서드의 매개 변수로 List<Dinner> 컬렉션을 전달할 수 있도록 수정해 보자.

```
public class FakeDinnerRepository : IDinnerRepository {
private List<Dinner> dinnerList;
public FakeDinnerRepository(List<Dinner> dinners) {
dinnerList = dinners;
}
public IQueryable<Dinner> FindAllDinners () {
```

```

return dinnerList.AsQueryable();
}
public IQueryable<Dinner> FindUpcomingDinners() {
return (from dinner in dinnerList
where dinner.EventDate > DateTime.Now
select dinner).AsQueryable();
}
public IQueryable<Dinner> FindByLocation(float lat, float lon) {
return (from dinner in dinnerList
where dinner.Latitude == lat && dinner.Longitude == lon
select dinner).AsQueryable();
}
public Dinner GetDinner(int id) {
return dinnerList.SingleOrDefault(d => d.DinnerID == id);
}
public void Add(Dinner dinner) {
dinnerList.Add(dinner);
}
public void Delete(Dinner dinner) {
dinnerList.Remove(dinner);
}
public void Save() {
foreach (Dinner dinner in dinnerList) {
if (!dinner.IsValid)
throw new ApplicationException("Rule violations");
}
}
}
}

```

우리가 구현한 FakeDinnerRepository 클래스는 이제 데이터베이스에 연결될 필요가 없으며 대신 메모리 상의 Dinners 객체의 컬렉션을 이용하여 동작하게 되었다.

단위 테스트에 FakeDinnerRepository 클래스를 사용하기

이제 앞서 데이터베이스 연결 문제로 실패했던 DinnersController 클래스를 위한 단위 테스트 코드로 돌아와 이 코드를 다음과 같이 FakeDinnerRepository 클래스를 사용하도록 수정하자.

```

[TestClass]
public class DinnersControllerTest {
List<Dinner> CreateTestDinners() {
List<Dinner> dinners = new List<Dinner>();
for (int i = 0; i < 101; i++) {
Dinner sampleDinner = new Dinner() {
DinnerID = i,
Title = "Sample Dinner",
HostedBy = "SomeUser",
Address = "Some Address",
Country = "USA",
ContactPhone = "425-555-1212",
Description = "Some description",
EventDate = DateTime.Now.AddDays(i),
Latitude = 99,
Longitude = -99
};
dinners.Add(sampleDinner);
}
}
}

```

```

}
return dinners;
}
DinnersController CreateDinnersController() {
var repository = new FakeDinnerRepository(CreateTestDinners());
return new DinnersController(repository);
}
[TestMethod]
public void DetailsAction_Should_Return_View_For_Dinner() {
// Arrange
var controller = CreateDinnersController();
// Act
var result = controller.Details(1);
// Assert
Assert.IsInstanceOfType(result, typeof(ViewResult));
}
[TestMethod]
public void DetailsAction_Should_Return_NotFoundView_For_BogusDinner() {
// Arrange
var controller = CreateDinnersController();
// Act
var result = controller.Details(999) as ViewResult;
// Assert
Assert.AreEqual("NotFound", result.ViewName);
}
}
}

```

이제 테스트를 실행해보면 다음과 같이 두 테스트가 모두 성공하는 것을 볼 수 있다.

The screenshot shows a test runner window titled '테스트 결과' (Test Results). The status bar indicates '테스트 실행 완료 결과: 2/2개 성공 확인된 항목: 0' (Test execution complete, 2/2 tests passed, 0 items confirmed). The main table lists the following test results:

| 결과 | 테스트 이름 | 프로젝트 | 오류 메시지 |
|----|--|------------------|--------|
| 성공 | DetailsAction_Should_Return_View_For_Dinner | NerdDinner.Tests | |
| 성공 | DetailsAction_Should_Return_NotFoundView_For_BogusDinner | NerdDinner.Tests | |

그림 1-152

무엇보다 이 테스트들은 단 몇 초만에 수행되며 추가로 복잡한 로직이 필요하지도 않다. 이제 우리는 데이터베이스 연결이 없이도 DinnersController 클래스의 모든 액션 메서드에 대한 단위 테스트를 실행할 수 있게 되었다.

=====

참고: DI 프레임워크

(이 책에서처럼) 수동으로 DI 패턴을 구현하는 것도 나쁘지 않지만 애플리케이션 내의 의존성과 컴포넌트의 수가 늘어날수록 코드의 유지보수는 더욱 어렵게 된다.

그러나 이미 .NET 프레임워크 상에서 DI 패턴을 구현한 다양한 프레임워크들이 존재하며 이들을 이용하면 유연하게 객체 사이의 의존성을 관리할 수 있다. 이 프레임워크들은 간혹 “제어 전환 (Inversion of Control, 이하 IoC)” 컨테이너라고 불리기도 하며 응용 프로그램 설정을 통해 객체간

의 의존성을 런타임에 지정하거나 전달할 수 있는 매커니즘을 제공한다 (대부분 생성자 주입 방법을 사용한다). .NET 환경에서 가장 많이 사용되는 DI / IoC 프레임워크로는 AutoFac, Ninject, Spring.NET, StructureMap 그리고 Windsor 등이 있다.

ASP.NET MVC는 개발자가 컨트롤러 클래스의 해석과 인스턴스 과정에 관여하여 DI / IoC 프레임워크를 손쉽게 적용할 수 있는 확장 가능한 API를 제공하고 있다. DI / IoC 프레임워크를 사용하면 우리가 구현한 DinnersController 클래스의 생성자를 제거하여 DinnerRepository 클래스와의 의존성을 완벽하게 제거할 수 있다.

이 책에서는 NerdDinner 애플리케이션의 구현에 DI / IoC 프레임워크를 도입하지는 않는다. 그러나 향후에 NerdDinner 애플리케이션의 코드와 기능이 증가한다면 이들의 활용을 고려해 보아야 한다.

=====

Edit 액션 메서드에 대한 단위 테스트 구현하기

이번에는 DinnersController 클래스의 데이터 수정 기능을 검증하는 단위 테스트를 구현해보자. 우선 HTTP GET 방식의 Edit 액션 메서드를 구현한 코드는 아래와 같다.

```
//  
// GET: /Dinners/Edit/5  
[Authorize]  
public ActionResult Edit(int id) {  
    Dinner dinner = dinnerRepository.GetDinner(id);  
    if (!dinner.IsHostedBy(User.Identity.Name))  
        return View("InvalidOwner");  
    return View(new DinnerFormViewModel(dinner));  
}
```

먼저 유효한 Dinners 객체를 요청했을 때 DinnerFormViewModel 객체가 렌더링하려는 뷰가 올바르게 렌더링되는지 검증하기 위한 테스트 메서드를 다음과 같이 구현한다.

```
[TestMethod]  
public void EditAction_Should_Return_View_For_ValidDinner() {  
    // Arrange  
    var controller = CreateDinnersController();  
    // Act  
    var result = controller.Edit(1) as ViewResult;  
    // Assert  
    Assert.IsInstanceOfType(result.ViewData.Model,  
        typeof(DinnerFormViewModel));  
}
```

그러나 테스트를 실행해보면 Edit() 액션 메서드가 Dinner.IsHostedBy() 메서드에 전달되는 User.Identity.Name 속성에 액세스할 때 널 참조 예외(Null Reference Exception)가 발생하는 것을 알 수 있다.

Controller 기반 클래스의 User 객체는 현재 로그인 한 사용자를 캡슐화하며 ASP.NET MVC가 런타임에 컨트롤러 클래스의 인스턴스를 생성될 때 만들어진다. 현재 DinnersController 클래스는 웹 서버 환경에서 테스트하는 것이 아니기 때문에 User 객체가 존재하지 않는 것이다 (따라서 널 참조 예외가 발생한다).

User.Identity.Name 속성을 가상화하기

가상화 프레임워크(Mocking Framework)는 의존적인 객체의 가상화된 버전을 동적으로 생성함으로써 테스트를 보다 손쉽게 진행할 수 있도록 해준다. 예를 들어 DinnersController 클래스의 Edit 액션 메서드를 테스트하기 위해 가상화 프레임워크를 이용하여 DinnersController 클래스가 가상의 사용자 이름을 참조할 수 있도록 User 객체를 동적으로 생성할 수 있다. 이렇게 함으로써 앞서와 같이 테스트를 실행할 때 널 참조 예외가 발생하는 상황을 피할 수 있다.

이미 ASP.NET MVC와 함께 사용할 수 있는 다양한 가상화 프레임워크들이 마련되어 있다 (이들의 목록은 <http://www.mockframeworks.com>에서 찾을 수 있다). 이 책에서는 NerdDinner 애플리케이션의 테스트를 위해 오픈 소스 가상화 프레임워크인 "Moq"를 이용한다. Moq 프레임워크는 ["http://www.mockframeworks.com/moq"](http://www.mockframeworks.com/moq)에서 무료로 다운로드할 수 있다.

일단 Moq 프레임워크를 다운로드한 후에는 다음과 같이 NerdDinner.Tests 프로젝트에 참조로 추가해야 한다.

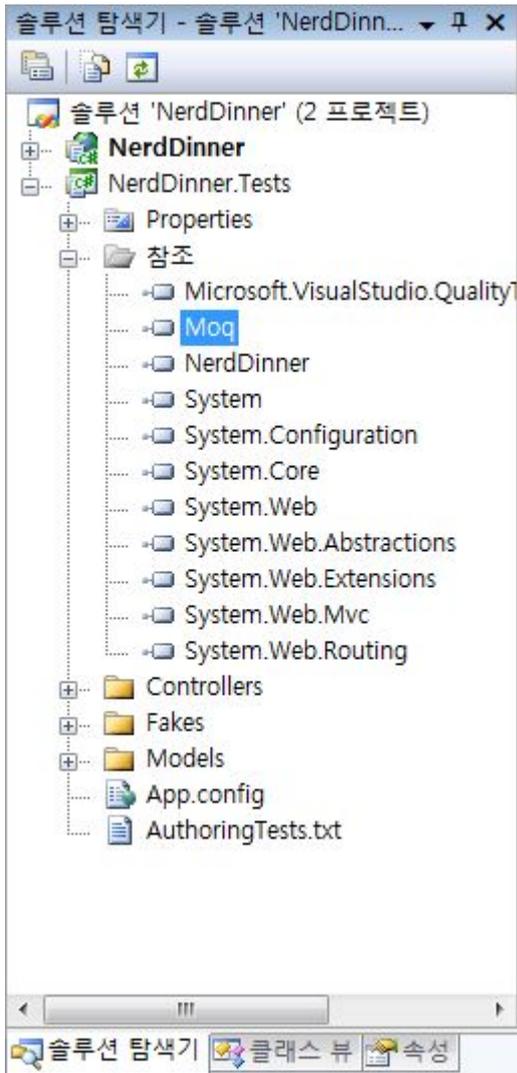


그림 1-153

그런 후 테스트 클래스에 “CreateDinnersControllerAs(사용자이름)” 형식의 헬퍼 메서드를 재정의한다. 이 메서드는 다음과 같이 사용자의 이름을 전달받아 DinnersController 클래스 인스턴스의 `User.Identity.Name` 속성에 대입한다.

```
DinnersController CreateDinnersControllerAs(string userName) {
var mock = new Mock<ControllerContext>();
mock.SetupGet(p => p.HttpContext.User.Identity.Name).Returns(userName);
mock.SetupGet(p => p.HttpContext.Request.IsAuthenticated).Returns(true);
var controller = CreateDinnersController();
controller.ControllerContext = mock.Object;
return controller;
}
```

위의 코드에서는 Moq 프레임워크의 Mock 객체를 이용하여 가상의 ControllerContext 객체 (ASP.NET MVC가 User, Request, Response, Session 등의 런타임 객체를 노출하기 위해 Controller 클래스에 전달하는 객체이다)를 생성한다. 그런 후 Mock 객체의 “SetupGet” 메서드를 호출하여 ControllerContext 클래스의 `HttpContext.User.Identity.Name` 속성이 헬퍼 메서드에 전달된 사용자

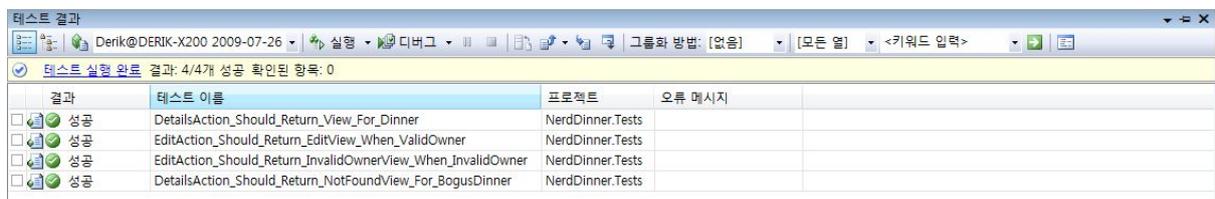
이름을 리턴하도록 한다.

우리는 ControllerContext 클래스의 모든 속성과 메서드를 가상화할 수 있으며 이를 증명하기 위해 Request.IsAuthenticated 속성(실제로는 테스트에서 사용되지 않지만 Request 객체를 가상화하는 방법을 보여주기 위해 사용하였다)을 가상화하기 위한 SetupGet() 메서드를 호출하고 있다. 그런 후에는 생성된 가상의 ControllerContext 클래스 인스턴스를 헬퍼 메서드가 리턴할 DinnersController 객체에 대입한다.

이제 지금까지 구현한 헬퍼 메서드를 이용하여 데이터 수정 시나리오를 테스트할 코드를 다음과 같이 작성한다.

```
[TestMethod]
public void EditAction_Should_Return_EditView_When_ValidOwner() {
    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");
    // Act
    var result = controller.Edit(1) as ViewResult;
    // Assert
    Assert.IsInstanceOfType(result.ViewData.Model,
        typeof(DinnerFormViewModel));
}
[TestMethod]
public void EditAction_Should_Return_InvalidOwnerView_When_InvalidOwner() {
    // Arrange
    var controller = CreateDinnersControllerAs("NotOwnerUser");
    // Act
    var result = controller.Edit(1) as ViewResult;
    // Assert
    Assert.AreEqual(result.ViewName, "InvalidOwner");
}
```

이제 테스트를 실행해보면 다음과 같이 테스트가 성공적으로 완료되는 것을 볼 수 있다.



The screenshot shows a test runner window titled '테스트 결과' (Test Results). The window displays the following table of test results:

| 결과 | 테스트 이름 | 프로젝트 | 오류 메시지 |
|----|---|------------------|--------|
| 성공 | DetailsAction_Should_Return_View_For_Dinner | NerdDinner.Tests | |
| 성공 | EditAction_Should_Return_EditView_When_ValidOwner | NerdDinner.Tests | |
| 성공 | EditAction_Should_Return_InvalidOwnerView_When_InvalidOwner | NerdDinner.Tests | |
| 성공 | DetailsAction_Should_Return_NotFoundView_For_BogusDinner | NerdDinner.Tests | |

그림 1-154

UpdateModel() 메서드 테스트하기

앞서 HTTP GET 방식의 Edit 액션 메서드를 위한 테스트 코드를 작성하였으므로 이번에는 아래의 HTTP POST 방식의 Edit 액션 메서드를 위한 테스트 코드를 작성해보자.

```
//
// POST: /Dinners/Edit/5
```

```

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Edit (int id, FormCollection collection) {
    Dinner dinner = dinnerRepository.GetDinner(id);
    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");
    try {
        UpdateModel(dinner);
        dinnerRepository.Save();
        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());
        return View(new DinnerFormViewModel(dinner));
    }
}

```

이 액션 메서드는 Controller 기반 클래스의 UpdateModel() 메서드를 사용하고 있는 새로운 테스트 시나리오를 필요로 한다. UpdateModel() 메서드는 폼으로부터 전송된 매개 변수와 Dinners 객체를 바인딩해 주는 메서드이다.

다음의 코드는 UpdateModel() 메서드를 사용하기 위해서 폼으로부터 전송된 매개 변수를 제공하는 방법을 보여준다. 폼으로부터 전송된 매개 변수들을 흉내내기 위해서는 FormCollection 객체를 생성하고 값을 추가하여 Controller 클래스의 "ValueProvider" 속성에 대입하면 된다.

첫 번째 테스트 메서드는 정상적으로 수정된 데이터를 저장한 경우 브라우저가 상세 보기 페이지로 이동하는지 여부를 확인한다. 두 번째 테스트 메서드는 올바르지 않은 입력값이 전달된 경우 액션 메서드가 수정 페이지를 다시 표시하고 올바른 에러 메시지를 보여주는지 여부를 확인한다.

```

[TestMethod]
public void EditAction_Should_Redirect_When_Update_Successful() {
    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");
    var formValues = new FormCollection() {
        { "Title", "Another value" },
        { "Description", "Another description" }
    };
    controller.ValueProvider = formValues.ToValueProvider();
    // Act
    var result = controller.Edit(1, formValues) as RedirectToRouteResult;
    // Assert
    Assert.AreEqual("Details", result.RouteValues["Action"]);
}
[TestMethod]
public void EditAction_Should_Redisplay_With_Errors_When_Update_Fails() {
    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");
    var formValues = new FormCollection() {
        { "EventDate", "Bogus date value!!!" }
    };
    controller.ValueProvider = formValues.ToValueProvider();
    // Act
    var result = controller.Edit(1, formValues) as ViewResult;
    // Assert
}

```

```
Assert.IsNotNull(result, "Expected redisplay of view");
Assert.IsTrue(result.ViewData.ModelState.Count > 0, "Expected errors");
}
```

전체 테스트 코드를 적용하기

지금까지 컨트롤러 클래스를 테스트하는데 필요한 핵심 개념들에 대해 살펴보았다. 이 방법들을 이용하면 애플리케이션의 테스트에 필요한 수백개의 테스트 메서드를 간단하게 구현해 낼 수 있다.

컨트롤러와 모델 객체가 실제 데이터베이스를 필요로 하지 않기 때문에 테스트 메서드들은 매우 빠르게 동작하며 실행하기도 쉽다. 우리는 수백개의 테스트 메서드를 수 초 안에 실행할 수 있으며 우리가 변경한 내용이 기존의 애플리케이션에 영향을 미치는지 여부를 손쉽게 판단할 수 있다. 단위 테스트를 이용하면 편리하게 애플리케이션을 지속적으로 개선하거나 리팩터링 혹은 다듬을 수 있다.

테스트는 이 장의 마지막 토픽으로 다루어졌지만 그렇다고 테스트가 개발 과정의 막바지에 수행해야 하는 작업은 아니다. 실상은 개발 과정에서 최대한 일찍 자동화된 테스트 코드를 작성해야 한다. 그렇게 하면 개발한 코드에 대한 즉각적인 피드백을 얻을 수 있으며 애플리케이션의 사용 행태에 대해 조금 더 깊게 생각해 볼 수 있음은 물론 애플리케이션의 계층과 의존성을 명확하게 디자인할 수 있다.

이 책의 마지막 장에서는 테스트 주도 개발 (Test Driven Development, 이하 TDD)에 대해 소개하며 ASP.NET MVC 프로젝트에 TDD를 적용하는 방법에 대해서도 설명한다. TDD는 결과 코드가 올바르게 동작하는지 확인하는 테스트 코드를 먼저 작성하는 과정을 반복하는 개발 방법을 말한다. TDD를 적용하면 각각의 기능을 구현할 때 여러분이 구현해야 할 기능들을 확인하는 테스트 코드를 먼저 작성하게 된다. 일단 테스트 코드를 작성하면 (그리고 그 테스트가 실패했음을 확인하면) 테스트를 통해 확인한 기능을 구현한다. 그 과정에서 여러분은 이 기능이 어떻게 동작하게 될 것인가에 대해 미리 고민하는 시간을 가졌기 때문에 요구 사항에 대해 보다 명확하게 이해하게 될 것이며 가장 좋은 방법으로 기능을 구현하게 될 것이다. 기능을 구현한 후에는 테스트를 다시 실행하고 그것이 올바르게 동작하는지에 대해 즉시 알 수 있게 된다. TDD에 대한 보다 자세한 내용은 제10장을 참고하기 바란다.

NerdDinner 애플리케이션 다시 둘러보기

이제 NerdDinner 애플리케이션의 초기 버전의 개발이 완료되어 웹을 통해 배포할 준비가 되었다.

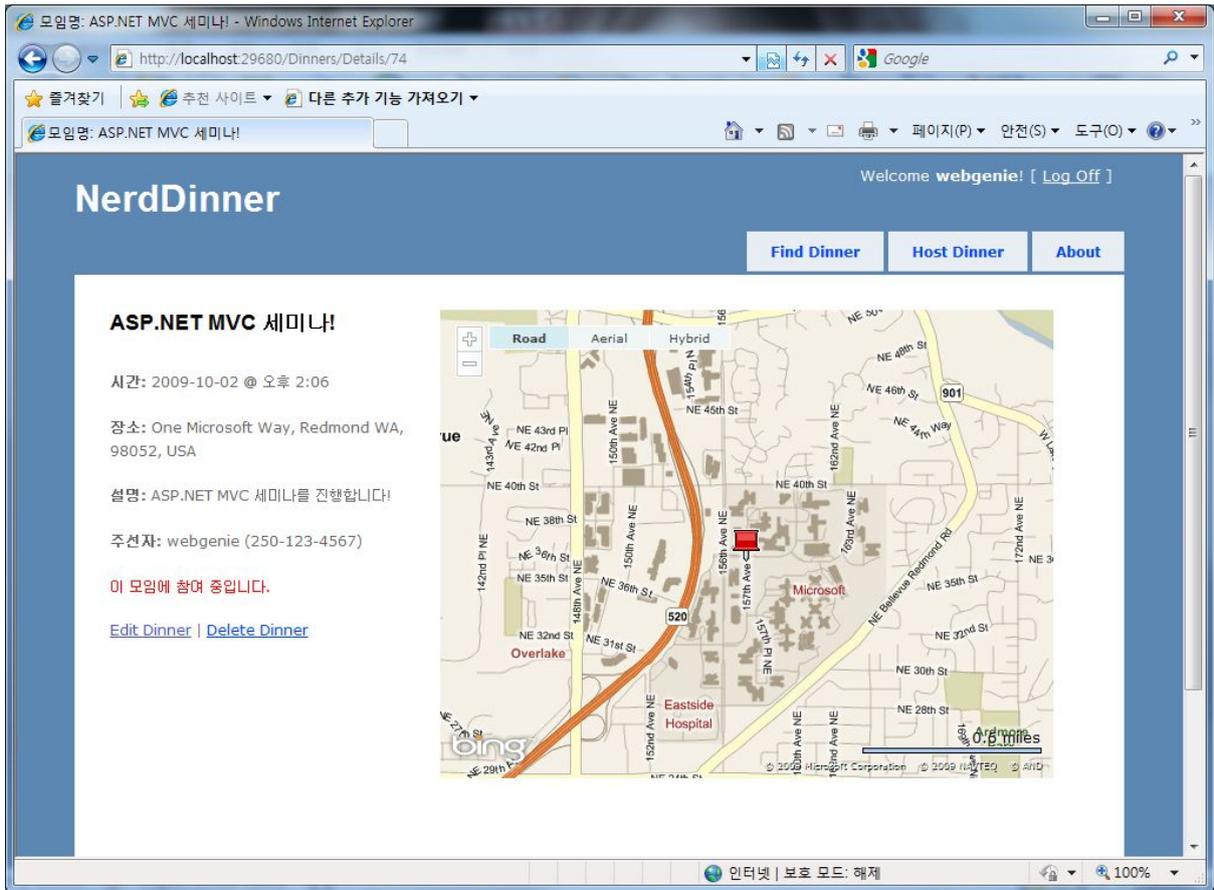


그림 1-155

NerdDinner 애플리케이션은 ASP.NET MVC의 다양한 기능들을 이용하여 구현되었다. 지금까지 소개한 애플리케이션의 개발 과정이 ASP.NET MVC의 핵심 모듈들이 어떻게 동작하는지 이해하며 이들이 애플리케이션 내에서 어떻게 통합되는지를 잘 보여줄 수 있었기를 바란다.

이후의 장들에서는 ASP.NET MVC를 더욱 깊게 살펴볼 것이며 각각의 기능에 대해 보다 상세히 설명할 것이다.

