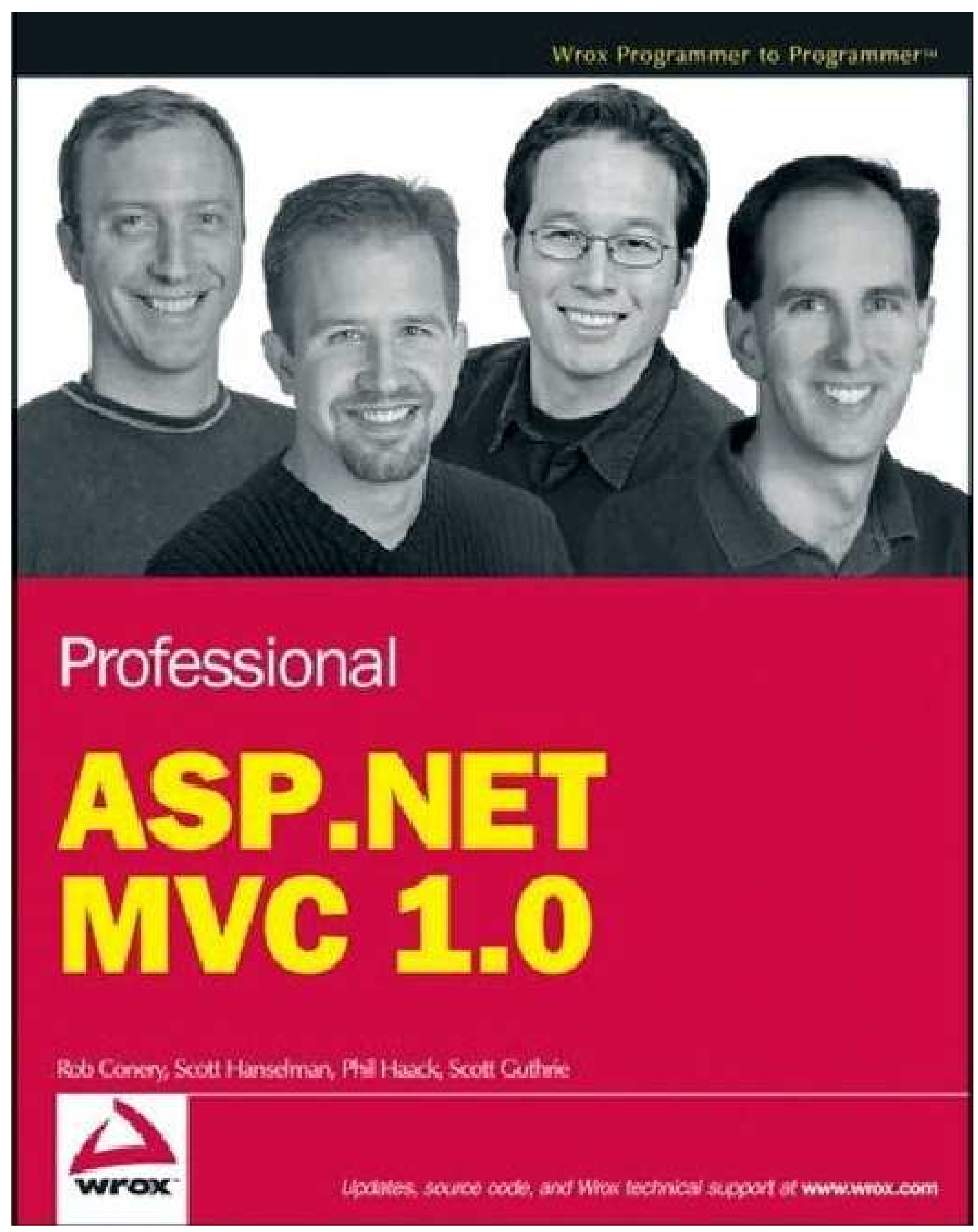
一步一步学习 ASP.NET MVC 1.0



http://www.agiledon.com

张逸

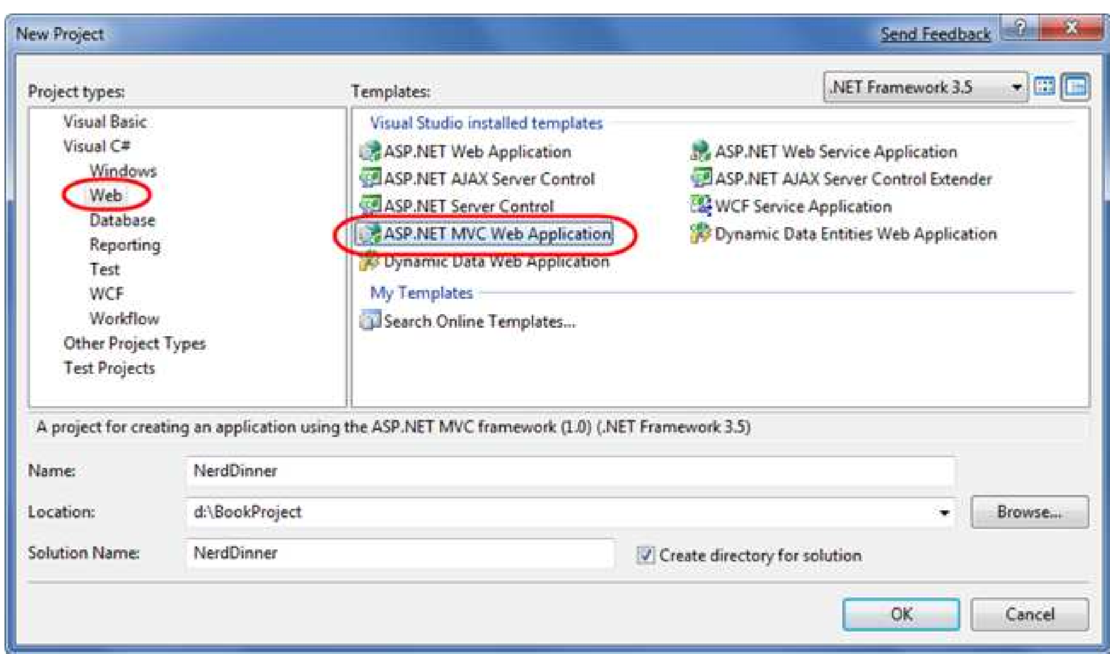
目 录

创建 MVC Web Application 4
检查 NerdDinner 项目的目录结构 5
运行 NerdDinner 应用程序 8
测试 NerdDinner 应用程序 10
创建数据库 10
设置表之间的外键关系 12
增加数据到 Dinners 数据表 13
创建 Model 模型 13
LINQ to SQL 14
控制器和视图（Controllers and Views） 25
添加 DinnersController 控制器 25
理解 ASP.NET MVC Routing 27
在 DinnersController 控制器中使用 DinnerRepository 29
控制器 Controller 使用视图 Views 30
实现 NotFound 视图模板 31
实现 Details 视图模板 33
实现 Index 视图模板 38
命名规范和\Views 目录结构 42
创建、更新、删除记录 44
实现 HTTP-GET 编辑 Action 方法 45
Html.BeginForm() 和 Html.TextBox() HTML 辅助方法 49
实现 HTTP-POST 的 Edit Action 方法 50
获取表单提交的值 51
处理编辑异常 53
理解 ModelState 和验证 HTML 辅助方法 54
Html 辅助方法和 ModelState 集成 55
完成 Edit Action 方法的实现 57
实现 HTTP-GET 的 Create Action 方法 58
实现 HTTP-POST 的 Create Action 方法 61
实现 HTTP-GET 的 Delete Action 方法 63
实现 HTTP-POST Delete Action 方法 65
模型绑定的安全性 66
基于用途来锁定绑定 67
基于类型来锁定绑定 67
CRUD 封装 68
ViewData 和 ViewModel 70
从 Controller 传递数据到 View 视图模板 71
使用 ViewData 字典 71
使用 ViewModel 模式 73
定制 ViewModel 类（Custom-shaped ViewModel Classes） 76
Partials 和 Master 页面 76
使用 Partial 视图模板 76
使用 Partial 视图模板简化代码 79

Master 页面 80
分页 84
认证和授权 91
理解认证和授权 91
Forms Authentication 和 AccountController 91
使用[Authorize]过滤器对/Dinners/Create 授权 94
创建 Dinners 时，使用 User.Identity.Name 属性 95
在编辑 Dinners 记录时，使用 User.Identity.Name 属性 96
显示/隐藏编辑和删除链接 98
AJAX 实现 RSVP 响应 99
显示用户是否已经回复了 99
实现 Register Action 方法 101
使用 AJAX 调用 Register Action 方法 101
添加 jQuery 动画 103
简化-重构 RSVP Partial 视图 105
集成 AJAX 地图 106
创建 Map Partial 视图 106
创建一个 Map.js 工具类库 107
集成地图到创建和编辑表单 109
集成地图到 Details 视图 113
在数据库和仓储中实现位置搜索 114
实现基于 JSON 的 AJAX 搜索 Action 方法 118
使用 jQuery 调用基于 JSON 的 AJAX 方法 119
单元测试 122
为什么需要单元测试？ 122
NerdDinner.Tests 项目 122
为 Dinner 模型类创建单元测试 124
运行测试 127
创建 DinnersController 单元测试 127
依赖注入（Dependency Injection） 129
提取 IDinnerRepository 接口 129
更新 DinnersController 支持构造器注入 131
创建 FakeDinnerRepository 类 131
在单元测试中使用 FakeDinnerRepository 134
创建 Edit Action 方法的单元测试 136
模仿 User.Identity.Name 属性 137
测试 UpdateModel() 139
单元测试总结 141
NerdDinner 范例程序总结 142

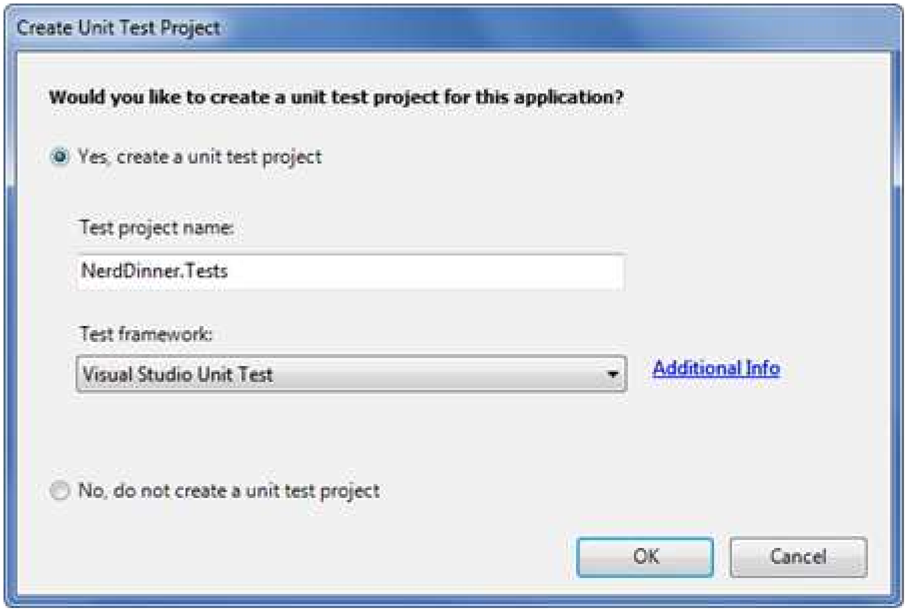
创建 MVC Web Application

在安装好 ASP.NET MVC 1.0 框架后，就可以创建 ASP.NET MVC Web Application 了。File -> New Project 创建 MVC Web Application，如下图所示。项目名称设置为 NerdDinner。



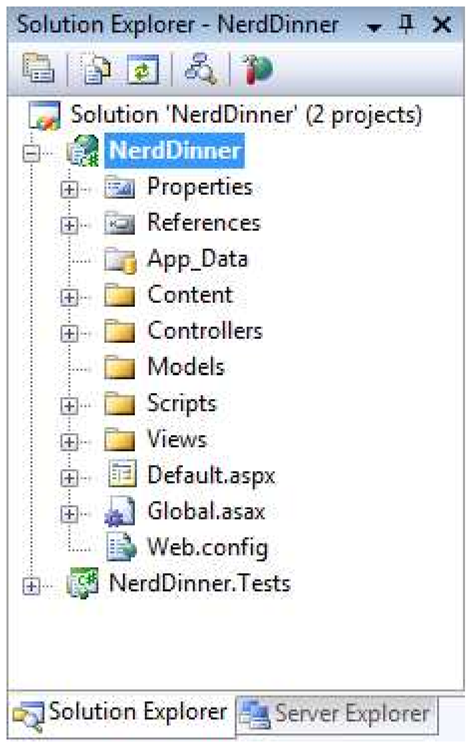
点击确定按钮后，Visual Studio 将弹出一个创建单元测试项目（Create Unit Test Project）的对话框，如下图所示。单元测试项目允许我们创建自动测试，验证应用程序的功能和行为。我们采用默认设置和名称，点击 OK 按钮。

Visual Stuido 将创建一个 Solution，包含有 2 个项目，一个是 MVC Web 应用程序，另一个是单元测试项目。



检查 NerdDinner 项目的目录结构

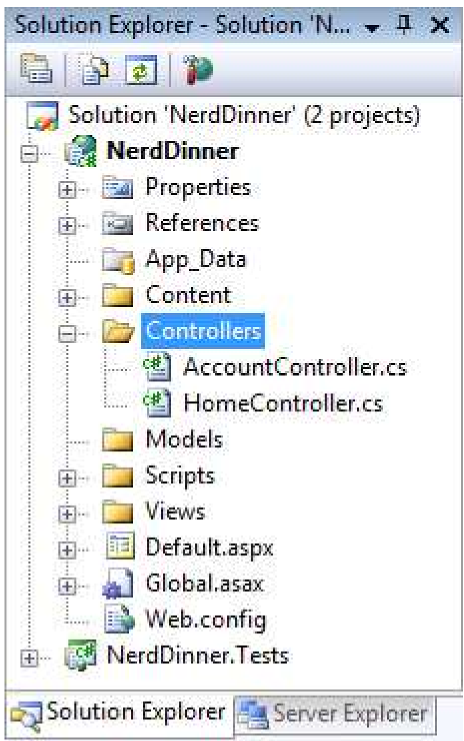
在使用 Visual Studio 创建 ASP.NET MVC 应用程序时，它会自动增加一些文件和目录到项目中，如下图所示。



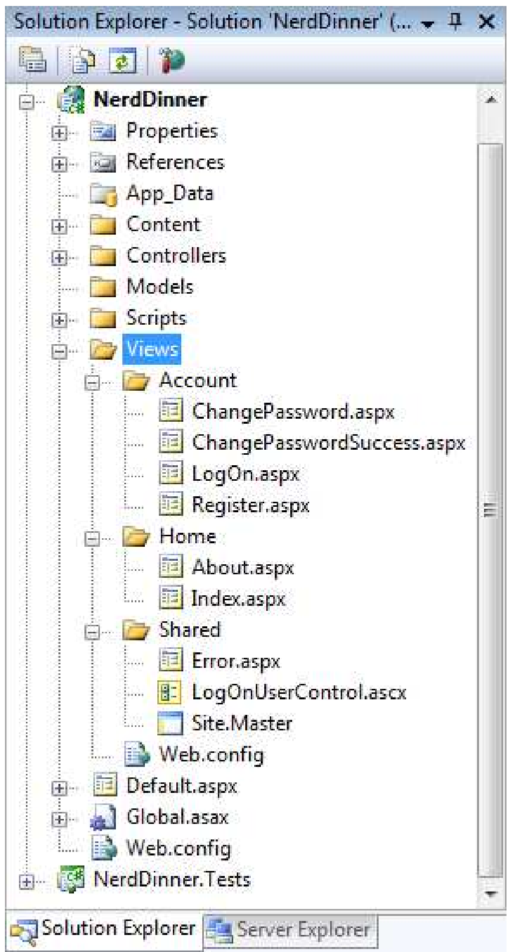
默认情况下，ASP.NET MVC 项目有 6 个顶级目录。
Controls – 放置 Controller 类，处理 URL 请求。
Models – 放置业务实体类，表示和操作数据。
Views – 放置 UI 模板文件，负责展示输出结果。
Scripts – 放置 Javascript 类库文件和.js 文件。
Contents – 放置 CSS 和图像文件，以及其他非动态的、非 Javascript 文件。
App_Data – 放置数据库文件。

ASP.NET MVC 应用程序不是必须要求这样的目录结构。事实上，大型系统的开发人员通常会将应用程序分为多个项目文件，这样是项目更易于管理（如，数据 Model 类通常在一个单独的类库项目中）。默认的项目结构提供了常规的目录结构，用来保存应用程序更加清晰。

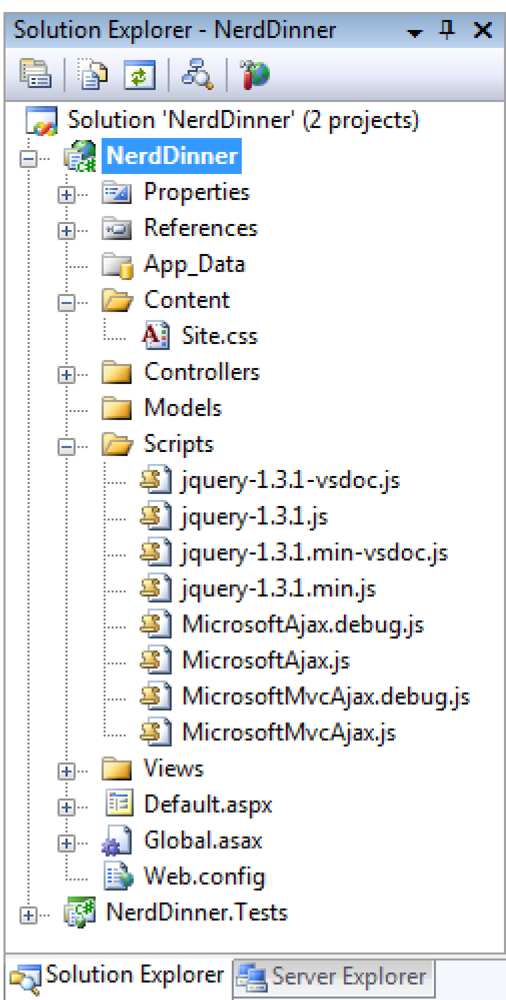
当我们展开/Controllers 目录时，可以发现默认情况下，Visual Studio 自动增加了 2 个 Controller 类 – HomeController 和 AccountController。



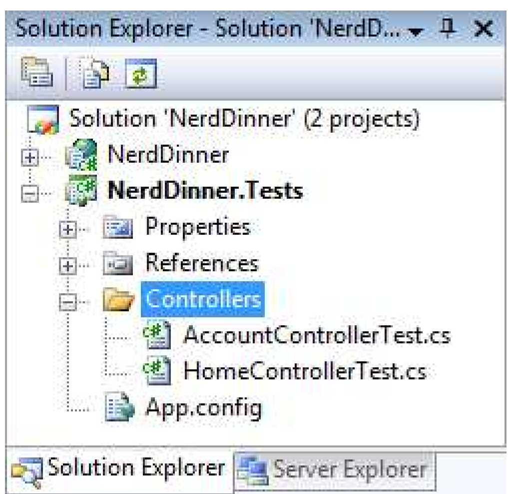
在展开/Views 目录时，发现自动添加了三个子目录，分别为：/Home, /Account 和 /Shard。同时，也添加了一些模板文件。



当展开/Content 和 /Scripts 文件后，会发现自动添加的 Site.css 文件和 Javascript 文件，其中 Site.css 文件用来对站点的 HTML 进行格式定义，Javascript 文件则使 Web 应用程序支持 ASP.NET AJAX 和 jQuery。



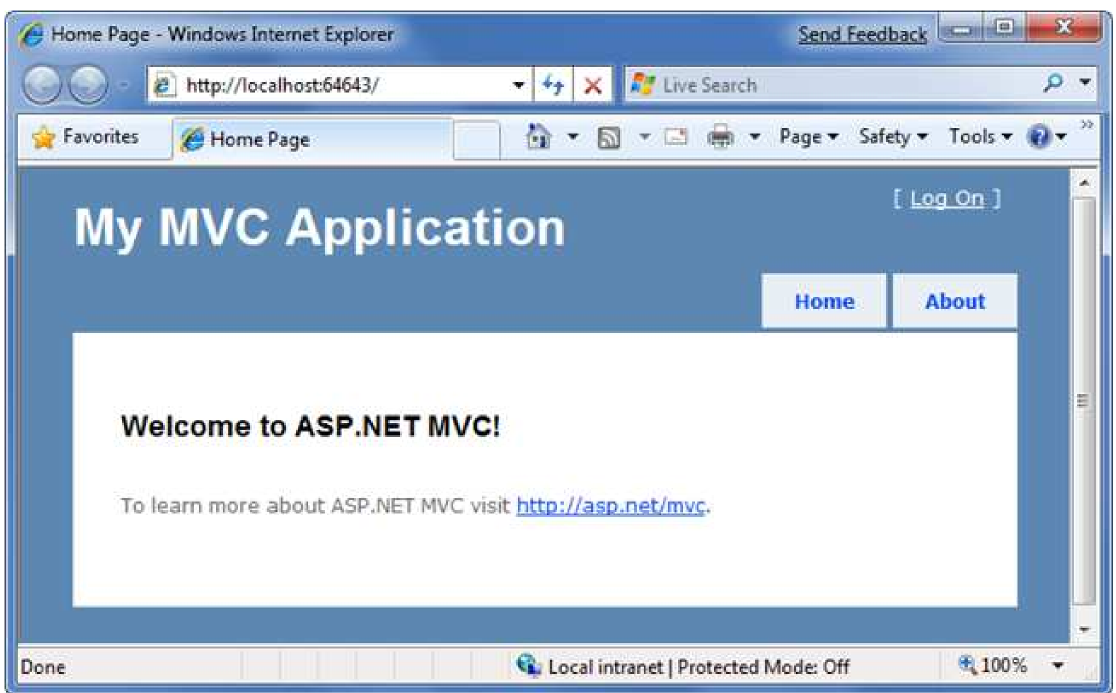
当展开 NerdDinner.Tests 项目时，会发现有 2 个类，包含了对 Controller 类的单元测试。如下图所示。



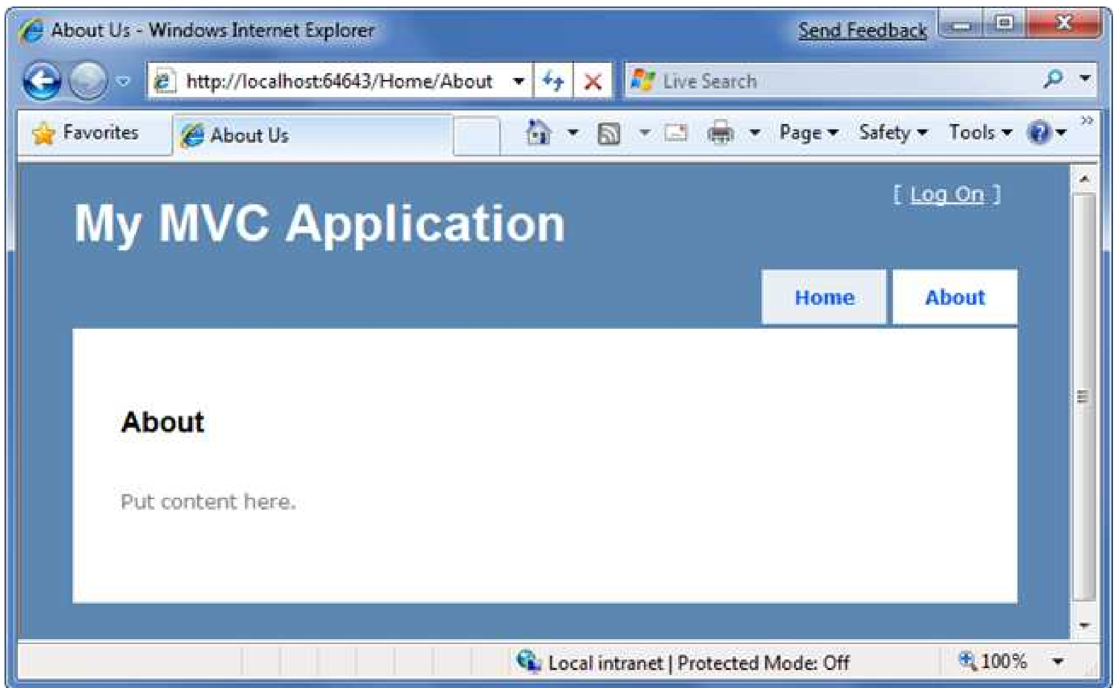
Visual Studio 增加的这些默认文件提供了一个可工作的应用程序的基本结构，包括 homepage、about 页面、登录/logout/注册等等页面，以及一个 unhandled 错误页面。

运行 NerdDinner 应用程序

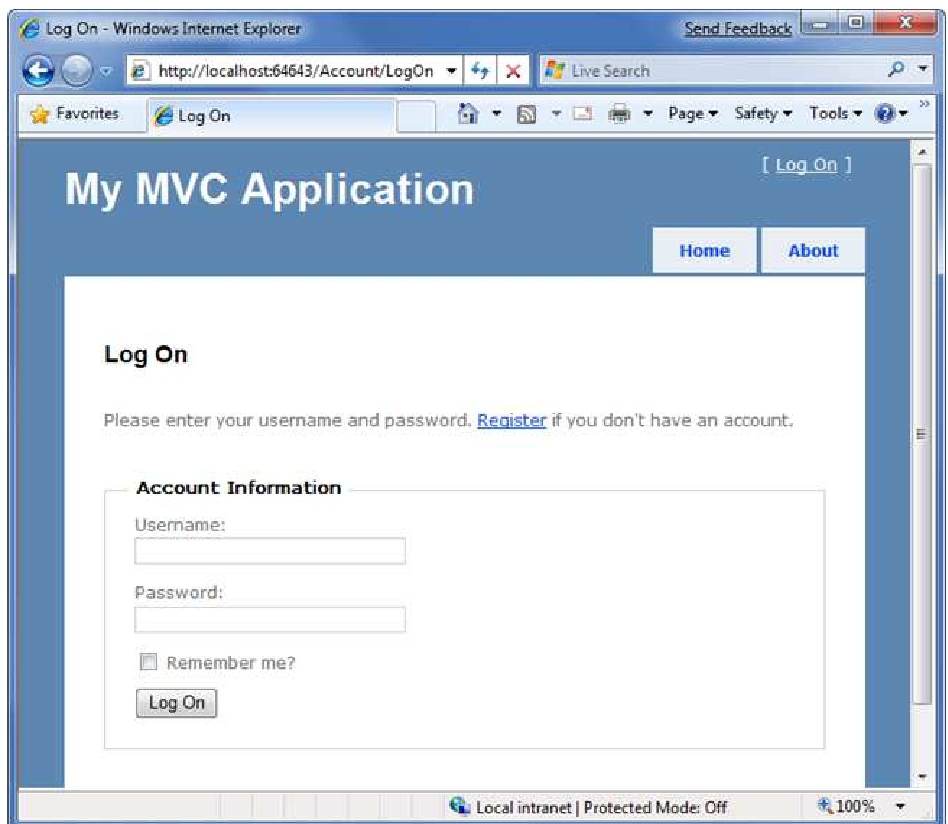
通过 Visual Studio 2008 运行上一步创建的 NerdDinner 应用程序，将启动内置的 ASP.NET Web Server。如下是 NerdDinner 应用程序的首页：



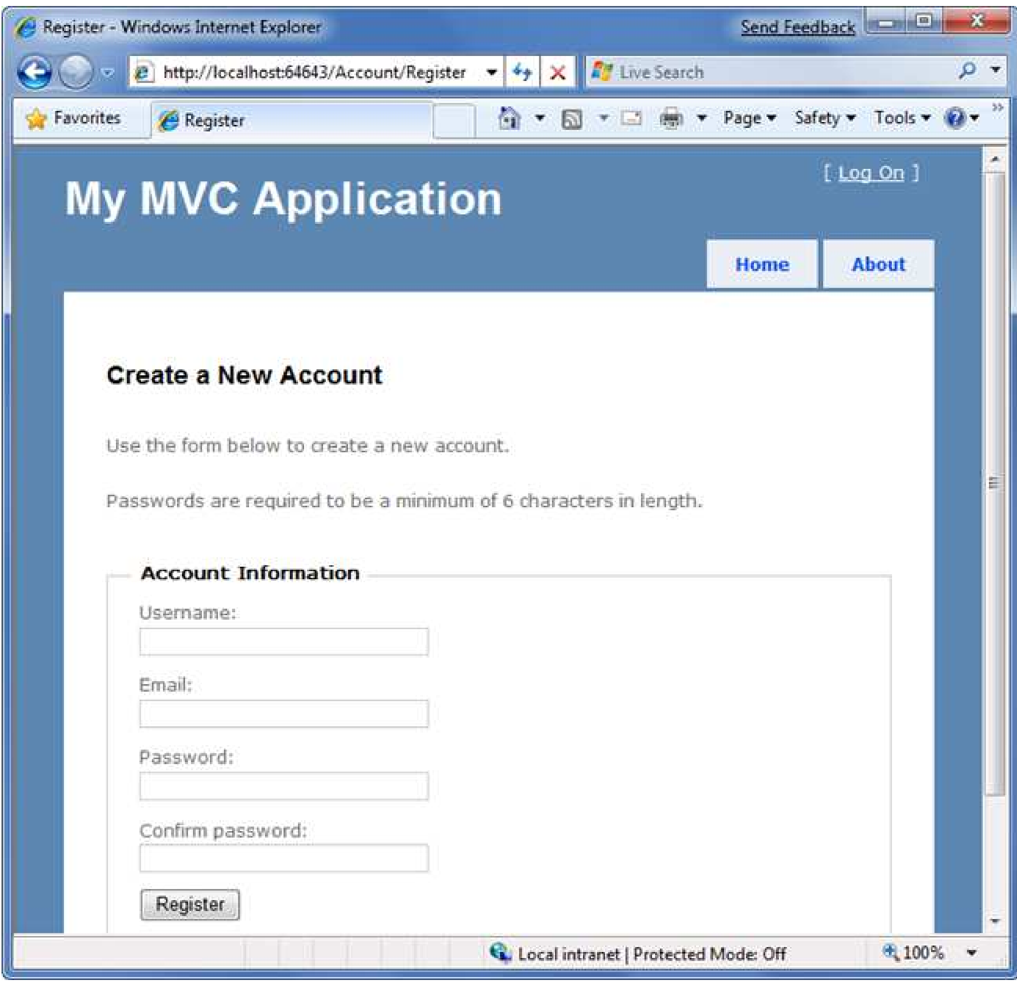
点击 About 链接，显示 about 页面，如下图所示：



点击右上角的 Log On 链接，进入 Login 登录页面，如下图所示：



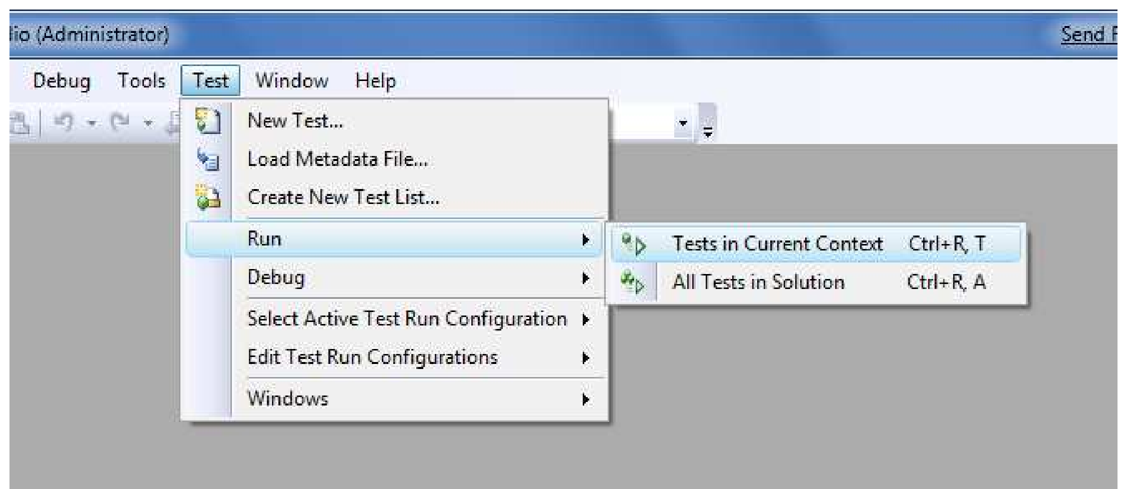
如果没有登录帐号，可以点击 Register 注册链接（URL 地址：/Account/Register），注册一个帐号：



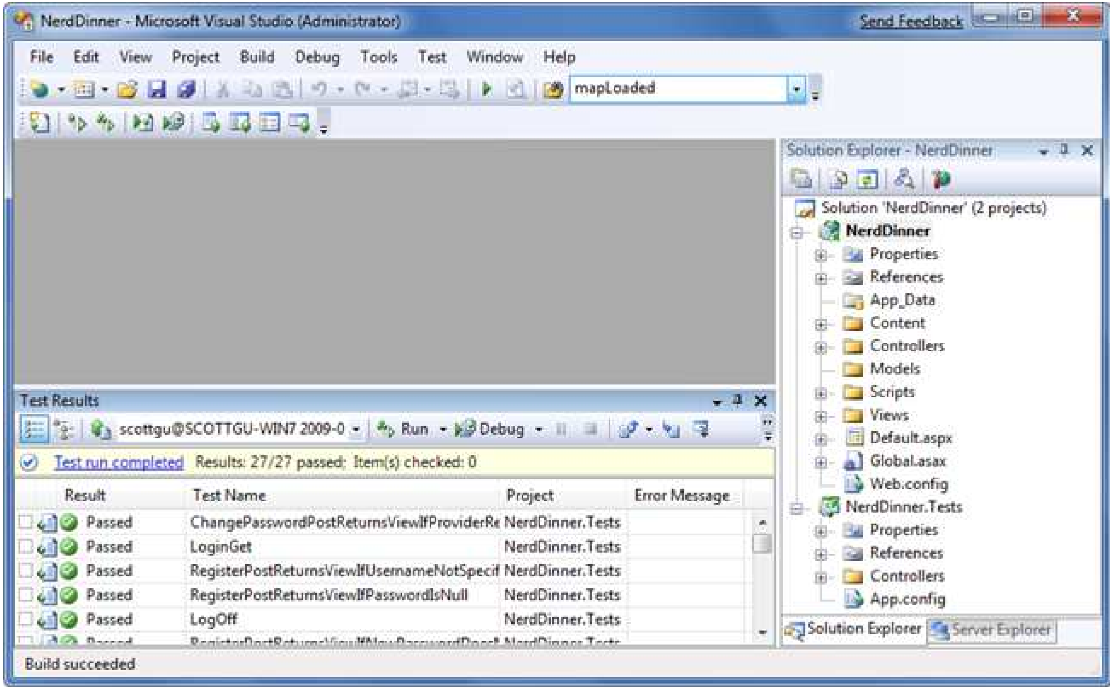
上述主页、about 页面、Logout/Register 页面功能的实现是我们在创建 MVC 项目时默认添加的，我们将使用这些代码作为应用程序的起始点。

测试 NerdDinner 应用程序

如果使用专业版或者更高级的 Visual Studio 2008 版本，则可以使用 Visual Studio 内置的单元测试 IDE 测试项目。



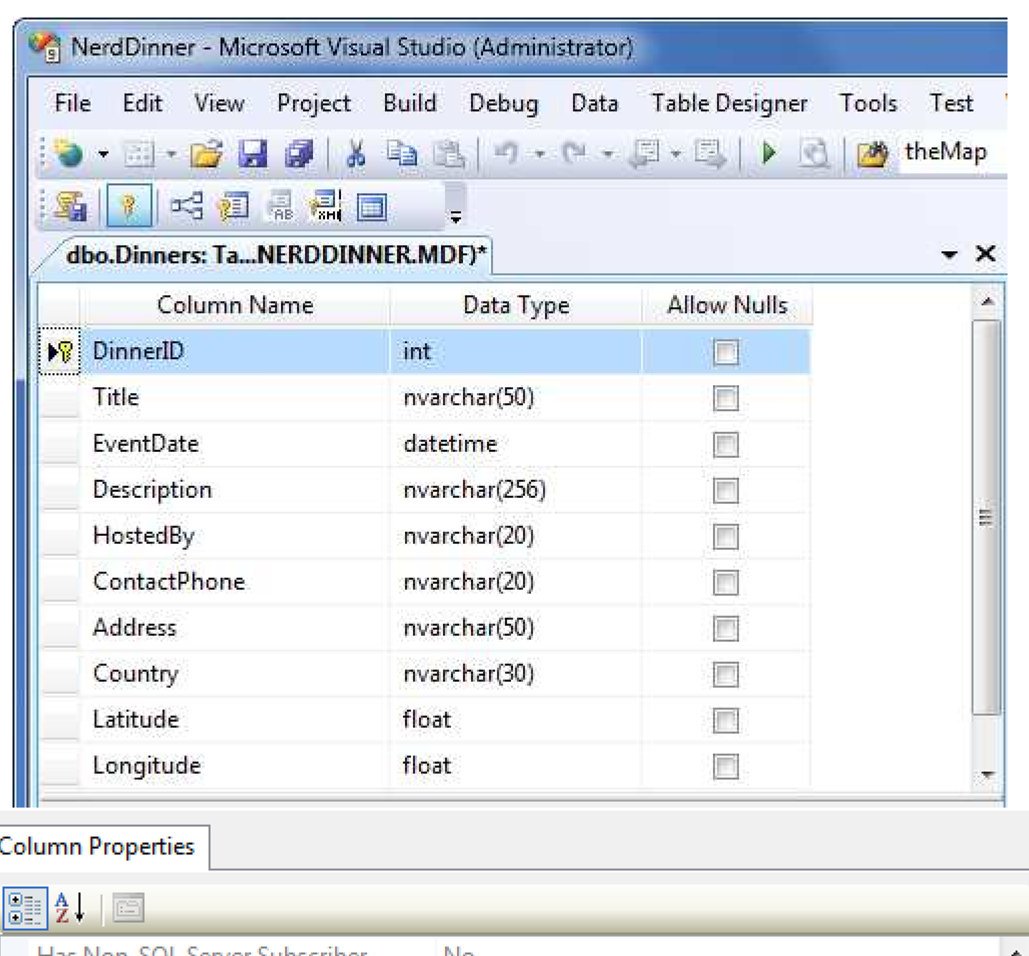
选择上述菜单项，将在 VS 2008 中打开 Test Results 面板，显示了 27 个单元测试的 pass/fail 状态，这些包含在我们新建的项目中，覆盖了内置的功能。如下图所示：

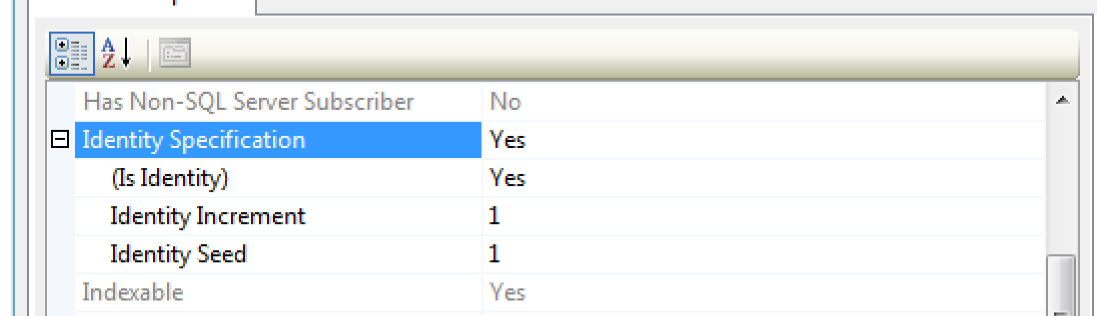


创建数据库

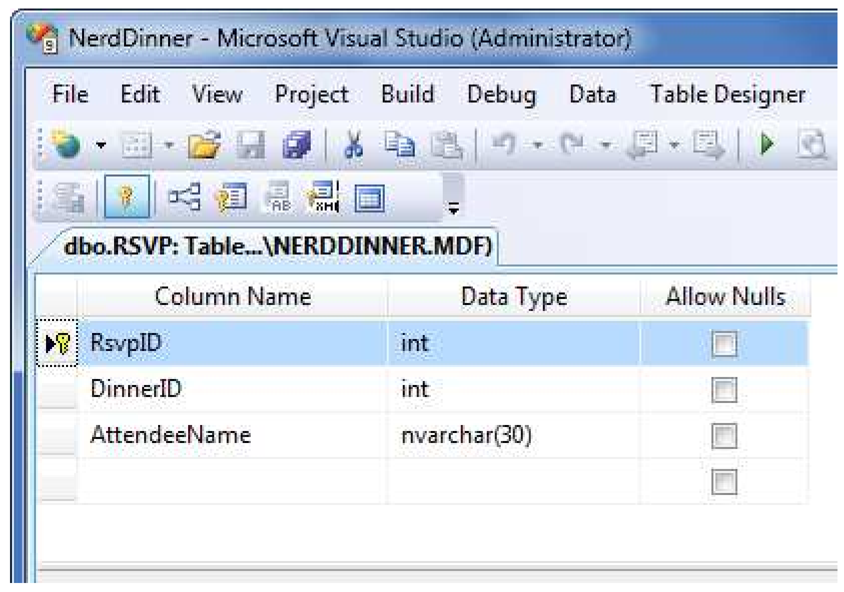
我们将使用数据库来存储 NerdDinner 应用程序的所有的 Dinner 和 RSVP 数据。这里，我们采用 SQL Server 2008 来创建和管理 NerdDinner 数据库。需要向数据库中增加 2 张表，一个表用来存放 Dinners 数据，另一个用来跟踪 RSVP。

如下图所示，有 NerdDinner 数据库，Dinners 数据表，其中 DinnerID 字段设置为主键，并且设置为自增长。





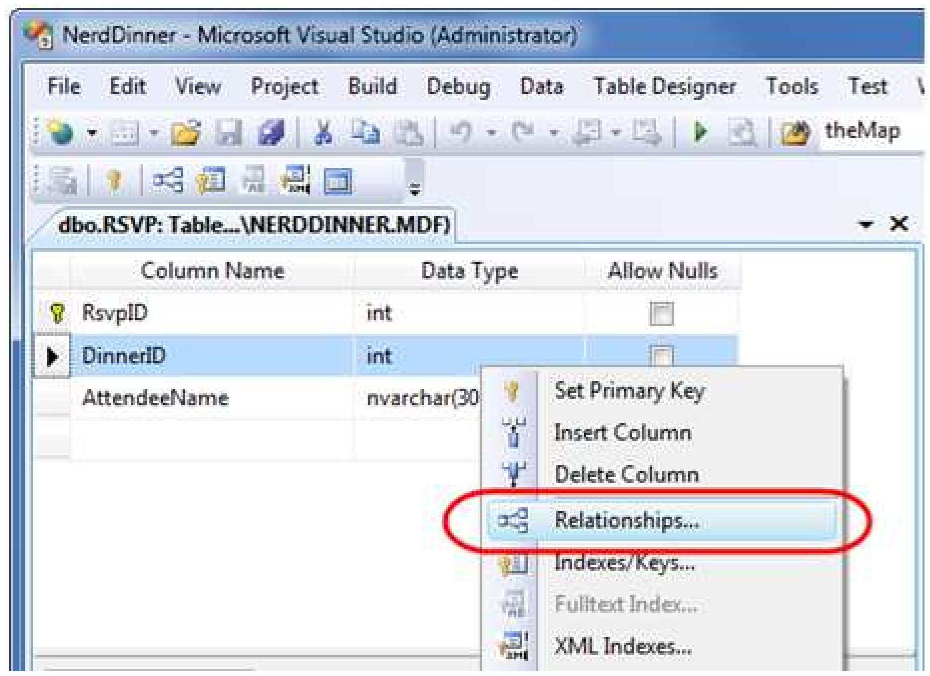
接下来按照相同的步骤，创建 RSVP 数据表，该表有 3 列。设置 RsvpID 列为主键，同时设置为 identity-自增长。最后，保存表名为 RSVP，如下图所示。



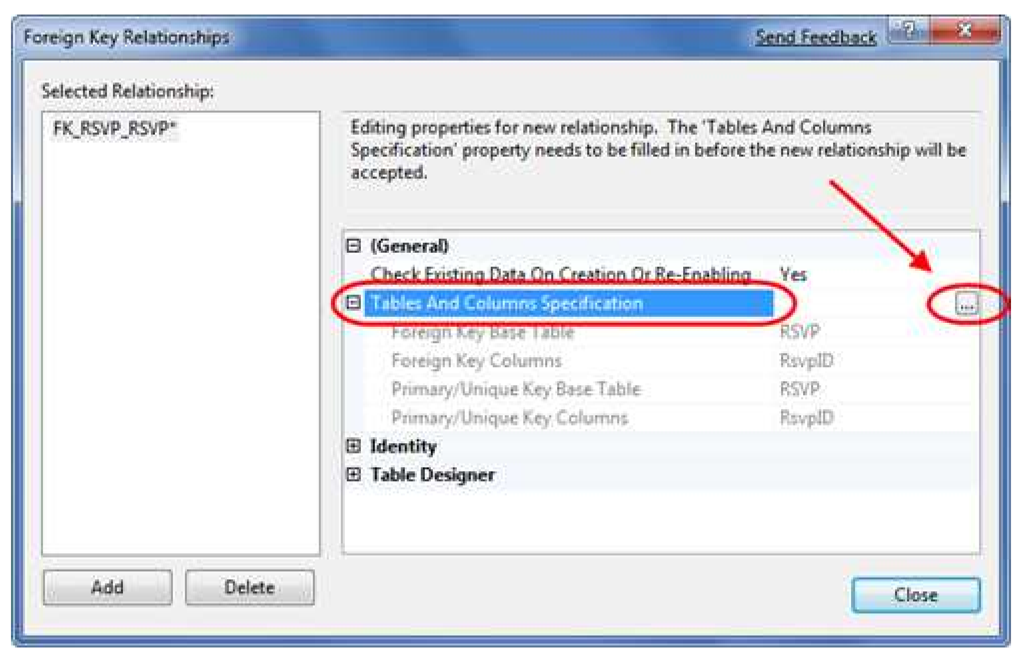
设置表之间的外键关系

NerdDinner 数据库有 2 个表，设置 one-to-many 1 对多关系，这样我们可以关联每一条 Dinner 记录到 0 或者多条 RSVP 记录。

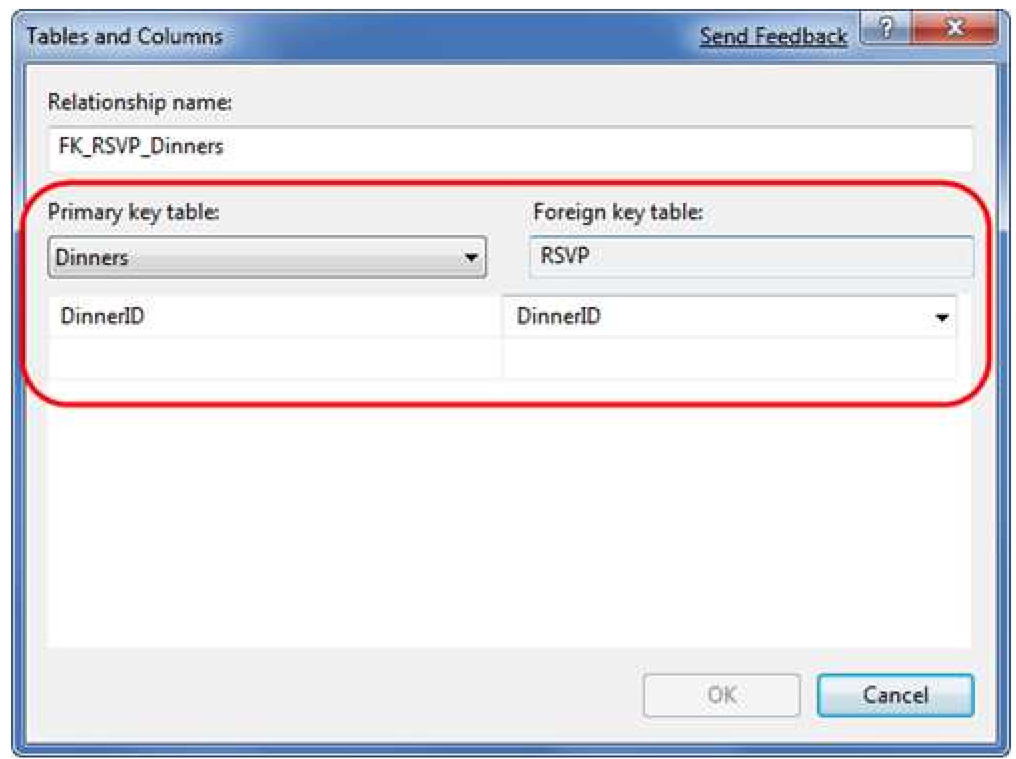
打开 RSVP 表，右键点击 DinnerID 字段，选择“关系…”菜单项，如下图所示。



弹出外键关系对话框，设置两个表之间的外键关系。点击添加按钮，添加一个新的关系到对话框。一旦添加关系后，进一步点击“表和列规范”后面的…按钮，如下图所示。

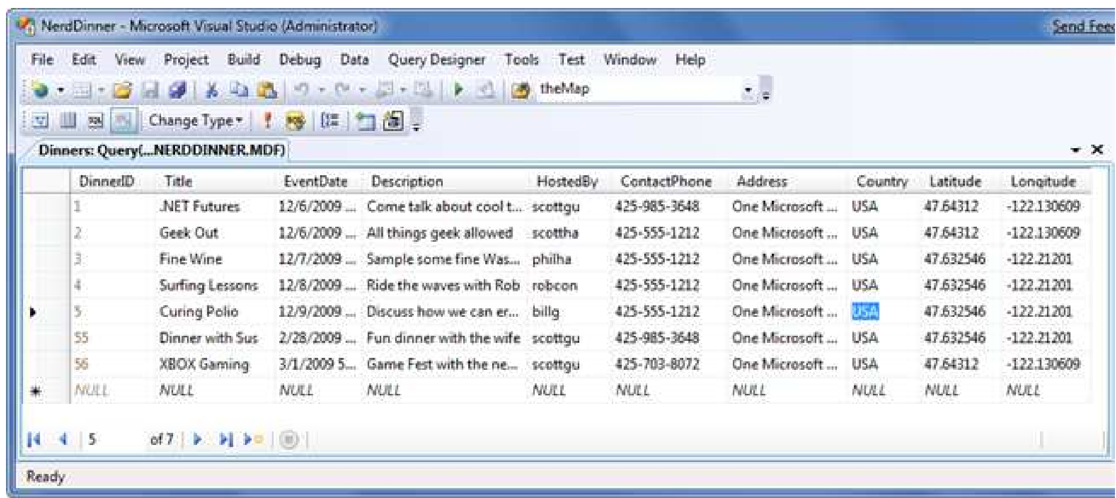


接下来弹出表和列对话框，指定相关的表和列的关系。如下图所示，进行 1 对多关系的设置。现在，RSVP 表中的每一条记录关联到 Dinners 表的一条记录。SQL Server 将负责维护数据的一致性，阻止用户增加没有对应 Dinner 记录的新 RSVP 行，也阻止用户删除还存在 RSVP 行关联的 Dinner 记录。



增加数据到 Dinners 数据表

下面我们增加一个示例数据到 Dinners 表，这些数据在随后的应用程序开发中会用到。



DinnerID	Title	EventDate	Description	HostedBy	ContactPhone	Address	Country	Latitude	Longitude
1	.NET Futures	12/6/2009 ...	Come talk about cool t...	scottgu	425-985-3648	One Microsoft ...	USA	47.64312	-122.130609
2	Geek Out	12/6/2009 ...	All things geek allowed	scottha	425-555-1212	One Microsoft ...	USA	47.64312	-122.130609
3	Fine Wine	12/7/2009 ...	Sample some fine Was...	philha	425-555-1212	One Microsoft ...	USA	47.632546	-122.21201
4	Surfing Lessons	12/8/2009 ...	Ride the waves with Rob	robcon	425-555-1212	One Microsoft ...	USA	47.632546	-122.21201
5	Curing Polio	12/9/2009 ...	Discuss how we can er...	billg	425-555-1212	One Microsoft ...	USA	47.632546	-122.21201
55	Dinner with Sus	2/28/2009 ...	Fun dinner with the wife	scottgu	425-985-3648	One Microsoft ...	USA	47.632546	-122.21201
56	XBOX Gaming	3/1/2009 5...	Game Fest with the ne...	scottgu	425-703-8072	One Microsoft ...	USA	47.64312	-122.130609
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

创建 Model 模型

在 Model-View-Controller 框架中，model 表示应用程序的数据对象，以及相应的业务领域逻辑，包括数据验证和业务规则。model 是 MVC 应用程序的核心部分，下面我们会看到 model 的作用。

ASP.NET MVC 框架支持使用任意数据访问技术，开发人员可以选择大量的.NET 数据访问技术实现 model，

如 LINQ to Entities、LINQ to SQL、NHibernate、LLBLGen Pro、SubSonic、WilsonORM、或者基本的 ADO.NET DataReaders、DataSets 等等。

对于本范例程序 NerdDinner，我们将采用 LINQ to SQL 创建一个简单的业务领域模型，非常接近于数据库的设计，并增加了一些定制的验证逻辑和业务规则。接下来实现一个 repository 类，帮助抽象化数据实体的实现，允许我们轻松实现单元测试。

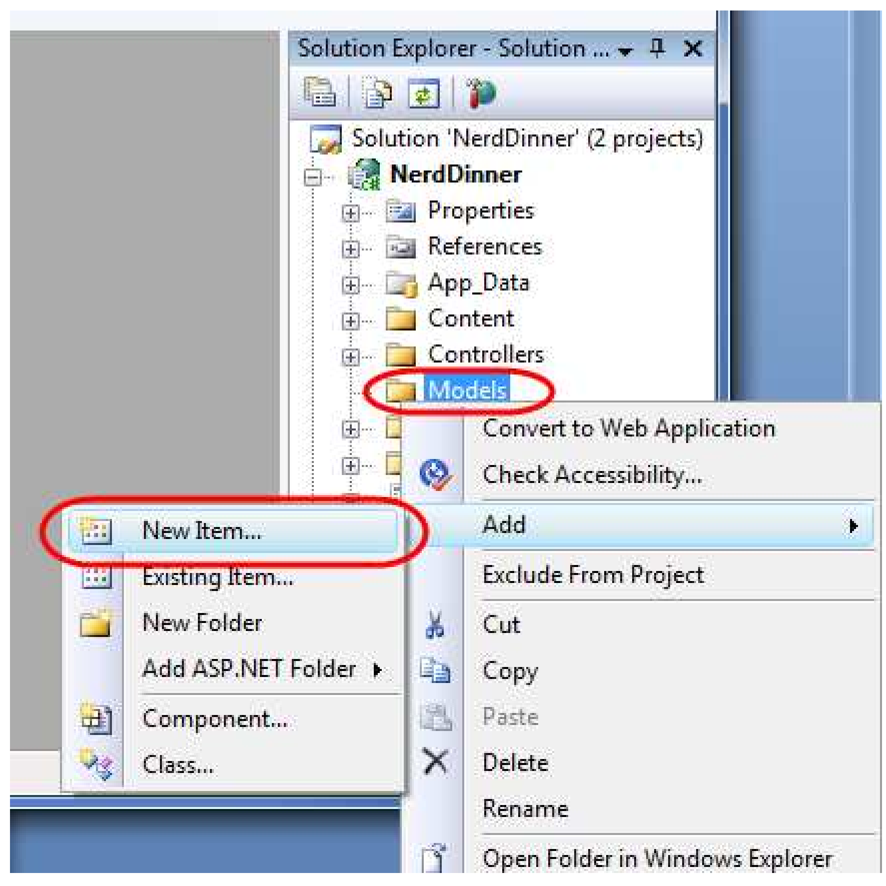
LINQ to SQL

LINQ to SQL 是.NET 3.5 的 ORM（Object Relational Mapper）部分。LINQ to SQL 提供了一个简单的方式映射数据表到.NET 类。对于我们的 NerdDinner 范例程序，我们将映射 Dinners 和 RSVP 数据表到 Dinner 和 RSVP 模型类。Dinners 和 RSVP 数据表中的列将映射为 Dinner 和 RSVP 模型类的属性。每一个 Dinner 和 RSVP 对象将表示 Dinners 或 RSVP 数据表中的一条单独的数据记录。

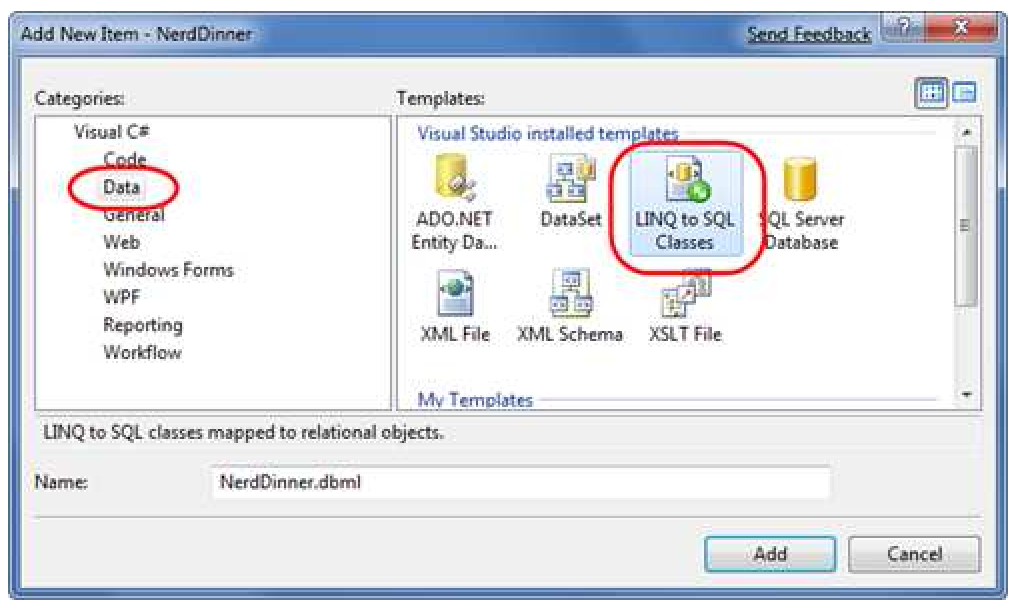
LINQ to SQL 避免让我们手动创建 SQL 脚本从数据库中检索和更新 Dinner 和 RSVP 对象。在访问或更新数据库记录时，LINQ to SQL 将负责生成合适的 SQL 执行逻辑。我们可以使用 VB 和 C#支持的 LINQ 语言写查询表达式，检索 Dinner 和 RSVP 对象。这样可以大量减少代码行，构建清晰的应用程序。

增加 LINQ to SQL 类到项目程序中

现在终于可以回到我们的 NerdDinner 应用程序了！右键点击 Models 文件夹，选择 Add -> New Item 菜单项。

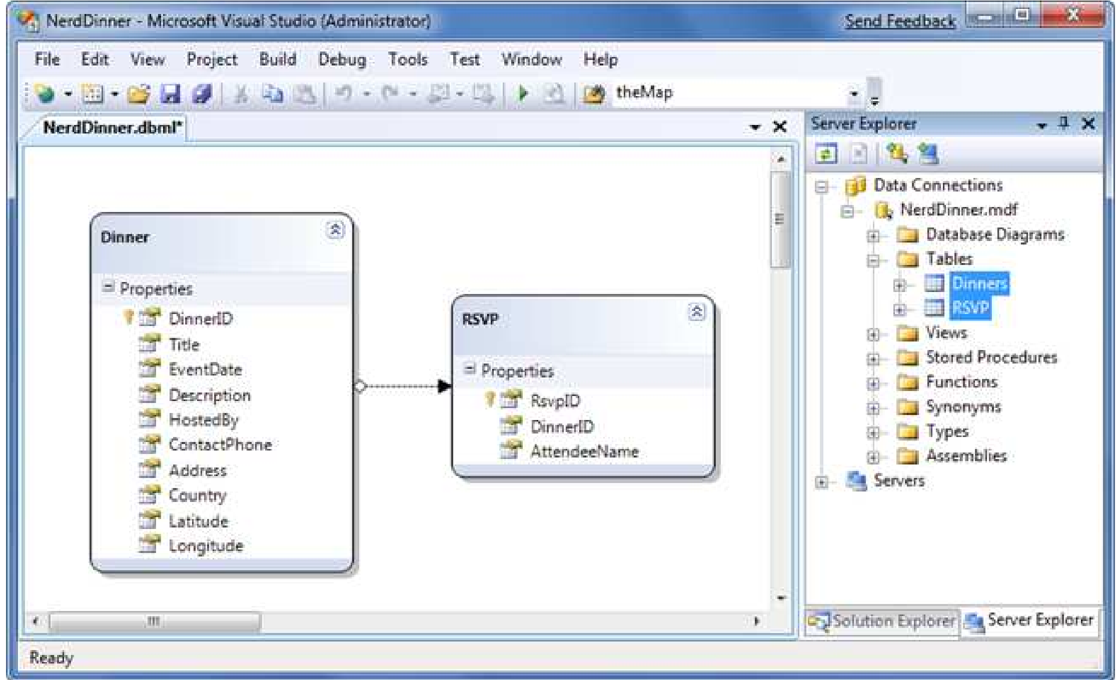


在弹出的 Add New Item 对话框，如下图所示。选择 LINQ to SQL Classes 模板，命名为 NerdDinner.dbml，然后点击 Add 按钮。Visual Studio 将添加 NerdDinner.dbml 文件到\Models 目录，并自动打开 LINQ to SQL 对象关系设计器。



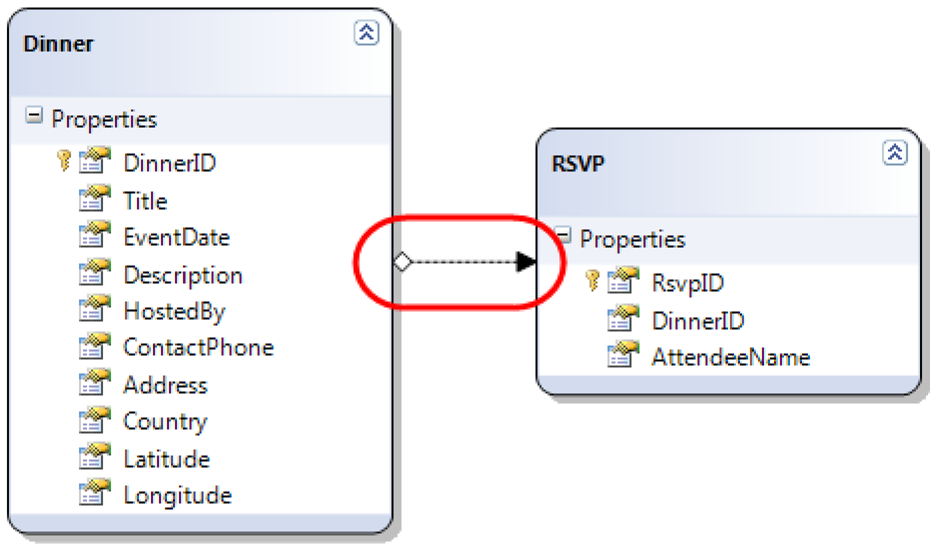
使用 LINQ to SQL 创建 Data Model Classes

LINQ to SQL 允许我们从现有的数据库快速数据模型类。我们可以在 Server Explorer 窗口打开 NerdDinner 数据库，并选择需要建模的数据表。将表 Dinners 和 RSVP 表拖拉到 NerdDinner.dbml 设计器中，如下图所示。



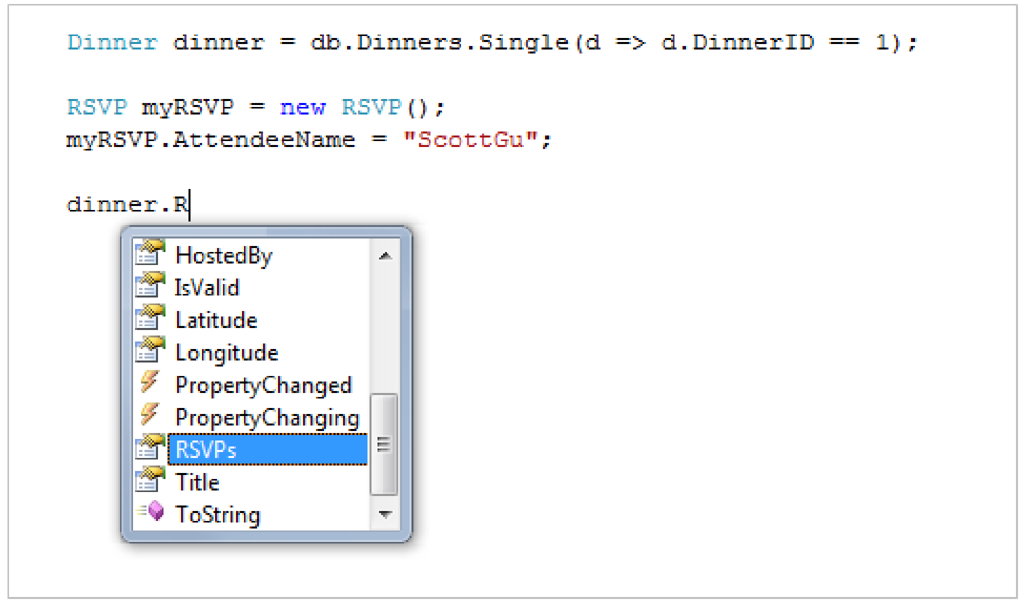
默认情况下，LINQ to SQL 设计器在基于数据表创建类时，自动复数化表和列名。例如，范例中的 Dinners 表自动产生 Dinner 类。这一类的命名有助于我们的模型类符合.NET 命名规范。但是，如果你不喜欢设计器自动生成的类或属性名，你可以在设计器中编辑或者通过属性列表进行更新。

另外，默认情况下，LINQ to SQL 设计器也会自动监视主键/外键关系，并基于这些关系，创建不同模型类之间的关联关系。如上图所示，当我们拖拉 Dinners 和 RSVP 表到 LINQ to SQL 设计器上时，自动创建 1 对多的关联关系，图中通过一个箭头进行表示。



上述关联关系将让 LINQ to SQL 增加一个强类型的 Dinner 属性到 RSVP 类中，开发人员可以使用指定 RSVP 的 Dinner 属性访问关联的实体。也会让 Dinner 类有一个强类型的 RSVP 集合属性，允许开发人员检索和更新 Dinner 关联的 RSVP 对象集合。

下面，我们看看当创建 RSVP 对象，并增加到 Dinner’s RSVP 集合是，Visual Studio 的智能提示：



从上可以看到 LINQ to SQL 如何为 Dinner 对象创建 RSVPs 集合，我们使用这个关联数据表 Dinner 和 RSVP 之间的一个外键关系。

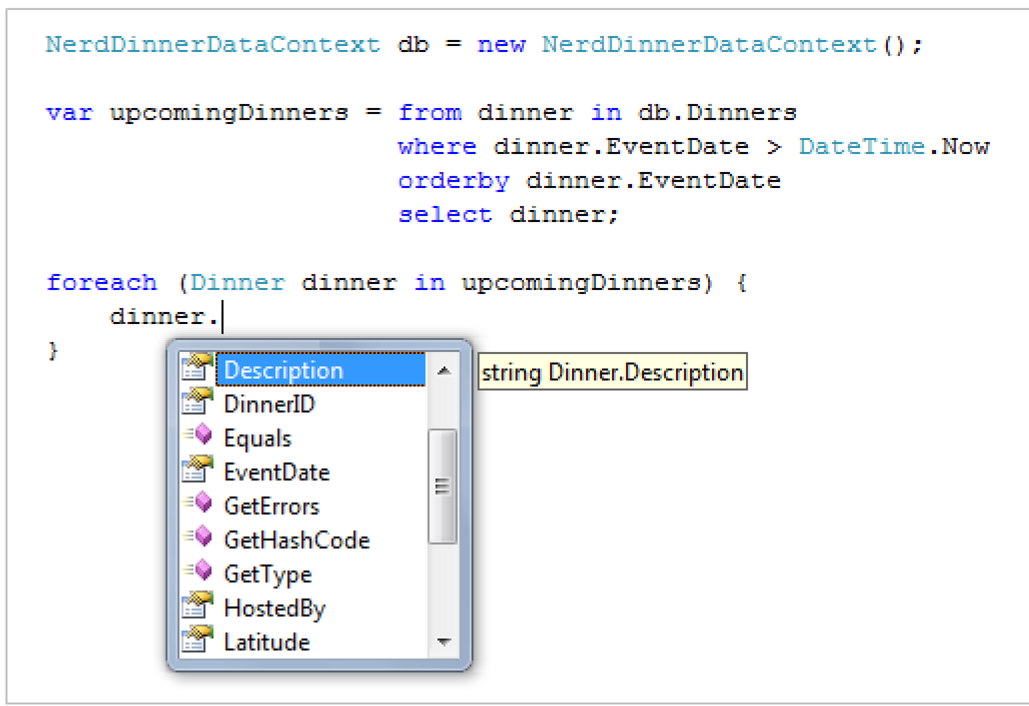
```
Dinner dinner = db.Dinners.Single(d => d.DinnerID == 1);

RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";

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

如果你不喜欢设计器的建模或命名的关联关系，你可以更改。点击设计器中的关联箭头，访问属性窗口，可以进行重命名、删除或者更改。对于 NerdDinner 应用程序，我们将使用默认的设置。

NerdDinnerDataContext 类

Visual Studio 自动生成.NET 类，表示 LINQ to SQL 设计器创建的模型和数据库关系，同时也会每一个 LINQ to SQL 设计器文件生成 LINQ to SQL DataContext 类。因为，我们命名 LINQ to SQL 类为 NerdDinner，创建的 DataContext 类将命名为 NerdDinnerDataContext，NerdDinnerDataContext 类将是我们与数据库交互的基本方式。

NerdDinnerDataContext 类公开了 2 个属性- Dinners 和 RSVPs，分别表示建模的 2 个数据表。我们使用 C# 写 LINQ 查询语句访问这些属性，从数据库去查询和检索 Dinner 和 RSVP 对象。如下的代码演示如何实例化 NerdDinnerDataContext 对象，并执行 LINQ 查询，获取系列 Dinners 对象。



NerdDinnerDataContext 对象跟踪对 Dinner 和 RSVP 对象的所有变更，允许我们轻松保存变更到数据库中。下面的代码演示如何使用 LINQ 查询从数据库中检索一个单一的 Dinner 对象，更新其中 2 个属性，接着保存变更到数据库。

```
NerdDinnerDataContext db = new NerdDinnerDataContext();

// Retrieve Dinner object that reprents row with DinnerID of 1
Dinner dinner = db.Dinners.Single(d => d.DinnerID == 1);

// Update two properties on Dinner
dinner.Title = "Changed Title";
dinner.Description = "This dinner will be fun";

// Persist changes to database
db.SubmitChanges();
```

代码中的 NerdDinnerDataContext 对象自动跟踪对 Dinner 对象的所有属性变更。当我们调用 SubmitChanges() 方法时，它对数据库执行合适的 SQL Update 语句，将新的数据更新到数据库。

创建 DinnerRepository 类

对于一个小应用程序而言，有时让 Conrollers 类直接使用 LINQ to SQL DataContext 类，并将 LINQ 查询语句写在 Controllers 中。但随着应用程序越来越大，这一方法的维护和测试将变得麻烦，并且导致重复的 LINQ 查询在多个地方出现。

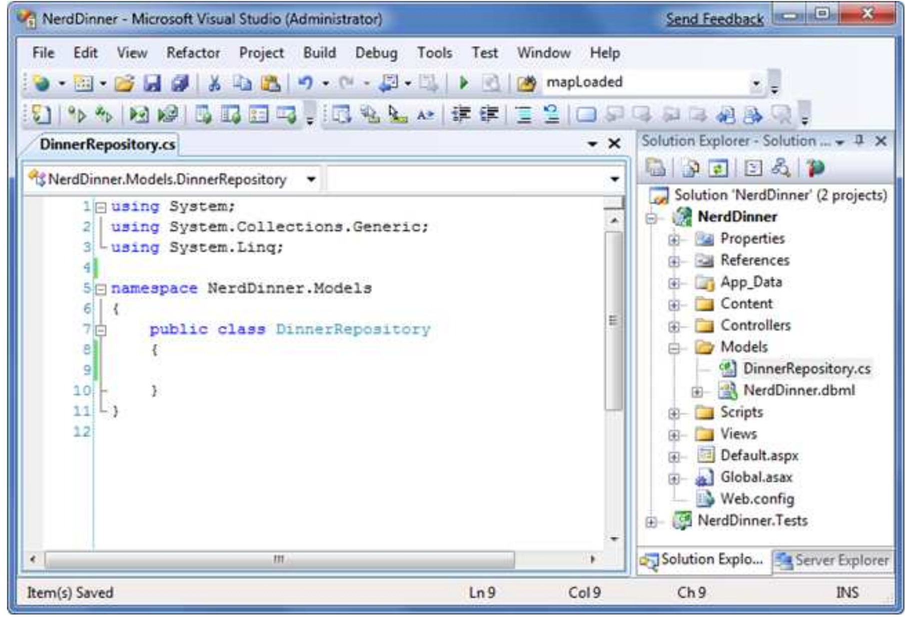
让维护和测试更方便的方法是使用 Repository 模式。Repository 类帮助封装数据查询和存储逻辑，从应用程序中抽象隔离具体的数据存储实现。除了是应用程序代码更加简洁外，使用 Repository 模式使将来更改数据库存储实现更加方便，并且有助于在没有真实数据库的情况下，进行应用程序进行单元测试。

对 NerdDinner 应用程序，我们将定义一个 DinnerRepository 类，类的原型如下：

```
public class DinnerRepository {

    // Query Methods
    public IQueryable<Dinner> FindAllDinners();
    public IQueryable<Dinner> FindUpcomingDinners();
    public Dinner           GetDinner(int id);

    // Insert/Delete
    public void Add(Dinner dinner);
    public void Delete(Dinner dinner);

    // Persistence
    public void Save();
}
```

备注：在本章后面部分，我们将从这个类中提取 IDinnerRepository 接口，允许在 Controllers 类实现依赖注入（Dependency Injection）。但在开始的时候，我们将开始一个简单的、直接工作的 DinnerRepository 类。

为了实现这个类，右键点击 Models 文件夹，选择 Add -> New Item 菜单项。在 Add New Item 对话框，我们选择 Class 模板，并命名文件为 DinnerRepository.cs。呵呵，终于又回到 NerdDinner 应用程序了。



接下来，我们实现 DinnerRepository 类，示例代码如下：

```
public class DinnerRepository {

    private NerdDinnerDataContext db = new NerdDinnerDataContext();

    //
    // Query Methods

    public IQueryable<Dinner> FindAllDinners() {
        return db.Dinners;
    }

    public IQueryable<Dinner> FindUpcomingDinners() {
        return from dinner in db.Dinners
               where dinner.EventDate > DateTime.Now
               orderby dinner.EventDate
               select dinner;
    }

    public Dinner GetDinner(int id) {
        return db.Dinners.SingleOrDefault(d => d.DinnerID == id);
    }

    //
    // Insert/Delete Methods

    public void Add(Dinner dinner) {
        db.Dinners.InsertOnSubmit(dinner);
    }

    public void Delete(Dinner dinner) {
        db.RSVPs.DeleteAllOnSubmit(dinner.RSVPs);
        db.Dinners.DeleteOnSubmit(dinner);
    }

    //
    // Persistence

    public void Save() {
        db.SubmitChanges();
    }
}
```

使用 DinnerRepository 类实现检索、更新、插入和删除操作

现在我们已经创建了 DinnerRepository 类，下面我们看看一些示例代码。

查询代码：

下面的代码使用 DinnerID 检索一条 Dinner 记录：

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);
```

下面的代码用来检索所有将来的 dinners，并遍历：

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve all upcoming Dinners
var upcomingDinners = dinnerRepository.FindUpcomingDinners();

// Loop over each upcoming Dinner and print out its Title
foreach (Dinner dinner in upcomingDinners) {
    Response.Write("Title" + dinner.Title);
}
```

插入和更新代码：

下面的代码演示新增 2 个 dinners，新增或更新的信息不会提交到数据库，直到调用 Save() 方法。LINQ to SQL 自动包装所有更新的数据库事务，因此在调用 Repository 的 Save() 方法时，或者所有的变更都发生，或者都不发生。

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Create First Dinner
Dinner newDinner1 = new Dinner();
newDinner1.Title = "Dinner with Scott";
newDinner1.HostedBy = "ScotGu";
newDinner1.ContactPhone = "425-703-8072";

// Create Second Dinner
Dinner newDinner2 = new Dinner();
newDinner2.Title = "Dinner with Bill";
newDinner2.HostedBy = "BillG";
newDinner2.ContactPhone = "425-555-5151";

// Add Dinners to Repository
dinnerRepository.Add(newDinner1);
dinnerRepository.Add(newDinner2);

// Persist Changes
dinnerRepository.Save();
```

下面的代码首先检索一个存在的 Dinner 对象，然后更新 2 个属性，最后调用 Repository 对象的 Save() 方法，提交更新到数据库。

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);
```

```
// Update Dinner properties
dinner.Title = "Update Title";
dinner.HostedBy = "New Owner";

// Persist changes
dinnerRepository.Save();
```

下面的示例代码首先检索一个 dinner 对象，然后添加一个 RSVP 对象。这里，使用 Dinner 对象的 RSVPs 集合对象。当调用 Resposity 对象的 Save()方法时，一条新的记录添加 RSVP 表中。

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);

// Create a new RSVP object
RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";

// Add RSVP to Dinner's RSVP Collection
dinner.RSVPs.Add(myRSVP);

// Persist changes
dinnerRepository.Save();
```

删除操作代码：

下面的代码首先检索一个已存在的 Dinner 对象，然后调用 Repository 对象的 Delete() 方法，标记该条记录删除。最后，在调用 Repository 对象的 Save() 方法时，从数据库表中删除该记录。

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);

// Mark dinner to be deleted
dinnerRepository.Delete(dinner);

// Persist changes
dinnerRepository.Save();
```

Model 类集成验证和业务规则逻辑

集成验证和业务规则逻辑是任何与数据打交道的应用程序的最重要部分。

Schema 验证

当使用 LINQ to SQL 设计器定义 Model 类时，数据模型类的属性类型和数据表的字段类型相关。例如，如果 Dinners 表中 EventDate 列是 datetime 类型，LINQ to SQL 创建的数据模型类相关属性也是 DateTime

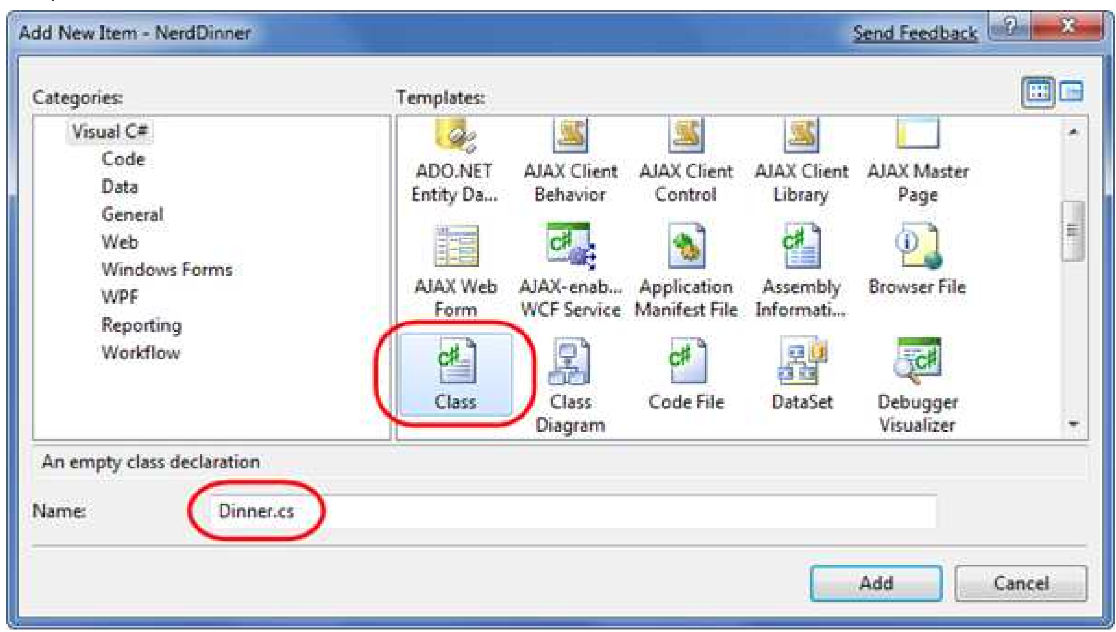
类型（这是内置的.NET 数据类型）。这意味着，如果你试图赋值整型或布尔型，将产生编译错误。
当使用字符串时，LINQ to SQL 将自动避开 SQL 值，因此你不必担心 SQL 注入攻击。

验证和业务规则逻辑

作为第一步，数据类型验证是非常有用的，但是还不足够。在大多数情况下，需要指定更丰富的验证逻辑。有很多不同的模式和框架可对模型类定义和应用验证。
在 NerdDinner 范例程序中，我们将采用相对简单和直接的模式，对 Dinner 模型类公开 IsValid 属性和 GetRuleViolations() 方法，IsValid 属性根据验证和业务规则返回 true 或 false，GetRuleViolations() 则返回所有错误的业务逻辑列表。

我们通过添加 Partial class 到项目中，来实现 IsValid 和 GetRuleViolations()方法。Partial 类用来增加方法/属性/事件到 VS 设计器生成的类中（如 LINQ to SQL 设计器生成的 Dinner 类），有助于区分我们编写的代码。

在项目中\Models 文件夹，添加新的类，并命名为 Dinner.cs。如下图所示：



点击 Add 按钮，添加 Dinner.cs 文件到项目中，并默认打开。接着，我们使用如下的代码来实现基本的业务规则/验证机制。

```
public partial class Dinner {

    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, string propertyName) {
        ErrorMessage = errorMessage;
        PropertyName = propertyName;
    }
}
```

上述代码提供了集成验证和业务规则的简单框架。现在，我们可以增加如下规则到 GetRuleViolations() 方法中。

```
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;
}
```

这里，我们使用 C#的 yield return 特性，返回有序的 RuleViolation 集合。

yield return 语句返回集合的一个元素，并移动到下一个元素上。yield break 可停止迭代。包含 yield 语句的方法或属性也称为迭代块。迭代块必须声明为返回 IEnumerator 或 IEnumerable 接口。这个块可以包含

多个 yield return 语句或 yield break 语句，但不能包含 return 语句。

前 6 个检查确保 Dinner 的字符串属性不能是 null 或空，最后一个规则比较有趣，调用 PhoneValidator.IsValidNumber() 方法，该方法将添加项目，用来验证 ContactPhone 数字格式符合相应的规则。

我们使用.NET 的正则表达式来实现电话号码的验证，下面是我们添加到项目中的一个简单 PhoneValidator 的实现，实现指定国家的正则模式检查。

```
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;
        }
    }
}
```

现在，当我们创建或更新 Dinner 对象时，验证逻辑规则将生效。开发人员可以主动判断是否 Dinner 对象是有效，并在不抛出异常的情况下，检索所有冲突列表。

```
Dinner dinner = dinnerRepository.GetDinner(5);

dinner.Country = "USA";
dinner.ContactPhone = "425-555-BOGUS";

if (!dinner.IsValid) {

    var errors = dinner.GetRuleViolations();

    // do something to fix the errors
}
```

如果我们试图保存一个无效状态的 Dinner 对象，在调用 DinnerRepository 的 Save() 方法时，将产生异常。

这是因为 Dinner 对象存在冲突的业务规则，Dinner.OnValidate() 分部方法(Partial Method)抛出异常。我们可以捕获这一异常，并主动检索冲突列表，进行修复。

```
Dinner dinner = dinnerRepository.GetDinner(5);

try {

    dinner.Country = "USA";
    dinner.ContactPhone = "425-555-BOGUS";

    dinnerRepository.Save();
}
catch {

    var errors = dinner.GetRuleViolations();

    // do something to fix errors
}
```

因为验证和业务规则在业务领域模型层实现，而不是在用户界面 UI 层，这些可以应用在应用程序的所有场景中。随后，我们可以更改或增加业务规则，并让这些代码应用到 Dinner 对象上。我们只需要在一个地方灵活更改业务规则，而不需要在整个应用程序和用户界面层到处更改，这是一个良好的应用程序设计。

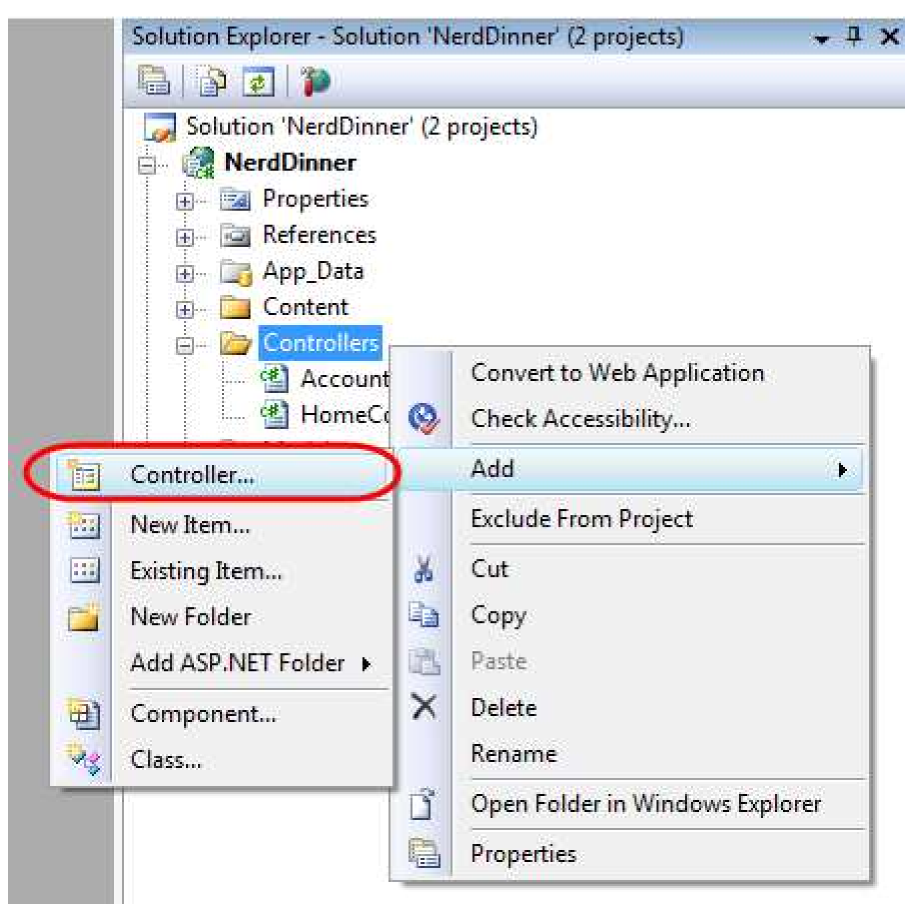
控制器和视图（Controllers and Views）

传统的 Web 框架，如 ASP/PHP/ASP.NET Web Forms 等等，请求的 URL 地址都是映射到特定的文件。如，请求的 URL 地址，像/Products.aspx 或/Products.php，由 Products.aspx 或 Products.php 文件负责处理。基于 Web 的 MVC 框架映射 URL 地址到服务器端代码有点不同，不是映射 URL 地址到特定文件，而是映射到类的方法上。这些类就是 MVC 中的 Controller 控制器，它们负责处理进来的请求和用户输入，接收和保存数据，并返回结果给客户端（显示 HTML、下载文件、或重定向到不同的 URL 等等）。

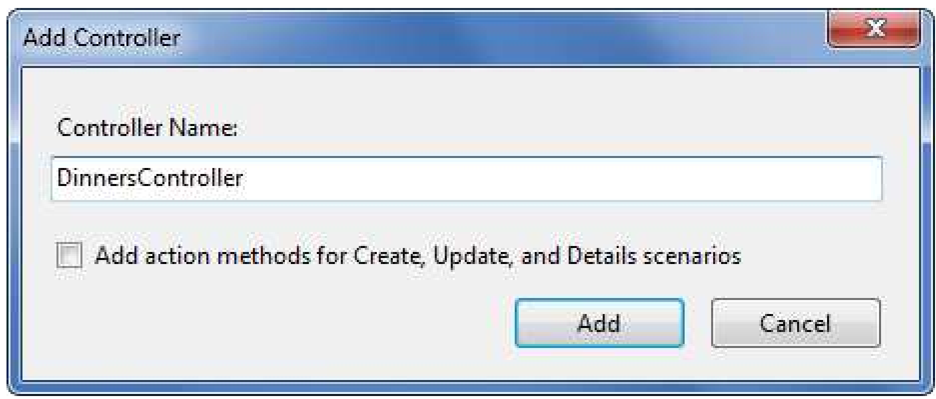
现在，我们已经创建一个基本的 model，下一步将添加控制器类到项目中，为 Dinners 网站用户提供数据列表显示和导航。

添加 DinnersController 控制器

右键点击 Web 项目中的 Controllers 文件夹，选择 Add ->Controller，如下图所示。



在弹出的 Add Controller 对话框，控制器名称输入 DinnersController。点击 Add 按钮，在项目文件中添加 DinnersController.cs 文件。



添加 Index() 和 Details() 方法到 DinnersController 类

我们想让访问者使用我们的应用程序浏览即将到来的宴会列表，用户可以点击任一宴会 Dinner 查看详细的信息。应用程序将发布如下的 URL 地址：
/Dinners/ -- 显示即将来临的宴会列表。
/Dinners/Details/[id] – 显示特定宴会的详细信息，通过 URL 地址中的 id 参数来匹配数据库中的 DinnerID。例如，/Dinners/Details/2 将在 HTML 页面显示 DinnerID=2 的详细信息。

下面，我们添加 2 个公共的 action 方法到 DinnersController 类中。

```
public class DinnersController : Controller {
```

```
    //
    // HTTP-GET: /Dinners/

    public void Index() {
        Response.Write("<h1>Coming Soon: Dinners</h1>");
    }

    //
    // HTTP-GET: /Dinners/Details/2

    public void Details(int id) {
        Response.Write("<h1>Details DinnerID: " + id + "</h1>");
    }
}
```

接下来，我们运行 NerdDinner 范例程序，通过浏览器调用方法。在 URL 地址栏输入/Dinners/将调用 Index()方法，并返回如下响应。

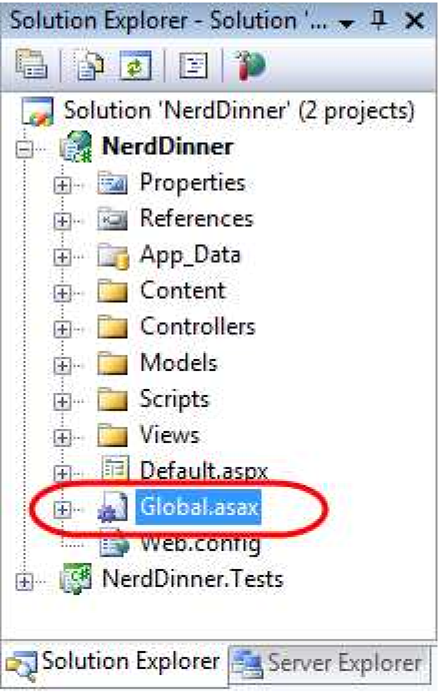
在地址栏输入/Dinners/Details/2，将触发 Details() 方法运行，返回如下结果：

你可能想知道-ASP.NET MVC 怎么知道调用 DinnersController 控制器的这些方法？下面我们看看 MVC 路由（routing）是如何工作的。

理解 ASP.NET MVC Routing

ASP.NET MVC 包含强大的 URL 路由引擎，提供了很好的灵活性来控制 URL 如何映射到控制器类。它允许我们完全定制 ASP.NET MVC 如何选择 controller 类，调用哪一个方法，以及从 URL/Querystring 中自动解析变量值，并作为参数传递给方法。ASP.NET MVC 路由也提供 SEO（Search Engine Optimization）优化的灵活性。

默认情况下，新的 ASP.NET MVC 项目已经注册了预配置的 URL 路由规则，这样允许我们轻松启动应用程序，而不需要配置任何东西。可以在项目中的 Application 类中看到默认的路由规则注册。在范例项目的根目录，双击 Global.asax 文件。



默认的 ASP.NET MVC 路由规则注册在该类的 RegisterRoutes() 方法中。

```
public void RegisterRoutes(RouteCollection routes) {

    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default",                                    // Route name
        "{controller}/{action}/{id}",                 // URL w/ params
        new { controller="Home", action="Index",id="" }  // Param defaults
    );
}
```

上述调用的 routes.MapRoute() 方法注册了一个默认的路由规则，映射请求的 URL 到 controller 类。使用的 URL 格式为：/{controller}/{action} /{id}，这里 controller 是指需要实例化的类名，action 是将调用的公共方法的名称，id 是一个可选的参数，嵌入在 URL 地址中，用来传递参数给方法。传递给 MapRoute() 方法的第三个参数是一组 controller/action/id 默认值，在 URL 没有指定时，Controller=Home、Action=Index、Id=””。

下面这个表格演示 URL 地址如何使用默认的/{controller}/{action}/{id} 规则进行映射：

URL	Controller Class	Action Method	Parameters Passed
/Dinners/Details/2	DinnersController	Details(id)	id=2
/Dinners/Edit/5	DinnersController	Edit(id)	id=5
/Dinners/Create	DinnersController	Create()	N/A
/Dinners	DinnersController	Index()	N/A
/Home	HomeController	Index()	N/A
/	HomeController	Index()	N/A

在 DinnersController 控制器中使用 DinnerRepository

现在我们开始先前创建的 model 替代现有的 Index() 和 Details() action 方法的实现。我们使用之前创建的 DinnerRepository 类来实现这些方法。

首先，需要 using NerdDinner.Models 命名空间，然后定义一个 DinnerRepository 实例作为 DinnerController 类的成员变量。在本章的后面部分，我们将引入依赖注入（Dependency Injection）的概念，演示另外一种方法来获取 DinnerRepository 的引用，实现更好的单元测试，但是现在，我们仍旧使用 DinnerRepository 实例。代码如下所示：

```
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) {
            Dinner dinner = dinnerRepository.GetDinner(id);
        }
    }
}
```

现在，我们已经准备好使用数据模型对象来产生 HTML 响应了。

控制器 Controller 使用视图 Views

尽管在 action 方法中可以写代码拼装 HTML，通过 Response.Write() 方法返回给客户端，但是这种方法并不可取。更好的办法是，在 DinnersController 的 action 方法中仅仅出来业务和数据逻辑，并将相关的数据传递给单独的视图（view）模板，view 负责生成 HTML 展示。随后我们会看到，view 模板就是一个简单的文本文件，通常包含 HTML 标识和嵌入的脚本。

将控制器 controller 逻辑和视图 view 展示分开带来很多好处，尤其是它帮助清晰隔离了应用程序代码和用户界面展示的脚本。这可以非常方便地对应用程序逻辑进行单元测试，而不需要 UI 展示逻辑，也可以方便修改 UI 脚本，而不会更改应用程序的代码，当然也有助于开发人员和用户界面设计人员在项目中的合作。

下面，我们更新 DinnersController 类，通过更新 2 个 action 方法原型（将返回类型从 void 更新为 ActionResult），使用视图模板来返回 HTML 用户界面响应。接着，我们调用 Controller 基类的 View()方法来返回 ViewResult 对象。

```
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) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
```

```
            return View("NotFound");
        else
            return View("Details", dinner);
    }
}
```

上面代码中，我们使用的 View() 辅助方法的原型如下：

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

View() 辅助方法的第一个参数是 view 模板文件，用来生成 HTML 响应。第二个参数是 model 对象，包含了 view 模板生成 HTML 响应所需要的数据。

在 Index() action 方法中，我们调用 View() 辅助方法，表示我们将使用 Index 视图模板来生成 HTML 形式的宴会（dinners）列表。我们传递一组 Dinner 对象给 view 模板，用来生成清单。
在 Details() action 方法中，我们通过 id 值检索特定的 Dinner 对象。如果有找到 Dinner 对象，则调用 View() 方法，使用 Details 视图模板展示检索到的 Dinner 对象。如果没有找到 Dinner 对象，则使用 NotFound 视图模板展示一个错误信息，表示 Dinner 对象不存在（一个重载的 View() 方法仅仅需要传入视图模板名称 – View(“NotFound”)）。

下面，我们开始实现 NotFound、Details、和 Index 视图模板。

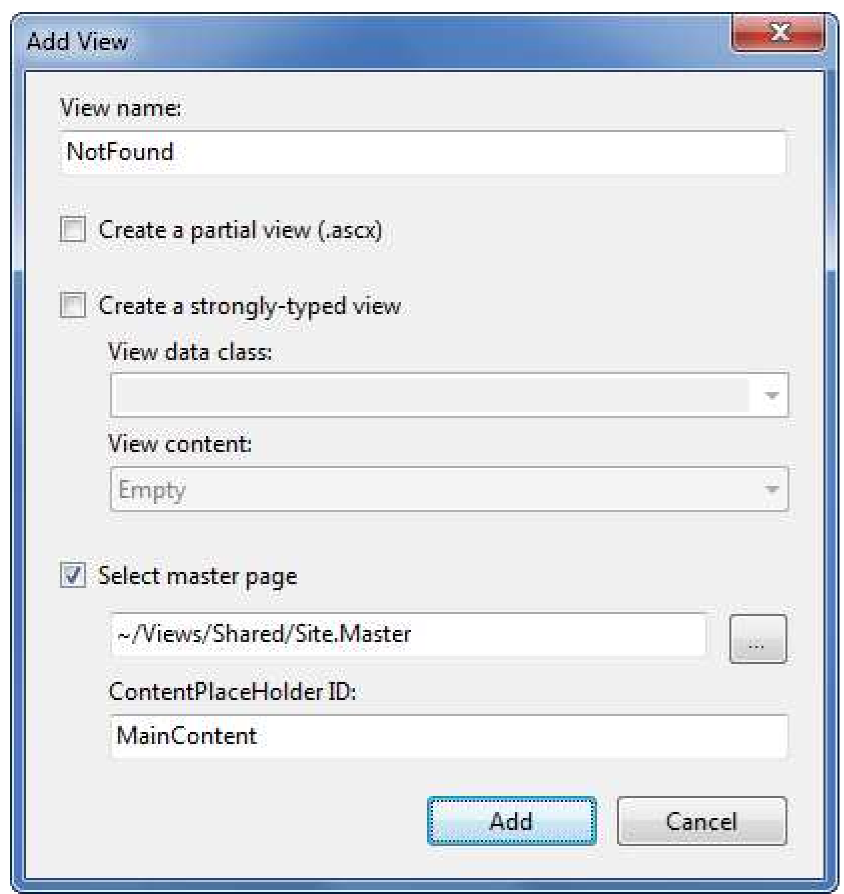
实现 NotFound 视图模板

我们开始实行 NotFound 视图模板 – 显示一条友好的错误信息，说明请求的宴会（dinner）没有找到。首先将光标定位在一个 controller action 方法体中，然后右键点击，在弹出菜单中选择 Add View 菜单项，就可以创建一个新的视图模板了，如下图所示。

```
//
// GET: /Dinners/Details/2

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
```

Add View...
Go To View...
Refactor
Organize Usings
Create Unit Tests...
Create Private Accessor

在弹出的 Add View 对话框，输入 NotFound，如下图所示。默认情况下，对话框中的视图名称就与光标所在的 action 方法中的视图名称一致，这里为 Details。因为我们第一步实现 NotFound 视图模板，因此在这里需要更名为 NotFound。



点击 Add 按钮，Visual Studio 将在\Views\Dinners 目录创建一个新的 NotFound.aspx 视图模板，如果该目录不存在，也会自动创建目录。

默认情况下，视图模板有 2 个 content regions，可以用来放置内容和代码。第一块允许我们定制 HTML 页面的标题，第二块允许我们定制 HTML 页面的主要内容（main content）。

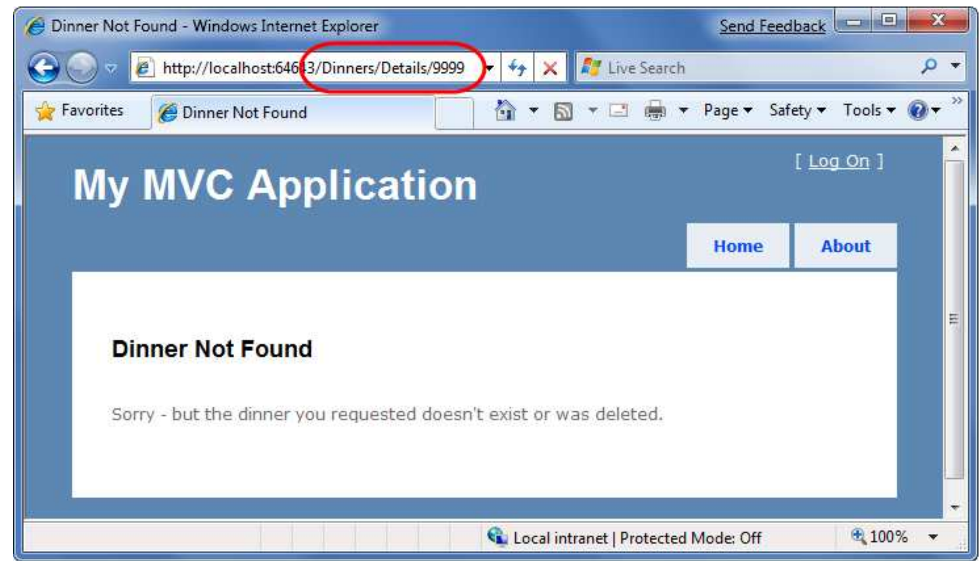
```
NotFound.aspx | DinnersController.cs
Server Objects & Events | (No Events)
1  <%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" MasterPageFile="~/Views/Sh
2
3  <asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
4      NotFound
5  </asp:Content>
6
7  <asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
8
9      <h2>NotFound</h2>
10
11 </asp:Content>
12
Design | Split | Source | <asp:Content#Main>
```

为了实现 NotFound 实体模板，我们添加了一些基本的内容，如下图所示：

```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Dinner Not Found
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Dinner Not Found</h2>
```

```
    <p>Sorry - but the dinner you requested doesn't exist or was deleted.</p>
</asp:Content>
```

下面运行 NerdDinner 范例程序，并在浏览器地址栏输入：/Dinners/Details/9999。该 dinner 在数据库中根本不存在，因此 DinnersController.Details() action 方法将呈现 NotFound 视图模板，如下图所示：



你可能会注意到上图中，我们创建的视图模板继承了其他 HTML 脚本，显示在主要内容周围。这是因为我们创建的视图模板使用了 Master Page 模板，在整个站点实现一致的布局效果。在本章后续部分，我们会介绍更多关于 Master Page 的技术。

实现 Details 视图模板

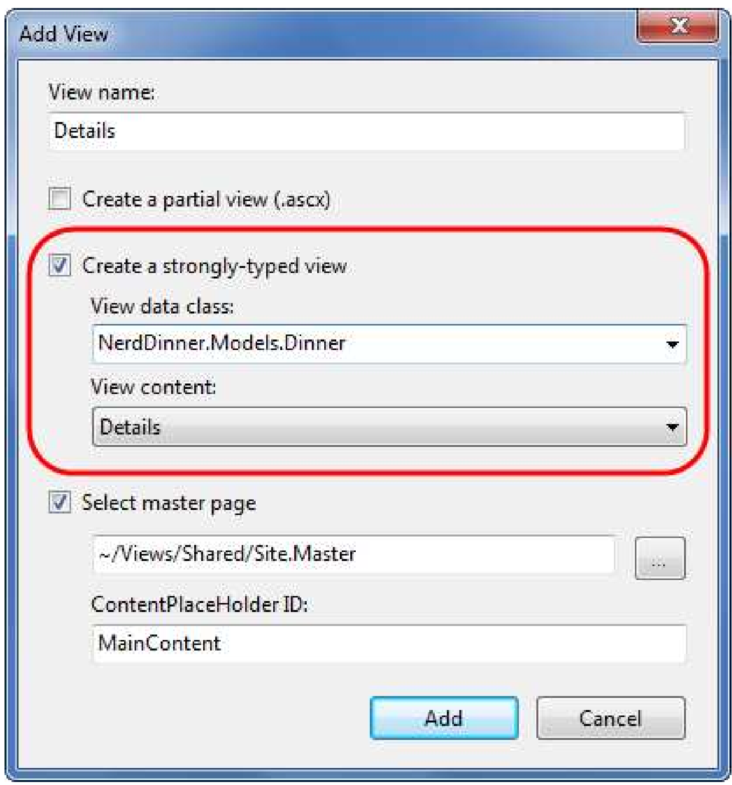
下面，我们将实现 Details 视图模板，用来显示单一 Dinner 模型的数据。首先，将光标定位到 Details action 方法（当然是在 DinnersController 控制器代码窗口），右键点击，在弹出的菜单中选择 Add View 菜单项。

```
//
// GET: /Dinners/Details/2

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
```

Add View...
Go To View...
Refactor
Organize Usings
Create Unit Tests...

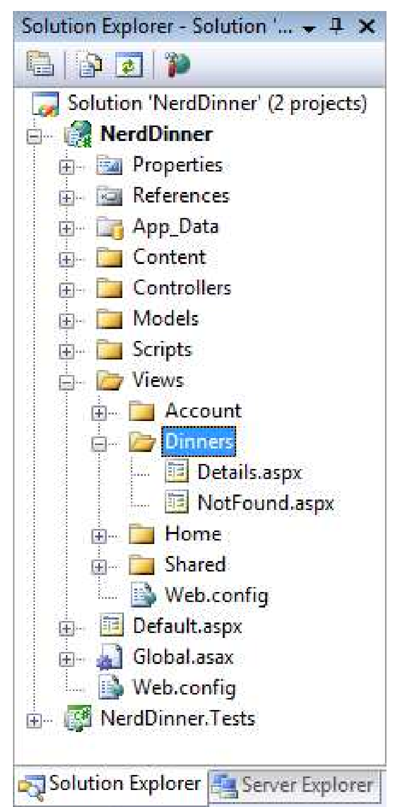
与上一步创建 NotFound 视图模板一样，将弹出 Add View 对话框。这里，我们使用默认的 Details 作为视

图名称，另外选择 Create a strongly-typed View（创建强类型视图）复选框，并从下拉列表框中选择从 Controller 传递到 View 的 Model 模型类型。针对正在创建的 Details 视图，我们将传递 Dinner 对象（该类型完整的名称是 NerdDinner.Models.Dinner）。如下图所示：



与 NotFound 视图模板创建 Empty View 不一样，这里我们选择 Details 模板。如上图所示，在 View Content 下拉列表框中选择 Details 选项。系统将根据传入的 Dinner 模型自动为 details 视图模板生成一个原始的实现。这样，方便我们快速开始视图模板的实现。

在点击 Add 按钮之后，Visual Studio 自动在\Views\Dinners 目录下创建 Details.aspx 视图模板。

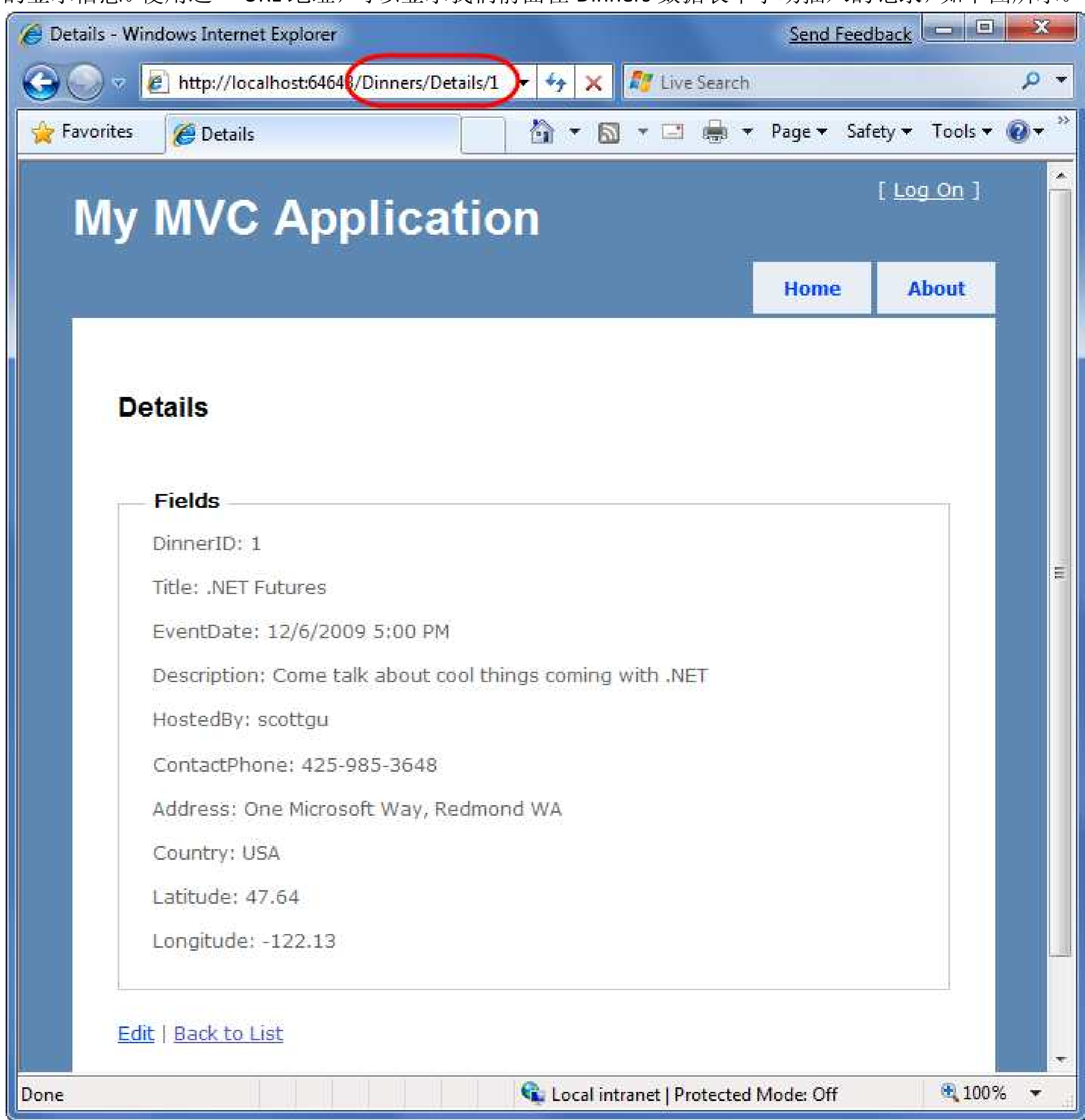


在 Details.aspx 视图模板中，已经根据 Dinner 模型创建一个初始的实现。VS 引擎使用.NET 发射机制，查询模型类的所有公共属性，并自动根据每一个字段的类型，添加相应的内容到视图模板中。

```
<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)) %>
```

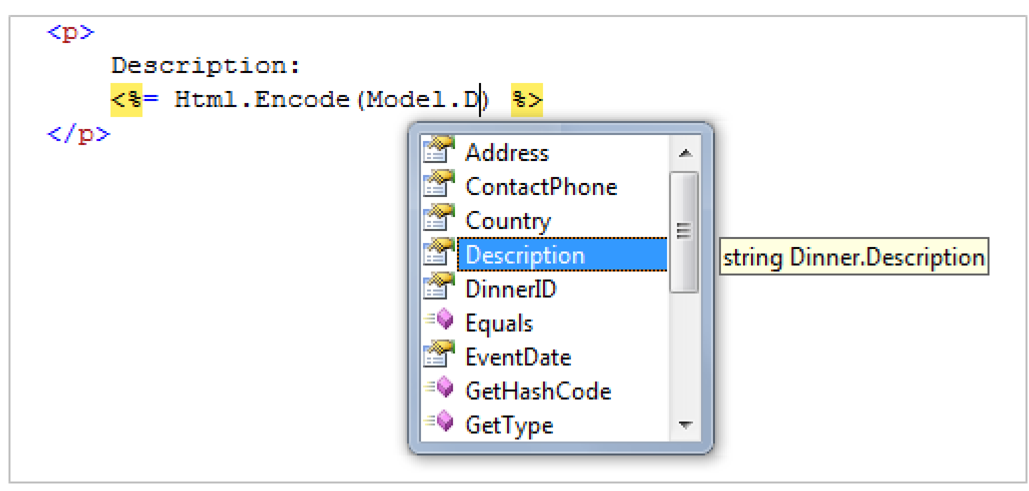
```
        </p>
        <p>
            Longitude:
            <%= Html.Encode(String.Format("{0:F}",Model.Longitude)) %>
        </p>
    </fieldset>

    <p>
        <%=Html.ActionLink("Edit","Edit", new { id=Model.DinnerID }) %>|
        <%=Html.ActionLink("Back to List", "Index") %>
    </p>
</asp:Content>
```

下面，我们运行 NerdDinner Web 应用程序，并在地址栏中输入/Dinners/Details/2，可以看到 Details 视图的显示信息。使用这一 URL 地址，可以显示我们前面在 Dinners 数据表中手动插入的记录，如下图所示。



就这么简单和快速，Details.aspx 页面提供了一个初始的实现。下面，我们将根据需要，进一步定制用户界面。当我们仔细查看 Details.aspx 页面时，发现该页面包含一些静态的 HTML 和嵌入展示代码。在视图模板呈现时，<% %> 代码块就会执行，<%= %> 代码块执行其中的代码，并呈现结果到视图的输出中展示。

我们也可以在视图中编写代码，通过使用强类型的 Model 属性，访问传入到 Controller 控制器的 Dinner 模型对象。在访问 Model 属性时，Visual Studio 编辑器提供了代码智能提示，如下图所示。

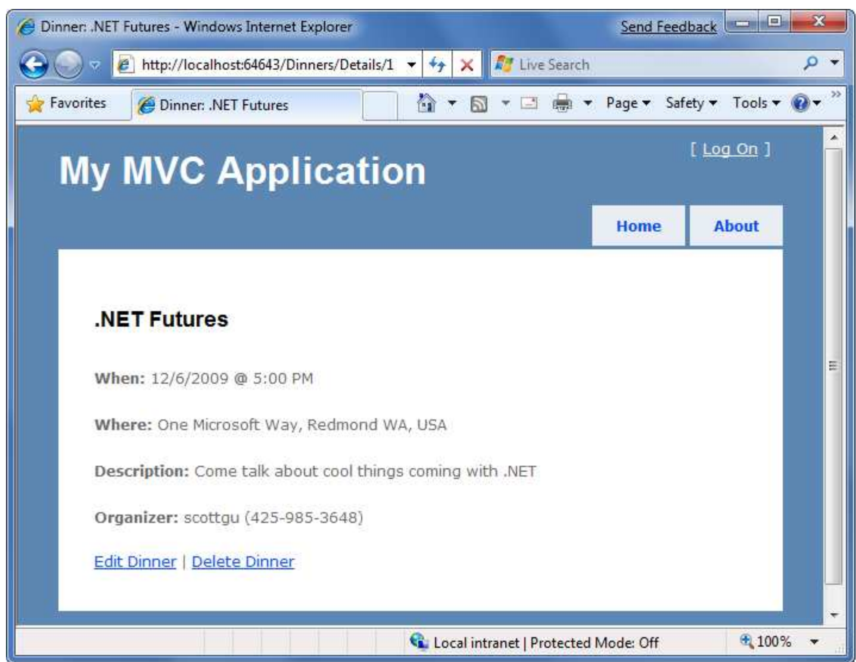


下面我们更改一下 Details 视图模板的代码，如下所示：

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Details:<%=Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
<h2><%=Html.Encode(Model.Title) %></h2>
    <p>
        <strong>When:</strong>
        <%=Model.EventDate.ToShortDateString() %>

        <strong>@</strong>
        <%=Model.EventDate.ToShortTimeString() %>
    </p>
    <p>
        <strong>Where:</strong>
        <%=Html.Encode(Model.Address) %>,
        <%=Html.Encode(Model.Country) %>
    </p>
     <p>
        <strong>Description:</strong>
        <%=Html.Encode(Model.Description) %>
    </p>
    <p>
        <strong>Organizer:</strong>
        <%=Html.Encode(Model.HostedBy) %>
        (<%=Html.Encode(Model.ContactPhone) %>)
    </p>
```

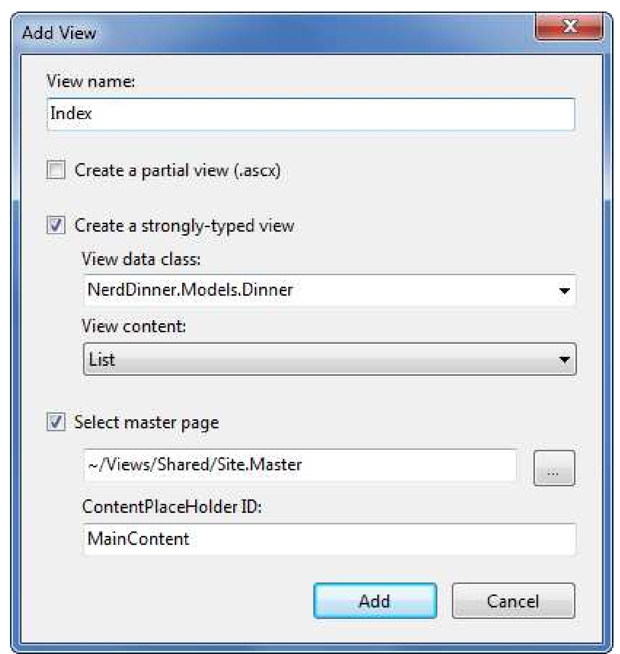
```
        <%=Html.ActionLink("Edit", "Edit", new { id=Model.DinnerID }) %> |
        <%=Html.ActionLink("Back to List", "Index") %>
</asp:Content>
```

再次访问/Dinners/Details/2 地址，这次呈现页面如下：

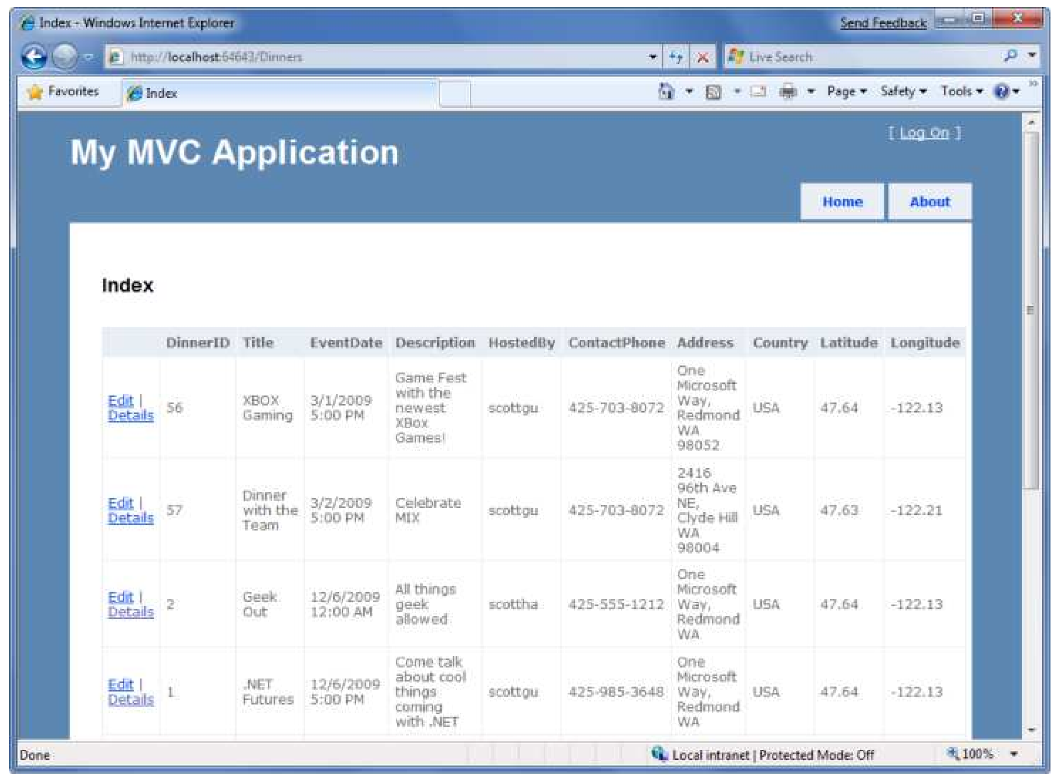


实现 Index 视图模板

现在我们开始实行 Index 视图模板 – 生成即将来临的宴会 Dinners 列表。首先，将光标定位在 Index action 方法（当然是在 DinnersController 类中），接着右键点击，选择弹出菜单中的 Add View 菜单项。在弹出 Add View 对话框中，视图模板的名称为 Index，选择 Create a strongly-typed View 复选框。这里，我们选择 NerdDinner.Models.Dinner 为模型类型，View Content 列表框选择 List，用来生成初始的列表脚本。



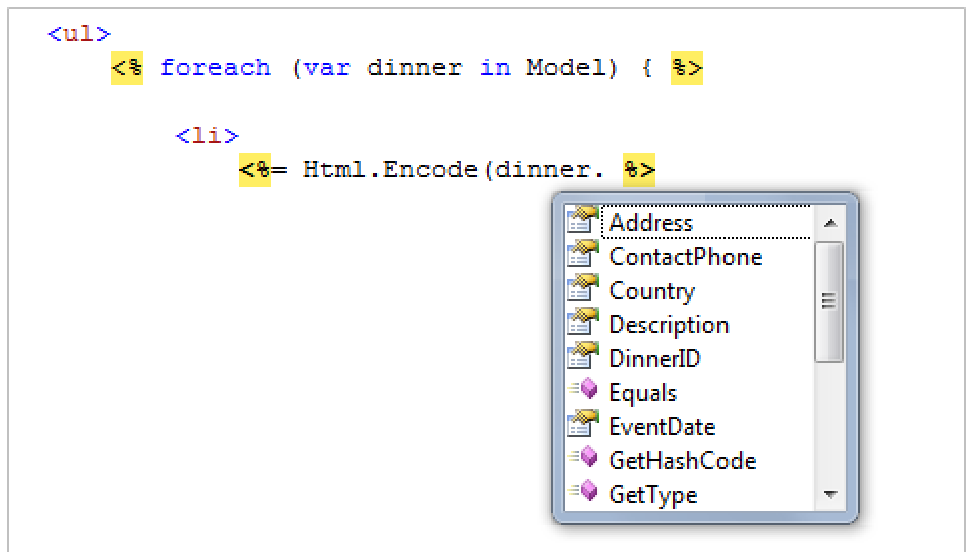
点击 Add 按钮，Visual Studio 自动在\Views\Dinners 目录创建新的 Index.aspx 视图模板，且自动实现一个初始的实现，将传入到视图的数据以 HTML 表格的形式呈现。当我们运行范例程序时，访问/Dinners/地址，将看到如下的效果：



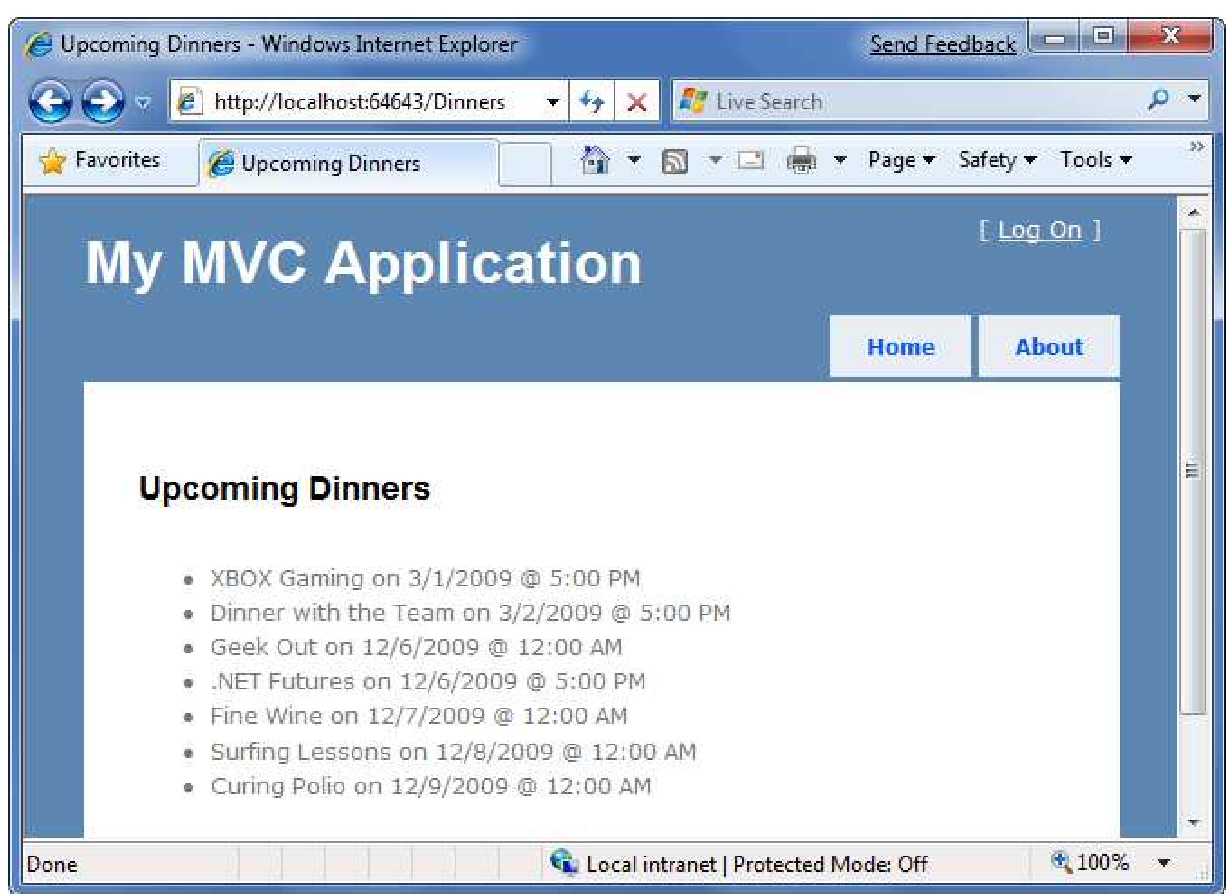
上面的表格显示即将来临的宴会 Dinners 列表，但是不是我们想要的格式。我们可以更新 Index.aspx 视图模板，展示较少的列，并使用 元素来呈现列表，而不是使用表格。示例代码如下：

```
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Upcoming Dinners</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>
```

上述代码使用 var 关键字在 foreach 循环语句中，循环 Model 中的每一个 dinner 对象。对 C# 3.0 不熟悉的人可能认为使用 var 意味着 dinner 对象是 late-bound，实际上这是编译器根据强类型 Model （类型为 IEnumerable<Dinner>）推断出 dinner 的类型，并编译本地变量 dinner 为 Dinner 类型，同时提供代码智能提示和编译期间的类型检查。



再次浏览器中访问/Dinners 地址时，发现视图已经更新了，显示效果如下：



现在看上去不错了，但还不够完美。最后一步是实现用户点击列表中每一条记录时，可以看到详细信息。通过使用 HTML 超链接指向 DinnersController 控制器的 Details action 方法，来实现这一点。

在 Index 视图中，有 2 种方法可以生成超链接。第一种方法是手动编写 HTML 的<a> 元素，<% %>代码块嵌在 HTML <a>元素中，如下所示：

```
<% 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>

<% } %>
```

第二种方法是利用 ASP.NET MVC 内置的 Html.ActionLink() 辅助方法，该方法支持通过编程的方式创建 HTML <a>元素，链接控制器 Controller 的另一个 action 方法。

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

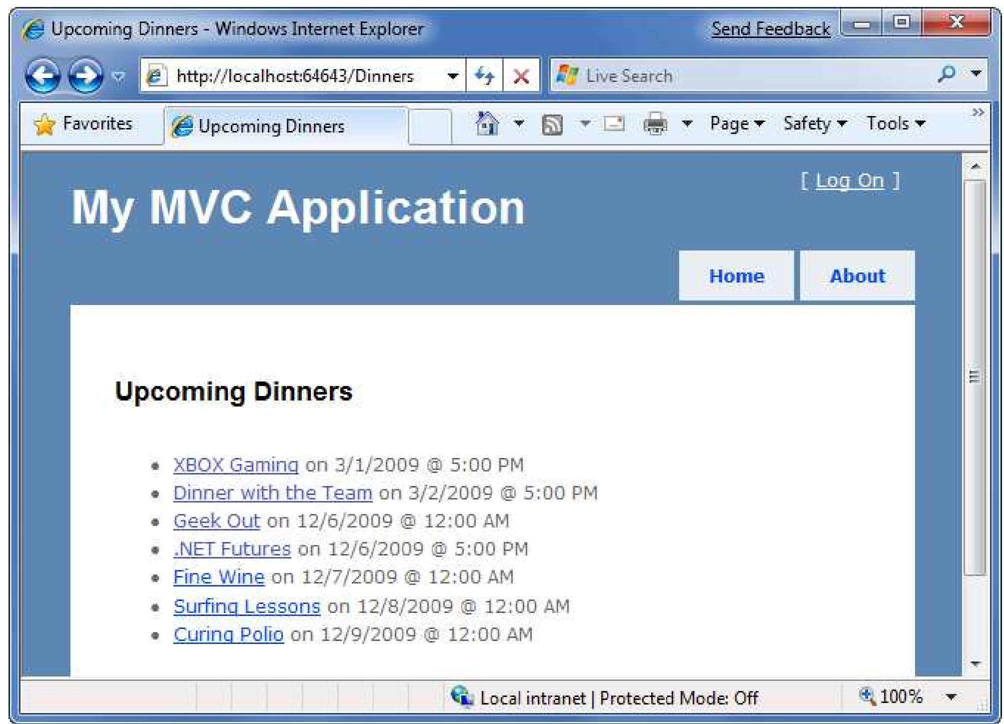
Html.ActionLink() 辅助方法的第一个参数是链接文本（示例代码是 dinner 的主题），第二个参数是控制器的 action 方法名称（示例代码是 Details 方法），第三个参数是一组传入给 action 方法的参数（以匿名类型的名称/值对的方式来实现）。上述代码中，指定链接的 dinner 的 id 参数。因为 ASP.NET MVC 默认的 URL 路由规则是{Controller}/{Action}/{id}，Html.ActionLink() 辅助方法将生成如下的输出：

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

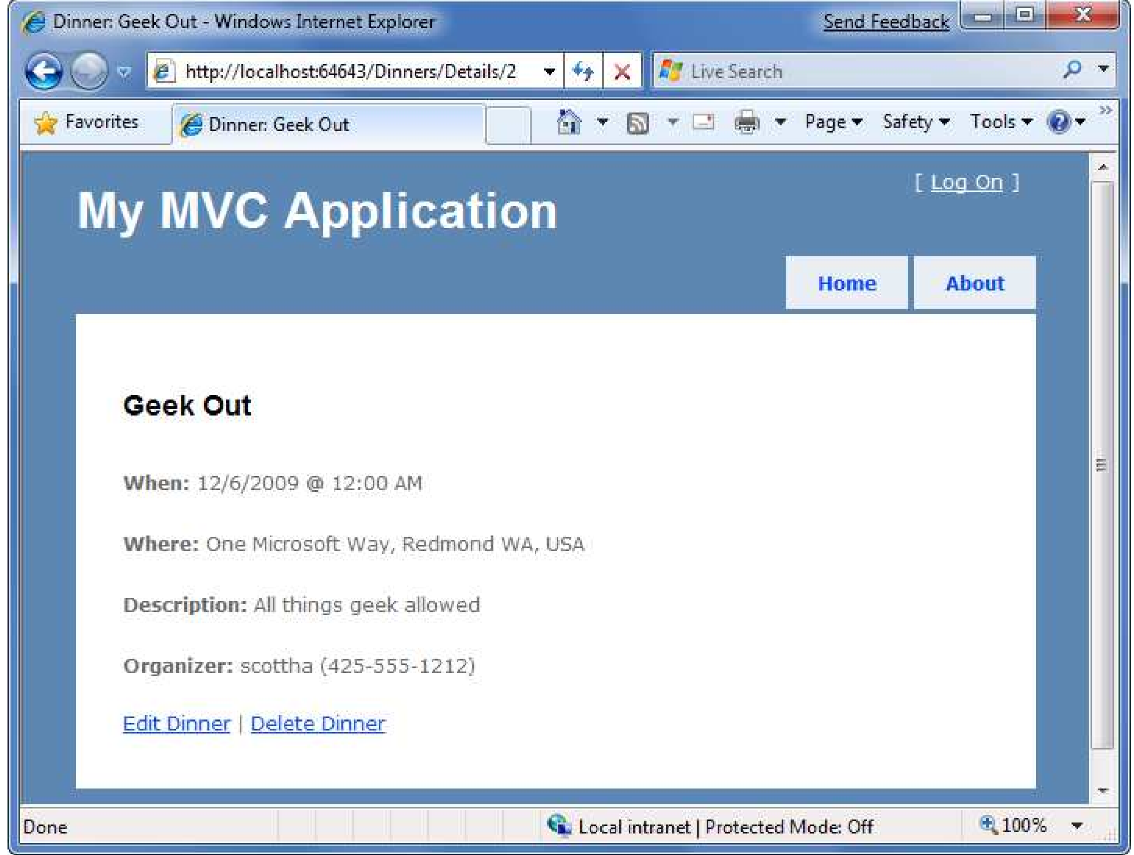
在 Index.aspx 视图文件中，我们将使用 Html.ActionLink() 辅助方法，让列表中的每一条记录指向正确的 details 地址：

```
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Upcoming Dinners</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>
```

现在，我们再次访问/Dinners 网址，将看到如下的 dinner list 页面：



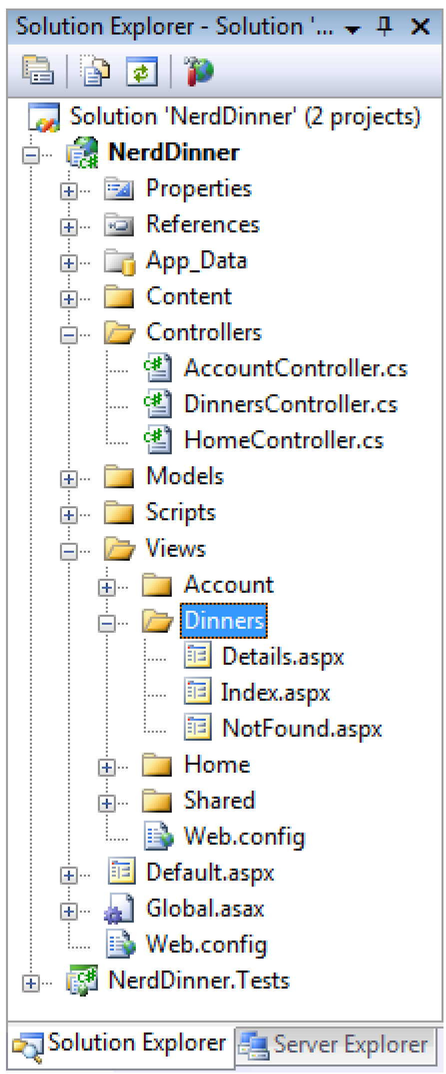
当我们点击 dinners 列表的任何一条记录，就可以进入 dinner 的详细页面：



命名规范和\Views 目录结构

默认情况下，ASP.NET MVC 应用程序使用约定的目录命名和结构，来解析视图模板。开发人员在 Controller 控制器中引用视图时，不必指定完整的路径。ASP.NET MVC 将自动寻找应用程序中

\Views\[ControllerName]\目录，并找到相应的视图模板文件。
例如，我们创建的 DinnersController 类 – 直接引用 3 个视图模板：Index / Details / NotFound。ASP.NET MVC 默认到应用程序的\Views\Dinners 目录下寻找这些文件。



从上图中，可以看到 3 个 Controller 控制器类（DinnersController、HomeController 和 AccountController），其中后面 2 个控制器是在创建项目时默认添加的，另外在\Views 目录中有 3 个子目录，分别对应 Controller 类。
Home 和 Accounts 控制器引用的视图会自动到相应的\Views\Home 和\Views\Account 目录中寻找。\Views\Shared 子目录则用来存放共享的视图模板，可供多个控制器使用。ASP.NET MVC 在解析一个视图模板时，首先检查\Views\[Controller] 特定目录，如果找不到，则继续到\Views\Shared 子目录中寻找。

关于对每一个视图模板的命名，建议的方法是视图模板保持和 action 方法相同的名称。例如，Index action 方法使用 Index 视图来呈现视图结果，Details action 方法使用 Details 视图来呈现结果。这样，有助于很快知道哪一个视图模板和哪一个 action 方法关联。

当视图模板和控制器调用的 action 方法有相同的名称时，开发人员可以不必显式指定视图模板的名称。我们仅仅需要传递模型对象给 View() 辅助方法（不需要制定视图名称），ASP.NET MVC 将自动推断出我们使用的视图 \Views\[controllerName]\[ActionName]，并展示结果。
这样，我们可以稍微简洁一些代码，避免在代码中重复 view 的名称。

```
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);
    }
}
```

上面的代码是实现 Dinner 列表和显示详细信息的全部代码。

创建、更新、删除记录

我们已经介绍了控制器和视图，已经如何实现数据列表和详细信息的显示。接下来，我们进一步扩展 DinnersController 类实现编辑、创建和删除 Dinners 记录。

DinnersController 处理 URL 请求
之前我们添加了 2 个 action 方法到 DinnersController 控制器，实现支持如下 2 个 URL：/Dinners 和 /Dinners/Details/[id]。

URL	VERB	Purpose
/Dinners/	GET	Display an HTML list of upcoming dinners.
/Dinners/Details/[id]	GET	Display details about a specific dinner.

现在，我们增加更多的 action 方法，实现如下 3 个 URL 地址：
/Dinners/Edit/[id]
/Dinners/Create

/Dinners/Delete/[id]

这些 URL 分别支持编辑已存在的 Dinners，创建新的 Dinners 和删除 Dinners。

我们将同时支持 HTTP GET 和 HTTP POST 方法访问这些新的 URL 地址。HTTP GET 对这些 URL 地址的请求将显示初始的 HTML 视图（edit 将显示 Dinner 数据；create 将显示一个空白的窗口；delete 则显示一个确认对话框）。HTTP POST 对这些 URL 的请求将保存/更新/删除 DinnerRepository 的 Dinner 数据。

URL	VERB	Purpose
/Dinners/Edit/[id]	GET	Display an editable HTML form populated with Dinner data.
	POST	Save the form changes for a particular Dinner to the database.
/Dinners/Create	GET	Display an empty HTML form that allows users to define new Dinners.
	POST	Create a new Dinner and save it in the database.
/Dinners/Delete/[id]	GET	Display a confirmation screen that asks the user whether they want to delete the specified dinner.
	POST	Deletes the specified dinner from the database.

下面开始实行 edit 方案。

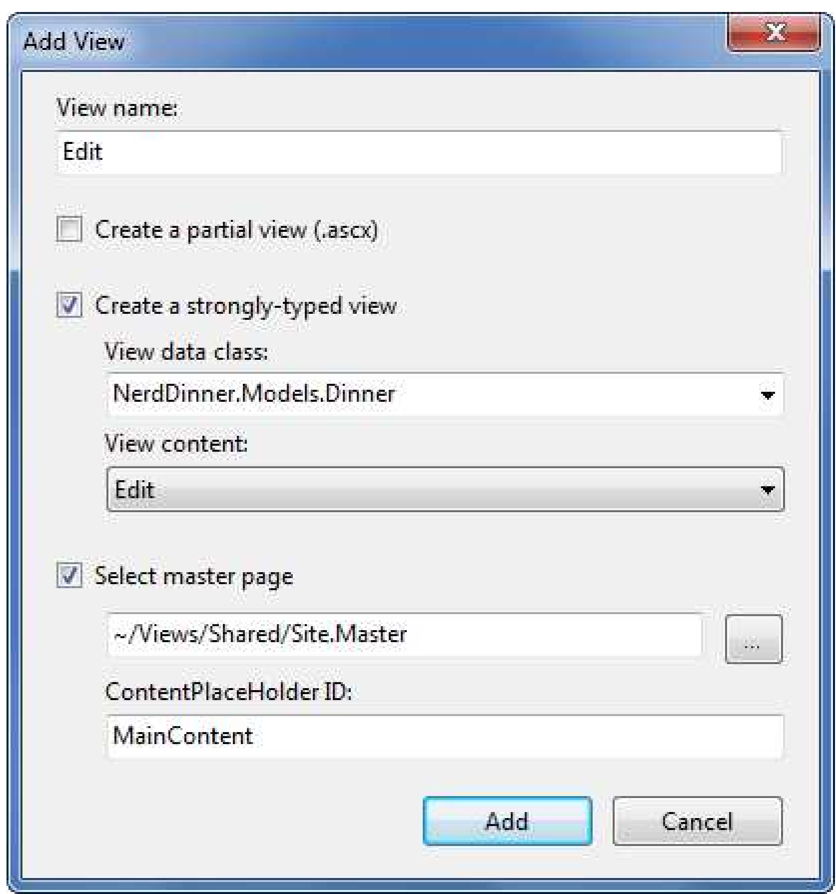
实现 HTTP-GET 编辑 Action 方法

我们首先实现 edit action 方法的 HTTP GET 动作。在访问/Dinners/Edit/[id]地址时，该方法将被调用。实现代码如下：

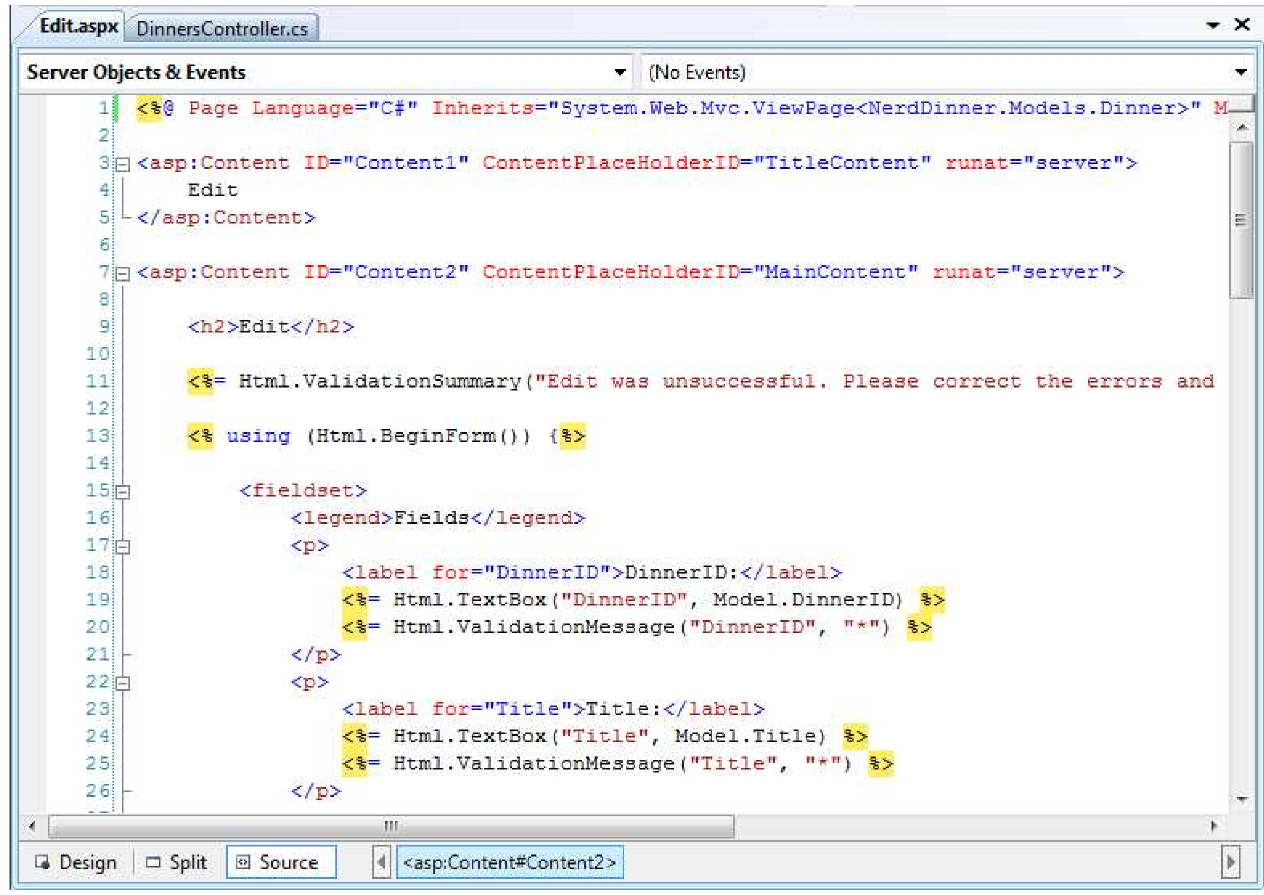
```
//
// GET: /Dinners/Edit/2

public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    return View(dinner);
}
```

上述代码使用 DinnerRespository 检索一个 Dinner 对象，接着在一个视图模板中程序 Dinner 对象。因为在 View() 辅助方法中没有显式传入视图的名称，因此它将基于惯例，使用默认的视图模板：/Views/Dinners/Edit.aspx。

现在，我们开始创建视图模板，步骤与前面创建视图模板一样。在 DinnersController 的 Edit action 方法中点击鼠标右键，选择 Add View 菜单项。在弹出的 Add View 对话框，设置 View data class 为 NerdDinner.Models.Dinner，表示将传入 Dinner 对象（Model）给视图模板。同时，设置 View content 为 Edit，表示将创建一个初始的编辑模板。如下图所示：



在点击 Add 按钮后，Visual Studio 自动在\Views\Dinners 目录中创建一个新的 Edit.aspx 视图模板文件，并自动在 VS 编辑器中打开，提供一个初始的 Edit 模板的实现。

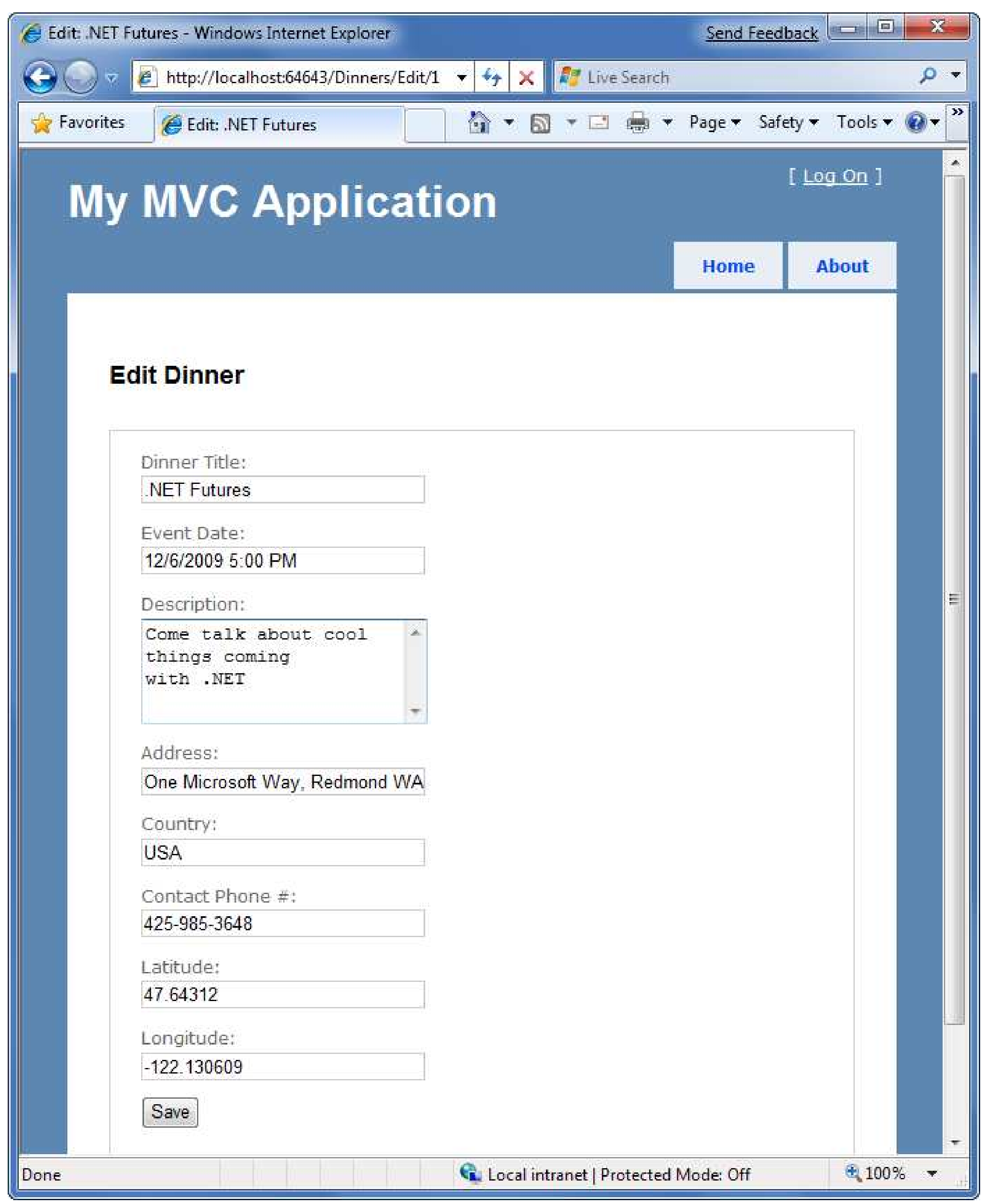


下面我们对默认生成的 Edit 视图进行一些更新，代码如下（删除了一些不想公开的属性）：

```
<asp:Content ID="Content2" 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">EventDate:</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">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>
```

现在运行应用程序，请访问/Dinners/Edit/2 网址，将看到如下的页面：



上述页面的 HTML 脚本如下所示，就是一个标准的 HTML-有一个<form>元素。在点击 Save 保存按钮后，向 /Dinners/Edit/2 执行 HTTP POST 动作。其中 HTML 元素 - <input type=”text” />元素用来展示可编辑的文本框。

```
<form action="/Dinners/Edit/1" method="post">

    <fieldset>
        <p>
            <label for="Title">Dinner Title:</label>
            <input id="Title" name="Title" type="text" value=".NET Futures" />
        </p>
        <p>
            <label for="EventDate">Event Date:</label>
            <input id="EventDate" name="EventDate" type="text" value="12/6/2009 5:00 PM" />
        </p>

        <!-- Some Fields Omitted for Brevity -->

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>

</form>
```

Html.BeginForm() 和 Html.TextBox() HTML 辅助方法

Edit.aspx 视图模板使用了一些 HTML 辅助方法：Html.ValidationSummary()，Html.BeginForm()，Html.TextBox()，和 Html.ValidationMessage()等等。处理生成 HTML 脚本外，这些辅助方法还提供了内置的错误处理和验证支持。

Html.BeginForm() 辅助方法

Html.BeginForm() 辅助方法用来输出 HTML <form>元素。在 Edit.aspx 视图模板中，你会发现我们使用了 C#的 using 语句。左括号{ 表示开始<form>元素，右括号 } 表示结束</form> 元素：

```
<% using (Html.BeginForm()) { %>
    <fieldset>
        <!-- Fields Omitted for Brevity -->
        <p>
            <input type="submit" value="Save"/>
        </p>
    </fieldset>
<% } %>
```

如果你认为使用 using 语句并不直观，你也可以使用 Html.BeginForm() 和 Html.EndForm() 组合，示例代码如下：

```
<% Html.BeginForm();   %>
    <fieldset>
        <!-- Fields Omitted for Brevity -->
        <p>
            <input type="submit" value="Save"/>
        </p>
    </fieldset>
<% Html.EndForm(); %>
```

调用不传参数的 Html.BeginForm() 方法将输出一个 form 元素，和 HTTP-POST 方法和当前请求的 URL 地

址，这就是为什么 Edit 视图生成如下 <form action="/Dinners/Edit/1" method="post"> 元素。当然，如果需要提交到不同的 URL 地址，我们可以传入显式的参数给 Html.BeginForm() 方法。

Html.TextBox() 辅助方法

Edit.aspx 视图使用 Html.TextBox() 辅助方法输出<input type="text" />元素：

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

上面的 Html.TextBox() 方法接收了仅仅一个参数 – 用来同时指定<input type="text" />元素的 id/name 属性，以及填充文本框值的 Model 属性。例如，传入的 Dinner 对象的 Title 属性值为".NET Futures"，因此 Html.TextBox("Title")方法将输出<input id="Title" name="Title" type="text" value=".NET Futures" />。另外，我们也可以使用 Html.TextBox() 的第一个参数来指定元素的 id/name 属性值，并显式给第二个参数传递值。

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

如果需要对输出结果进行格式化输出，则可以使用.NET 内置的 String.Format() 静态方法。Edit.aspx 视图模板使用这一方法来对 EventDate 值进行格式化（DateTime 类型），不显示秒信息：

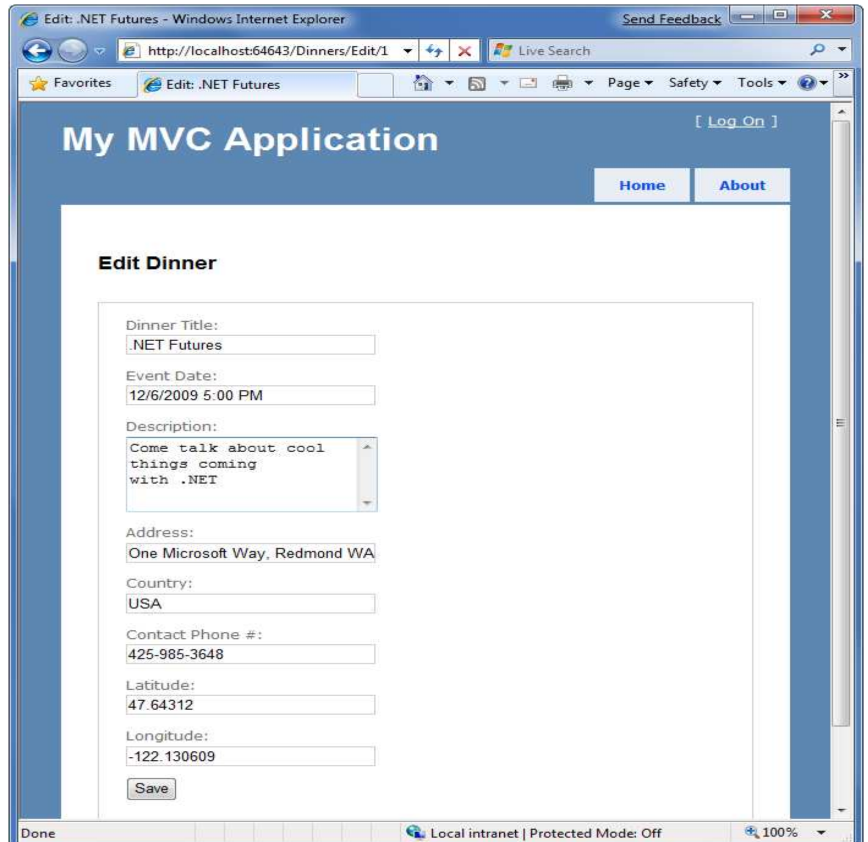
```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

Html.TextBox() 方法的第三个可选参数可用来输出额外的 HTML 属性。如下代码显示如何对<input type="text" />元素呈现额外的 size="30" 和 class="mycssclass" 属性。注意：这里为了避免冲突 C#的关键字 Class 冲突，采用@前缀。

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

实现 HTTP-POST 的 Edit Action 方法

现在，我们已经实现了支持 HTTP-GET 的 Edit action 方法。当用户请求/Dinners/Edit/2 地址时，接收一个 HTML 页面。



点击 Save 保存按钮，将触发表单提交到/Dinners/Edit/2 网址，并通过 HTTP POST 提交<input> 表单中的值。下面，我们开始实现 HTTP POST 的 Edit action 方法 – 负责处理保存操作。

通过添加一个重载的 Edit action 方法到 DinnersController 类中，并设置 AcceptVerbs 属性，表示该方法负责处理 HTTP POST 动作。

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
   ...
}
```

当对重载的 action 方法添加[AcceptVerbs] 属性后，ASP.NET MVC 根据进来的 HTTP 动作，自动分发请求给合适的 action 方法。HTTP POST 请求/Dinners/Edit/[id] 将有上述 Edit 方法负责处理，然而所有其他的 HTTP 请求/Dinners/Edit/[id] 将有之前定义的 Edit 方法负责（该方法没有[AcceptVerbs]属性）。

获取表单提交的值

在 HTTP POST 的 Edit 方法中，有很多方法可以获取表单参数值。一个简单的办法是使用 Controller 基类的 Request 属性来访问 form 集合，并直接获取提交的参数值：

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

    // Retrieve existing dinner
    Dinner dinner = dinnerRepository.GetDinner(id);

    // Update dinner with form posted values
    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"];

    // Persist changes back to database
    dinnerRepository.Save();

    // Perform HTTP redirect to details page for the saved Dinner
    return RedirectToAction("Details", new { id = dinner.DinnerID });
}
```

上述方法有一点繁琐，特别是增加异常处理逻辑之后。

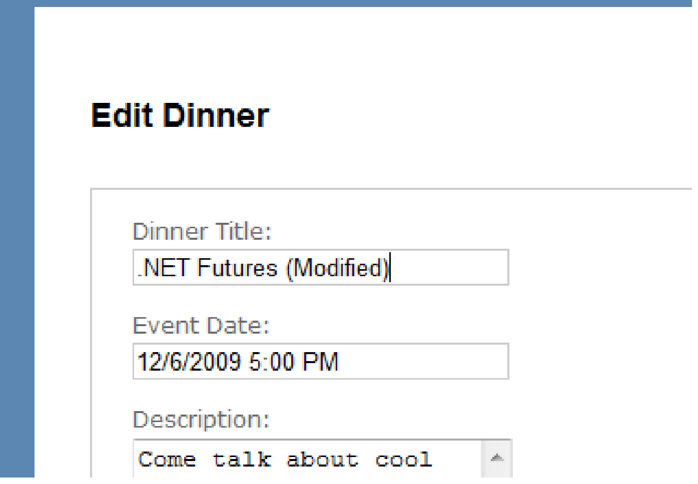
一个更好的方法是使用 Controller 基类的内置方法 UpdateModel()。该方法支持使用传入的表单参数更新

对象的属性，它使用反射机制来解析对象的属性名称，接着基于客户端传入的参数值自动赋值给对象相关属性。

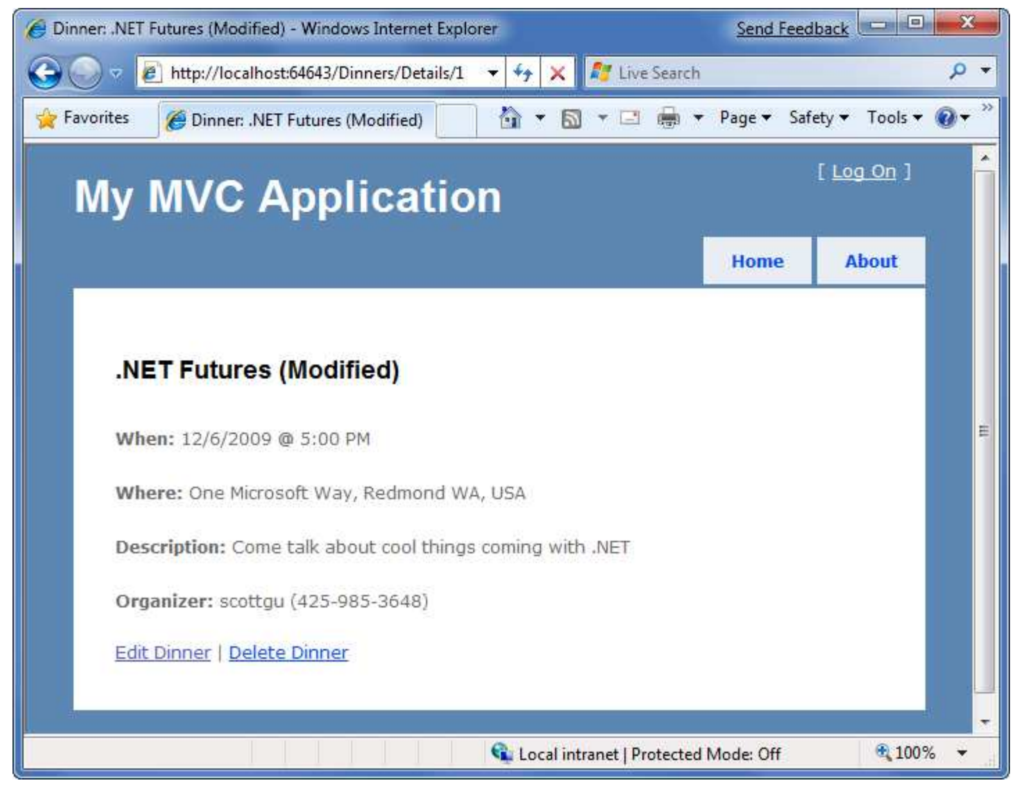
下面，我们使用 UpdateModel() 方法来实现之前的 HTTP-POST Edit Action 方法：

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
    Dinner dinner = dinnerRepository.GetDinner(id);

    UpdateModel(dinner);
    dinnerRepository.Save();
    return RedirectToAction("Details", new { id = dinner.DinnerID });
}
```

再次方法/Dinners/Edit/2 网址，并更改 Dinner 的标题和事件日期：



点击 Save 保存按钮，执行表单提交，触发 Edit 方法的调用，并将更新的值持久化到数据库。接着，重定向到详细页面 – Details 视图（显示最新保存的数据）。



处理编辑异常

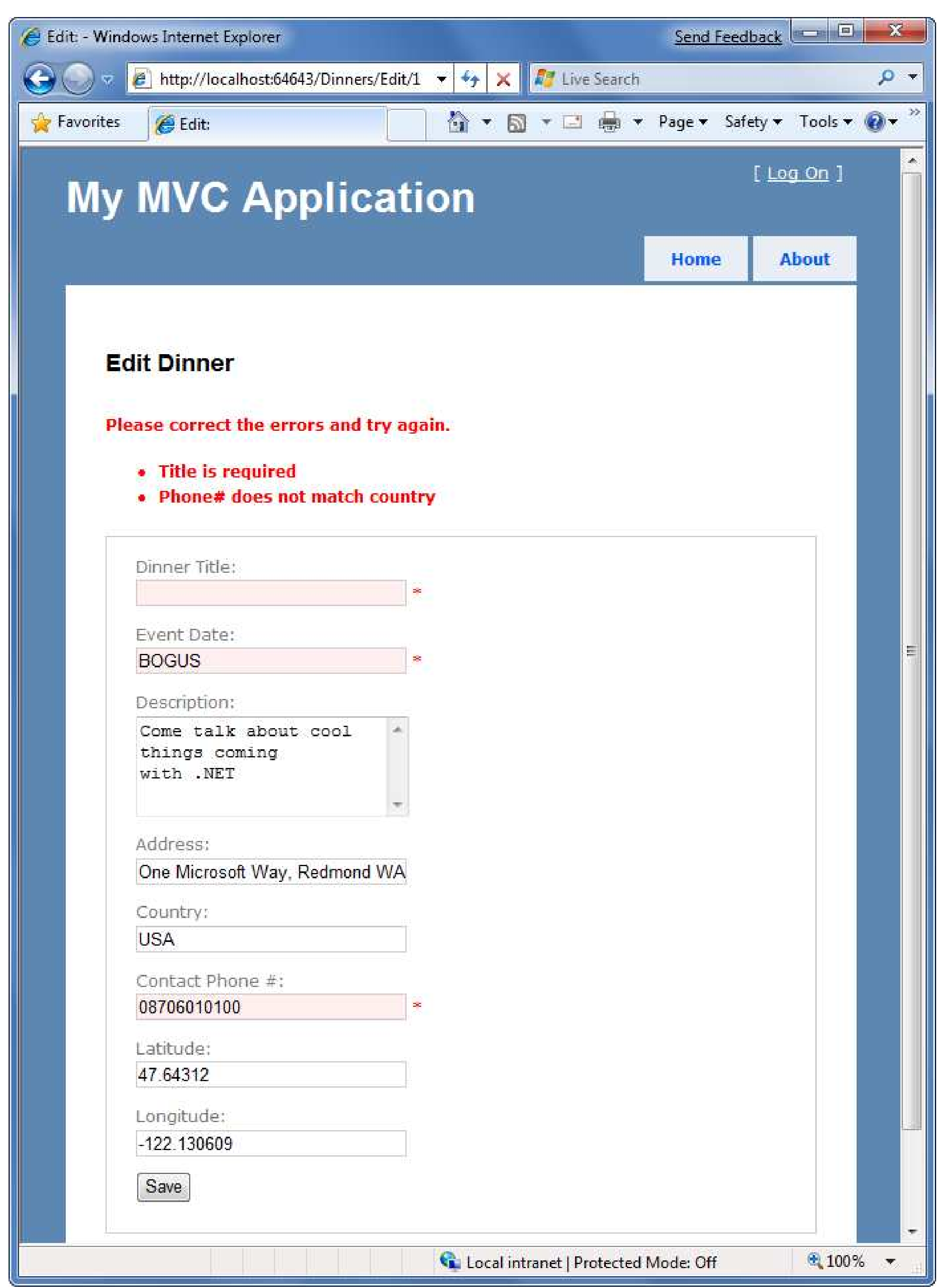
当前 HTTP-POST 实现方法工作正常 – 当然也会出现异常。当用户在编辑表单时犯错误了，我们需要确保表单显示错误信息，指导用户去纠正。这包括用户提交了错误的数据（如错误的日期格式），或者存在业务规则冲突等等。当发生错误是，表单需要保持用户初始录入的数据，这样他们不必重复录入一遍。这个过程需要重复多次，直到最终成功提交表单。

ASP.NET MVC 包括一些友好的内置功能，使异常处理和重新显示表单更容易。为了演示这些功能，下面我们再次更新 Edit Action 方法，代码如下：

```
//
// 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 {

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

        return View(dinner);
    }
}
```

上述代码与之前的实现基本一致，除了增加了一个 try/catch 异常捕获代码块。如果调用 UpdateModel() 方法出现异常，或者保存 DinnerRepository 时出现异常（如果我们试图保存一个无效的对象-规则冲突，将抛出异常），异常捕获代码块将触发执行。在 catch 代码块中，首先遍历 Dinner 对象中所有规则冲突，并添加到 ModelState 对象中，接着重新显示视图。

下面为了模拟异常信息，重新运行应用程序，编辑 Dinner，并将 Title 清空，事件日期 EventDate 设置为“ BOGUS”，电话格式等等，然后点击 Save 保存按钮，现在 HTTP POST 触发 Edit 方法，但不能成功保存 Dinner 信息（因为有异常么），并重新显示表单：



现在，应用程序有更好的异常处理机制。有无效输入的文本框通过红色*进行提示，并且错误信息也显示在界面上。表单同时也保留了用户最初录入的信息 – 这样他们不必重复录入。所有这些是如何实现的呢？这是因为我们使用了一些内置的 ASP.NET MVC 功能，使输入验证和异常处理更加容易。

理解 ModelState 和验证 HTML 辅助方法

Controller 类有一个 ModelState 属性集合，可以用来提示传递到视图的 model 对象是否有错误。ModelState

中 Error 记录识别模型属性的名称和错误信息，并允许指定友好的错误信息。

在 UpdateModel() 辅助方法给 model 对象的属性赋值时，如遇到异常或错误，会自动写道 ModelState 集合中。例如，Dinner 对象的 EventDate 属性的类型为 DateTime，当 UpdateModel() 方法不能将"BOGUS"字符串赋值给 EventDate，UpdateModel() 方法将添加一条记录到 ModelState 集合，说明在给该属性赋值时，发生错误。

开发人员也可以显式写代码，添加错误记录到 ModelState 集合中，如下代码所示。我们在 catch 错误处理异常块中，根据 Dinner 对象中的 Rule Vilations（规则冲突）信息，添加到 ModelState 集合中。

```
[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 {

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

        return View(dinner);
    }
}
```

Html 辅助方法和 ModelState 集成

HTML 辅助方法，如 Html.TextBox()，在输出内容时，会检查 ModelState 集合。如果发现该属性有异常或错误，将呈现用户输入的内容和 CSS 错误类。

例如，在 Edit 视图中，我们使用 Html.TextBox() 辅助方法呈现 Dinner 对象的 EventDate 属性：

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

当有错的时候呈现视图时，Html.TextBox() 方法检查 ModelState 集合，检查是否有错误关联到 Dinner 对象的 EventDate 属性。当发现有错误时，将显示用户提交的"BOGUS"输入作为参数值，同时对<input type="textbox" />元素添加 CSS 错误类，如下所示：

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

你可以定制 CSS 错误类的样式。默认的 CSS 错误类 – input-validation-error 定义在\content\site.css 文件

中，样式定义如下：

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

CSS 样式对输入无效元素的文本框显示如下：

Event Date:
BOGUS *

Html.ValidationMessage() 辅助方法

Html.ValidationMessage() 辅助方法用来输出特定 Model 属性相关的 ModelState 错误信息：

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

上述代码输出： The value ‘BOGUS' is invalid

Html.ValidationMessage() 辅助方法也支持第二个参数，允许开发人员覆盖错误消息：

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

上述代码输出：*，而不是默认的错误信息。

Html.ValidationSummary() 辅助方法

Html.ValidationSummary() 辅助方法将呈现总结的错误消息，通过
元素列出在 ModelState 集合中所有详细的错误消息：

Edit Dinner

Please correct the errors and try again.

- Title is required
- Phone# does not match country

Dinner Title:
*

Event Date:
BOGUS *

Description:
Come talk about cool things coming

Html.ValidationSummary() 辅助方法接收一个可选的字符串参数 – 定义一个概括性的错误消息，并显示在所有详细错误信息的前面：

```
<%= Html.ValidationSummary("Please correct the errors and try again.") %>
```

你也可以定义 CSS 设置错误消息的样式。

使用 AddRuleViolations 辅助方法

初始的 HTTP-POST Edit 的实现方法使用了一个 foreach 循环语句，遍历 Dinner 对象的 Rule Violations，并添加到 controller 的 ModelState 集合：

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

    return View(dinner);
}
```

为了使代码更简洁一点，我们添加 ControllerHelpers 类到 NerdDinner 项目中，并实现了 AddRuleViolations 扩展方法，添加了一个对 ASP.NET MVC ModelStateDictionary 类的辅助方法。该扩展方法封装了使用 RuleViolation 错误信息填充 ModelStateDictionary 集合类的逻辑：

```
public static class ControllerHelpers {

   public static void AddRuleViolations(this ModelStateDictionary modelState, IEnumerable<RuleViolation> errors) {

       foreach (RuleViolation issue in errors) {
           modelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
       }
   }
}
```

接下来，我们更新 HTTP-POST Edit 方法，使用上述扩展方法实现 Dinner 的 Rule Violations 填充 ModelState 集合。

完成 Edit Action 方法的实现

下面的代码实现了控制器中 Edit 的所有逻辑：

```
//
// 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.AddRuleViolations(dinner.GetRuleViolations());
        return View(dinner);
    }
}
```

关于 Edit 方法的实现的优点，不仅 Controller 类，而且 View 视图模板都不必关心 Dinner 模型类的特定验证方法或者业务规则。以后，我们可以针对 Model 类增加额外的业务规则，而不必要求 Controller 和 View 更改代码。这样，我们可以根据需求，以最小的更改代码量，灵活改进应用程序。

实现 HTTP-GET 的 Create Action 方法

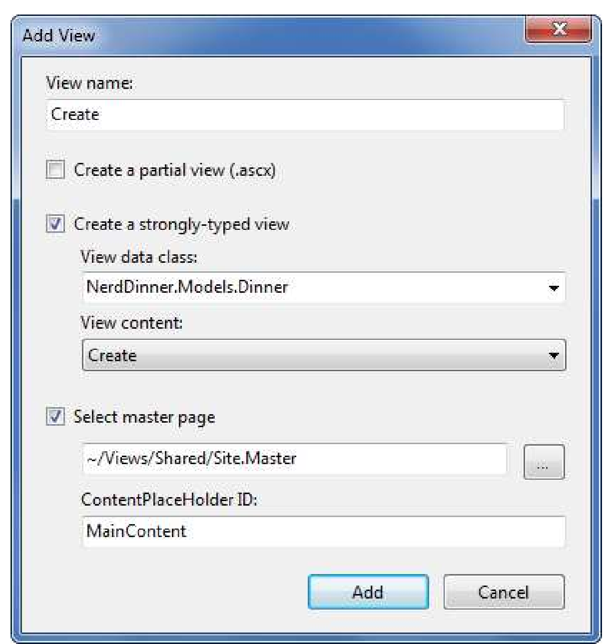
前面我们完成了实现 DinnersController 类的 Edit 方法，接下来实现 Create 方法 – 允许用户添加新的 Dinners 记录。

下面开始实现 Create Action 方法的 HTTP GET 行为。在用户访问/Dinners/Create 地址时，将调用该方法，实现代码如下：

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

上述代码创建一个 Dinner 对象，并对 EventDate 属性赋值 DateTime.Now.AddDays(7)。接着基于新的 Dinner 对象呈现视图。因为我们没有显式传入名称给 View() 辅助方法，因此它将基于惯例和默认路径解析视图模板：/Views/Dinners/Create.aspx。

下面创建视图模板。右键在 Create Action 方法内点击，并选择 Add View 菜单项。在 Add View 对话框表示将传入 Dinner 对象给视图模板，并选择 Create 模板。如下图所示：

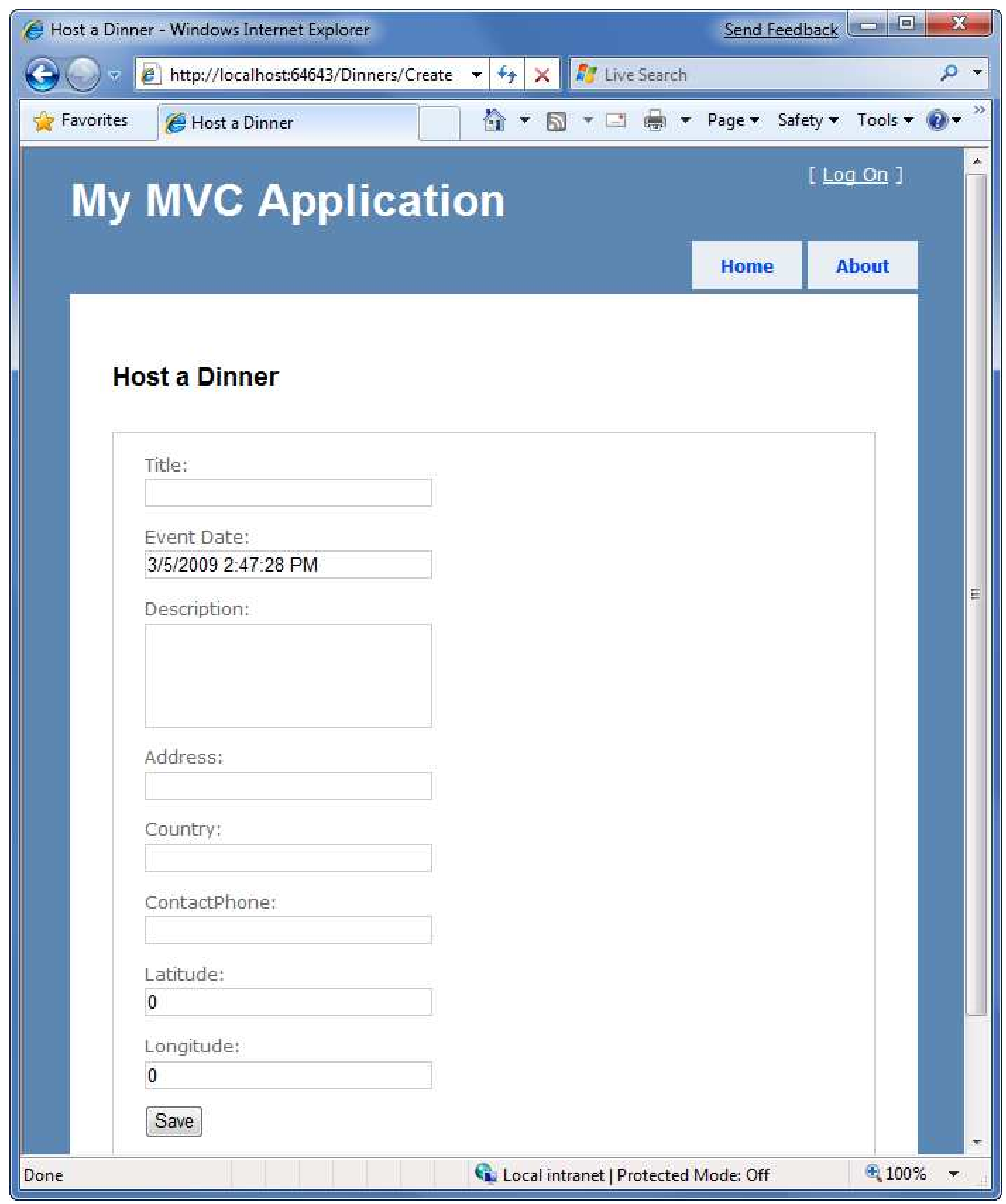


当点击 Add 按钮，Visual Studio 自动在\Views\Dinners 目录下创建 Create.aspx 视图文件。下面对默认创建的 Create.aspx 文件进行修改，修改之后的代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="TitleContent" runat="server">
    Host a Dinner
</asp:Content>

<asp:Content ID="Content3" 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">EventDate:</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>
```

现在，我们运行 NerdDinner 应用程序，并在浏览器中访问/Dinners/Create 网址，将根据 Create Action 方法的实现，呈现如下界面：



实现 HTTP-POST 的 Create Action 方法

我们已经实现了 Create Action 方法的 HTTP-GET 版本。当用户点击 Create 创建按钮时，将执行表单提交到/Dinners/Create 地址，并使用 HTTP POST 动作提交 HTML 表单参数。

现在开始实现 HTTP POST 动作的 Create Action 方法。添加一个重载的 Create action 方法到 DinnersController 类，并设置 AcceptVerbs 属性，表示该方法负责处理 HTTP POST 的请求：

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

在 HTTP-POST 的 Create 方法中，有很多办法可以访问表单提交过来的参数值。

一个方法是创建新的 Dinner 对象，并使用 UpdateModel() 辅助方法（就像 Edit 方法一样），将表单传递过来的值赋给 Dinner 对象。接着，添加 Dinner 对象到 DinnerRepository，并持久化到数据库，并重定向到 Details action 方法，并显示新创建的 Dinner 对象：

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

另外一个方法是 Create() action 方法接收一个 Dinner 对象作为方法参数。接着 ASP.NET MVC 自动实例化一个新的 Dinner 对象，并使用表单输入的参数赋值给 Dinner 对象的属性，然后传递给 action 方法，代码如下：

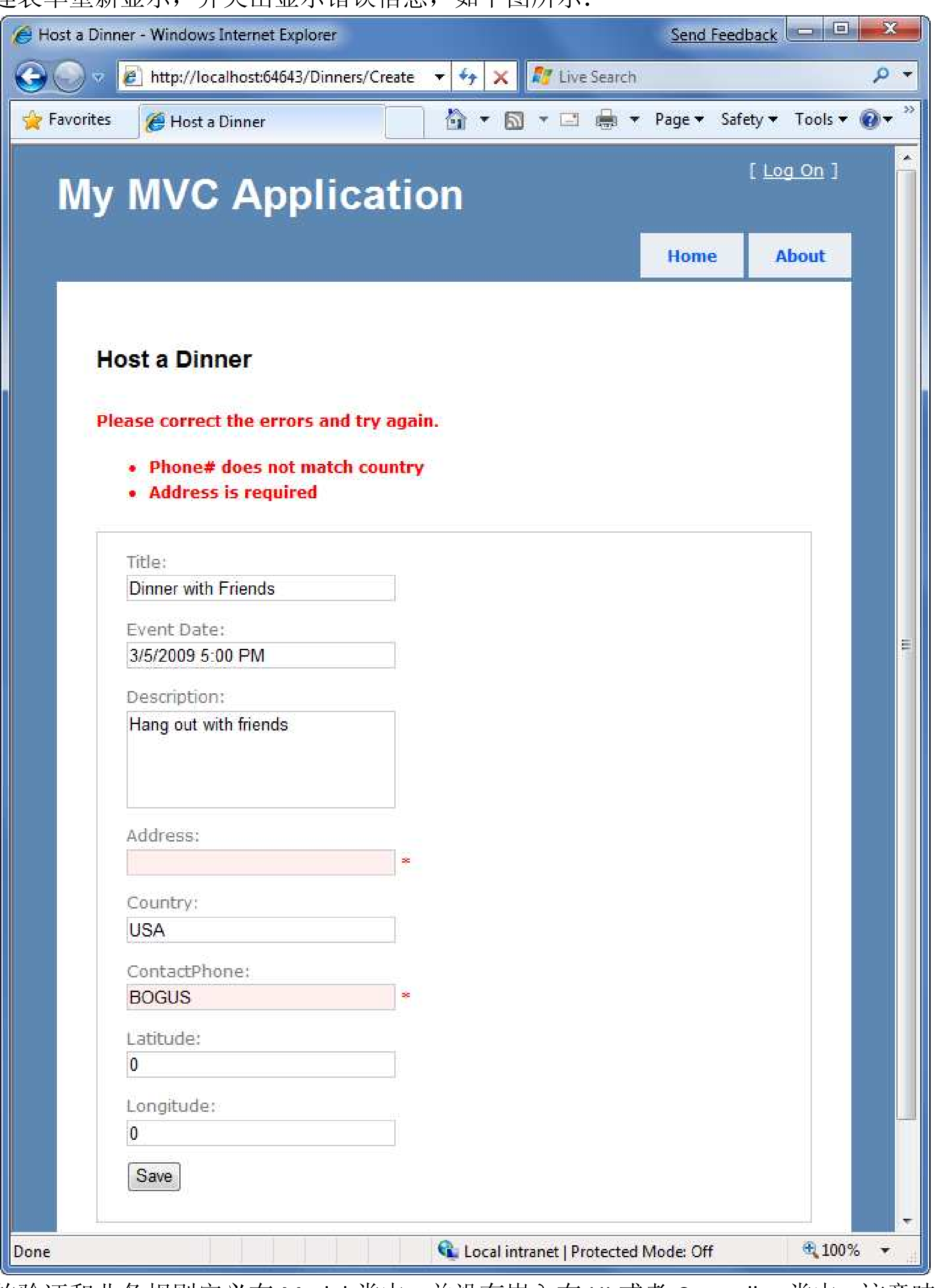
```
// POST: /Dinners/Create
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Dinner dinner) {
    if (ModelState.IsValid) {
        try {
            dinner.HostedBy = "SomeUser";
            dinnerRepository.Add(dinner);
            dinnerRepository.Save();
            return RedirectToAction("Details",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

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

上述方法通过检查 ModelState.IsValid 属性，来判断 Dinner 对象是否成功通过表单提交过来的参数值进行赋值了。如果存在输入无效的数据，将返回 False（如传入“BOGUS” 给 EventDate 属性）。如果存在任何问题，action 方法将重新显示表单。

如果输入的参数是有效的，action 方法将试图添加和保存新的 Dinner 对象到 DinnerRepository。代码块在 try/catch 块中，如果有任何业务规则冲突，将重新显示表单（这将导致 dinnerRepository.Save()方法抛出异常）。

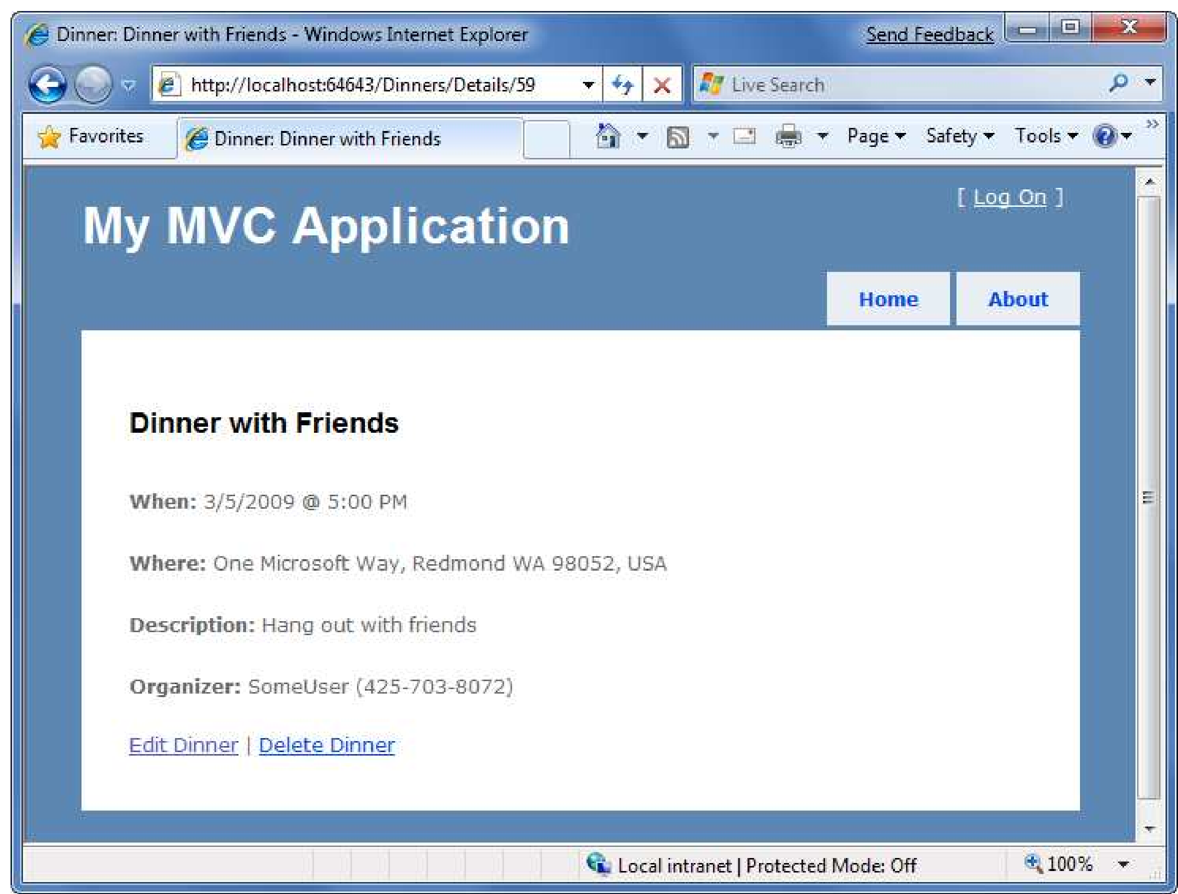
为了测试错误处理机制，我们请求/Dinners/Create URL 地址，并填写了 Dinner 详细信息。不正确的输入将导致新建表单重新显示，并突出显示错误信息，如下图所示：



上述所有的验证和业务规则定义在 Model 类中，并没有嵌入在 UI 或者 Controller 类中。这意味着今后当

我们需要更新验证或业务规则时，只需要在一个地方更新就可以了，并应用到整个应用程序中，而不必更新 Edit 或 Create action 方法。

当我们修改无效的输入后，再次点击创建 Create 按钮，DinnerRepository 的 Save 操作将会成功，一个新的 Dinner 对象将添加到数据库。接着，重定向到/Dinners/Details/[id] 地址 – 并显示新创建的 Dinner 对象的详细信息：



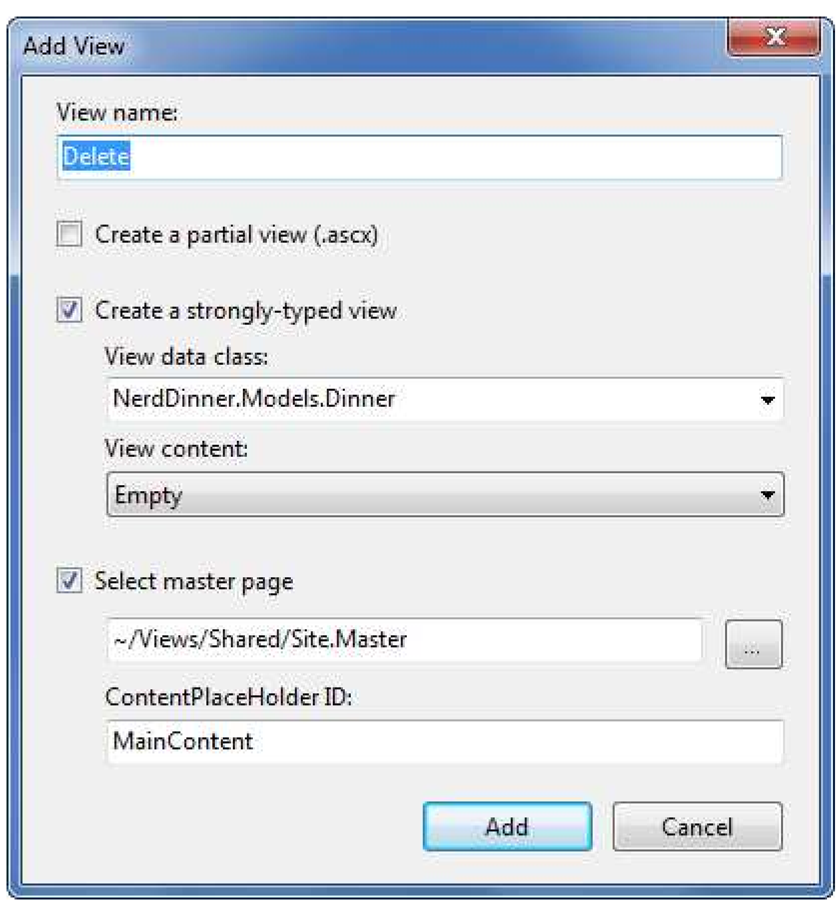
实现 HTTP-GET 的 Delete Action 方法

现在开始添加 Delete 功能到 DinnersController 类。当用户访问/Dinners/Delete/[id] URL 网址时，将访问 delete action 方法，下面是该方法的实现：

```
//
// 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);
}
```

action 方法试图检索将要删除的 Dinner 对象。如果 Dinner 对象存在，就呈现该 Dinner 对象的视图。如

果对象不存在（或者已经删除了），将返回一个 NotFound 的视图，之前我们已经创建了这个视图。创建 Delete 视图的方法不重复了，如下图所示：



点击 Add 按钮后，Visual Studio 自动在\Views\Dinners 目录下创建一个新的 Delete.aspx 视图模板文件。

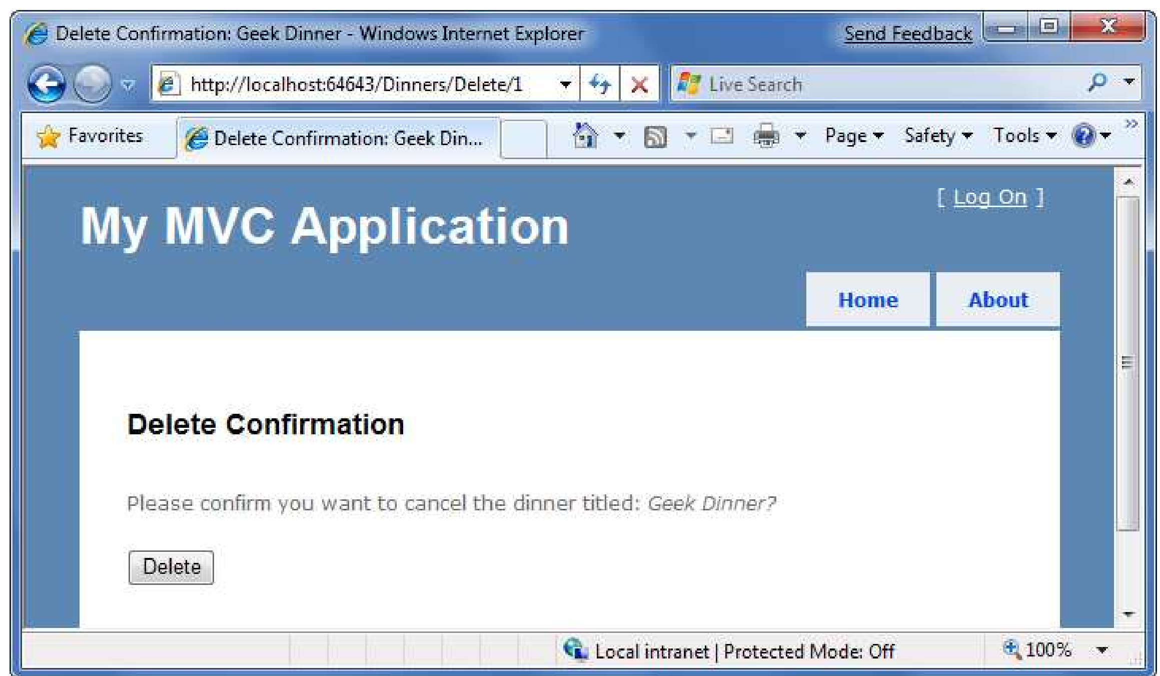
我们将修改该文件，实现删除确认。代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="TitleContent" runat="server">
    Delete Confirmation:   <%=Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="Content3" 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>
```

上述代码显示将要删除的 Dinner，并输出<form>元素，当用户点击删除 Delete 按钮时，就提交表单到 /Dinners/Delete/[id] 地址。

运行应用程序，访问/Dinners/Delete/[id] 网址，id 标识一个有效的 Dinner 对象，在浏览器中显示效果如下：



实现 HTTP-POST Delete Action 方法

现在实现 Delete action 方法的 HTTP POST 动作，代码如下：

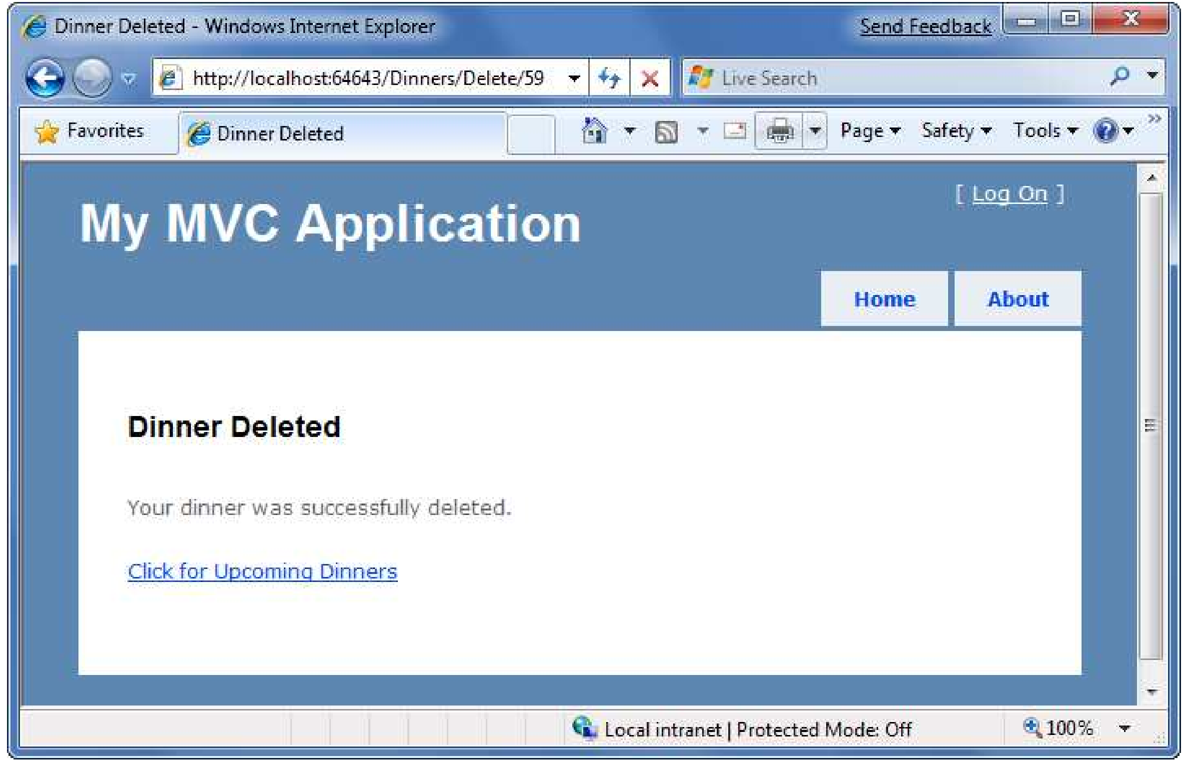
```
//
// HTTP POST: /Dinners/Delete/1
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(int id, string confirmButton) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");

    dinnerRepository.Delete(dinner);
    dinnerRepository.Save();

    return View("Deleted");
}
```

Delete Action 方法的 HTTP-POST 版本首先检索需要删除的 Dinner 对象。如果没有找到（可能已经删除了），则显示 NotFound 视图模板。如果找到了，则从 DinnerRepository 中删除，并展示 Deleted 视图模板。

现在需要创建 Deleted 视图模板，在 Add View 对话框中，不需要选择强类型的 Model 类型，代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="TitleContent" runat="server">
    Dinner Deleted
</asp:Content>

<asp:Content ID="Content3" 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>
```

再次运行范例程序，访问/Dinners/Delete/[id] URL 地址，在确认删除窗口中，点击删除 Delete 按钮，执行 HTTP-POST 操作到/Dinners/Delete/[id] 地址，触发 DinnersController 中的 Delete 方法的 Post 版本，从数据库中删除指定的 Dinner 记录，并显示已删除 Deleted 视图模板：



模型绑定的安全性

我们已经讨论了 2 中不同的方法使用 ASP.NET MVC 内置的模型绑定功能。第一个方法是使用 UpdateModel() 方法更新一个已存在的模型对象的属性；第二个方法是传递模型对象，作为 action 方法的参数。这两项技术都非常强大和有用。

功能虽然强大，但也需要考虑用户输入的安全性，包括绑定对象到表单的输入。一定要通过 HTML 编码所有用户输入值，避免 HTML 和 Javascript 注入攻击（备注：我们的范例应用程序使用 LINQ to SQL，可以自动编码所有参数，避免这些类型的攻击）。不能仅仅依赖于客户端的验证，总是要采用服务端的验证，阻止攻击者试图传入无效的数据。

默认情况下，UpdateModel() 方法试图根据匹配的表单参数值，更新所有的属性。同样地，作为参数传递给 action 方法的模型对象，基于表单参数设置全部模型对象的属性。

基于用途来锁定绑定

你可以基于用途，提供显式的可更新的属性列表，锁定绑定策略。这可以通过传递一个额外的字符串数组参数给 UpdateModel() 方法：

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

UpdateModel(dinner, allowedProperties);
```

作为传递给 Action 方法的参数 – 模型对象也支持[Bind]属性，允许指定 include list 或者允许的属性列表，如下所示：

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

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

基于类型来锁定绑定

你可以基于类型来锁定绑定规则。这样你一旦制定绑定规则，就可以在所有 Controllers 和 Action 方法中应用了，包括 UpdateModel 方法和 Action 方法的参数。

通过添加[Bind]属性在类型上，或者在应用程序中的 Global.asax 文件中（如果类型不是我们自己定义的情况下，非常有用），来定制类型绑定规则。戒指使用 Bind 属性的 Include 和 Exclude 属性来控制类或接口中哪些属性是可绑定的。

我们的 NerdDinner 应用程序将使用这一技术，添加[Bind]属性，显示可绑定的属性：

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

注意到 RSVPs 集合、DinnerID 和 HostedBy 属性都不允许通过绑定来设置。基于安全的原因，这些属性只能在 action 方法中使用显式的代码进行设置。

CRUD 封装

ASP.NET MVC 包括一些内置的功能，帮助实现表单提交的场景。我们在 DinnerRepository 类中使用了大量这些功能提供 CRUD 的支持。
我们使用以 Model 模型为中心的方法来实现我们的范例程序。这意味着所有的验证和业务规则都在模型层（Model Layer）中定义 – 而不是在 Controllers 控制器和 View 视图中。Controller 类和 View 视图模板都不必了解模型类实现的业务规则。

这样，可以保持我们的应用程序架构简洁和易于测试。在将来，我们可以添加额外的业务规则到模型层（Model Layer）中，而不必更改 Controller 类和 View 视图，就可以使用了。这一特性提供了我们很多的灵活性来改进我们的应用程序。

DinnersController 控制器实现了 Dinner 列表和显示详细信息，还有创建、编辑和删除操作等等。该类的完整代码如下：

```
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.AddRuleViolations(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.AddRuleViolations(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");
    }
}
```

ViewData 和 ViewModel

现在我们将进一步扩展 DinnersController，实现丰富表单编辑功能。这里我们讨论 2 种方法，用来将数据从 Controller 传递到 View：ViewData 和 ViewModel。

从 Controller 传递数据到 View 视图模板

MVC 模式一个典型的特征是严格的功能隔离。Model 模型、Controller 控制器和 View 视图各自定义了作用和职责，且相互之间以定义好的方式进行沟通。这有助于提升测试性和代码重用。
当 Controller 决定呈现 HTML 响应给客户端是，它负责显式传递给 View 模板所有需要的数据。View 模板从不执行任何数据查询或应用程序逻辑 – 仅仅负责呈现 Model 或 Controller 传递过来的数据。
目前，DinnersController 控制器传递给 View 模板的 Model 模型数据非常简单和直接 – Index() 方法是 Dinner 对象列表，Details()、Edit()、Create() 和 Delete() 方法则是传递一个 Dinner 对象。当增加更多 UI 特性时，我们经常需要传递更多数据，在视图模板中展示 HTML 响应。例如，我们需要改变 Edit 和 Create 视图中 Country 字段（从 HTML 文本框到下拉列表框）。我们将生成一个动态的、支持的国家列表，而不是在视图模板中硬编码的下拉列表框。我们需要从 Controller 同时传递 Dinner 对象和支持的国家列表给 View 模板。下面看看通过 2 种方式来实现。

使用 ViewData 字典

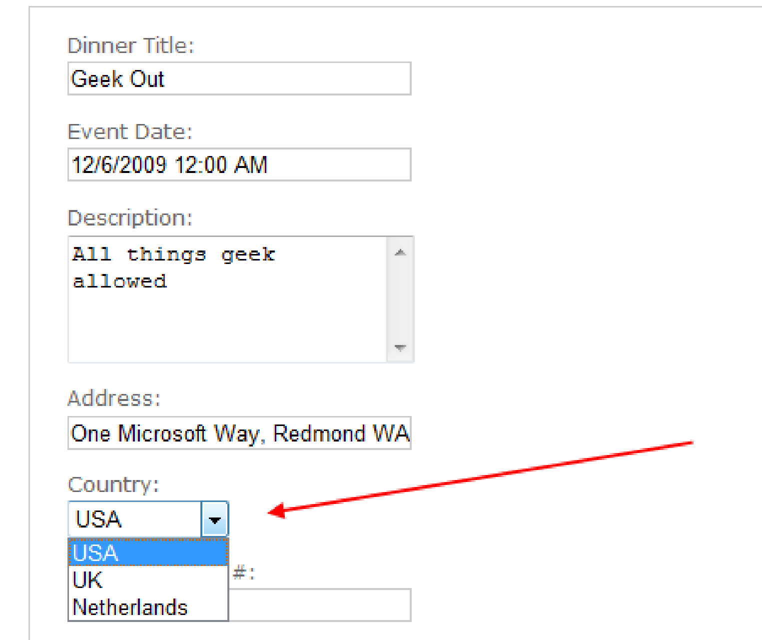
Controller 基类公开了一个 ViewData 字典属性，用来从 Controllers 传递额外的数据给 Views 视图。
例如，为了实现将 Edit 视图中 Country 国家的文本框改为下拉列表框，我们更新 Edit() Action 方法，传入一个 SelectList 对象（除了 Dinner 对象外），该对象将作为 Country 下拉列表框的 Model 类。

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

上述 SelectList 构造函数接收 2 个参数，第一个是国家列表，添加下拉列表，第二个是当前选择的值。
下面更新 Edit.aspx 视图模板，使用 Html.DropDownList() 辅助方法代码 Html.TextBox() 辅助方法：

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

上述 Html.DropDownList() 辅助方法接收 2 个参数，第一个是输出的 HTML 表单元素的名称，第二个是通过 ViewData 字典传入的 SelectList 模型类，必备那个使用 C#的关键字 as 转换 dictionary 为 SelectList。
现在我们在浏览器中访问/Dinners/Edit/2，发现 Edit 视图模板中 Country 国家文本框已经更新为下拉列表框了。



因为我们也会从 HTTP POST Edit 方法中呈现 Edit 视图模板（当有错误时，否则会进入 Details 视图模板），因此我们也需要更新 HTTP POST Edit 方法，在发生错误进入 Edit 视图模板时，添加 SelectList 到 ViewData 中，代码如下：

```
//
// 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.AllCountries, dinner.Country);

        return View(dinner);
    }
}
```

现在 DinnersController 控制器中 Edit 方法完全支持下拉列表框了。

使用 ViewModel 模式

ViewData 字典方法的优点是非常快和容易实现。部分开发人员不喜欢使用基于字符串的字典（string-based dictionaries），因为一些输入错误会导致错误，但是不能在编译期间发现。在使用 View 视图模板中使用强类型时，非强类型的 ViewData 字典也需要使用 as 操作符或类型转换。

另一个可选的方法是 ViewModel 模式。当时有这一模式时，我们需要针对特定的 View 创建强类型的类，公开 View 模板需要的动态参数值或内容。Controller 类接着填充和传递这些类给 View 模板去使用。这样可以实现类型安全、编译期间检查和编辑器智能提示等等。

例如，针对 Dinner 的 Edit 视图，我们创建一个 DinnerFormViewModel 类，公开了 2 个强类型的属性：Dinner 对象和 SelectList 模型类（用来填充国家下拉列表框）。

```
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.AllCountries, dinner.Country);
    }
}
```

接着，我们更新 Edit() Action 方法，使用从 repository 检索到的 Dinner 对象创建 DinnerFormViewModel 对象，并传递给视图模板：

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

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

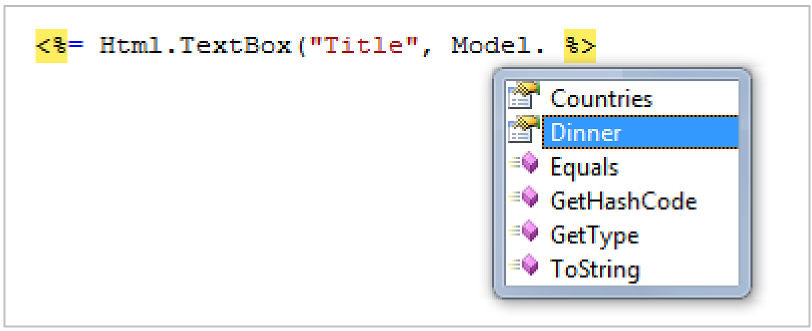
接下来，更新视图模板，在 Edit.aspx 页面文件中，更改顶部的 inherits 属性，从

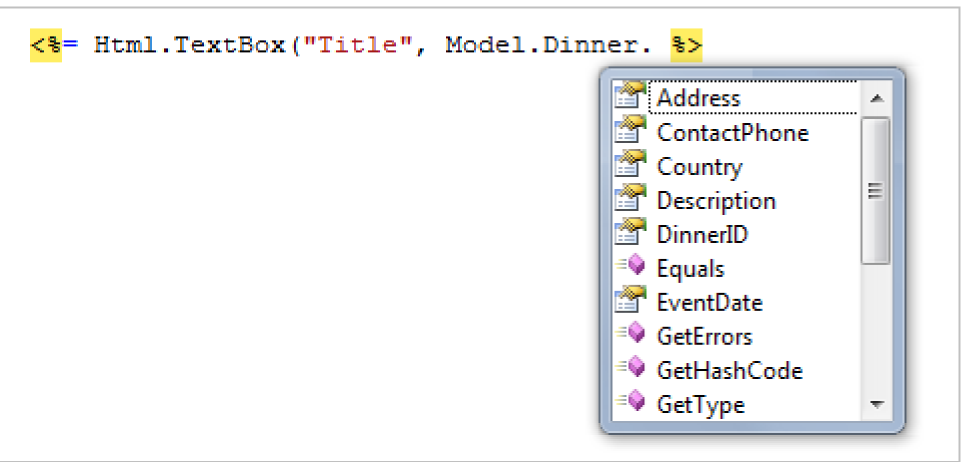
```
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Models.Dinner>
```

更改为

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

一旦完成上述操作后，View 模板中 Model 属性的智能提示将更新为传入的 DinnerFormViewModel 对象模型：





下面我们需要更新视图中的代码。对于表单中的 HTML 元素的名称不需要更新，仍旧保持为 Title、Country 等等，我们需要更新 HTML 辅助方法，使用 DinnerFormViewModel 类来获取属性值。

```
<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>
```

同样地，我们也需要更新 Edit Post 方法，在产生错误时，使用 DinnerFormViewModel 类传递给视图模板：

```
//
// 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));
    }
}
```

我们也更新 Create() Action 方法，重用相同的 DinnerFormViewModel 类，在 View 中实现 Country 下拉列表框。下面是 HTTP-GET 的实现代码：

```
//
// GET: /Dinners/Create

public ActionResult Create() {

    Dinner dinner = new Dinner() {
        EventDate = DateTime.Now.AddDays(7)
    };

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

下面是 HTTP-POST Create 方法的实现代码：

```
//
// 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(new DinnerFormViewModel(dinner));
```

}
现在 Edit 和 Create 视图都支持通过下列列表框来选择国家了。

定制 ViewModel 类（Custom-shaped ViewModel Classes）

在上面的实现方案中，DinnerFormViewModel 类直接公开了 2 个公有属性：Dinner 模型对象和 SelectList 模型属性。这一方法适合于 View 模板中 HTML 用户界面元素和业务 Model 对象比较接近的场景。
如果不符合这一情况，可以考虑创建定制的 ViewModel 类，根据视图的使用情况创建优化的对象模型 – 该对象模型可能完全不同于底层的业务模型对象（Domain Model Object）。例如，该 ViewModel 类有可能公开不同的属性或者从多个 Model 对象中汇总的属性。
定制的 ViewModel 类不仅可用来从 Controller 传递数据到 View 去呈现，而且可用来处理从表单提交回来给 Controller 的 action 方法的数据。针对后一种情况，你可以让 Action 方法根据表单提交回来的数据更新 ViewModel 对象，接着使用 ViewModel 实例来映射或者获取时间的业务模型对象（Domain Model Object）。
定制 ViewModel 类提供了很好的灵活性，在任何时候，你发现 View 模板中的呈现代码或 Action 方法中表单提交代码越来越开始复杂时，你可以考虑使用定制的 ViewModel 了。通常，这意味着业务模型对象和 View 视图中的用户界面元素不一致，一个中介的定制 ViewModel 类就可以发挥作用了。

Partials 和 Master 页面

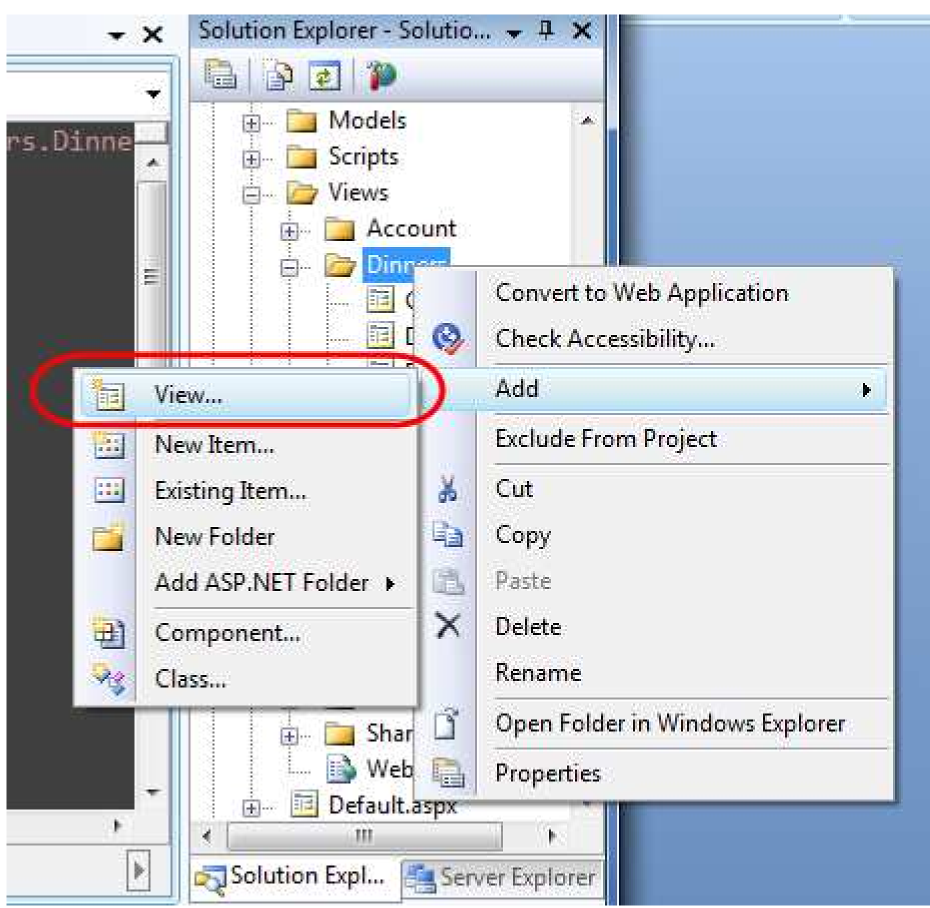
ASP.NET MVC 的一个设计理念是“Do Not Repeat Yourself”原则（通常称为 DRY）。DRY 设计帮助排除重复的代码和逻辑，让应用程序更快创建和更容易维护。
我们已经看到 DRY 原则应用在 NerdDinner 应用程序的一些方面了。如验证逻辑在 Model 层实现，在 Controller 的编辑和创建方法中均可执行；我们也跨越 Edit、Details 和 Delete 方法重用 NotFound 视图模板；对 View 视图模板采用命名规范，这样在调用 View() 辅助方法是不需要显式指定名称；另外在 Edit 和 Create Action 方法中重用 DinnerFormViewModel 类。

下面我们在 View 模板中应用 DRY 原则，减少重复的代码。

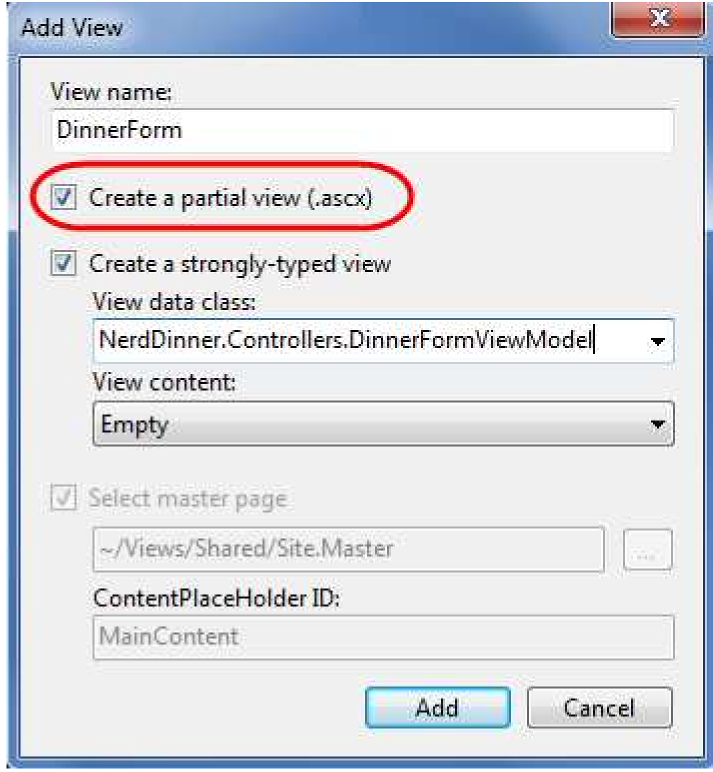
回顾 Edit 和 Create 视图模板
目前，我们使用 2 个不同的 View 视图模板 – Edit.aspx 和 Create.aspx – 来显示 Dinner 表单界面。我们可以比较一下这两个视图模板，发现非常相似。仅仅浏览器页面标题和表单标题不同，表单的布局和输入控件是一致的。
打开 Edit.aspx 和 Create.aspx 视图模板，我们发现这 2 个页面包含相同的表单布局和输入控件代码。这一重复意味着将来任何时候，如果我们添加或更改 Dinner 对象的属性，我们不得不改变两次 – 不推荐这样做。

使用 Partial 视图模板

ASP.NET MVC 支持定义 Partial View 模板，封装一个页面的部分视图呈现逻辑。Partial 提供了一个有效的方法定义视图呈现逻辑，然后在应用程序中多个页面重用。
为了消除 Edit.aspx 和 Create.aspx 视图模板的重复，我们创建一个 Partial View 模板，命名为 DinnerForm.ascx，封装两个页面中表单布局和输入元素相同的部分。具体操作：右键点击/Views/Dinners 目录，选择 Add->View 菜单项：



在弹出的 Add View 对话框中，对新的视图命名 DinnerForm，并选择 Create a partial view 复选框，并指明将传入 DinnerFormViewModel 类：



在点击 Add 按钮后，Visual Studio 自动在\Views\Dinners 目录下创建一个新的 DinnerForm.ascx 视图模板。

接着，我们从 Edit.aspx/Create.aspx 视图模板中复制重复的表单到新的 DinnerForm.ascx Partial 视图模板：

```
<%= 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>

<% } %>
```

接着，我们更新 Edit 和 Create 视图模板，调用 DinnerForm partial 模板，消除表单的重复内容。通过在视图模板中调用 Html.RenderPartial(“DinnerForm”) 来实现：

Create.aspx 视图文件：

```
<asp:Content ID="Content2" 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 视图文件：

```
<asp:Content ID="Content2" 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>
```

在调用 Html.RenderPartial() 方法时，你可以显式限定 partial 模板的路径，如~Views/Dinners/DinnerForm.ascx。在上述代码中，我们利用 ASP.NET MVC 基于约定的命名规范，仅仅指定 DinnerForm 作为 partial 模板。ASP.NET MVC 首先基于约定查找视图目录（对 DinnersController 而言，查找/Views/Dinners 目录），如果没有发现 partial 模板，则继续在/Views/Shared 目录下查找。

在调用 Html.RenderPartial()方法，并传入 partial 视图名称，ASP.NET MVC 将传入视图模板使用的 Model 和 ViewData 字典对象给 partial 视图。另外，Html.RenderPartial() 也有重载的版本，支持传入不同的 Model 对象和 ViewData 字典给 partial 视图去使用。当你仅仅想传递 Model 或 ViewModel 子集时，这一方法非常有用。

使用 Partial 视图模板简化代码

我们已经创建了 DinnerForm Partial 视图模板消除重复的视图呈现逻辑，这是创建 partial 视图模板的大部分原因。

有时，即使仅仅在一个地方调用，也需要创建 partial 视图模板。一个非常复杂的视图模板（View Template）通过提取和分割成多个 patial 视图模板，可以大大简化视图模板的阅读和维护。

例如，范例项目中 Site.master 文件的代码如下。代码非常易于阅读，因为显示登录/退出的逻辑显示在右上角，封装在 LogOnUserControl Partial 视图中。

```
<div id="header">
    <div id="Div1">
        <h1>My MVC Application</h1>
    </div>
```

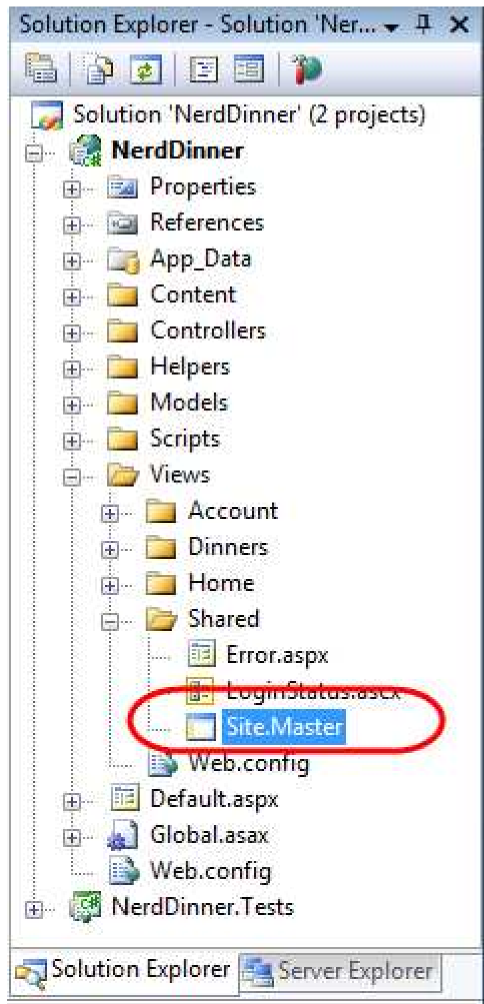
```
    <div id="logindisplay">
        <% Html.RenderPartial("LogOnUserControl"); %>
    </div>

    <div id="menucontainer">

        <ul id="menu">
            <li><%=Html.ActionLink("Home", "Index", "Home")%></li>
            <li><%=Html.ActionLink("About", "About", "Home")%></li>
        </ul>
    </div>
</div>
```

在任何时候，如果你发现比较难理解视图模板中的 HTML 代码，就应该考虑是否提取和重构该视图模板为多个合适的 partial 视图。

Master 页面

除了支持 Partial 视图外，ASP.NET MVC 也支持创建 master 页面模板，该模板用来定义网站的通用的页面布局和上层的 HTML 脚本。添加到 master 页面的 Content placeholder 控件则用来定义可替换的区域，可以被其他视图来重载或填充。这提供了一个非常有效的方法在整个应用程序中应用相同的页面布局设计。

默认情况下，一个新的 ASP.NET MVC 项目自动创建了一个 master 页面，文件名为 Site.master，存放在\Views\Shared 目录下。



默认的 Site.master 文件如下，定义了网站的外观 HTML，以及顶端的导航菜单。该文件包含了 2 个可替换的 content placeholder 控件 – 一个是标题，另一个是页面的主要内容。

```
<%@ 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 id="Head1" 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="Div1">
                <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="Div2">
            <asp:ContentPlaceHolder ID="MainContent" runat="server" />
        </div>
    </div>
</body>
</html>
```

我们为 NerdDinner 范例程序创建的所有视图模板（List，Details，Edit，Create 和 NotFound 等等）都基于 Site.master 模板。当我们使用 Add View 对话框创建视图模板时，默认会在视图顶部添加<% @Page %>指令，该指令中的 MasterPageFile 属性标识引用了 Site.master 文件。

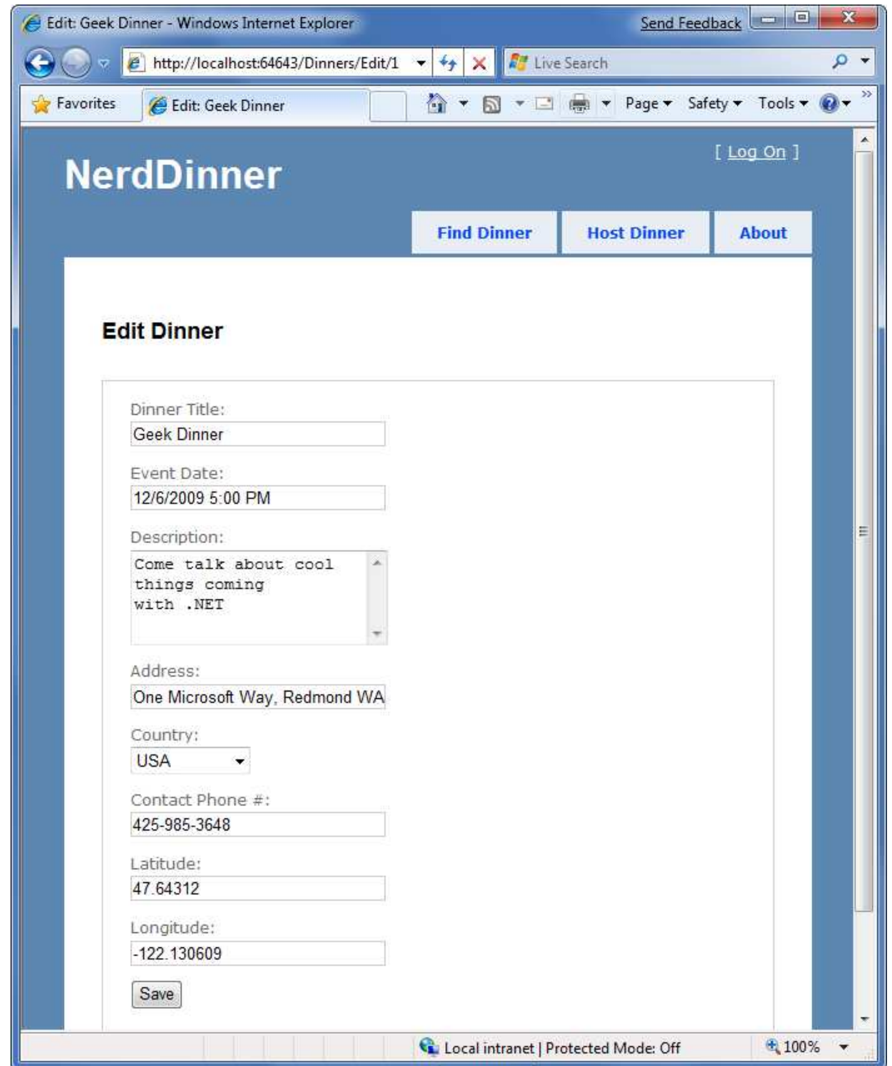
```
<%@ Page
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerViewModel>"
 MasterPageFile="~/Views/Shared/Site.Master" %>
```

这样当我们更改 Site.master 文件内容时，所有的更新会自动应用到视图模板上。下面，我们更新 Site.master 的头部部分（header section），将 My MVC Application 更改为 NerdDinner。同时，更新导航菜单，第一个 tab 页面为 “Find a Dinner” （DinnersController 的 Index() action 方法负责处理）。另外，还需要增加一个新的 tab 页面，命名为 “Host a Dinner” （DinnersControllers 的 Create() action 方法负责处理）：

```
<div id="header">
    <div id="Div1">
        <h1>NerdDinner</h1>
    </div>
    <div id="logindisplay">
        <% Html.RenderPartial("LoginStatus"); %>
    </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>
```

保存 Site.master 文件，并更新浏览器，我们会发现导航菜单已经更新了，如下图所示：

访问/Dinners/Edit/[id] 网址，编辑视图页面如下：



Partial 和 Master 页面提供非常灵活的选项来清晰组织视图，并帮助消除重复的视图内容和代码，使视图模板更易于阅读和维护。

分页

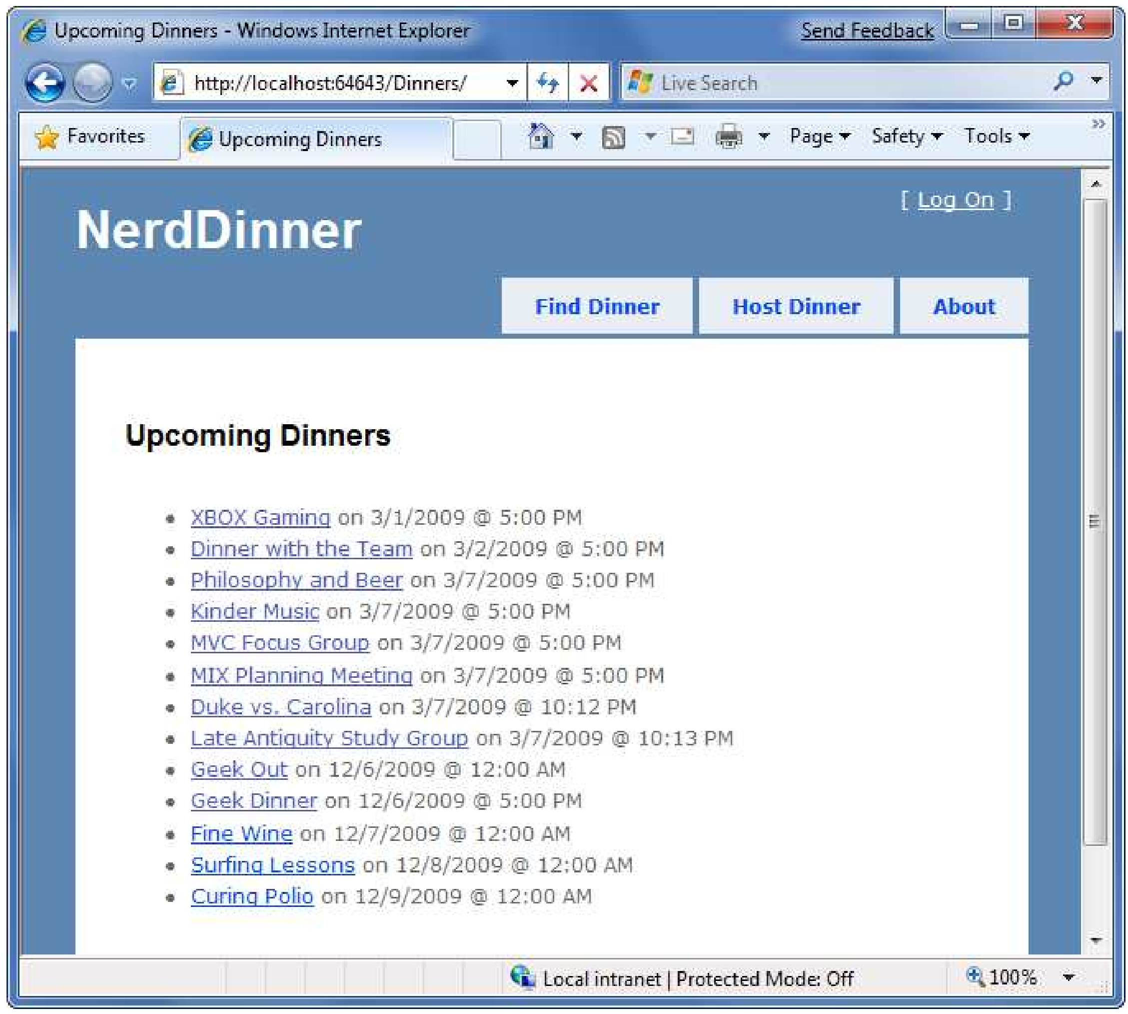
如果 dinners 列表记录过多，为了让用户更方便浏览，我们需要实现分页显示，而不是一次显示大量的记录列表。

Index() Action 方法更新

DinnersController 类的 Index() action 方法代码如下：

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

当用户请求/Dinners 地址时，将返回所有即将来临的 dinners 列表，并在页面全部显示出来。



理解 IQueryable<T>

IQueryable<T> 接口是.NET 3.5 的 LINQ 中引入的，实现了强大的 deferred execution，我们将利用这一特

性实现分页功能。

在 DinnerRepository 类中，FindUpcommingDinners() 方法将返回 IQueryable<Dinner>对象。

```
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() 方法返回的 IQueryable<Dinner> 对象封装了使用 LINQ to SQL 从数据库中检索 Dinner 对象的查询。重要的是，该语句并不会对数据库执行查询，直到我们试图去访问或者遍历查询中的数据，或者我们调用 ToList() 方法。调用 FindUpcomingDinners() 方法在执行查询之前，可以选择添加额外的操作或过滤（chained operations/filters），当查询数据时，LINQ to SQL 会聪明地对数据库执行组合查询。

为了执行分页逻辑，我们更新了 Index() action 方法，在调用 ToList() 方法之前，对返回的 IQueryable<Dinner>序列应用了附加的 Skip 和 Take 操作符。

```
//
// GET: /Dinners/

public ActionResult Index() {

    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
    var paginatedDinners = upcomingDinners.Skip(10).Take(20).ToList();

    return View(paginatedDinners);
}
```

上述代码跳过数据库中前 10 条即将来临的 Dinners 记录，接着返回 10 条记录。LINQ to SQL 聪明地构造了优化的 SQL 查询，在 SQL 数据库端实现上述逻辑，仅仅返回我们想要的 10 条记录，使查询性能更加优化和可扩展性。

在 URL 中添加 page 参数值

我们将在 URL 中包含 page 参数值，标识用户请求的分页，而不是在代码中硬编码特定的页面范围。

使用 Querystring 参数值

下面的代码演示如何使用更新 Index() action 方法，支持 querystring 参数，URL 地址如/Dinners?page=2。

```
//
// 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);
}
```

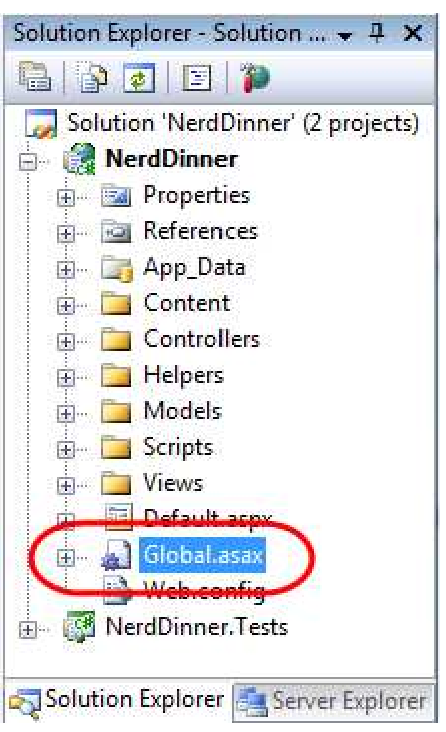
上述 Index() 方法有一个参数 page，该参数定义为 nullable 整型。/Dinners?page=2 地址表示 2 将传递为参数值。/Dinners 地址（没有 querystring 参数）将传入一个空值。

我们通过 page 乘以页面记录数（page size，本范例为 10）来决定需要跳过的 dinners 记录条数。同时，我们使用了 C#操作符（??），该操作在处理 nullable 类型时，非常有用。如果 page 参数为空（null），上述代码将对 page 赋值为 0。

使用嵌入的 URL 参数值

除了了 querystring 外，另外的一个办法是嵌入 page 参数到实际的 URL 中，如/Dinners/Page/2 或者/Dinners/2。ASP.NET MVC 包含了一个强大的 URL 路由引擎，可以轻松支持上述场景。

我们可以注册定制的路由规则，映射任何进来的 URL 或 URL 格式到任何 Controller 控制器类和 Action 方法。可以通过打开项目中 Global.asax 文件来实现。



使用 MapRoute() 辅助方法，注册一个新的映射规则，像第一次调用 routes.MapRoute()方法：

```
public void RegisterRoutes(RouteCollection routes) {
```

```
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "UpcomingDinners",                          // Route name
        "Dinners/Page/{page}",                      // URL with params
        new { controller = "Dinners", action = "Index" } // Param defaults
    );

    routes.MapRoute(
        "Default",                                  // Route name
        "{controller}/{action}/{id}",               // URL with params
        new { controller="Home", action="Index",id="" }  // Param defaults
    );
}

void Application_Start() {
    RegisterRoutes(RouteTable.Routes);
}
```

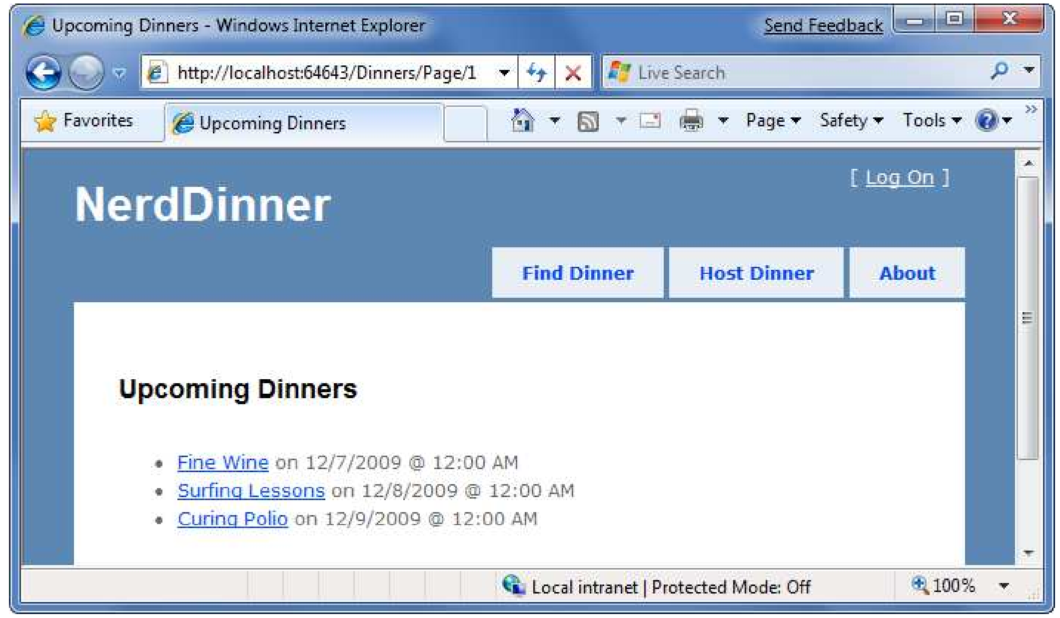
在上述代码中，我们注册了一个新的路由规则 – UpcommingDinners，URL 格式为/Dinners/Page/{page}，其中{page} 是嵌入在 URL 的参数。MapRoute() 方法将符合上述 URL 格式的请求映射到 DinnersController 类的 Index() action 方法。

我们使用与 Querystring 方案中完全相同的 Index() 方法，处理 page 参数将来自于 URL，而不是 querystring：

```
//
// 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);
}
```

现在访问/Dinners URL 时，将返回前 10 条记录，而不是全部记录了。

接下来访问/Dinners/Page/1 URL 时，将看到下一页的 Dinners 记录。



添加页面导航界面

实现页面分页的最后一步是在视图模板中添加上一页、下一页的导航界面，方便用户在页面之间切换。为了正确实现这一步，我们需要知道数据库中 Dinners 的记录数和需要分为多少页。接着，我们计算当前请求的页面是开始页或是最后页，并正确显示或隐藏上一页和下一页的导航链接。我们将在 Index() action 方法中实现这一逻辑。另一选择是，我们在项目中添加一个辅助类，封装这一逻辑，实现更多地方的重用。

下面是一个简单 PaginatedList 辅助类，继承.NET Framework 内置的 List<T>集合类。它实现了一个可重用的集合类，用来在任何 IQueryable 序列中实现分页。在 NerdDinner 范例程序中，用来对 IQueryable<Dinner> 结果集进行分页，但是它也可用于其他应用程序中对 IQueryable<Product>或 IQueryable<Customer>结果集进行分页。

```
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);
        }
    }
}
```

上述类公开的 4 个属性，PageIndex、PageSize、TotalCount 和 TotalPages 等等，另外公开了 2 个辅助属性：HasPreviousPage 和 HasNextPage，这 2 个属性分别表示是否页面数据在集合的开始页或者结束页。上述代码将引起执行 2 次 SQL 查询，第一次检索 Dinner Objects 总记录数(不返回对象，仅仅是执行 SELECT COUNT 语句，返回一个整型值)，第二次检索数据库，返回当前页需要的数据列表。

接着，我们更新 DinnersController.Index() 辅助方法，从 DinnerRepository.FindUpcomingDinners() 结果集中创建一个 PaginatedList<Dinner> 对象，并传递给视图模板。

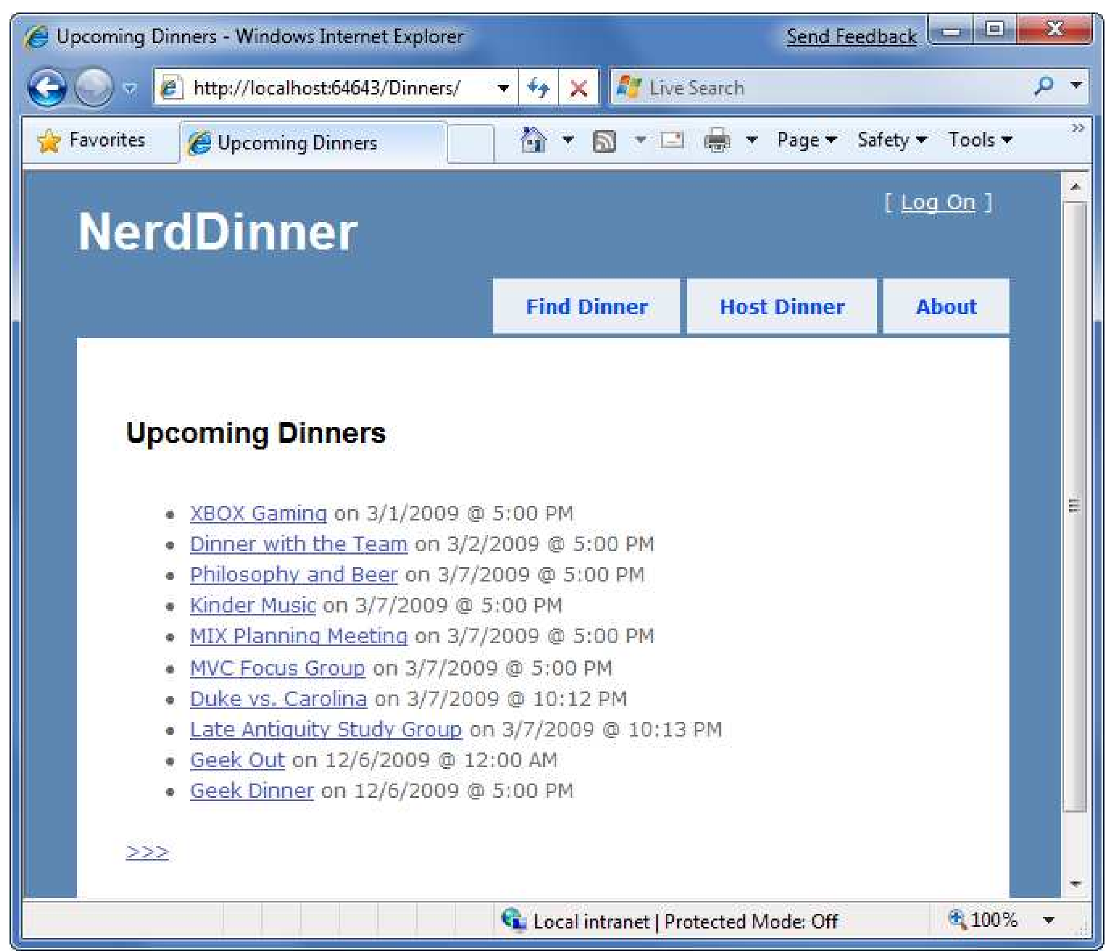
```
// GET: /Dinners/
//       /Dinners/Page/2
public ActionResult Index(int? page) {
    const int pageSize = 10;

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

接着更新\Views\Dinners\Index.aspx 视图模板，从继承 ViewPage<IEnumerable<Dinner>>更新为 ViewPage<NerdDinner.Helpers.PaginatedList<Dinner>>，然后添加如下代码到视图模板的底部，显示或隐藏上一页和下一页的导航链接：

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

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

<% } %>
```

上述代码使用 Html.RouteLink() 辅助方法生成 HTML 超链接，这一方法与我们前面使用的 Html.ActionLink() 辅助方法比较相似。区别是生成 URL 地址时，使用 Global.asax 文件中设置的路由规则。确保生成的 URL 和 Index() Action 方法有相同的格式：/Dinners/Page/{page} – 其中{page} 参数值基于当前页的序号来提供。

现在，我们访问 NerdDinner 应用程序时，每次将展示 10 条 Dinners 记录。



在页脚也提供了上一页和下一页的导航链接，实现页面跳转。

认证和授权

现在 NerdDinner 范例程序可以让访问网站的任何人创建和编辑任何 Dinner 的信息。下面我们改变这些，仅仅注册和登录的用户才允许创建新的 Dinner，并且增加限制，仅仅 Dinner 的主持人才允许编辑 Dinner 的详细信息。
为了实现上述功能，我们将使用认证和授权来保护应用程序。

理解认证和授权

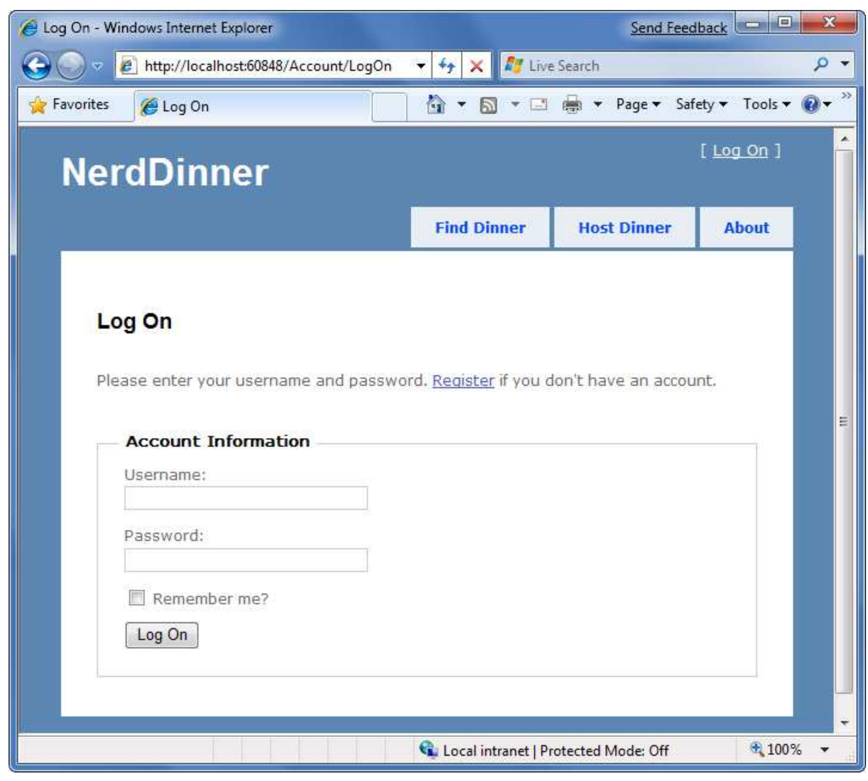
认证是识别和验证访问应用程序的客户，简而言之，就是识别访问网站的终端用户是谁。
ASP.NET 支持多种方式来认证浏览器用户。对 Internet 应用程序而言，最通用的认证方法是 Forms Authentication。Form Authentication 允许开发人员创建一个 HTML 登录表单，连接数据库或其它密码表，验证用户提交的用户名/密码。如果用户名/密码是正确的，开发人员接着让 ASP.NET 生成一个加密的 HTTP cookie，识别用户随后的请求。我们将在 NerdDinner 范例程序中使用 Form Authentication。

授权是判断是否一个验证的用户有权限访问一个特定的 URL 或资源，执行一些操作。例如，在 NerdDinner 范例程序中，我们将授权仅仅登录的用户可以访问/Dinners/Create URL 网址，并创建一个新的 Dinner 对象。我们也授权仅仅 Dinner 的主持人可以编辑该条记录，并拒绝其他人修改。

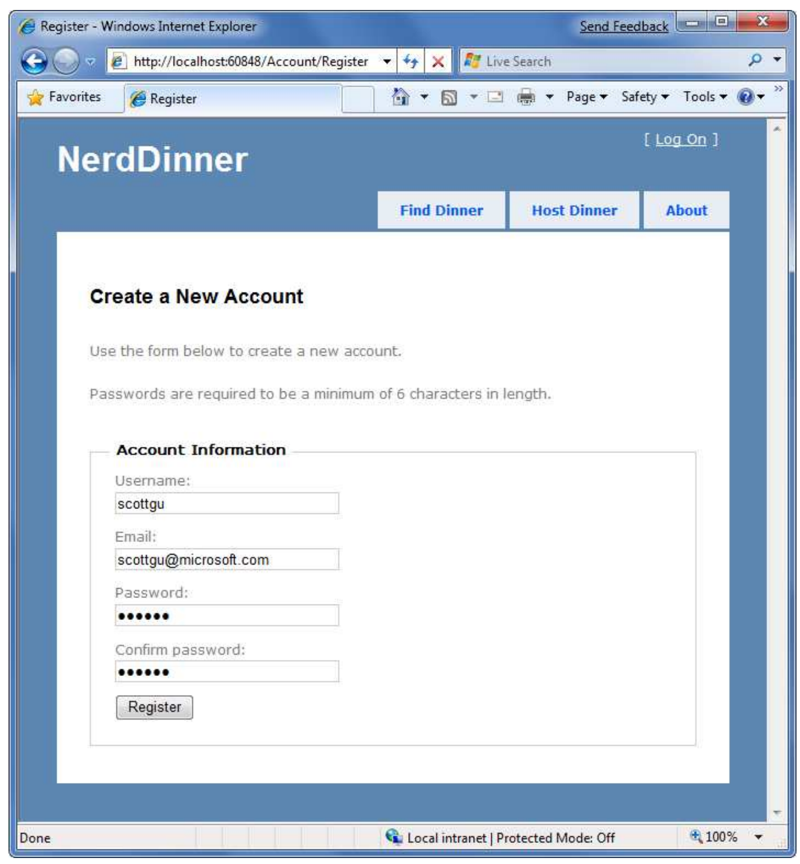
Forms Authentication 和 AccountController

在创建 ASP.NET MVC 应用程序时，ASP.NET MVC 的默认 Visual Studio 项目模板自动实现了 Forms Authentication，也自动添加一个预先创建的账户登录实现，使得站点集成安全验证更加容易。

当未登录的用户访问 NerdDinner 范例程序时，默认的 Site.Master Master 页面在右上角显示一个 Log On（登录）链接。点击 Log On 链接，引导用户到/Account/LogOn URL。



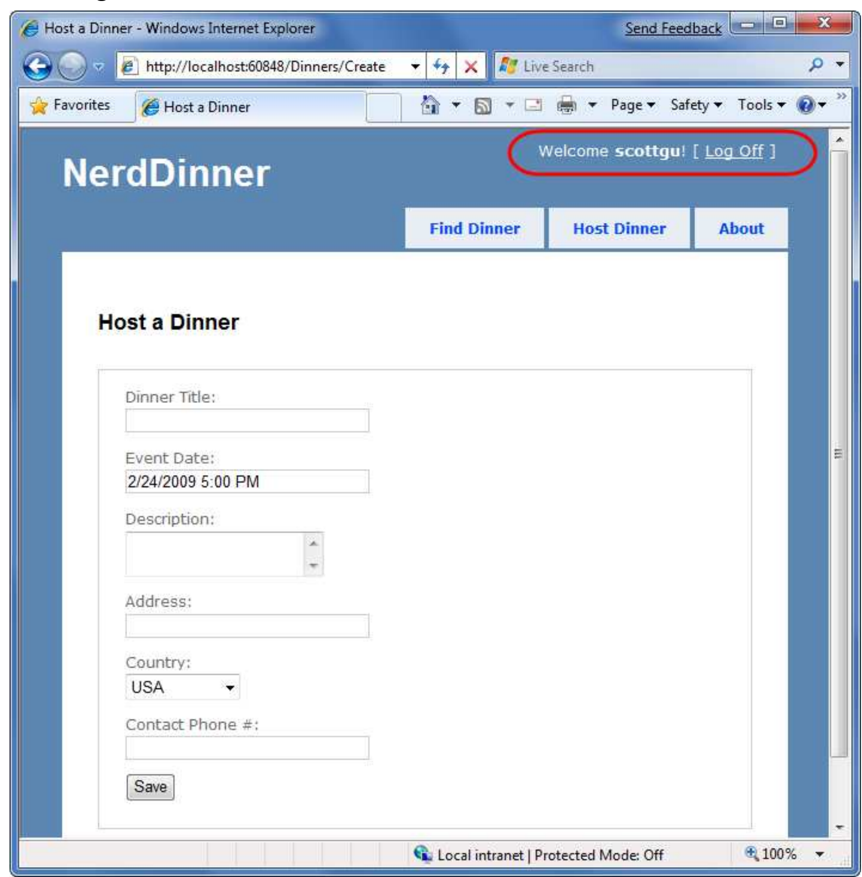
没有注册的访问者可点击 Register 注册链接，进入/Account/Register 地址，允许输入帐号的信息：



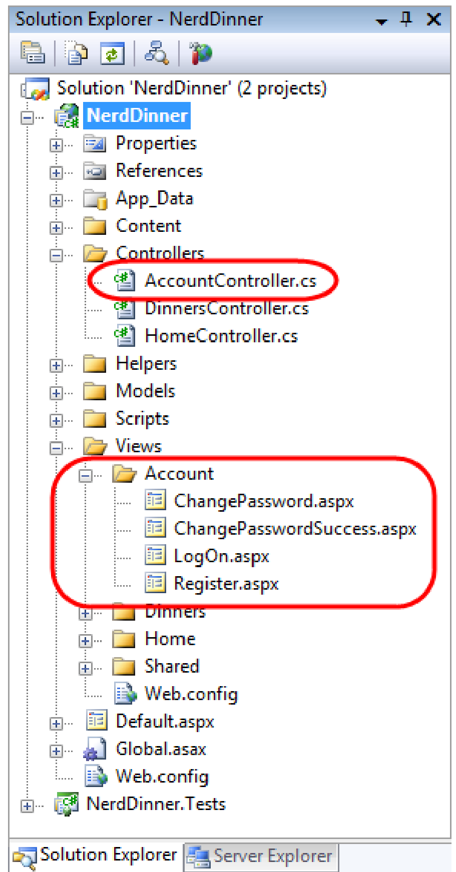
点击 Register 按钮在 ASP.NET Membership 系统创建一个新的用户，并使用 Forms Authentication 认证用户。

当用户登录后，Site.Master 更改右上角的输出为 Welcome [username]!，以及一个 Log Off 的链接，一旦

点击 Log Off 链接将退出登录。



上述的登录、退出和注册功能都在 AccountController 类中实现，该类是在创建 ASP.NET MVC 项目时自动创建的，相应的用户界面通过视图模板实现的，存放在\Views\Account 目录：

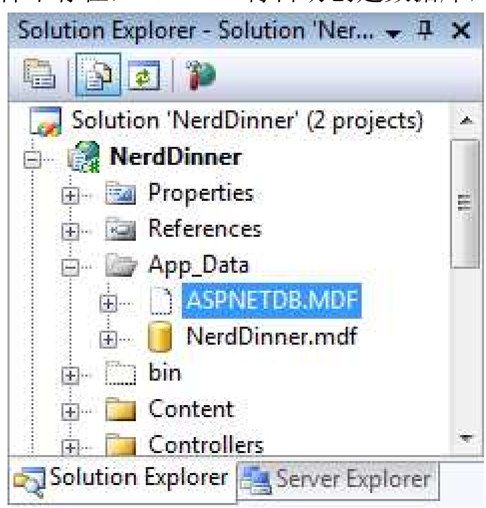


AccountController 类使用 ASP.NET Forms Authentication 系统生成加密的认证 cookies，和 ASP.NET

Membership API 来存放和验证用户名/密码。ASP.NET Membership API 是可扩展的，并实现任何使用任何密码库。ASP.NET 内置的 membership provider 实现了将用户名/密码存放在 SQL 数据库，或者活动目录（Active Directory）中。

我们可以配置 NerdDinner 范例程序中的 membership provider，打开根目录的 web.config 配置文件，查找<membership>节点。当项目创建时，默认的 web.config 已经添加了，并注册了 SQL Membership Provider，并配置了使用 ApplicationServices 连接字符串来指定数据库。

默认的 ApplicationServices 连接字符串（web.config 配置文件中的<connectionStrings>节点）配置使用 SQL Express，指向 ASPNETDB.MDF SQL Express 数据库，该 DB 文件存放在 App_Data 目录。如果在 Membership API 第一次使用时，该数据库文件不存在，ASP.NET 将自动创建数据库，并准备合适的数据库 schema。



如果不想使用 SQL Express，而想使用 SQL Server 实例（或连接一个远程数据库），我们需要做的只是更新 web.config 配置文件中的 ApplicationServices 连接字符串，并确保合适的 membership schema 在该数据库中已经正确地创建了。你可以在如下目录：

\Windows\Microsoft.NET\Framework\v2.0.50727\

运行 aspnet_regsql.exe 工具，添加合适的 membership schema，以及其他 ASP.NET 应用程序服务到数据库中。

使用[Authorize]过滤器对/Dinners/Create 授权

对 NerdDinner 范例程序，我们不必写任何代码来实现安全认证和帐号管理实现。用户可以注册新帐号，登录/退出网站。现在，我们添加授权逻辑到范例应用程序中，并使用认证状态和访问者的用户名，来控制他们在站点中能做什么，和不能做什么。

首先，我们添加授权逻辑到 DinnersController 类的 Create Action 方法上。我们要求访问/Dinners/Create URL 网址的用户必须登录，如果他们没有登录，将重定向到登录页面，以便他们可以登录。

实现这一逻辑非常简单，我们需要做的是添加 [Authorize] 过滤器属性到 Create Action 方法上，如下所示：

```
//
// GET: /Dinners/Create

[Authorize]
public ActionResult Create() {
    ...
}
```

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

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Create(Dinner dinnerToCreate) {
   ...
}
```

ASP.NET MVC 支持创建 action 过滤器，实现可重用的逻辑应用到 action 方法上。[Authorize] 过滤器是 ASP.NET MVC 内置的 action 过滤器，开发人员可以定义授权规则应用到 action 方法和 Controller 类上。如果[Authorize] 没有带任何参数，则强制要求用户请求 action 方法时必须登录，否则重定向到登录页面。当重定向时，原来的请求 URL 会作为 querystring 参数跟在 URL 后面（如，/Account/LogOn?ReturnUrl=%2fDinners%2fCreate），在用户登录完成后，AccountController 再次重定向用户返回开始请求的页面。

[Authorize] 过滤器也支持指定 Users 或 Roles 属性，该属性要求用户必须登录，此外还要求用户必须在允许的用户列表中或允许的角色成员。例如，如下代码仅允许 2 个指定的用户，“scottgu”和“billg” 访问/Dinners/Create 路径：

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

将特定的用户名直接写在代码中不易于将来的代码维护，更好的办法是定义一个 roles（角色），然后通过数据库或者活动目录（Active Directory）映射用户到角色中。ASP.NET 提供了一个内置的角色管理 API 和一组内置的 Role provider（包括 SQL 和活动目录），帮助实现用户/角色的映射。我们接着更新代码，仅允许 admin 角色的用户访问/Dinners/Create URL。

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

创建 Dinners 时，使用 User.Identity.Name 属性

在查询当前登录的用户名时，我们可以使用 Controller 基类公开的 User.Identity.Name 属性。

之前，我们实现 HTTP-POST 的 Create() action 方法时，Dinner 的 HostedBy 属性是让用户在页面手动输入的。这里，我们可以更新代码，使用 User.Identity.Name 属性给 HostedBy 赋值，同时自动为支持人增加一条 RSVP 记录：

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

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Create(Dinner dinner) {

    if (ModelState.IsValid) {

        try {
```

```
                dinner.HostedBy = User.Identity.Name;

                RSVP rsvp = new RSVP();
                rsvp.AttendeeName = User.Identity.Name;
                dinner.RSVPs.Add(rsvp);

                dinnerRepository.Add(dinner);
                dinnerRepository.Save();

                return RedirectToAction("Details", new { id=dinner.DinnerID });
            }
            catch {
                ModelState.AddModelErrors(dinner.GetRuleViolations());
            }
        }

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

因为我们给 Create() 方法添加了[Authorize] 属性，ASP.NET MVC 将确保仅登录的用户才允许访问/Dinners/Create 地址，并执行该方法。同时，User.Identity.Name 属性将一定有一个有效的用户名。

在编辑 Dinners 记录时，使用 User.Identity.Name 属性

下面针对编辑 Dinners 的功能进行授权，限制仅仅 Dinner 的主持人能够编辑 Dinner 的属性。
为了实现这一功能，我们首先在 Dinner 对象（先前我们创建的 Dinner.cs Partial 类）添加一个 IsHostedBy(username)的辅助方法。该方法根据传入的用户名是否匹配 Dinner 的 HostedBy 属性，来返回 true 或 false，并封装了大小写无关的字符串比较：

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

我们接着在 DinnersController 类中的 Edit() Action 方法上添加[Authorize] 属性，确保访问/Dinners/Edit/[id] URL 的用户必须已经登录。
在 Edit 方法中，使用 Dinner.IsHostedBy(username) 辅助方法验证登录的用户是否匹配 Dinner 的主持人。如果用户不是 host，则显示 InvalidOwner 视图，并终止请求，代码如下：

```
//
// 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(dinnerToEdit.GetRuleViolations());

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

参考之前的步骤，创建一个新的 InvalidOwner 视图模板，并添加如下错误信息的显示：

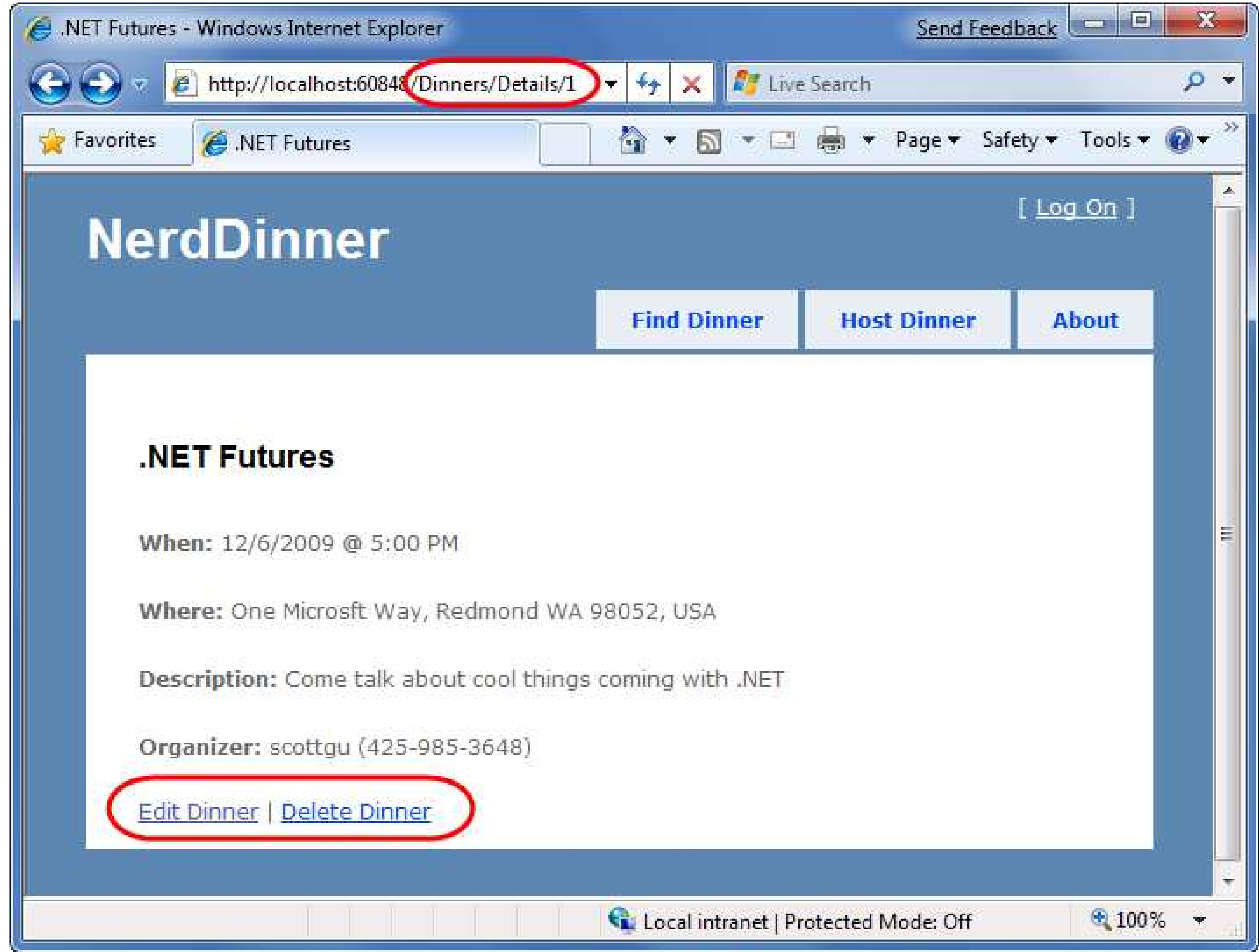
```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    You Don't Own This Dinner
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Error Accessing Dinner</h2>
    <p>Sorry - but only the host of a Dinner can edit or delete it.</p>
</asp:Content>
```

现在当一个未授权的用户视图编辑一条 Dinner 记录时，将显示如下错误信息页面：

重复上述步骤，也对 DinnersController 中的 Delete() action 方法进行授权控制，确保仅 Dinner 的主人才允许删除该记录。

显示/隐藏编辑和删除链接

下图 Dinner 的详细信息页面，显示了编辑和删除 Dinner 记录的链接，该链接分别指向 DinnersController 类的 Edit 和 Delete action 方法：



目前，不管当前访问者是否是该 Dinner 的主人，编辑和删除的链接都会显示。下面，我们将改变这一情况，仅仅当访问者是 Dinner 的 owner 时，才显示上述链接。

DinnersController 中的 Details() action 方法查询一个 Dinner 对象，接着作为 Model 对象传递给视图模板，代码如下：

```
//
// GET: /Dinners/Details/5

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");

    return View(dinner);
}
```

下面将更新视图模板，利用 Dinner.IsHostedBy() 辅助方法来协助显示/隐藏编辑和删除链接，代码如下：

```
<% 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}) %>

<% } %>
```

AJAX 实现 RSVP 响应

现在开始实现让已登录的用户回复参加晚宴了，我们将在 Dinner 的详细页面，使用 AJAX 技术实现这一功能。

显示用户是否已经回复了

用户访问/Dinners/Details/[id] URL，可以查看特定 Dinner 的详细信息。关于 Details action 方法的实现，也不在这里描述了。

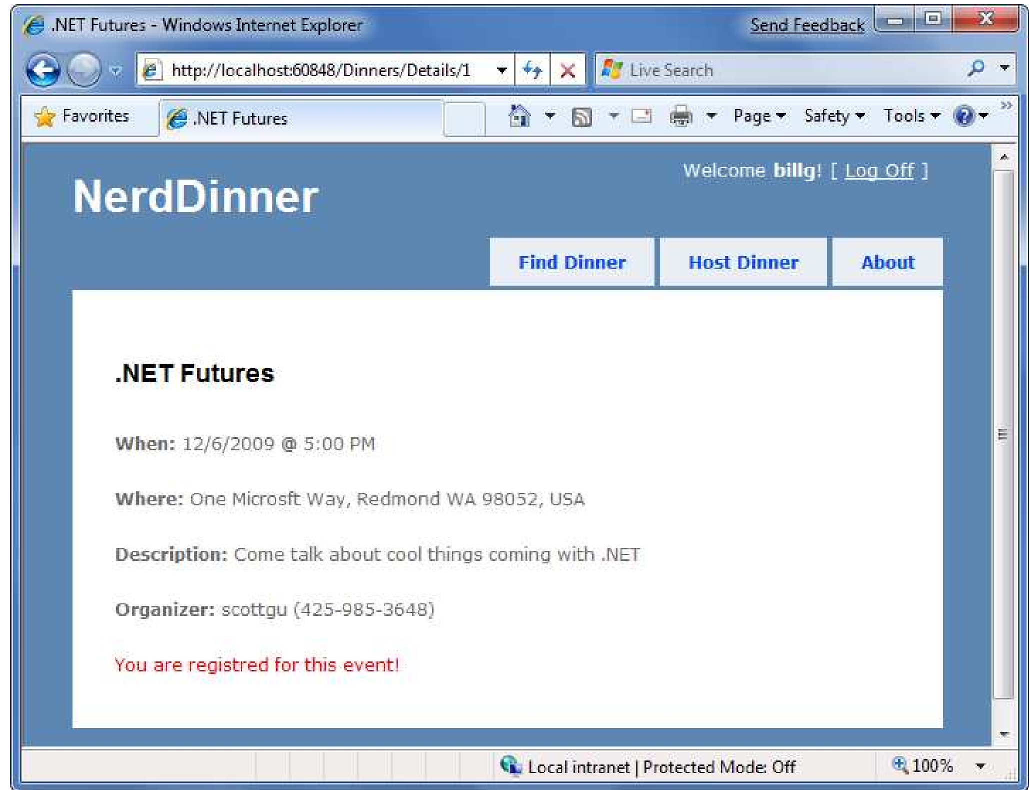
实现 RSVP 支持的第一步是在 Dinner 对象（Dinner.cs partial 类）中添加一个 IsUserRegistered(username) 的辅助方法，该方法基于是否用户已经回复了该 Dinner 晚宴，显示 true 或 false。

```
public partial class Dinner {

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

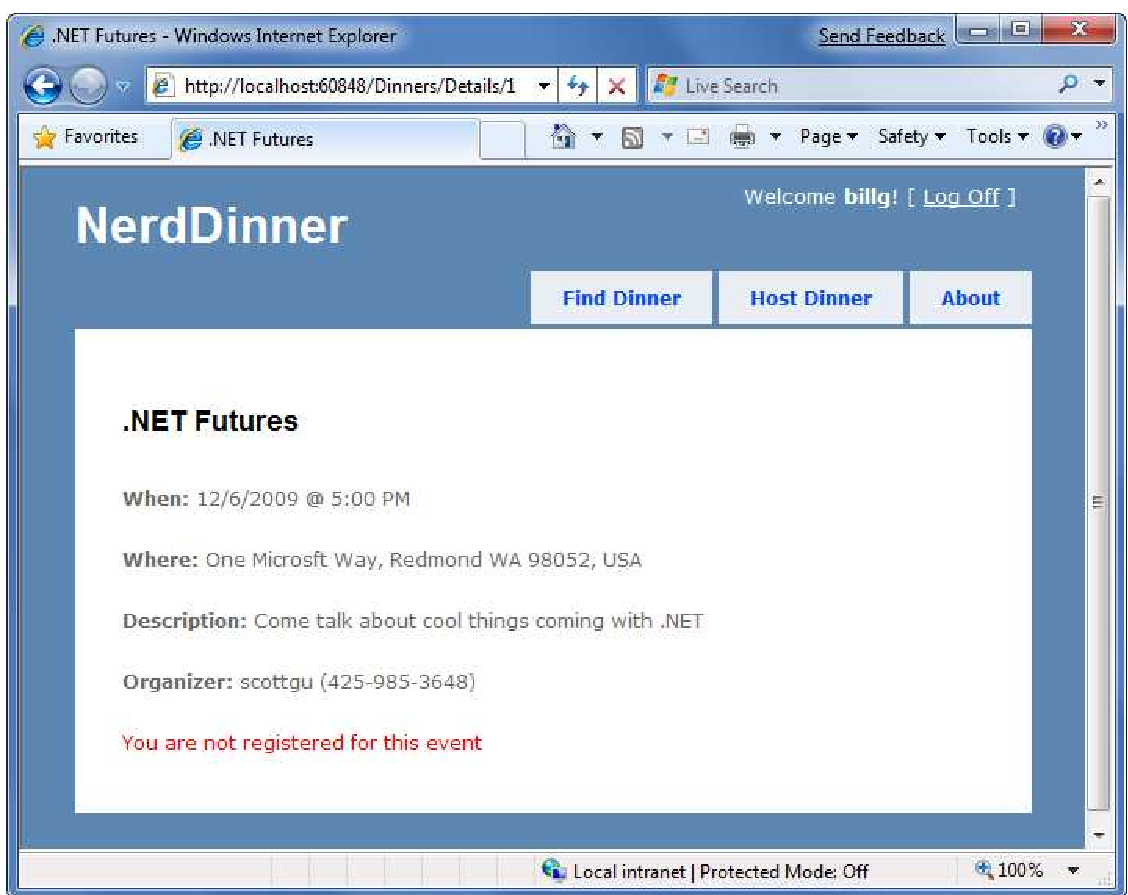
接着添加如下代码到 Details.aspx 视图模板，显示合适的信息，标识用户是否已经注册该晚宴或者没有：

```
<% if (Request.IsAuthenticated) { %>
    <% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>
        <p>You are registred for this event!</p>
    <% } else {  %>
        <p>You are not registered for this event</p>
    <% }  %>
<% } else { %>
    <a href="/Account/Logon">Logon</a> to RSVP for this event.
<% } %>
```

现在当用户访问一个已注册的 Dinner 时，将看到如下信息：



当访问一个未注册的 Dinner 时，将看到如下信息：



实现 Register Action 方法

下面实现功能，让用户可以在详细页面登记宴会。为了实现这一功能，我们创建一个新的 RSVPController 类。

在 RSVPController 类中实现 Register action 方法，传入 Dinner 的 id 作为参数，查找相应的 Dinner 对象，检查登录的用户是否注册了该晚宴，如果没有，则添加 RSVP 对象到数据库中：

```
public class RSVPController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // AJAX: /Dinners/RSVPForEvent/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.RSVPs.Add(rsvp);
            dinnerRepository.Save();
        }

        return Content("Thanks - we'll see you there!");
    }
}
```

上述代码返回一个简单的字符串作为 action 方法的输出，我们也可以将这一消息写在视图模板中。但是既然是一条简单的字符串，我们就是要 Controller 基类的 Content() 辅助方法，返回一条字符串。

使用 AJAX 调用 Register Action 方法

我们将使用 AJAX，在详细页面视图中调用 Register action 方法，相当容易实现。首先，我们添加 2 个脚本库的引用：

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

第一个库引用核心的 ASP.NET AJAX 客户端脚本库，这个文件大概 24k（压缩之后），包含核心的客户端 AJAX 功能。第二个库包含有工具函数，与 ASP.NET MVC 内置的 AJAX 辅助方法集成。

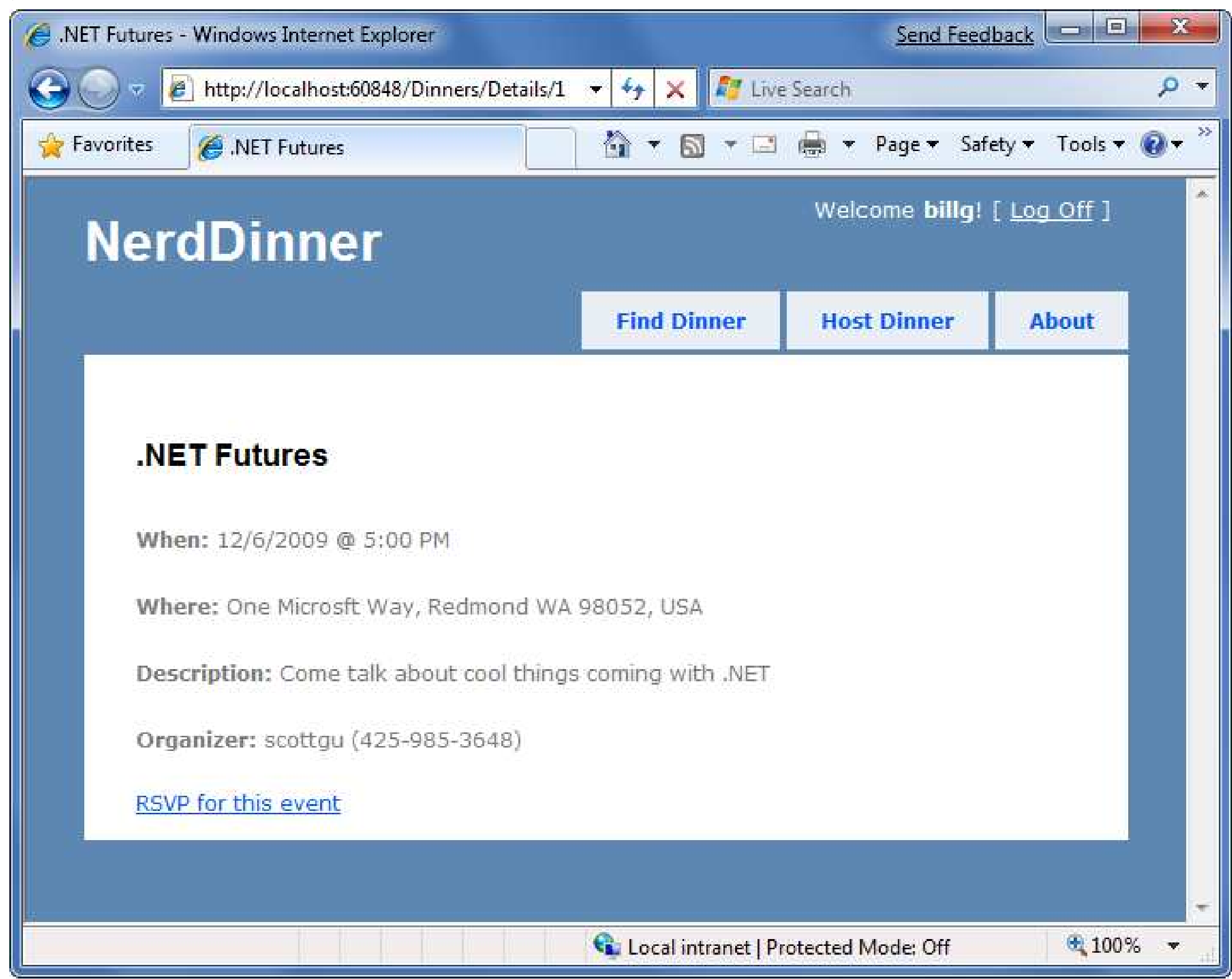
接着，我们更新之前创建的视图模板代码。不要仅仅输出“你还没有注册该晚宴！”，而是产生一个链接。

当点击这一链接时，执行 AJAX 调用，并调用 RSVPController 中的 Register action 方法：

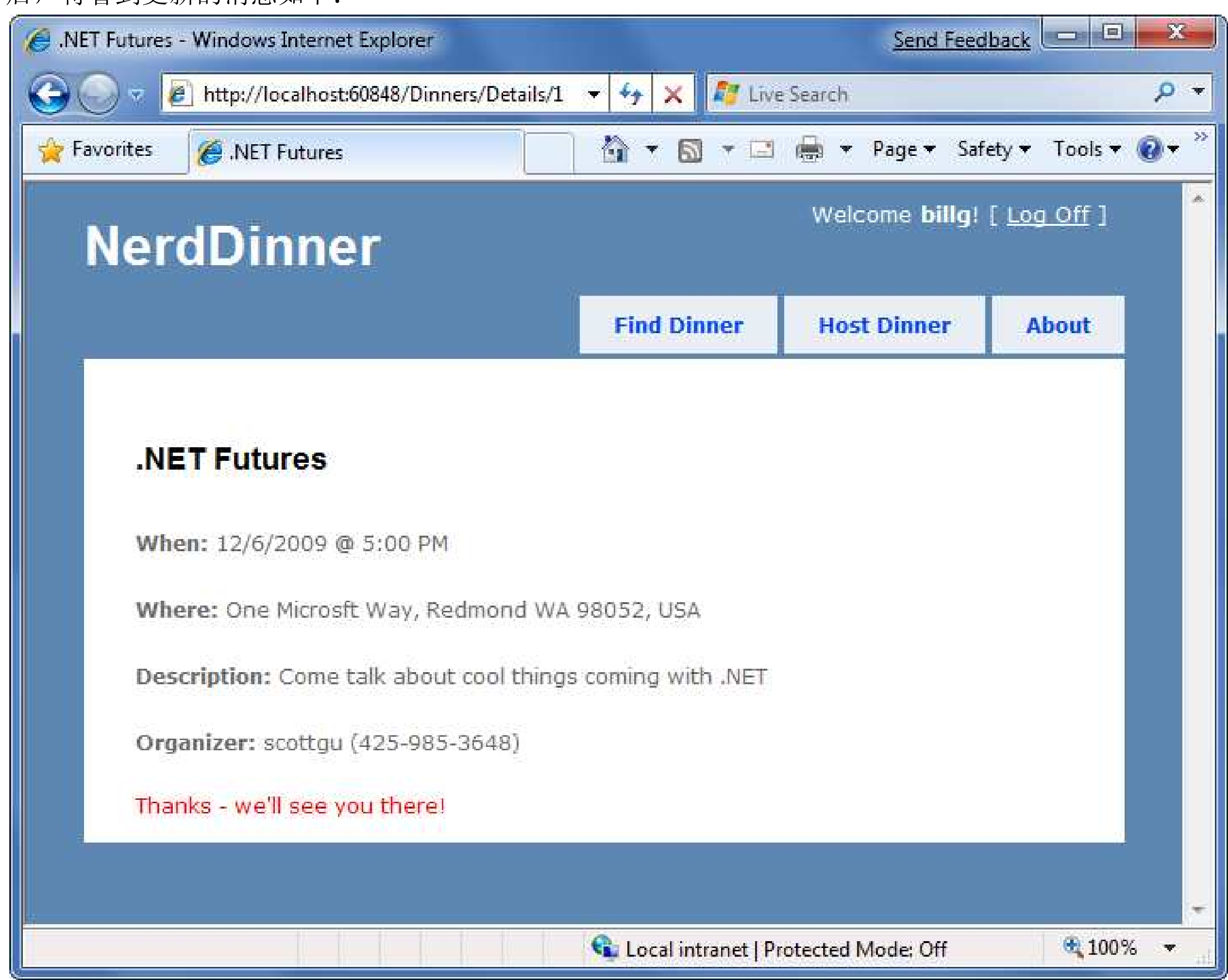
```
<div id="rsvpmsg">
<% if(Request.IsAuthenticated) { %>
    <% if(Model.IsUserRegistered(Context.User.Identity.Name)) { %>
        <p>You are registred for this event!</p>
    <% } else { %>
        <%= Ajax.ActionLink( "RSVP for this event",
                             "Register", "RSVP",
                             new { id=Model.DinnerID },
                             new AjaxOptions { UpdateTargetId="rsvpmsg"}) %>
    <% } %>
<% } else { %>
    <a href="/Account/Logon">Logon</a> to RSVP for this event.
<% } %>
</div>
```

上述代码使用的 Ajax.ActionLink() 辅助方法是 ASP.NET MVC 内置的，与 Html.ActionLink() 辅助方法类似，但不是执行一个标准的导航，而是进行 AJAX 调用一个 action 方法。上面，我们调用了 RSVPController 的 Register action 方法，并传入 Model.DinnerID 作为 id 参数。最后的 AjaxOptions 参数表示接收 action 方法返回的内容，并更新页面中 id 为 rsvpmsg 的<div>元素。

现在当用户访问一个没有注册的 Dinner 记录时，将看到“注册该晚宴”的链接：



当用户点击“注册该晚宴”链接，将 AJAX 调用 RSVP Controller 的 Register action 方法，在完成方法调用后，将看到更新的消息如下：



上述 AJAX 调用触发的网络带宽和通信量是非常轻量级的。当用户点击“注册该晚宴”链接时，发送一个很小的 HTTP POST 请求到/Dinners/Register/1 网址：

POST /Dinners/Register/2 HTTP/1.1
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Referer: http://localhost:1167/Dinners/Details/2

Register action 方法的响应消息也非常简单：

HTTP/1.1 200 OK
2.Content-Type: text/html; charset=utf-8
3.Content-Length: 29
4.Thanks - we'll see you there!

这一轻量级的调用非常快，即使网络很忙的情况也工作正常。

添加 jQuery 动画

前一步我们实现的 AJAX 功能工作又快又好。有时会很快，以至于用户没有注意到 RSVP 链接已经更新为新的文本了。为了让输出的文本更加明显，我们对更新的信息添加一个简单的动画，以获得用户的注意。

默认的 ASP.NET MVC 项目模板已经包含了 jQuery – 一个非常优秀和流行的开源 Javascript 库（微软也支持）。jQuery 提供了大量的功能，包括一个 HTML DOM 选择和效果库。
为了使用 jQuery，首先需要引用 jQuery 脚本。因为我们几乎在整个网站多处使用 jQuery，因此我们在 Site.master 页面中添加脚本引用，这样所有页面都可以使用。

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

提示：确保你已经安装了 VS 2008 SP1 的 JavaScript 智能提示补丁，对 JavaScript 文件（包括 jQuery）提供了丰富的智能提示。你可以从这里下载：http://tinyurl.com/vs2008javascripthotfix

使用 jQuery 编写代码经常使用通用的$() JavaScript 方法，使用 CSS 选择器查询一个或多个 HTML 元素。例如，$(“#rsvpmsg”) 选择所有 id 为 rsvpmsg 的 HTML 元素，$(“.something”) 则选择所有带”something” 的 CSS 类的 HTML 元素。你甚至可以编写更高级的查询，如使用$(“input[@type=radio][@checked]”) 返回所有选择的 Radio 按钮。

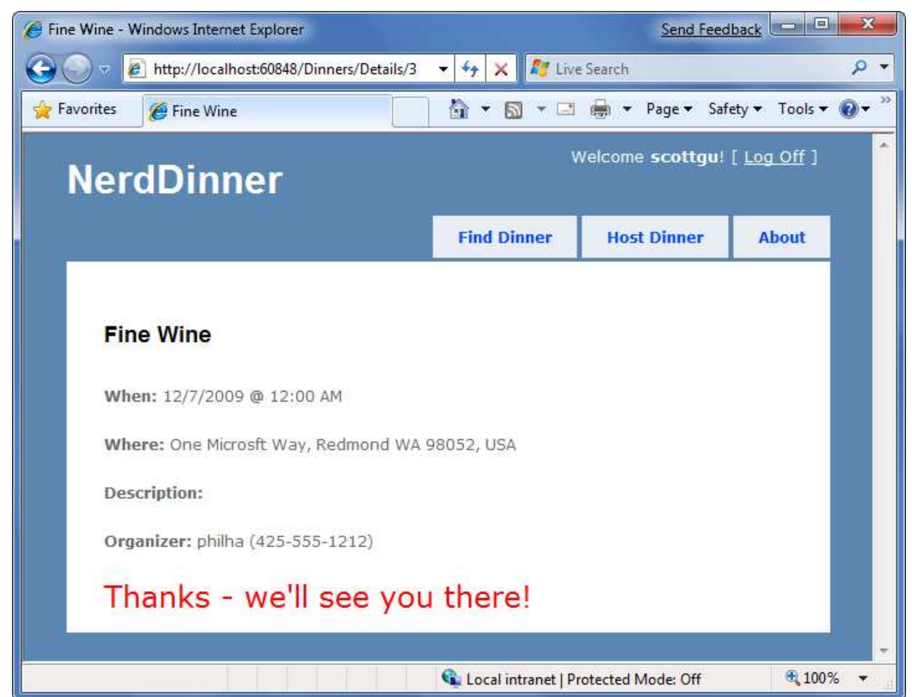
一旦你选择了元素，你可以调用方法，如隐藏它们：$(“rsvpmsg”).hide();
对于本示例中，我们将定义一个简单的 JavaScript 功能，命名 AnimateRSVPMessage，功能为：选择 rsvpmsg 的<div>元素，并动画调整文本的大小。下面的代码实现在 400 毫秒内增大文本的大小。

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

在 AJAX 完成调用之后，通过传递其名称给 Ajax.ActionLink() 辅助方法，来调用上述 JavaScript 函数（通过 AjaxOptions 的 OnSuccess 事件属性）。

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

现在当点击” RSVP for this event”链接，并且 AJAX 调用完成后，返回的消息将动画地变大。



除了提供 OnSuccess 事件，AjaxOptions 对象公开了 OnBegin、OnFailure 和 OnComplete 事件，以及很多其他的属性和有用的选项。

简化-重构 RSVP Partial 视图

Details 视图模板有点变得太长了，理解开始变得有点困难了。为了提高代码的可读性，下面创建一个新的 partial 视图 – RSVPStatus.ascx，封装 Details 视图的所有 RSVP 视图代码。
右键点击\Views\Dinners 文件夹，选择 Add->View 菜单项。设置 Dinner 对象作为强类型的 ViewModel，接着从 Details.aspx 视图复制/粘到 RSVP 内容。

下面继续创建另外一个 partial 视图 – EditAndDeleteLinks.ascx，该视图封装了编辑和删除的链接代码。也设置 Dinner 对象作为强类型的 ViewModel，并从 Details.aspx 视图中复制/粘帖 – 编辑和删除的逻辑。

这样，在 Details 视图模板的底部包含了 2 个 Html.RenderPartial()方法的调用：

```
<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>When:</strong>
            <%=Model.EventDate.ToShortDateString() %>

            <strong>@</strong>
            <%=Model.EventDate.ToShortTimeString() %>
        </p>
        <p>
            <strong>Where:</strong>
            <%=Html.Encode(Model.Address) %>,
            <%=Html.Encode(Model.Country) %>
        </p>
         <p>
            <strong>Description:</strong>
            <%=Html.Encode(Model.Description) %>
        </p>
        <p>
            <strong>Organizer:</strong>
            <%=Html.Encode(Model.HostedBy) %>
            (<%=Html.Encode(Model.ContactPhone) %>)
        </p>

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

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

    </div>
</asp:Content>
```

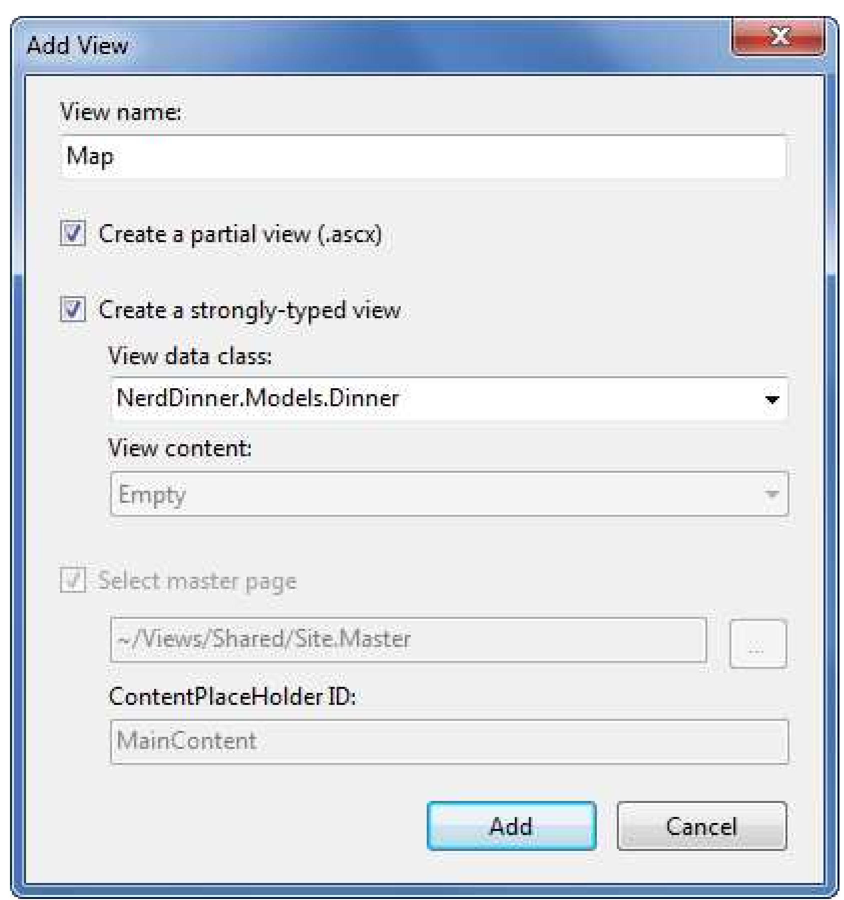
这样代码变得更加简洁，易于阅读和维护。下一步将演示如何集成 AJAX 地图。

集成 AJAX 地图

下面我们在范例程序中集成 AJAX 地图，使应用程序更加可视化。这样，在用户创建、编辑或者查看宴会信息时，可以看到宴会的地理位置信息。

创建 Map Partial 视图

我们计划在应用程序中多个地方使用地图功能。为了保持代码的简洁，我们封装通用的地图功能在一个单一的 partial 视图模板中，然后在多个 Controller action 方法和视图中重用。对该新建的 partial 视图命名为 map.ascx，创建在\Views\Dinners 目录。
如下图所示，创建 map.ascx partial 视图，并设置传递 Dinner 模型类作为 View Data Class：



进一步更新 Map.ascx 文件，代码如下：

```
<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>
```

第一部分<script>引用指向微软的 Virtual Earth 6.2 地图库，第二部分<script>引用指向 map.js 文件，该文件将封装通用的 JavaScript 地图逻辑。<div id="theMap">元素是一个 HTML 容器，Virtual Earth 将使用该容器来承载地图。

接下来包含了 2 段嵌入的<script>代码块，视图相关的 2 个 JavaScript 功能。第一个函数是当页面准备好运行客户端脚本时，使用 jQuery 调用一个函数。它调用 LoadMap() 辅助函数，该函数定义在 Map.js 脚本文件中，用来加载 virtual earth 地图控件。第二个是一个回调事件句柄，添加一个别针在地图上，标识位置。

你会注意到我们在客户端脚本中使用服务端的<%= %>代码块，嵌入 Dinner 的经度和维度属性。这一技术非常有用，可以输出动态值在客户端脚本中使用（不需要一个单独的 AJAX 回调到服务端去检索值 – 使得响应更快）。<%= %>代码块在视图在 Server 端呈现时将执行，因此，HTML 输出将嵌入的 JavaScript 值（如，var latitude = 47.64312）。

创建一个 Map.js 工具类库

开始创建 Map.js 文件，用来封装地图的 JavaScript 功能，并实现上述的 LoadMap 和 LoadPin 方法。右键点击项目中的\Scripts 目录，并选择 Add -> New Item 菜单项，选择 JScript 项目，命名为 Map.js。

下面是添加到 Map.js 文件中的 JavaScript 代码，该代码复制和 Virtual Earth 交互，并显示地图和 Dinner 对象的位置标识：

```
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;

    // Makes the control bar less obtrusize.
    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);

    //Make a nice Pushpin shape with a title and description
    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(where) {
    var numberOfResults = 20;
    var setBestMapView = true;
    var showResults = true;

    map.Find("", where, null, null, null,
            numberOfResults, showResults, true, true,
            setBestMapView, callbackForLocation);
}
```

```
function callbackForLocation(layer, resultsArray, places,
            hasMore, VEErrorMessage) {

    clearMap();

    if (places == null)
        return;

    //Make a pushpin for each place we find
    $.each(places, function(i, item) {
        description = "";
        if (item.Description !== undefined) {
            description = item.Description;
        }
        var LL = new VELatLong(item.LatLong.Latitude,
                        item.LatLong.Longitude);

        LoadPin(LL, item.Name, description);
    });

    //Make sure all pushpins are visible
    if (points.length > 1) {
        map.SetMapView(points);
    }

    //If we've found exactly one place, that's our address.
    if (points.length === 1) {
        $("#Latitude").val(points[0].Latitude);
        $("#Longitude").val(points[0].Longitude);
    }
}

function clearMap() {
    map.Clear();
    points = [];
    shapes = [];
}
```

集成地图到创建和编辑表单

我们将集成地图到现有的创建和编辑视图。这点比较容易做到，不需要更改任何 Controller 代码。因为创建和编辑视图共享一个通用的 DinnerForm partial 视图，展示 Dinner 的用户界面，因为我们只需要在一个地方添加地图，就可以在创建和编辑视图中看到了。

我们需要做的是，打开\Views\Dinners\DinnerForm.ascx partial 视图，并更新它包含新的 map partial 视图。如下是更新后的 DinnerForm 视图的代码（注：为了代码的简洁性，这里忽略了 HTML 表单的元素）：

```
<%= Html.ValidationSummary() %>

<% using (Html.BeginForm()) { %>

    <fieldset>

        <div id="dinnerDiv">
            <p>
                [HTML Form Elements Removed for Brevity]
            </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>

<% } %>
```

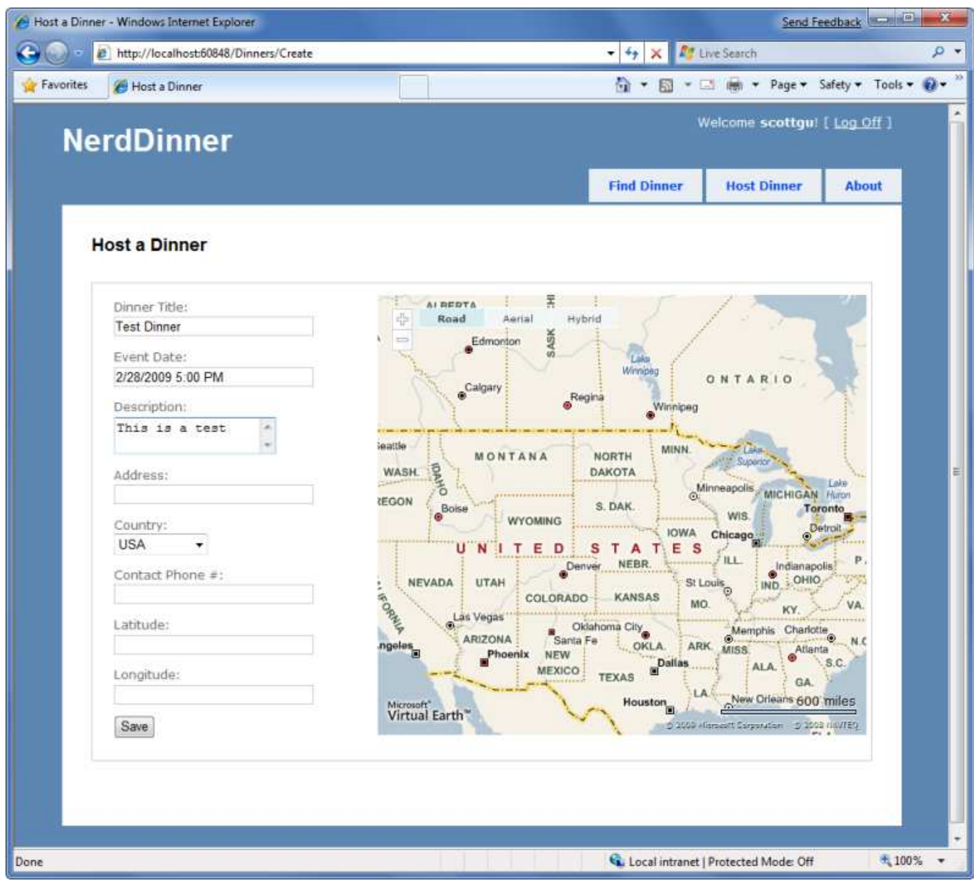
DinnerForm partial 视图接收类型为 DinnerFormViewModel 作为模型类型（因为它既需要一个 Dinner 对象，也需要一个 SelectList 填充国家下拉列表框）。Map partial 视图仅仅需要类型为 Dinner 的模型类型，因此当我们呈现 Map partial 视图时，我们传递 DinnerFormViewModel.Dinner 属性给它。

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

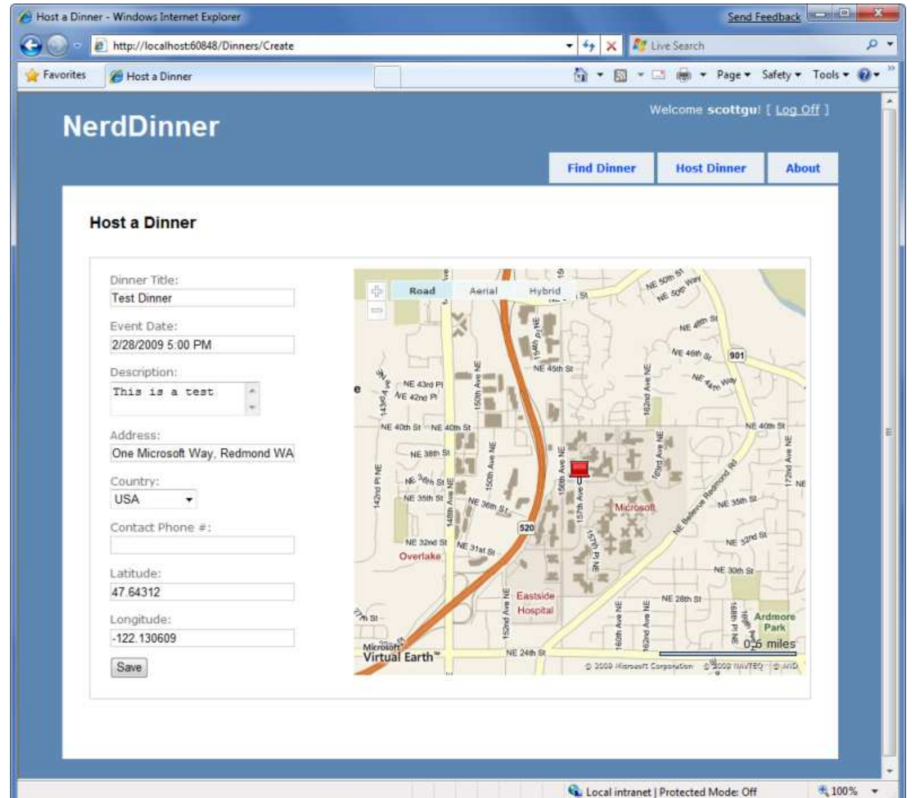
我们添加到 partial 的 JavaScript 函数使用了 JQuery 将"blur"事件附加到了"Address"HTML textbox 上。你或许听说过"focus"事件，它能够在用户单击或者跳转到 textbox 的时候触发。"blur"事件与之

相反，是离开的时候触发。上述事件处理器会在触发时清除经度和纬度 textbox 值，然后在我们的地图上指定新的位置。根据我们给出的地址，虚拟地图会返回一个值，然后在 map.js 文件中定义的回调事件处理器会使用该值更新经度和纬度 textbox。

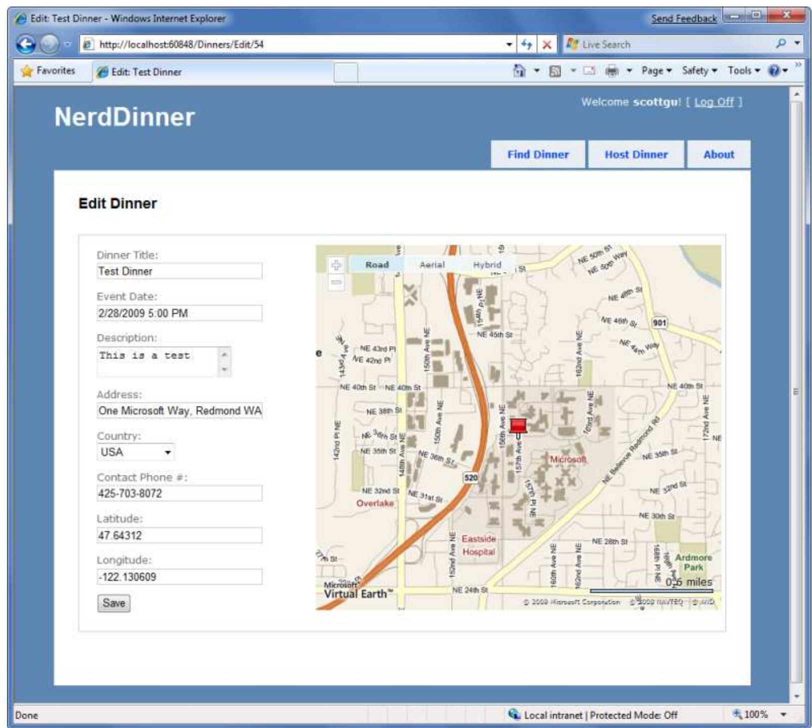
现在，再次运行应用程序，点击“Host Dinner”标签，我们会看到显示的默认地图：



当我们输入地址，地图会动态更新，并显示位置所在，事件处理器会根据位置值生成经度和纬度：



如果我们保存的用餐，然后再次打开进行编辑，就会在加载页面时看到显示的地图位置：

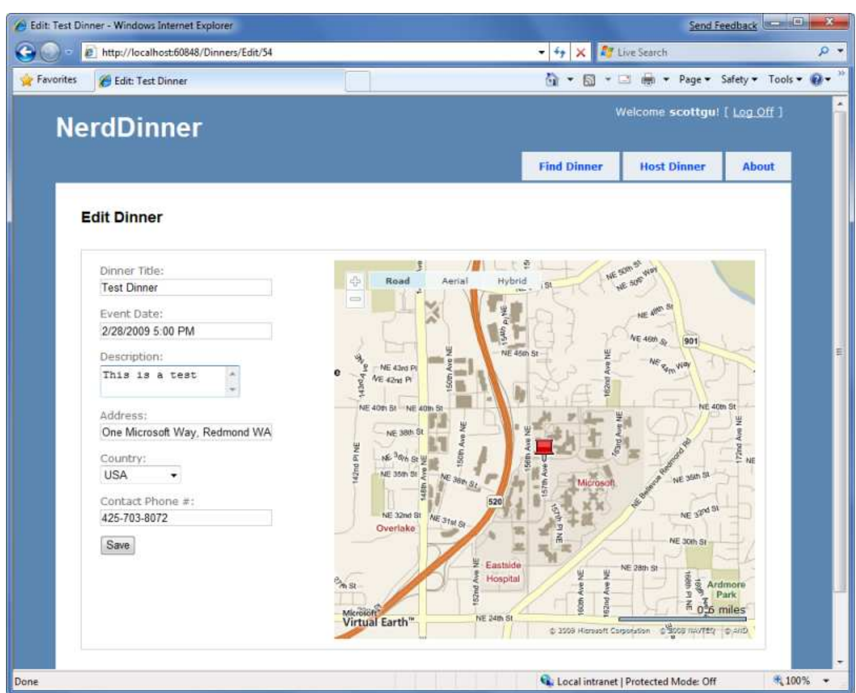


每次地址字段发生变化，地图和经度/纬度就会相应更新。

现在，地图显示了 Dinner 的地点，我们也能够修改从可见的 textbox 中修改经度和纬度值。由于地图会在输入地址的时候自动更新经度和纬度值，所以可以将它们隐藏起来。要实现这一目的，可以使用 Html.Hidden()辅助方法：

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

现在显示的界面就更加友好，避免显示经度和纬度值（在数据库中会为每次 Dinner 存储它们）：



集成地图到 Details 视图

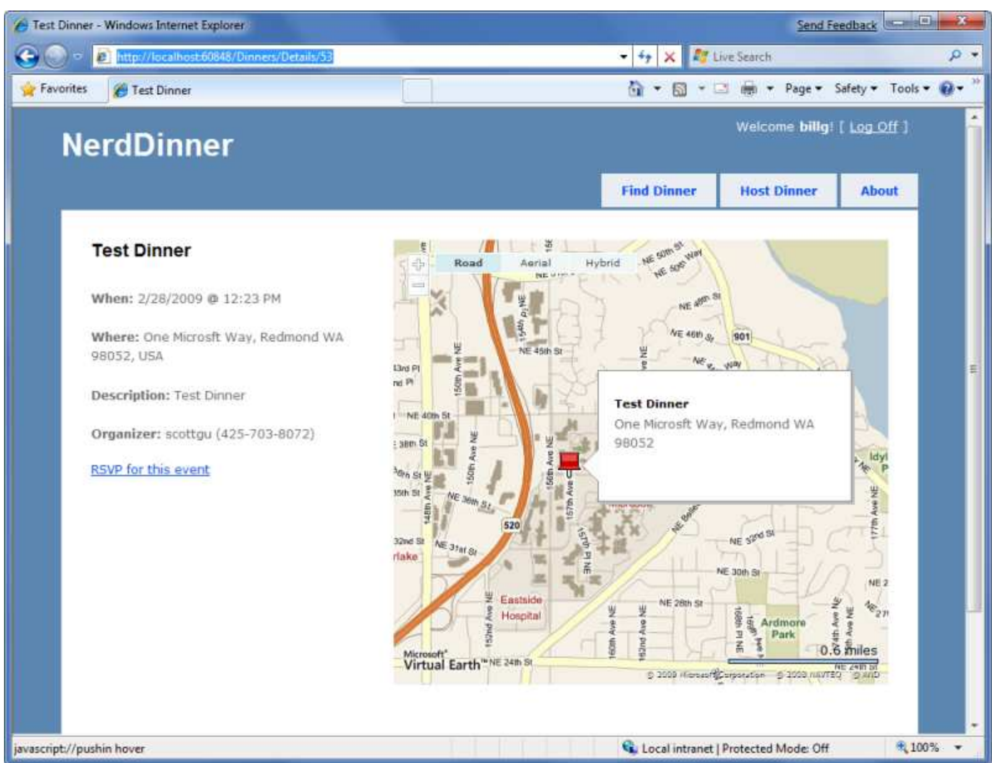
现在，我们已经将地图集成到了创建和编辑表单，接着是将地图集成到 Details。我们需要做的在 Details 视图中调用<% Html.RenderPartial("map"); %>。

下面是集成了地图的完整 Details 视图的源代码：

```
<asp:Content ID="Content1" 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>When:</strong>
            <%=Model.EventDate.ToShortDateString() %>

            <strong>@</strong>
            <%=Model.EventDate.ToShortTimeString() %>
        </p>
        <p>
            <strong>Where:</strong>
            <%=Html.Encode(Model.Address) %>,
            <%=Html.Encode(Model.Country) %>
        </p>
         <p>
            <strong>Description:</strong>
            <%=Html.Encode(Model.Description) %>
        </p>
        <p>
            <strong>Organizer:</strong>
            <%=Html.Encode(Model.HostedBy) %>
            (<%=Html.Encode(Model.ContactPhone) %>)
        </p>

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

    </div>

    <div id="mapDiv">
        <%Html.RenderPartial("map"); %>
    </div>
```

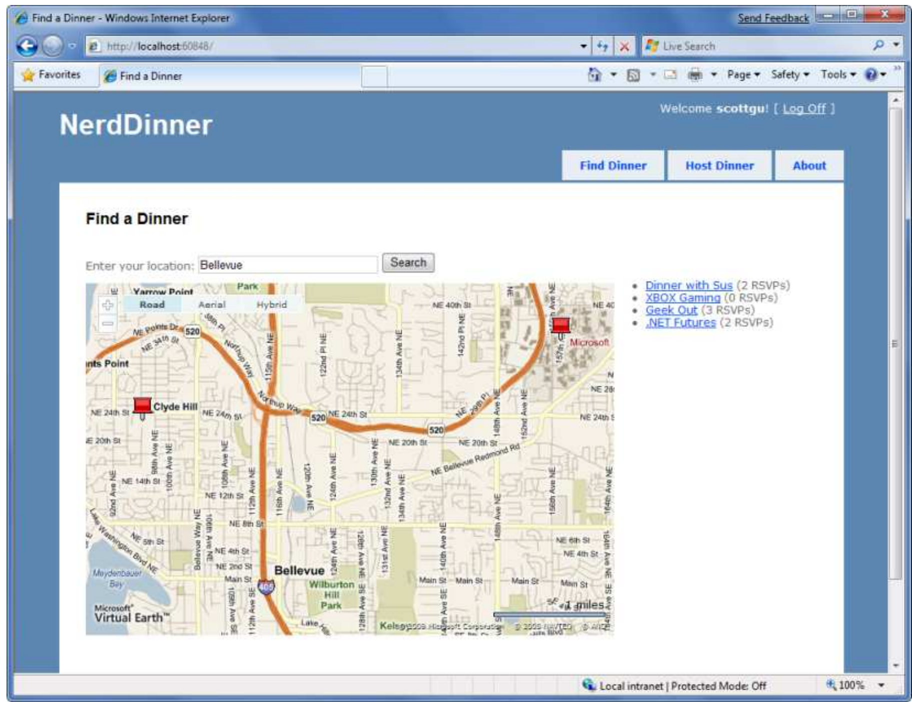
```
</asp:Content>
```

现在可以指向/Dinners/Details/[id] URL，就可以看到 Dinner 的详细情况，以及在地图中的用餐地点（当光标移到推针图案上时，就会显示用餐的标题以及地址），同时还包含了用 AJAX 实现的指向 RSVP 的链接：



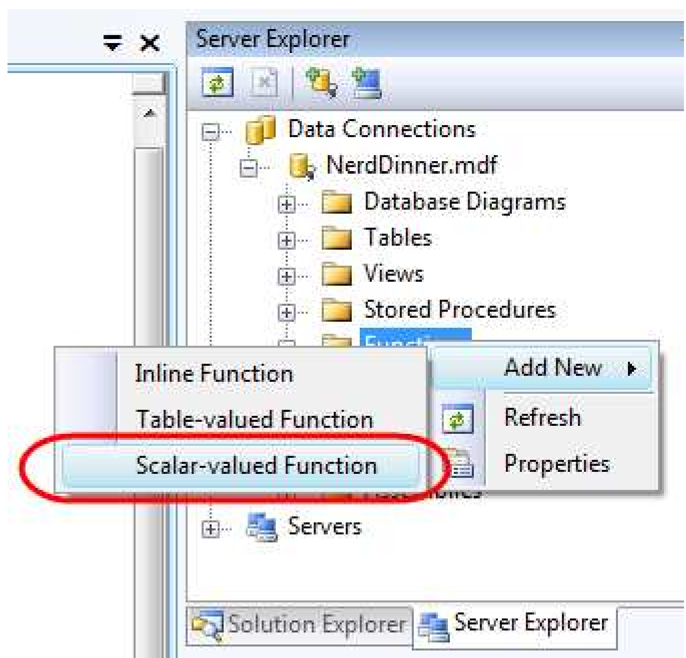
在数据库和仓储中实现位置搜索

为了完成 AJAX 实现，可以将地图移到首页，允许用户通过图形搜索附近的用餐信息：



现在，我们会在数据库以及数据仓储层提供支持，用以提供根据地址半径对 Dinners 进行搜索。我们可以使用 SQL 2008 提供的 geospatial 特性来实现，或者使用 Gary Dryden 提供的 SQL 方法，参见文章：http://www.codeproject.com/KB/cs/distancebetweenlocations.aspx 以及 Rob Conery 在博客中提出的使用 LINQ to SQL 方式：http://blog.wekeroad.com/2007/08/30/linq-and-geocoding/。

要实现这一技术，可以在 Visual Studio 中打开“Server Explorer”，选择 NerdDinner 数据库，然后右键单击其下的“function”子节点，选择创建一个新的“Scalar-valued function”：

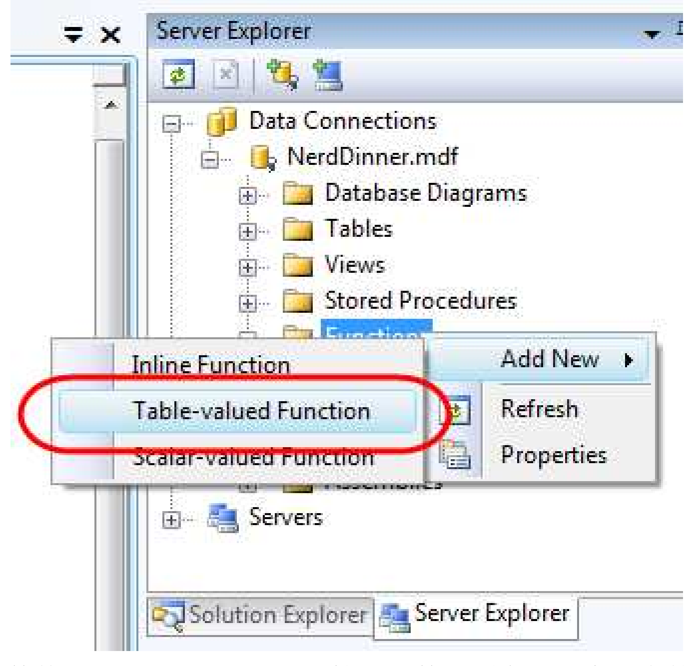


然后粘贴如下的 DistanceBetween 函数：

```
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
```

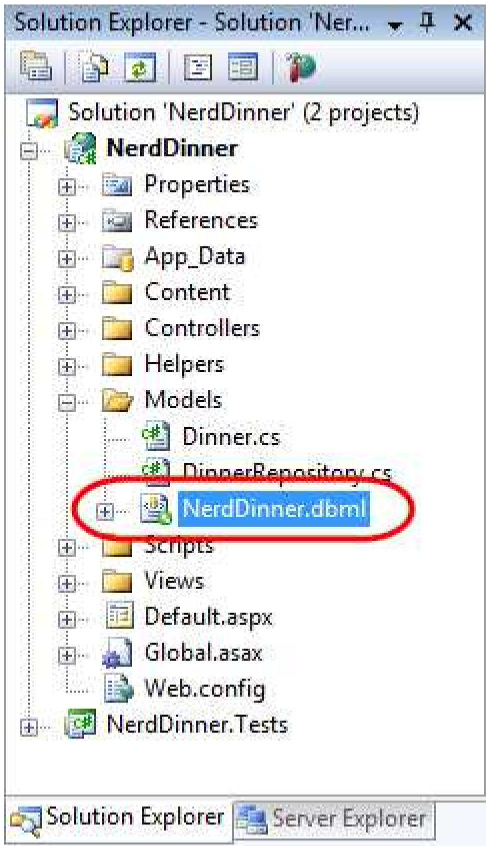
然后我们可以在 SQL 服务器中创建一个新的 table-valued 函数，命名为“NearestDinners”：



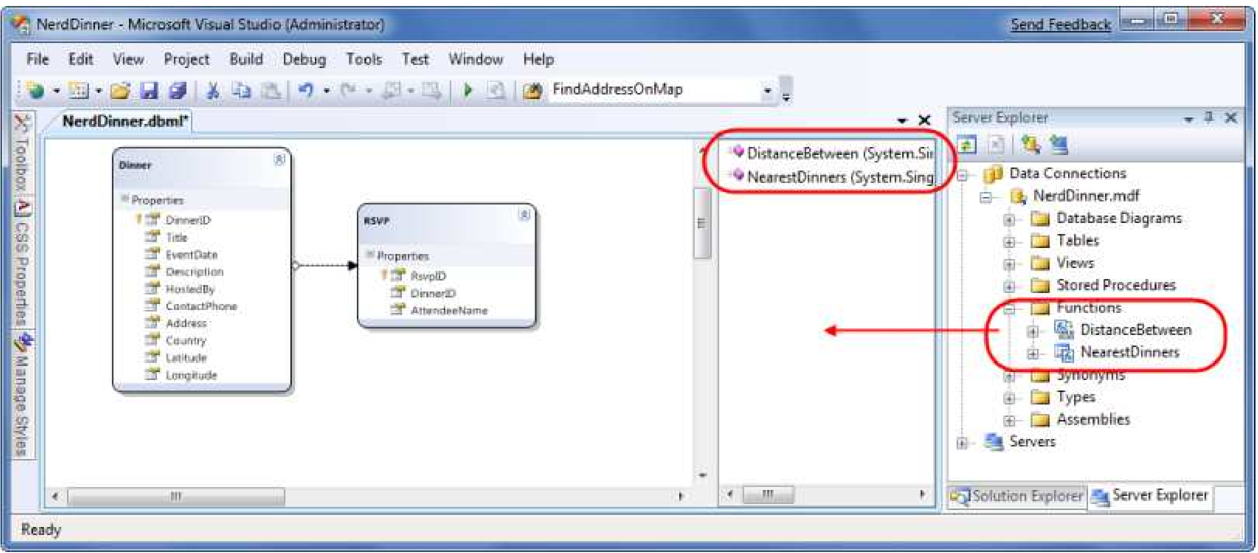
“NearestDinners”表函数使用了 DistanceBetween 辅助函数返回在 100 米以内的所有 Dinners：

```
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
```

要调用这一函数，我们首先通过双击在\Model 目录下的 NerdDinner.dbml 文件，打开 LINQ to SQL 设计器：



然后，我们将 NearestDinners 和 DistanceBetween 函数拖到 LINQ to SQL 设计器中，它们就可以作为 NerdDinnerDataContext 类的方法：



我们可以在 DinnerRepository 类中公开一个“FindByLocation”查询方法，使用 NearestDinner 函数返回距离指定位置 100 米内的即将举行的 Dinner：

```
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;
}
```

实现基于 JSON 的 AJAX 搜索 Action 方法

我们已经实现了一个控制器的 Action 方法，使用了新的 FindByLocation()方法，返回一个 Dinner 列表，用以生成地图。我们还可以让该方法以 JSON（JavaScript Object Notation）格式返回 Dinner 数据，这样就可以在客户端使用 JavaScript 来获得。

要实现这一目的，我们需要创建一个新的“SearchController”类，方法是右键单击\Controllers 目录，然后选择 Add->Controller 菜单项。然后在新的 SearchContoller 类中实现“SearchByLocation”action 方法：

```
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());
    }
}
```

方法中使用了 Controller 基类的 Json()辅助方法，使用基于 JSON 格式的 dinners 序列。JSON 是一个标准的文本，用来显示简单的数据结构。下面的例子就是 JSON 格式的包含了两个 JsonDinner 对象的列表：

```
[{"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}]
```

使用 jQuery 调用基于 JSON 的 AJAX 方法

现在我们可以使用 SearchController 类的 SearchByLocation()Action 方法改变 NerdDinner 应用程序的主页。我们可以打开/Views/Home/Index.aspx 视图模板，并更新为包含文本框、搜索按钮、地图和名为 dinnerList 的<div>元素：

```
<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>
```

然后，我们可以添加两个 JavaScript 函数到页面中：

```
<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>
```

第一个 JavaScript 函数会在装载页面时装载地图。第二个 JavaScript 函数则在搜索按钮连接了一个 JavaScript 单击事件处理器。当按钮被按下时，会调用添加到 Map.js 文件中的 FindDinnersGivenLocation()JavaScript 函数：

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

```
}
```

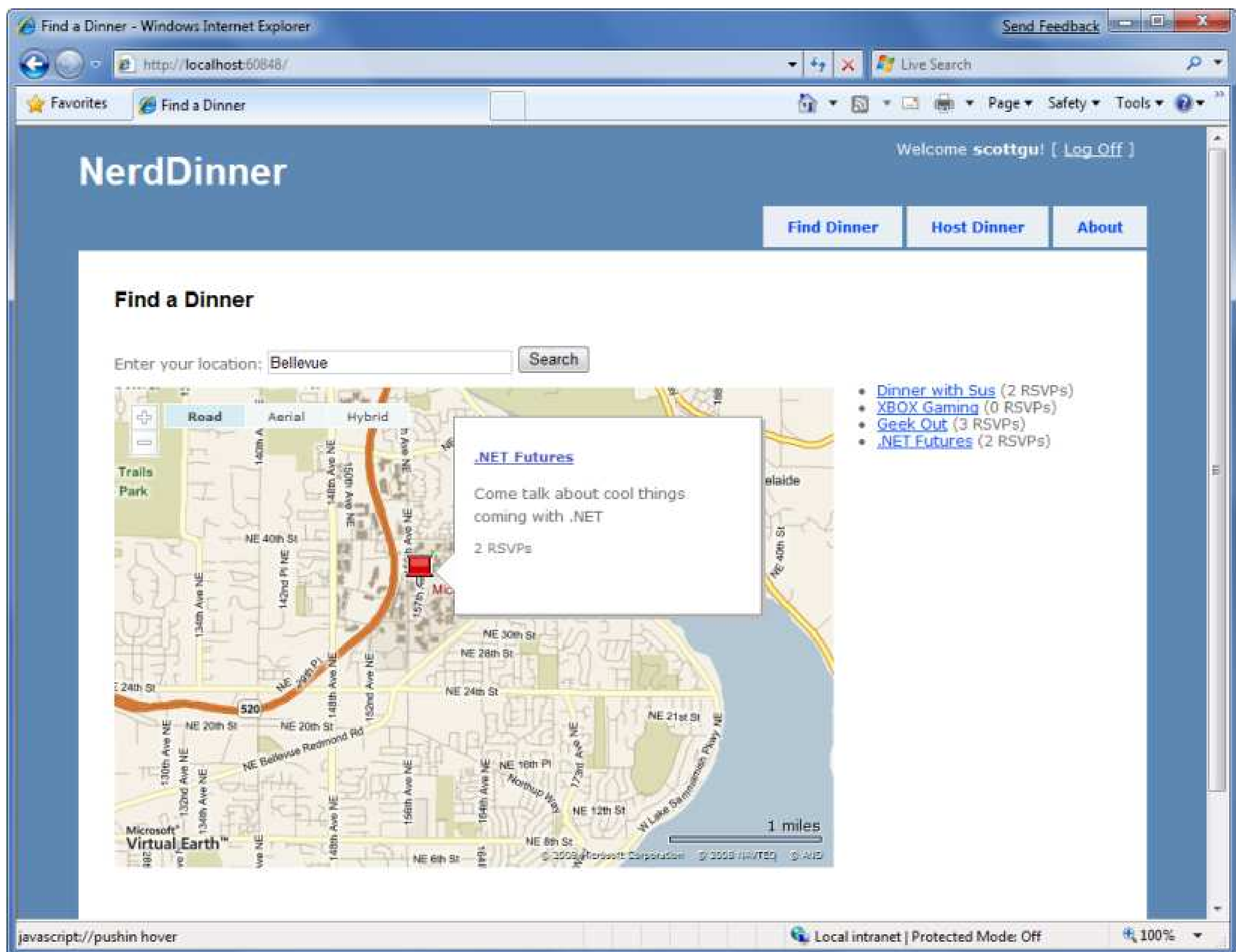
FindDinnersGivenLocation()函数会调用 Virtual Earth 控件的 map.Find()，找到输入位置的中心。当虚拟地图服务返回时，map.Find()方法会调用 callbackUpdateMapDinners 回调方法。

callbackUpdateMapDinners()方法才是完成真正工作的方法。它使用了 jQuery 的$.post()辅助方法执行一个 AJAX 调用，调用 SearchController 的 SearchByLocation()Action 方法。它定义了一个内联函数，当$.post()方法完成后，会调用该函数，然后从 SearchByLocation()方法返回的 JSON 格式的 dinners 结果为作为一个名为"dinners"变量被传递。然后对返回的 dinners 进行遍历，使用 dinner 的经度和纬度以及其他属性添加到地图的一个新的推针上。它还会添加一个 dinner 入口到 dinners 的 HTML 列表，放在地图右侧。然后为推针和 HTML 列表连接一个 hover 事件，这样当用户将光标移到上方时，会显示 dinner 的详细内容：

```
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";

            // Add Pin to Map
            LoadPin(LL, '<a href="/Dinners/Details/' + dinner.DinnerID + '">'
                        + dinner.Title + '</a>',
                        "<p>" + dinner.Description + "</p>" + RsvpMessage);

            //Add a dinner to the <ul> dinnerList on the right
            $('#dinnerList').append($('<li/>')
                            .attr("class", "dinnerItem")
                            .append($('<a/>').attr("href",
                                      "/Dinners/Details/" + dinner.DinnerID)
                            .html(dinner.Title))
                            .append(" ("+RsvpMessage+")"));
        });

        // Adjust zoom to display all the pins we just added.
```

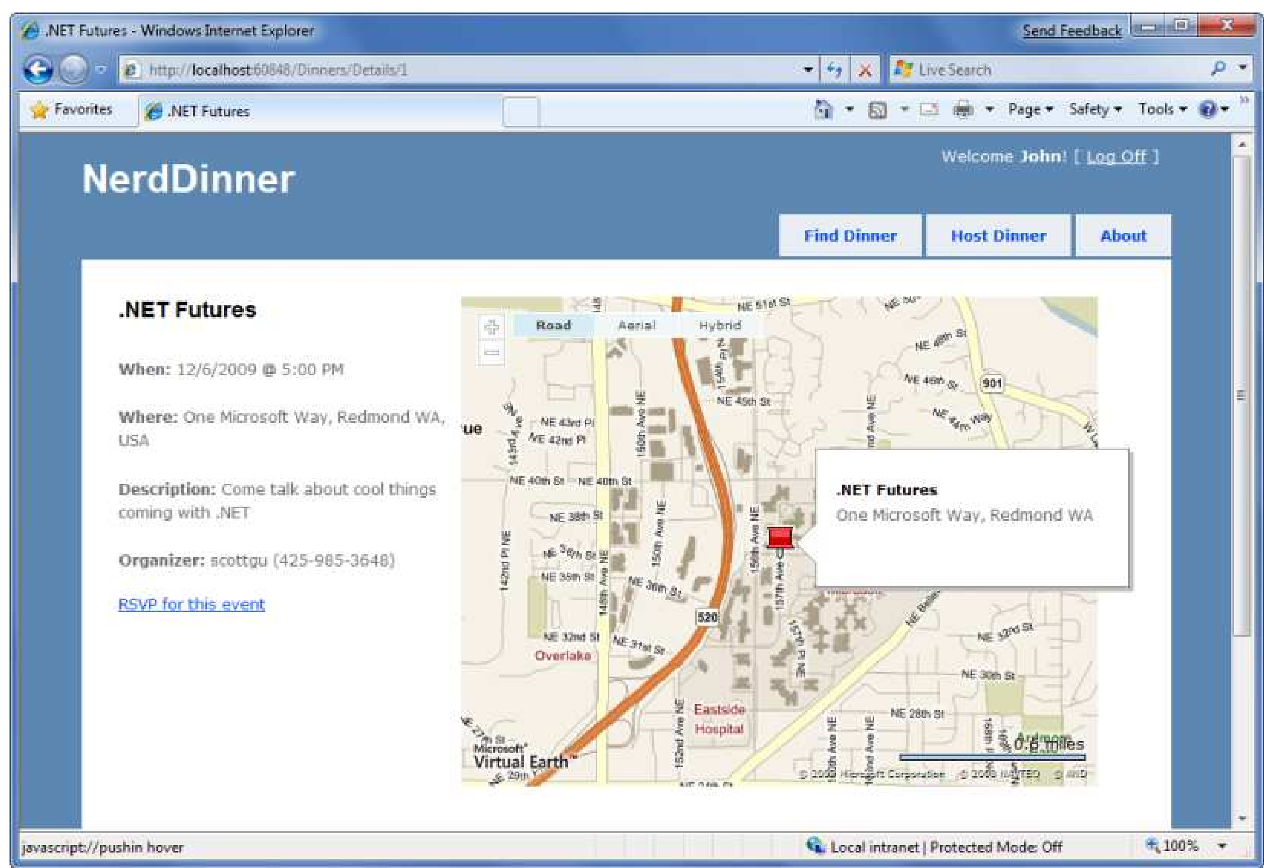
```
        map.SetMapView(points);

        // Display the event's pin-bubble on hover.
        $(".dinnerItem").each(function(i, dinner) {
            $(dinner).hover(
                function() { map.ShowInfoBox(shapes[i]); },
                function() { map.HideInfoBox(shapes[i]); }
            );
        });
    }, "json");
```

现在，当我们运行应用程序并访问主页时，会显示一个地图。当我们输入一个城市的名称会显示接下来会在附近举行的 dinner：



光标移到 dinner 上时会显示其细节。

单击 HTML 列表的右侧或者在弹出框的 Dinner 标题，会转向 dinner，通过此我们可以选择 RSVP：



单元测试

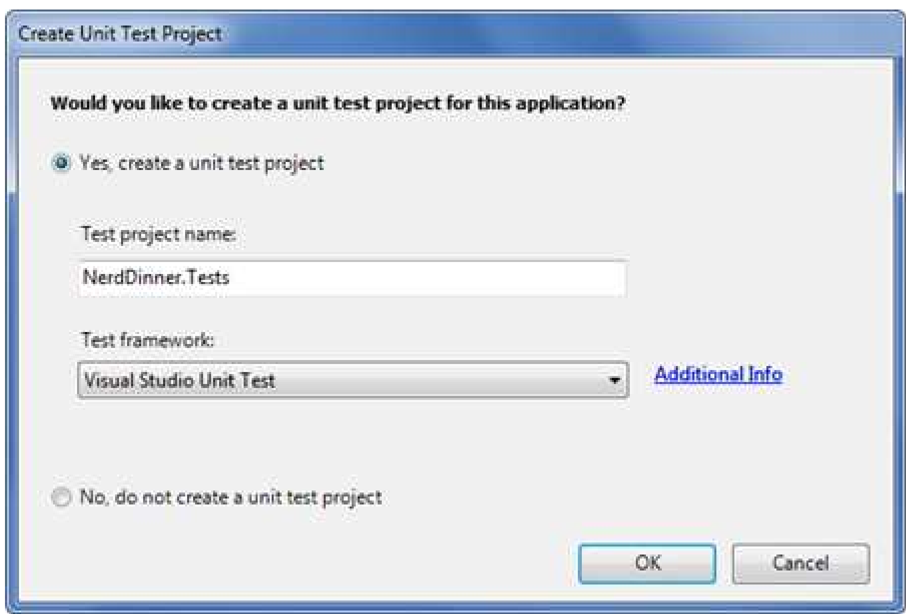
让我们开发一套自动的单元测试来验证 NerdDinner 范例程序的功能，这样在将来，我们可以自信地修改和改进应用程序。

为什么需要单元测试？

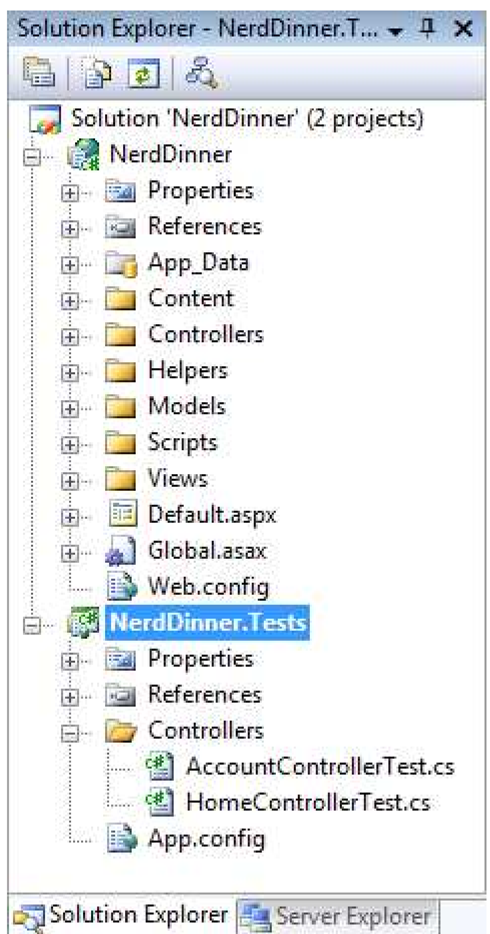
某天早上，你突然灵光一现，对正在工作的应用程序有个新的想法。你认为如果你实现一个更新，将是三整个应用程序得到极大的改进。这也许就是重构，精简代码、添加新的功能或者修复 bug。
当你坐在电脑前时，你面临的一个问题是 – 做这些更改，到底有多安全？是否这些更新有副作用或者破坏一些功能？这些更新可能很简单，只需要几分钟就可以完成，但是是否需要数个小时来手动测试整个应用程序？是否有可能你忘记一些，并导致有问题的应用程序上线到生产环境？做这些更新是否真的值得所有的付出？
自动化单元测试提供安全性，让你不断地增强你的应用程序，并避免害怕更新代码。自动化测试快速验证应用程序的功能，让你变得更加自信，有能力改进代码。也有助于创建易于维护和长久生命周期的项目，从而产生更高的投资回报。
ASP.NET MVC 框架使得单元测试更加容易和自然，也支持 TDD（Test Driven Development）工作流，开发基于测试优先。

NerdDinner.Tests 项目

当我们在最开始创建 NerdDinner 范例程序时，会弹出一个对话框询问是否想同时创建单元测试项目。



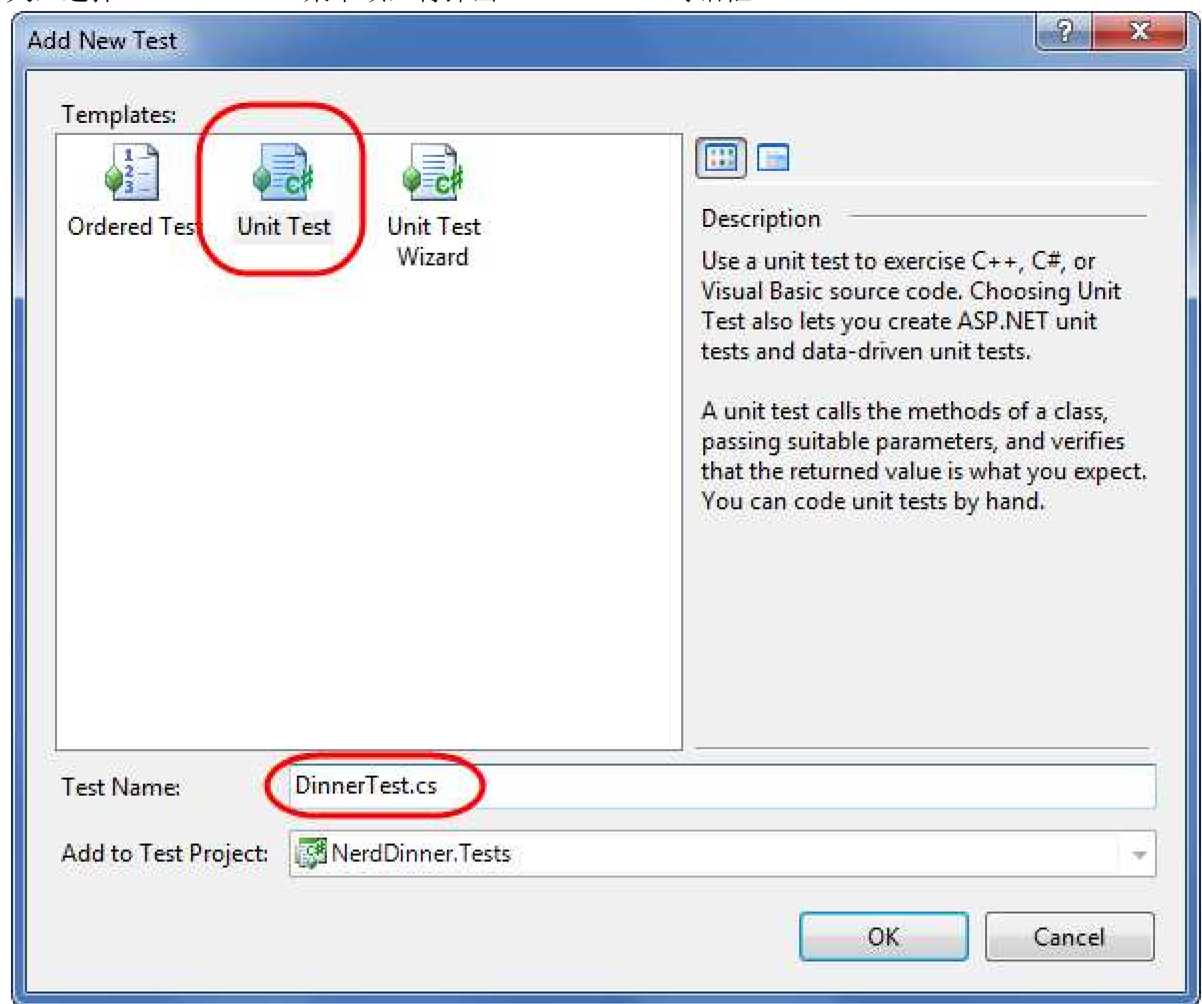
选择“Yes, create a unit test project”选项，就会添加一个测试项目到解决方案中。



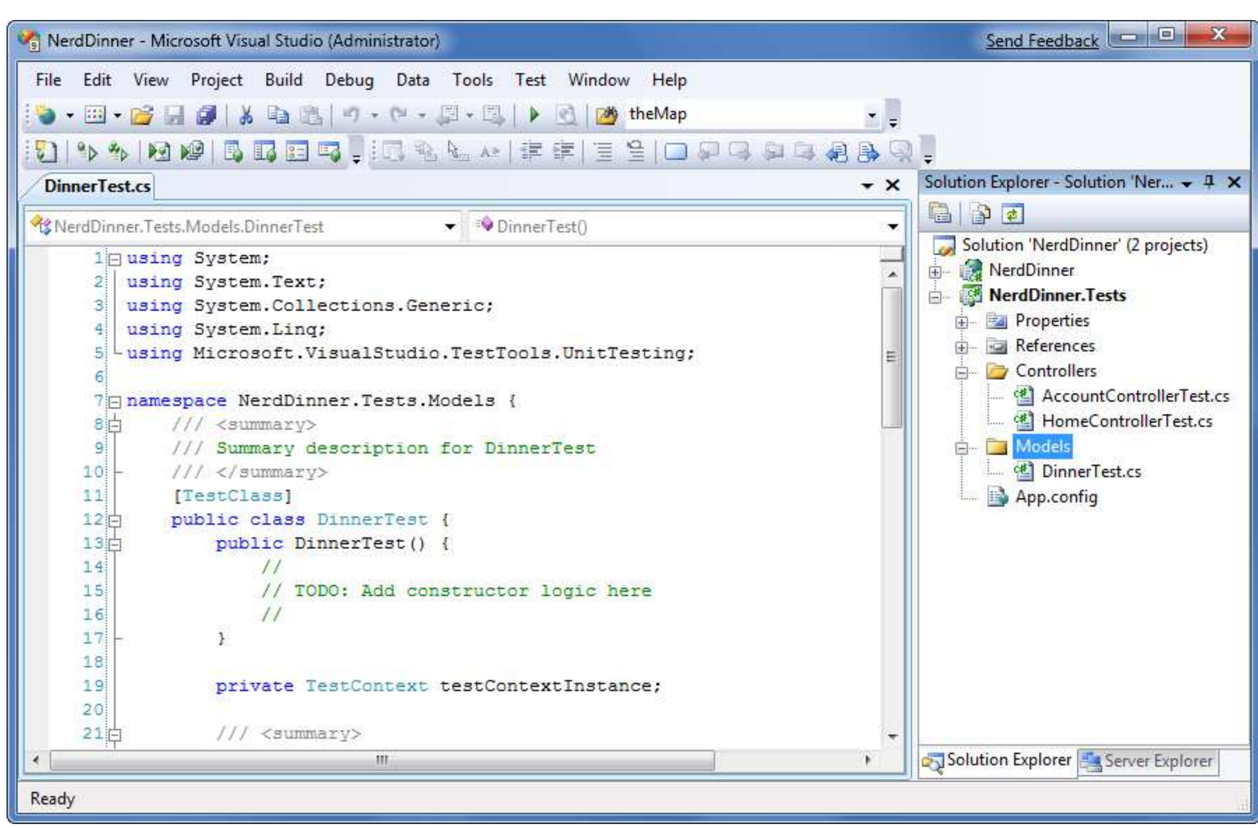
NerdDinner.Tests 项目引用了 NerdDinner 应用程序项目的程序集，让我们轻松实现自动化测试，并验证应用程序的功能。

为 Dinner 模型类创建单元测试

添加一些测试到 NerdDinner.Tests 项目，验证我们在创建模型层（Model Layer）创建的 Dinner 类。
在测试项目创建一个新的文件夹 – Models，在这里我们存放一些模型相关的测试。接着，右键单击文件夹，选择 Add->New Test 菜单项，将弹出 Add New Test 对话框。



我们选择创建 Unit Test 单元测试，命名为 DinnerTest.cs：



默认 Visual Studio 的单元测试模板有一些代码，且有些杂乱。让我们清理代码如下：

```
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 类中的[TestClass] 属性标识该类将包含测试，以及可选的初始化和代码。我们可以添加其他公有的方法，并设置 [TestClass] 属性。

下面是我们测试 Dinnerlei 的其中第一个（总共有 2 个）测试，第一个测试验证：如果一个新的 Dinner 对象创建过程中没有设置正确的属性，测试方法则认为 Dinner 对象是无效的。

接下来是第二个测试，该测试验证如果一个 Dinner 对象正确设置了所有属性，则 Dinner 对象是有效的。

```
[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);
    }
}
```

你应该注意到测试方法的名称非常清楚（甚至有点冗长）。我们这样命名是因为我们需要创建成千上百的测试方法，通过方法名称，我们可以迅速地了解的每一个方法的意图和行为（特别是在查看错误列表时）。测试方法的名称总是命名为正在测试的功能，我们使用 Noun_Should_Verb 命名模式。

我们使用 AAA 测试模式创建测试方法 – 分别代表 Arrange、 Act 和 Assert：

Arrange – 设置测试单元；

Act – 执行测试单元，并捕获结果；

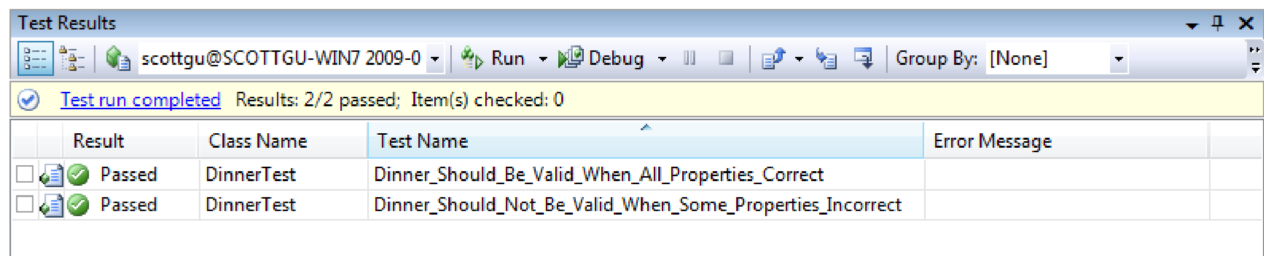
Assert – 验证执行行为；

当我们编写测试时，应尽量避免有太多的单个的测试。每一个测试应该验证一个单一的概念（这样，可以轻松定位到错误的原因）。一个好的设计向导是尽量针对每一个测试有一个 assert（断言）语句。如果你在一个测试方法中有多个 assert 语句，确保它们都在测试同一个概念。如不确定，则创建另外一个测试方法。

运行测试

Visual Studio 2008 专业版（或其他高级版本）包含了一个内置的测试运行器，可以在 IDE 中运行 Visual Studio Unit Test 项目。选择 Test -> Run -> All Tests in Solution 菜单项，运行所有的单元测试。或者将光标定位到一个特定的测试类或测试方法中，选择 Test -> Run -> Test in Current Context 菜单项，运行部分单元测试。

下面我们将光标定位到 DinnerTest 类中，选择 Test -> Run -> Test in Current Context 菜单项，运行我们刚才定义的 2 个测试方法。随后 Test Results 窗口自动在 Visual Studio 中出现，我们将可以看到测试结果：



备注：VS 测试结果窗口默认没有显示类名称（Class Name）。可以右键点击 Test Results 窗口，选择 Add/Remove Columns 菜单项，然后添加 Class Name 选项。

上述仅仅测试了一小部分，且都通过了测试。下面进行创建其他的测试，来验证特定的规则，并覆盖 2 个辅助方法 – IsUserHost() 和 IsUserRegistered() – 这是之前添加到 Dinner 类中的方法。针对 Dinner 类的这些测试让我们今后可以更容易和更安全地添加新的业务规则和验证。我们可以添加新的业务逻辑到 Dinner 类中，然后在几秒钟内验证这些更新是否破坏了任何之前的逻辑和功能。

认识到一个清楚的测试方法的名称有助于尽快理解政治测试的内容。另外，还建议使用 Tools->Options 菜单项，打开 Test Tools -> Test Execution 配置窗口，选择“Double-clicking a failed or inconclusive unit test result displays the point of failure in the test”复选框。当在测试结果窗口双击错误记录时，会立即跳到断言错误的地方。

创建 DinnersController 单元测试

下面创建一个单元测试验证 DinnersController 的功能。右键点击测试项目中的 Controllers 文件夹，选择 Add->New Test 菜单项，创建一个单元测试，命名为 DinnersControllerTest.cs。
创建 2 个测试方法验证 DinnersController 中的 Details() action 方法。第一个将验证当请求一个存在的 Dinner 对象时，一个视图将返回。第二个将验证如果请求一个不存在的 Dinner 对象时，NotFound 视图将返回。

```
[TestClass]
public class DinnersControllerTest {
```

```
    [TestMethod]
    public void DetailsAction_Should_Return_View_For_ExistingDinner() {

        // Arrange
        var controller = new DinnersController();

        // Act
        var result = controller.Details(1) as ViewResult;

        // Assert
        Assert.IsNotNull(result, "Expected View");
    }

    [TestMethod]
    public void DetailsAction_Should_Return_NotFoundView_For_BogusDinner() {

        // Arrange
        var controller = new DinnersController();

        // Act
        var result = controller.Details(999) as ViewResult;

        // Assert
        Assert.AreEqual("NotFound", result.ViewName);
    }
}
```

运行测试，上述 2 个测试都会失败：

Test Results

scottgu@SCOTTGU-WIN7 2009-0 ▾ Run ▾ Debug ▾ Group By: [None] ▾ [All Columr ▾

Test run failed Results: 0/2 passed; Item(s) checked: 2

Result	Class Name	Test Name	Error Message
Failed	DinnersControllerTest	DetailsAction_Should_Return_NotFoundView_For_BogusDinner	Test method NerdDinner.Tests.Co
Failed	DinnersControllerTest	DetailsAction_Should_Return_View_For_ExistingDinner	Test method NerdDinner.Tests.Co

查看错误信息，失败的原因是因为 DinnersRepository 类不能连接到数据库。NerdDinner 范例程序使用连接字符串连接到 SQL Server 数据库。因为 NerdDinner.Tests 项目还没有正确配置数据库连接信息。
只需要打开 NerdDinner.Tests 项目中 app.config 配置文件，配置相应的数据库连接信息就可以了：

```
  <connectionStrings>
    <addname="NerdDinnerConnectionString"connectionString="Data
Source=localhost;Initial Catalog=NerdDinner;Integrated Security=True"
     providerName="System.Data.SqlClient" />
  </connectionStrings>
```

单元测试代码使用真实的数据库，可能带来一些挑战，如：
1. 它会显著降低单元测试的执行速度。执行测试的时间越长，你会越少执行单元测试。理想情况是，你

希望单元测试能够在几秒内执行完成，像编译项目一样自然。
2. 它使设置和清理过程变得复杂。你希望每一个单元测试独立、互不依赖。如果连真实的数据库进行测试，你需要关心数据状态，并在不同的测试直接进行数据复位。

下一节将演示“依赖注入（Dependency Injection）”设计模式，可以帮助我们解决这些问题，并避免在测试过程中使用真实的数据库。

依赖注入（Dependency Injection）

现在 DinnersController 紧耦合 DinnerRepository 类，耦合（Coupling）指一个类显式依赖另外的一个类才能工作。

```
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);
}
```

因为 DinnerRepository 类需要访问数据库，DinnersController 类对 DinnerRepository 类的紧耦合导致 DinnersController action 方法的测试都需要连接数据库。
我们可以通过 Dependency Injection(依赖注入)设计模式来解决这一问题，类之间（如 Repository 类提供数据访问）不再创建隐式的依赖。而是，通过调用方的构造函数的参数，显式传递依赖关系。如果依赖关系通过接口来定义，我们就可以针对单元测试的情况，灵活传递虚假（Fake）的依赖实现。这样，我们在创建测试相关的依赖实现时，不必访问真实的数据库。

下面演示具体实现，首先对 DinnersController 实现依赖注入。

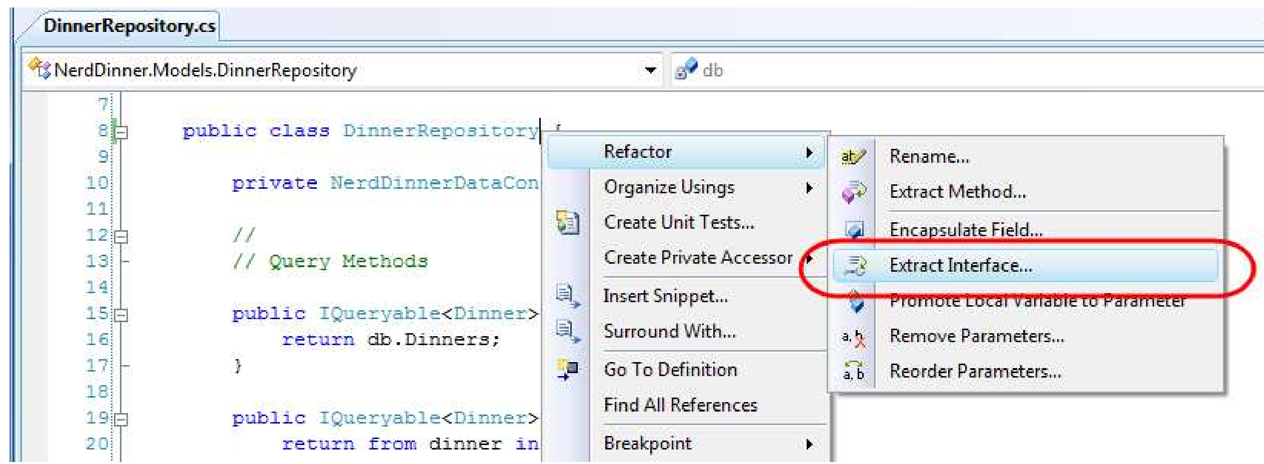
提取 IDinnerRepository 接口

第一步是创建新的 IDinnerRepository 接口，封装 Controller 检索和更新 Dinners 对象所需要的 Repository 契约。

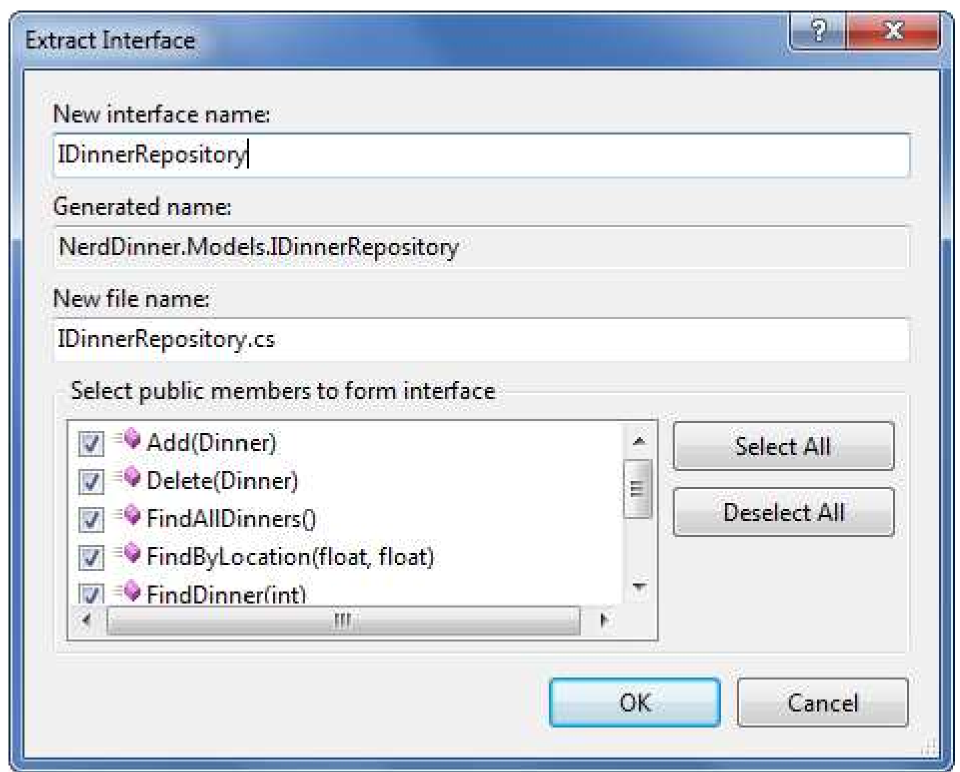
右键点击\Models 文件夹，选择 Add->New Item 菜单项，创建一个新的接口 IDinnerRepository.cs。

另外一种方法是，使用 Visual Studio 内置的重构工具（Refactoring tools），从现有的 DinnerRepository 类

中自动提取并创建一个接口文件。如通过 VS 提取这一接口文件，只需将光标定位到 DinnerRepository 类中，右键并选择 Refactor -> Extract Interface 菜单项：



随后，将弹出 Extract Interface 对话框，接口命名默认为 IDinnerRepository，并自动选择 DinnerRepository 类中的所有公共的方法，添加到接口中：



在点击 OK 按钮后，Visual Studio 将添加一个新的 IDinnerRepository 接口到应用程序中：

```
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();
}
```

现有的 DinnerRepository 类将更新为实现该接口：

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

更新 DinnersController 支持构造器注入

现在实现新的接口，更新 DinnersController 类。

目前，DinnersController 类是硬编码的，如 dinnerRepository 属性总是类型为 DinnerRepository 实例：

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();
    ...
}
```

我们更改上述代码，将 dinnerRepository 属性由 DinnerRespository 类型更改为 IDinnerRepository 接口类型，接着添加 2 个公共的 DinnersController 构造器。其中一个构造器允许传入 IDinnerRepository 类型的参数，另外一个是默认的构造器，使用现有的 DinnerRepository 的实现：

```
public class DinnersController : Controller {
    IDinnerRepository dinnerRepository;

    public DinnersController()
        : this(new DinnerRepository()) {
    }

    public DinnersController(IDinnerRepository repository) {
        dinnerRepository = repository;
    }
    ...
}
```

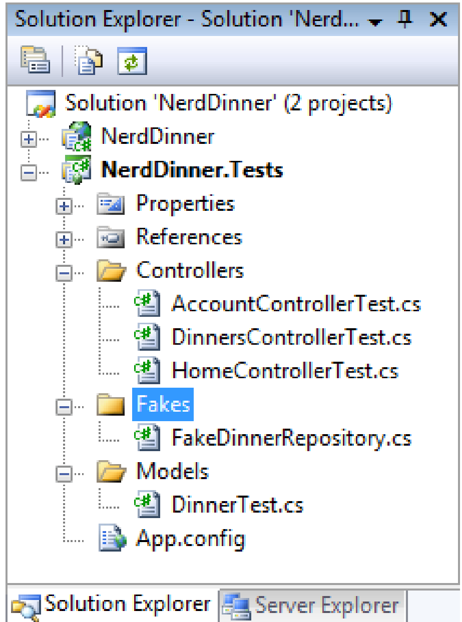
因为默认情况下 ASP.NET MVC 使用默认构造器创建控制器 Controller 类，DinnersController 控制器在运行时将继续使用 DinnerRepository 类执行数据访问。

但是，现在我们可以更新单元测试代码，使用带参数的构造器，传入一个虚假的 Dinner Repository 的实现。虚假的 Dinner repository 不需要访问真实的数据库，而是使用内存中的样本数据。

创建 FakeDinnerRepository 类

下面开始创建 FakeDinnerRepository 类。

首先，在 NerdDinner.Tests 项目中创建 Fakes 目录，接着添加一个新的 FakeDinnerRepository 类到该目录（右键点击该目录，选择 Add->New Class 菜单项）。



更新 FakeDinnerRepository 类，实现 IDinnerRepository 接口。接着右键点击，并选择 Implement interface IDinnerRepository 上下文菜单项：

FakeDinnerRepository.cs* IDinnerRepository.cs

NerdDinner.Tests.Fakes.FakeDinnerRepository

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using NerdDinner.Models;
6
7  namespace NerdDinner.Tests.Fakes {
8
9      public class FakeDinnerRepository : IDinnerRepository {
10     }
11 }
12
```

Implement interface 'IDinnerRepository'

Explicitly implement interface 'IDinnerRepository'

这样，Visual Studio 将自动添加 IDinnerRepository 接口成员到 FakeDinnerRepository 类中，并附有默认的基础（存根）实现：

```
public class FakeDinnerRepository : IDinnerRepository {

    public IQueryable<Dinner> FindAllDinners() {
        throw new NotImplementedException();
    }

    public IQueryable<Dinner> FindByLocation(float lat, float longValue){
        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();
    }
}
```

接着更新 FakeDinnerRepository 的实现代码，对作为构造函数参数传入的 List<Dinner>集合进行访问，而不是真实的数据库记录：

```
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");
        }
    }
}
```

现在，虚假的 IDinnerRepository 的实现不需要数据库了，可以工作在内存中的 Dinner 对象列表。

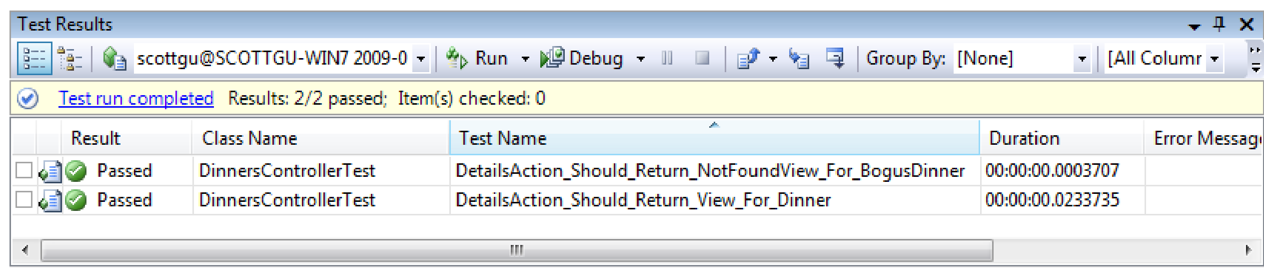
在单元测试中使用 FakeDinnerRepository

我们回到 DinnersController 单元测试，之前由于数据库不能访问，而有异常或失败。在 DinnersController 类中，我们将使用填充了内存中范例 Dinner 数据的 FakeDinnerRepository 类，来更新测试方法。示例代码如下：

```
[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);
    }
}
```

现在我们运行这些测试方法时，均验证通过：



最大的好处是，运行这些测试仅仅需要不到 1 秒，并且不需要任何复杂的安装/清理逻辑。现在，我们可以单元测试 DinnersController 类中的所有 action 方法（包括列表、分页、详细信息、创建、更新和删除等等），而不需要连接真实的数据库。

依赖注入框架

手动地实现依赖注入还是不错的，不过随着应用程序规模的增长，却很难维护依赖和组件。目前.NET 下有多个依赖注入框架，提供了依赖管理的灵活性。这些框架有时候也被称作“控制反转”容器。这些容器支持在运行时为对象传递依赖。在.NET 下常见的依赖注入框架/控制反转容器包括 AutoFac，Ninject，Spring.NET，StructureMap 和 Windsor。

创建 Edit Action 方法的单元测试

下面创建 DinnersController 的编辑功能的单元测试。首先，测试 Edit Action 方法的 HTTP-GET 版本：

```
//
// 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));
}
```

备注：上述代码中 DinnerFormViewModel 是之前我们定义的一个 Model 类。

我们将创建一个测试：当请求一个有效的 Dinner 对象时，验证返回的 DinnerFormViewModel 对象。

```
[TestMethod]
public void EditAction_Should_Return_View_For_ValidDinner() {

    // Arrange
    var controller = CreateDinnersController();
```

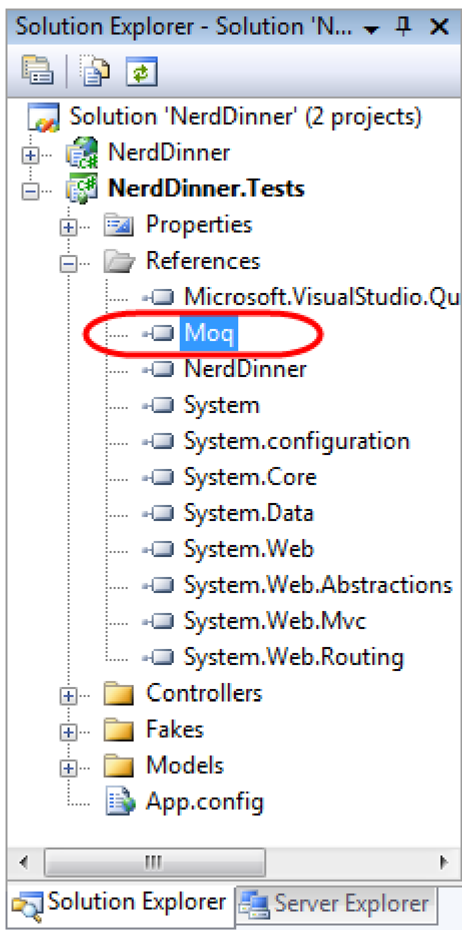
```
    // Act
    var result = controller.Edit(1) as ViewResult;

    // Assert
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(DinnerFormViewModel));
}
```

如果现在运行测试，发现测试会失败，这是因为 Edit 方法在访问 User.Identity.Name 属性，执行 Dinner.IsHostedBy()检查时，抛出 null 引用异常。
Controller 基类的 User 对象封装了登录用户的详细信息，在运行时创建 Controller 时，ASP.NET MVC 填充该对象。因为我们测试 DinnersController 时，没有运行在 web-server 的环境，因此 User 对象没有设置，导致 null 引用异常。

模仿 User.Identity.Name 属性

Mocking Framework 可以帮忙我们动态创建虚假的依赖对象，支持测试工作。例如，在 Edit Action 方法的测试中，我们可以使用一个 Mocking Framework，动态创建 User 对象，DinnersController 将使用该对象来模拟一个用户名。这样在运行测试时，可以避免 null 引用的发生。
有很多.NET Mocking Framework 可以应用于 ASP.NET MVC，访问 http://www.mockframeworks.com 可以查看到。对于 NerdDinner 应用程序的测试，我们使用一个开源的 Mocking Framework – Moq，可以从如下地址免费下载：http://www.mockframeworks.com/moq

下载后，在 NerdDinner.Tests 项目中添加对 Moq.dll 程序集的引用。



接着在测试类中添加一个重载的 CreateDinnersControllerAs(username) 辅助方法，接收 username 参数，该参数模仿 DinnersController 实例中的 User.Identity.Name 属性。

```
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;
}
```

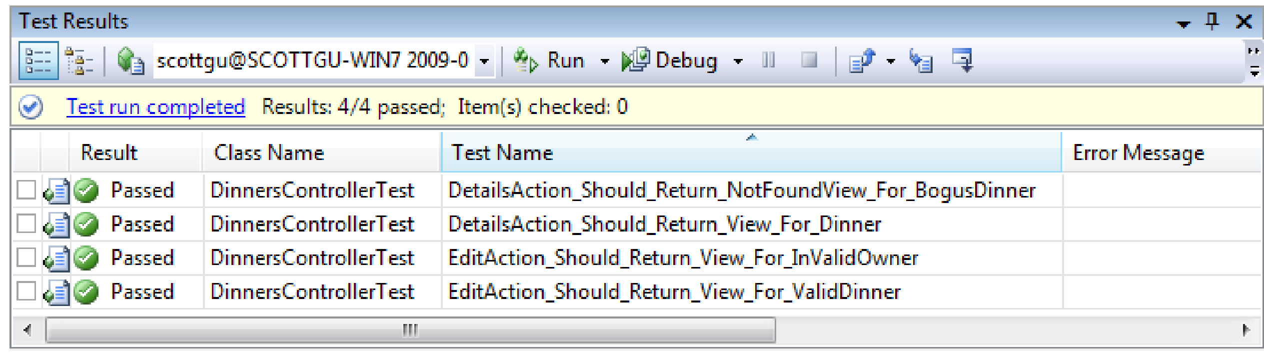
上述代码使用 Moq 创建一个 Mock 对象，虚拟一个 ControllerContext 对象（该对象是 ASP.NET MVC 传递给 Controller 类，公布运行时对象，如 User、Request、Response 和 Session）。调用 Mock 的 SetupGet 方法，表示 ControllerContext 的 HttpContext.User.Identity.Name 属性应该返回 username 字符串，该字符串是传递给辅助方法的参数。

我们可以模拟 ControllerContext 的任何属性和方法。为了证明这一点，我们也向 Request.IsAuthenticated 属性添加了 SetupGet() 的调用（该属性对于下面的测试是不需要的，但是可以证明如何模拟 Request 属性）。最后，将模拟的 ControllerContext 实例赋值给辅助方法需要返回 DinnersController 对象。

下面使用上述辅助方法编写单元测试，用不同的用户测试 Edit 方法：

```
[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");
}
```

现在通过所有测试：



测试 UpdateModel()

我们已经创建了测试 HTTP-GET 版本的 Edit Action 方法，下面继续创建测试 HTTP-POST 版本的 Edit Action 方法：

```
//
// 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));
    }
}
```

上述 Action 方法使用了 Controller 基类的 UpdateModel() 辅助方法，使用该辅助方法绑定表单提交的值到 Dinner 对象实例。

下面的 2 个测试演示了如何提供表单提交的值给 UpdateModel() 辅助方法使用。通过创建和填充一个 FormCollection 对象，接着赋值给 Controller 的 ValueProvider 属性，来实现测试。测试方法代码如下：

```
[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");
}
```

其中第一个测试验证：当成功保存后，浏览器重定向到 Details Action 方法。第二个测试验证：当提交无效的表单参数值时，重新显示带错误消息的 Edit 视图。

单元测试总结

我们已经完成了对 Controller 类进行单元测试的核心概念。我们可以使用这些技术轻松创建好几百简单测试，验证应用程序的功能。

因为 Controller 和 Model 测试不需要真实的数据库，这样可以非常快和容易运行。我们可以在几秒钟执行几百个自动化测试，并立即获得信息 – 是否更新破坏了现有的逻辑。这样，让我们有信心持续改进、重构和优化应用程序。

在本章的最后部分，我们介绍了测试相关技术，但并不表示测试是开发流程的最后阶段。相反，你应该在开发流程中尽早编写自动化测试。这样，你可以在开发过程中及时得到反馈结果，帮助你仔细思考应用程序的业务场景，并指导你设计清晰分层的、松耦合的应用程序。

《Professional ASP.NET MVC 1.0》这本书的随后章节将介绍 Test Driven Development（TDD），已经如何在 ASP.NET MVC 中使用。TDD 是一个迭代的开发过程。通过 TDD，首先编写验证将要实现的业务功能的测试。编写单元测试，可以帮助你清晰理解功能和如何工作。仅仅在完成测试代码的编写之后，才可是实现对应的实际功能。因为你已经思考了这些功能如何工作的业务场景，你可以更好地理解需求，以及如何最好地实现。当你完成了业务代码的编写之后，你可以重新运行测试，立即获得关于功能是否工作正常的反馈。

NerdDinner 范例程序总结

NerdDinner 范例应用程序终于完成了，已经可以发布了。

我们使用了大量的 ASP.NET MVC 功能来创建 NerdDinner 范例程序。希望这一开发过程演示了 ASP.NET MVC 核心功能是如何工作的，已经如何在一个应用程序中集成这些功能。