1

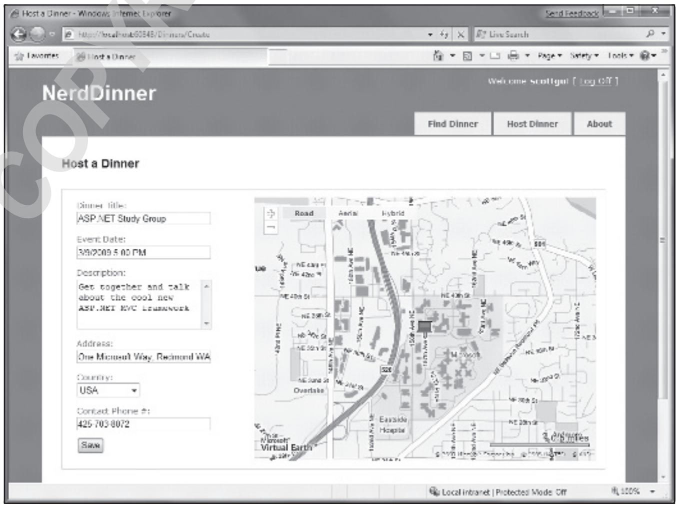
NerdDinner

COPYRIGHTED MATERIAL

The best way to learn a new framework is to build something with it. This first chapter walks through how to build a small, but complete, application using ASP.NET MVC, and introduces some of the core concepts behind it.

The application we are going to build is called "NerdDinner." NerdDinner provides an easy way for people to find and organize dinners online (Figure 1-1).

NerdDinner enables registered users to create, edit and delete dinners. It enforces a consistent set of validation and business rules across the application (Figure 1-2).


Host a Dinner - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Create
Live Search
Favorites
Host a Dinner
Page
Safety
Tools
NerdDinner
Welcome scottgu! [Log Off]
Find Dinner
Host Dinner
About
Road
Aerial
Hybrid
Virtual Earth
Local intranet | Protected Mode: Off
100%

Host a Dinner

Dinner Title: ASP.NET Study Group

Event Date: 3/9/2009 5:00 PM

Description: Get together and talk about the cool new ASP.NET MVC framework

Address: One Microsoft Way, Redmond WA

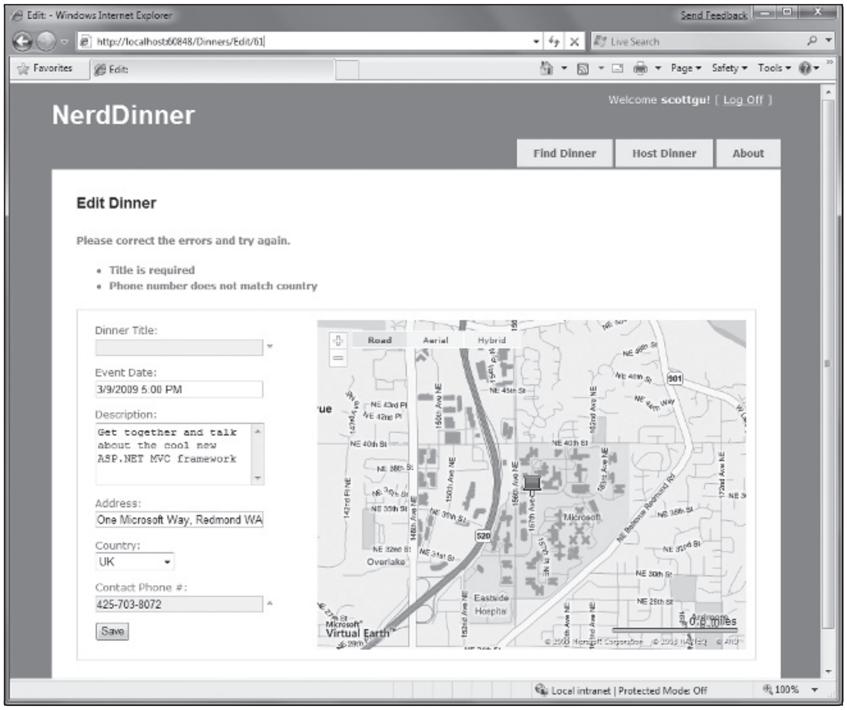
Country: USA

Contact Phone #: 425-703-8072

Save

Figure 1-1

Chapter 1 is licensed under the terms of Creative Commons Attribution No Derivatives 3.0 license and may be redistributed according to those terms with the following attribution: "Chapter 1 "NerdDinner" from Professional ASP.NET MVC 1.0 written by Rob Conery, Scott Hanselman, Phil Haack, Scott Guthrie published by Wrox (ISBN: 978-0-470-38461-9) may be redistributed under the terms of Creative Commons Attribution No Derivatives 3.0 license. The original electronic copy is available at `http://tinyurl.com/aspnetmvc`. The complete book Professional ASP.NET MVC 1.0 is copyright 2009 by Wiley Publishing Inc and may not redistributed without permission."


Edit: - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Edit/61
Live Search
Favorites
Edit:
Page ▾ Safety ▾ Tools ▾
Local intranet | Protected Mode: Off
100%

NerdDinner

Welcome **scottgu**! [Log Off]

Find Dinner | Host Dinner | About

Edit Dinner

Please correct the errors and try again.

- Title is required
- Phone number does not match country

Dinner Title:

Event Date: 3/9/2009 5:00 PM

Description: Get together and talk about the cool new ASP.NET MVC framework

Address: One Microsoft Way, Redmond WA

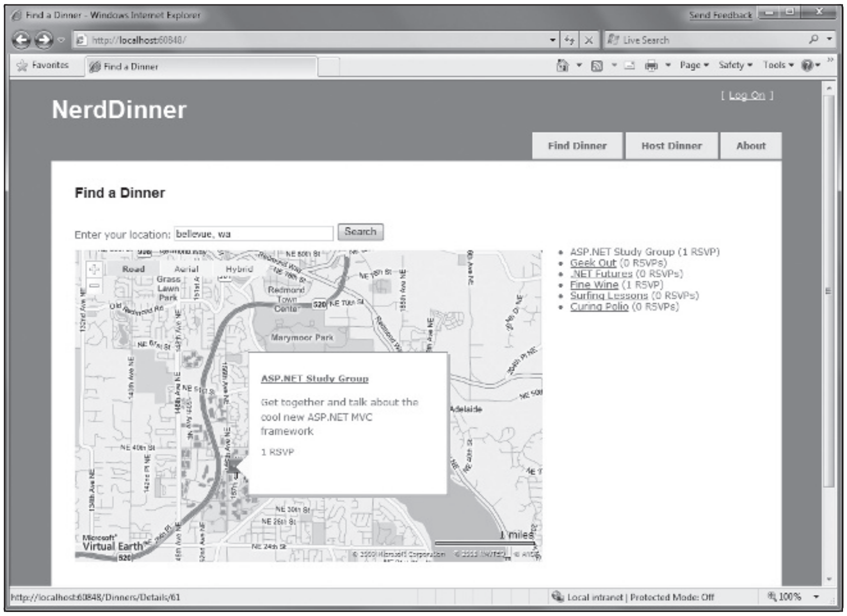
Country: UK

Contact Phone #: 425-703-8072

Save

Figure 1-2

Visitors to the site can search to find upcoming dinners being held near them (Figure 1-3):



NerdDinner

[Log On]

Find Dinner | Host Dinner | About

Find a Dinner

Enter your location: bellevue, wa Search

- ASP.NET Study Group (1 RSVP)
- Geek Out (0 RSVPs)
- .NET Futures (0 RSVPs)
- Fine Wine (1 RSVP)
- Surfing Lessons (0 RSVPs)
- Curing Polio (0 RSVPs)

ASP.NET Study Group

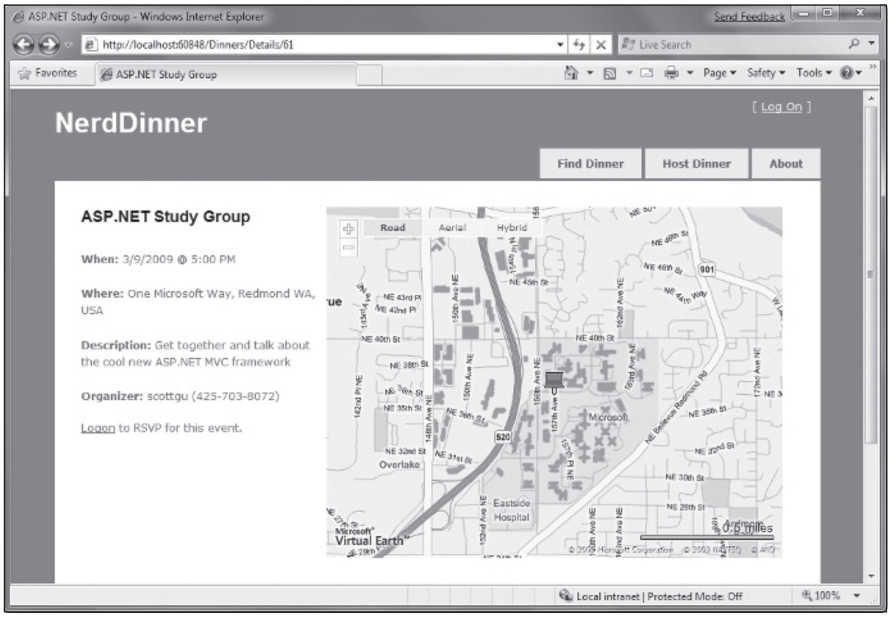
Get together and talk about the cool new ASP.NET MVC framework

1 RSVP

http://localhost:60848/Dinners/Details/61

Figure 1-3

Clicking a dinner will take them to a details page where they can learn more about it (Figure 1-4):



ASP.NET Study Group

When: 3/9/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

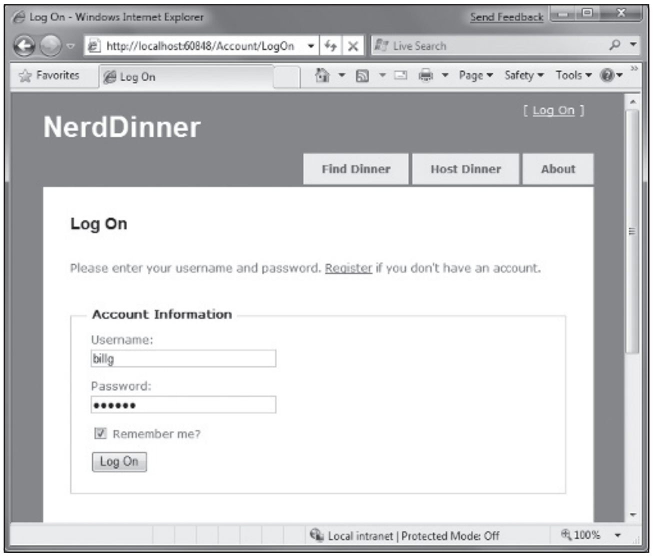
Description: Get together and talk about the cool new ASP.NET MVC framework

Organizer: scottgu (425-703-8072)

Logon to RSVP for this event.

Figure 1-4

If they are interested in attending the dinner they can log in or register on the site (Figure 1-5):



Log On

Please enter your username and password. Register if you don't have an account.

Account Information

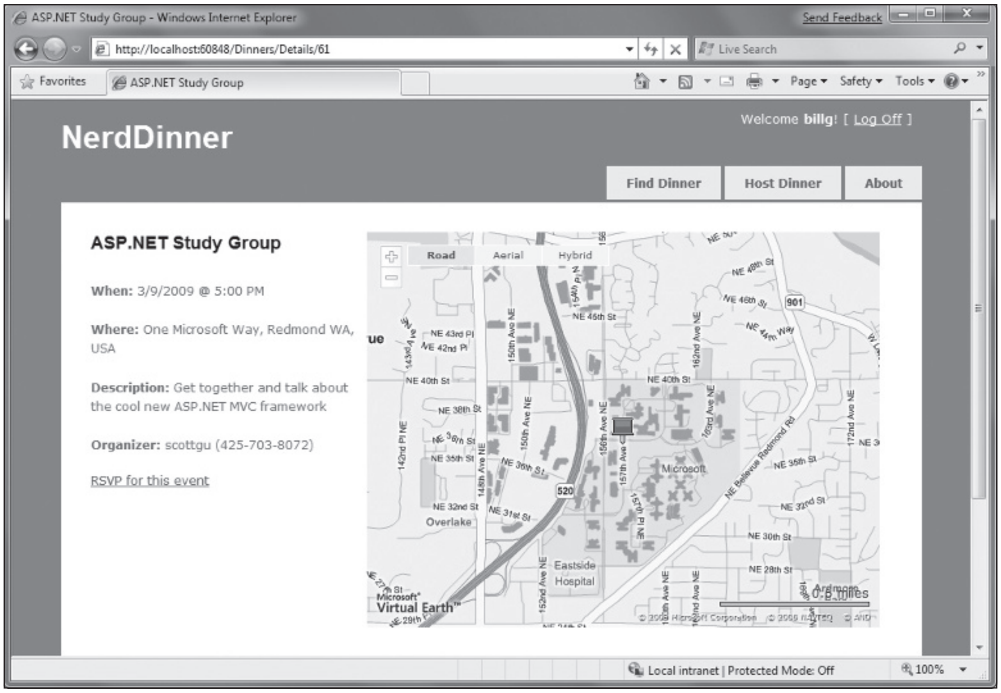
Username: billg

Password: ••••••

☑ Remember me?

Figure 1-5

They can then easily RSVP to attend the event (Figures 1-6 and 1-7):



ASP.NET Study Group - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/61
Live Search
Favorites
ASP.NET Study Group
Page
Safety
Tools
Welcome billg! [Log Off]
NerdDinner
Find Dinner
Host Dinner
About
Road
Aerial
Hybrid
Microsoft
Overlake
Eastside Hospital
Microsoft® Virtual Earth™
Local intranet | Protected Mode: Off
100%

ASP.NET Study Group

When: 3/9/2009 @ 5:00 PM

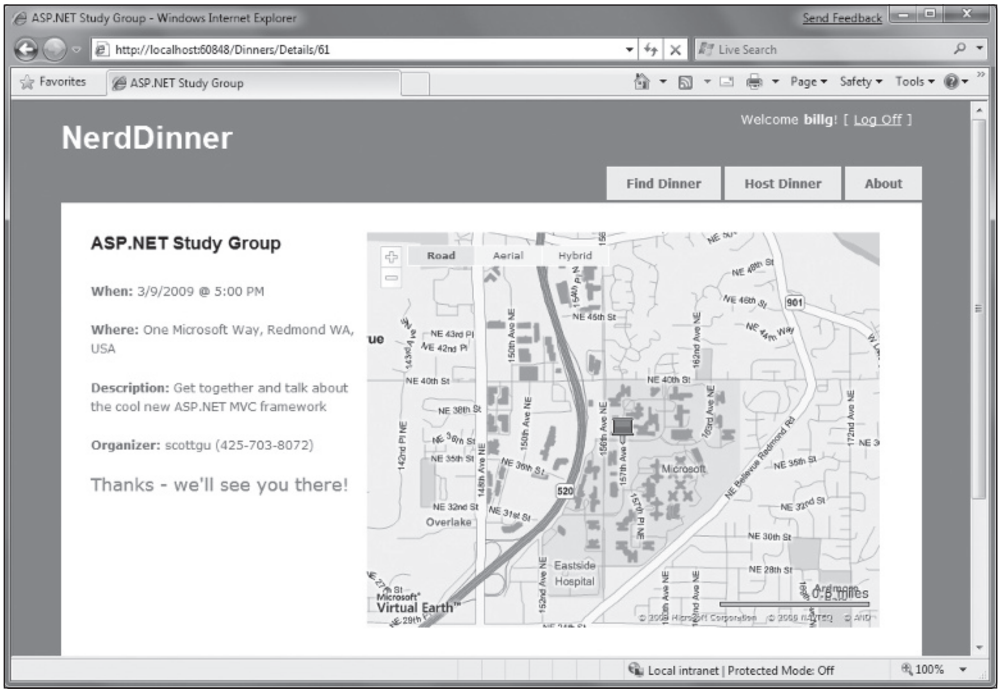
Where: One Microsoft Way, Redmond WA, USA

Description: Get together and talk about the cool new ASP.NET MVC framework

Organizer: scottgu (425-703-8072)

RSVP for this event

Figure 1-6



ASP.NET Study Group - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/61
Live Search
Favorites
ASP.NET Study Group
Page
Safety
Tools
Welcome billg! [Log Off]
NerdDinner
Find Dinner
Host Dinner
About
Road
Aerial
Hybrid
Microsoft
Overlake
Eastside Hospital
Microsoft® Virtual Earth™
Local intranet | Protected Mode: Off
100%

ASP.NET Study Group

When: 3/9/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

Description: Get together and talk about the cool new ASP.NET MVC framework

Organizer: scottgu (425-703-8072)

Thanks - we'll see you there!

Figure 1-7

We are going to begin implementing the NerdDinner application by using the File ➪ New Project command within Visual Studio to create a brand new ASP.NET MVC project. We'll then incrementally add functionality and features. Along the way we'll cover how to create a database, build a model with business rule validations, implement data listing/details UI, provide CRUD (Create, Update, Delete) form entry support, implement efficient data paging, reuse the UI using master pages and partials, secure the application using authentication and authorization, use AJAX to deliver dynamic updates and interactive map support, and implement automated unit testing.

You can build your own copy of NerdDinner from scratch by completing each step we walk through in this chapter. Alternatively, you can download a completed version of the source code here: `http://tinyurl.com/aspnetmvc`.

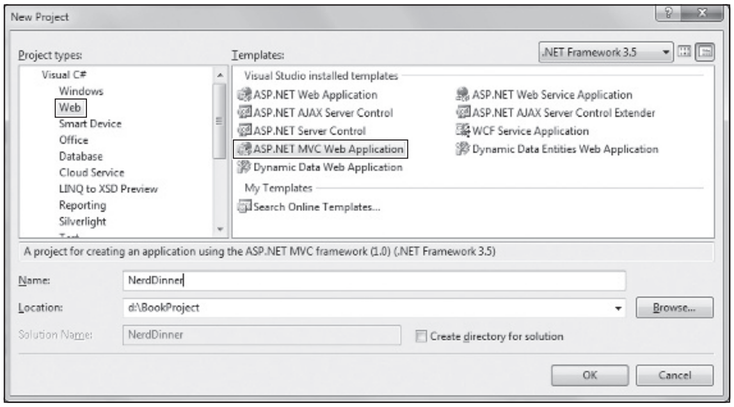
You can use either Visual Studio 2008 or the free Visual Web Developer 2008 Express to build the application. You can use either SQL Server or the free SQL Server Express to host the database.

You can install ASP.NET MVC, Visual Web Developer 2008, and SQL Server Express using the Microsoft Web Platform Installer available at `www.microsoft.com/web/downloads`.

File ➪ New Project

We'll begin our NerdDinner application by selecting the File ➪ New Project menu item within Visual Studio 2008 or the free Visual Web Developer 2008 Express.

This will bring up the New Project dialog. To create a new ASP.NET MVC application, we'll select the Web node on the left side of the dialog and then choose the ASP.NET MVC Web Application project template on the right (Figure 1-8):



A project for creating an application using the ASP.NET MVC framework (1.0) (.NET Framework 3.5)

Name: NerdDinner

Location: d:\BookProject

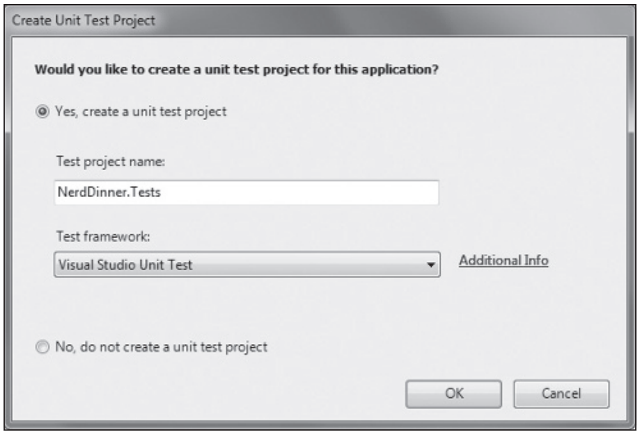
Solution Name: NerdDinner

☐ Create directory for solution

Figure 1-8

We'll name the new project **NerdDinner** and then click the OK button to create it.

When we click OK, Visual Studio will bring up an additional dialog that prompts us to optionally create a unit test project for the new application as well (Figure 1-9). This unit test project enables us to create automated tests that verify the functionality and behavior of our application (something we'll cover later in this tutorial).



Create Unit Test Project

Would you like to create a unit test project for this application?

● Yes, create a unit test project

Test project name: NerdDinner.Tests

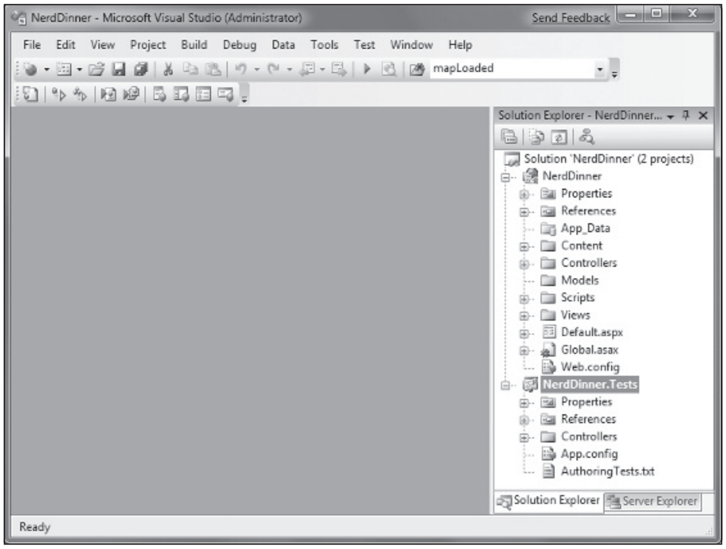
Test framework: Visual Studio Unit Test

Figure 1-9

The Test framework drop-down in Figure 1-9 is populated with all available ASP.NET MVC unit test project templates installed on the machine. Versions can be downloaded for NUnit, MBUnit, and XUnit. The built-in Visual Studio Unit Test Framework is also supported.

The Visual Studio Unit Test Framework is only available with Visual Studio 2008 Professional and higher versions). If you are using VS 2008 Standard Edition or Visual Web Developer 2008 Express, you will need to download and install the NUnit, MBUnit, or XUnit extensions for ASP.NET MVC in order for this dialog to be shown. The dialog will not display if there aren't any test frameworks installed.

We'll use the default `NerdDinner.Tests` name for the test project we create, and use the Visual Studio Unit Test Framework option. When we click the OK button, Visual Studio will create a solution for us with two projects in it — one for our web application and one for our unit tests (Figure 1-10):



NerdDinner - Microsoft Visual Studio (Administrator)
Send Feedback
File Edit View Project Build Debug Data Tools Test Window Help
mapLoaded
Solution Explorer - NerdDinner...
Solution Explorer
Server Explorer
Ready

```
Solution 'NerdDinner' (2 projects)
  NerdDinner
    Properties
    References
    App_Data
    Content
    Controllers
    Models
    Scripts
    Views
    Default.aspx
    Global.asax
    Web.config
  NerdDinner.Tests
    Properties
    References
    Controllers
    App.config
    AuthoringTests.txt
```

Figure 1-10

Examining the NerdDinner Directory Structure

When you create a new ASP.NET MVC application with Visual Studio, it automatically adds a number of files and directories to the project, as shown in Figure 1-11.

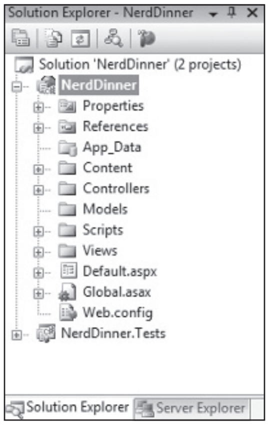


Figure 1-11

ASP.NET MVC projects by default have six top-level directories, shown in the following table:

Directory	Purpose
`/Controllers`	Where you put Controller classes that handle URL requests
`/Models`	Where you put classes that represent and manipulate data
`/Views`	Where you put UI template files that are responsible for rendering output
`/Scripts`	Where you put JavaScript library files and scripts (.js)
`/Content`	Where you put CSS and image files, and other non-dynamic/non-JavaScript content
`/App_Data`	Where you store data files you want to read/write.

ASP.NET MVC does not require this structure. In fact, developers working on large applications will typically partition the application up across multiple projects to make it more manageable (for example: data model classes often go in a separate class library project from the web application). The default project structure, however, does provide a nice default directory convention that we can use to keep our application concerns clean.

When we expand the `/Controllers` directory, we'll find that Visual Studio added two controller classes (Figure 1-12) — `HomeController` and `AccountController` — by default to the project:

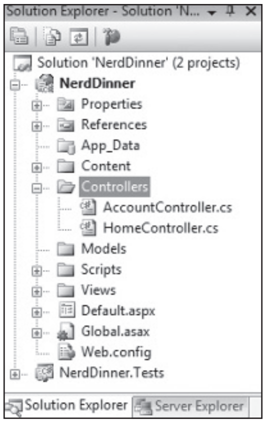


Figure 1-12

When we expand the /Views directory, we'll find three subdirectories — /Home, /Account and /Shared — as well as several template files within them, were also added to the project by default (Figure 1-13):

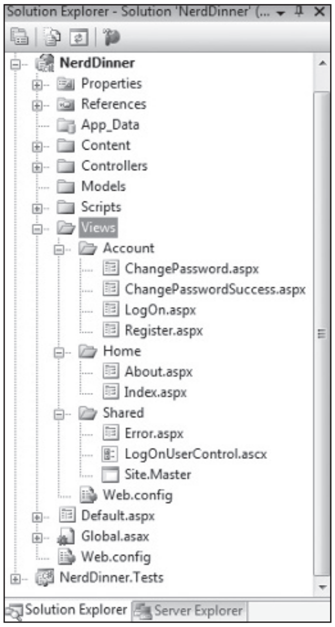
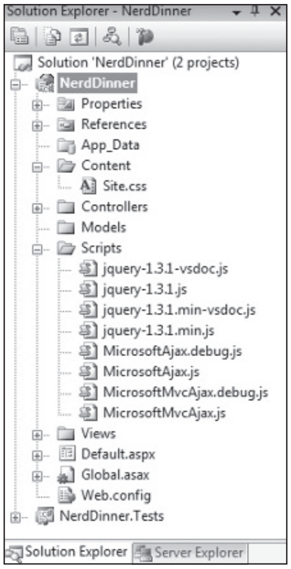


Figure 1-13

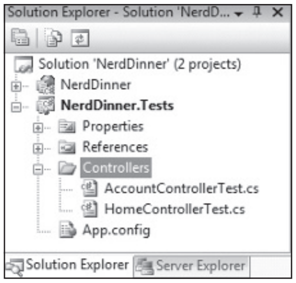
When we expand the /Content and /Scripts directories, we'll find a Site.css file that is used to style all HTML on the site, as well as JavaScript libraries that can enable ASP.NET AJAX and jQuery support within the application (Figure 1-14):



```
Solution 'NerdDinner' (2 projects)
  NerdDinner
    Properties
    References
    App_Data
    Content
      Site.css
    Controllers
    Models
    Scripts
      jquery-1.3.1-vsdoc.js
      jquery-1.3.1.js
      jquery-1.3.1.min-vsdoc.js
      jquery-1.3.1.min.js
      MicrosoftAjax.debug.js
      MicrosoftAjax.js
      MicrosoftMvcAjax.debug.js
      MicrosoftMvcAjax.js
    Views
    Default.aspx
    Global.asax
    Web.config
  NerdDinner.Tests
```

Figure 1-14

When we expand the `NerdDinner.Tests` project we'll find two classes that contain unit tests for our controller classes (Figure 1-15):



```
Solution 'NerdDinner' (2 projects)
  NerdDinner
  NerdDinner.Tests
    Properties
    References
    Controllers
      AccountControllerTest.cs
      HomeControllerTest.cs
    App.config
```

Figure 1-15

These default files, added by Visual Studio, provide us with a basic structure for a working application — complete with home page, about page, account login/logout/registration pages, and an unhandled error page (all wired-up and working out of the box).

Running the NerdDinner Application

We can run the project by choosing either the Debug ➪ Start Debugging or Debug ➪ Start Without Debugging menu items (Figure 1-16):

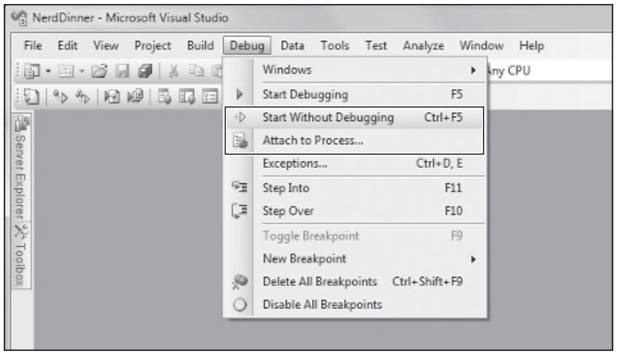


Figure 1-16

This will launch the built-in ASP.NET web server that comes with Visual Studio, and run our application (Figure 1-17):

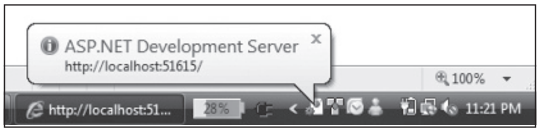
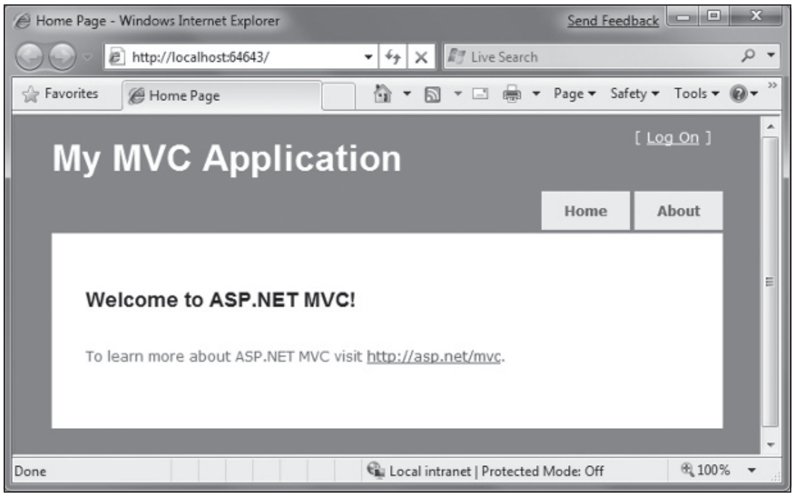

ASP.NET Development Server
http://localhost:51615/

Figure 1-17

FIgure 1-18 is the home page for our new project (URL: `/`) when it runs:



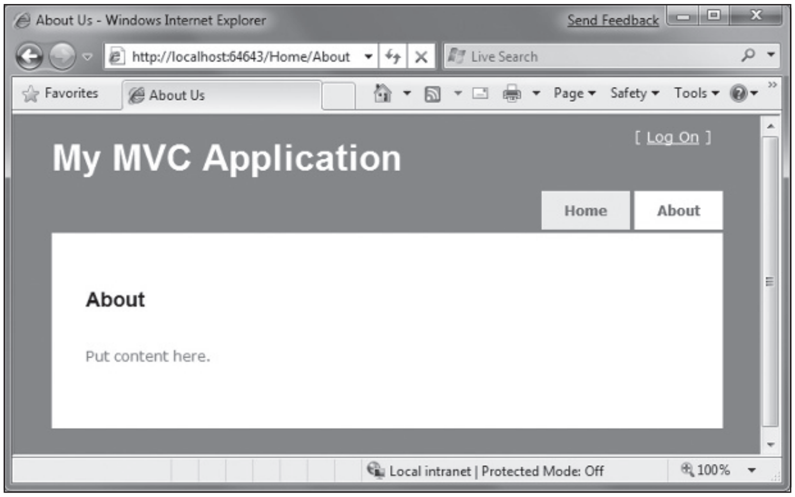
My MVC Application

Welcome to ASP.NET MVC!

To learn more about ASP.NET MVC visit http://asp.net/mvc.

Figure 1-18

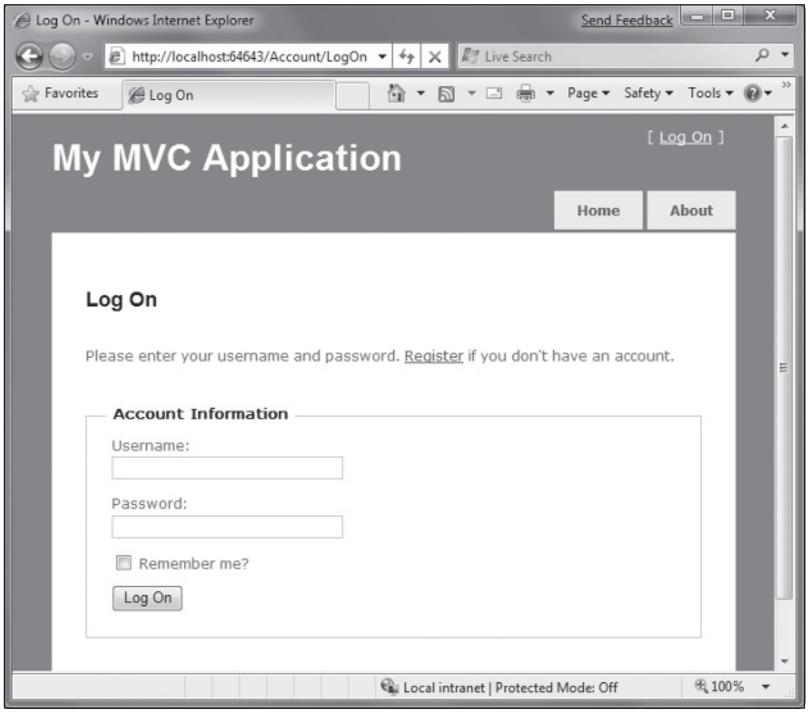
Clicking the About tab displays an About page (URL: `/Home/About`, shown in Figure 1-19):


About Us - Windows Internet Explorer
Send Feedback
http://localhost:64643/Home/About
Live Search
Favorites
About Us
Page
Safety
Tools
[Log On]
My MVC Application
Home
About
About
Local intranet | Protected Mode: Off
100%

Put content here.

Figure 1-19

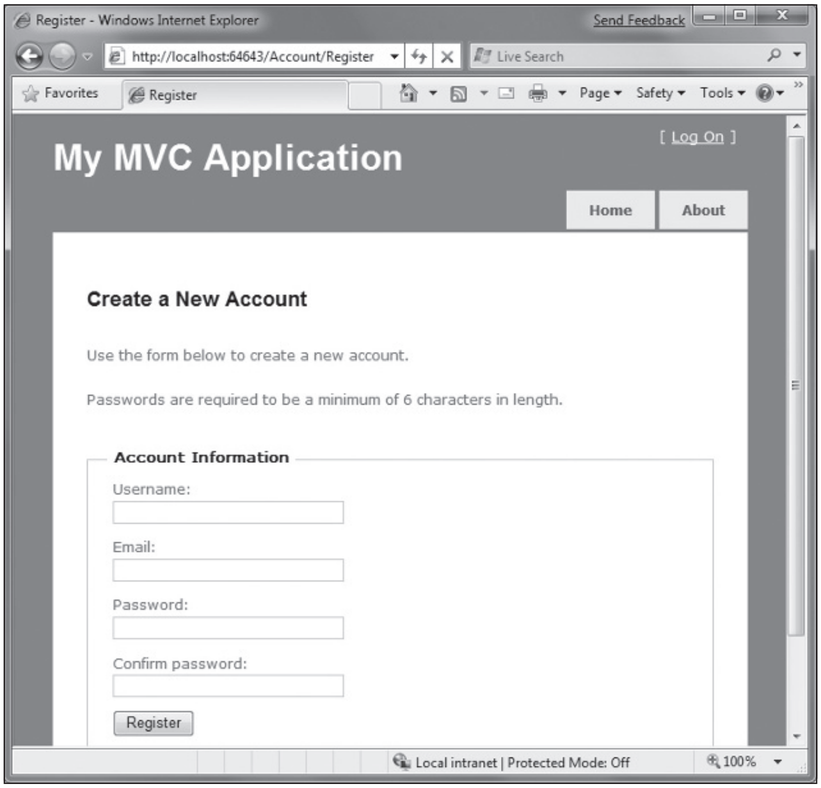
Clicking the Log On link on the top right takes us to a Login page shown in Figure 1-20 (URL: `/Account/LogOn`)



Please enter your username and password. Register if you don't have an account.

Figure 1-20

If we don't have a login account, we can click the Register link (URL: `/Account/Register`) to create one (Figure 1-21):


Register - Windows Internet Explorer
Send Feedback
http://localhost:64643/Account/Register
Live Search
Favorites
Register
Page ▾
Safety ▾
Tools ▾
My MVC Application
[Log On]
Home
About
Register
Local intranet | Protected Mode: Off
100%

Create a New Account

Use the form below to create a new account.

Passwords are required to be a minimum of 6 characters in length.

Account Information

Username:

Email:

Password:

Confirm password:

Figure 1-21

The code to implement the above home, about, and login/register functionality was added by default when we created our new project. We'll use it as the starting point of our application.

Testing the NerdDinner Application

If we are using the Professional Edition or higher version of Visual Studio 2008, we can use the built-in unit-testing IDE support within Visual Studio to test the project.

Choosing one of the above options in Figure 1-22 will open the Test Results pane within the IDE (Figure 1-23) and provide us with pass/fail status on the 27 unit tests included in our new project that cover the built-in functionality.

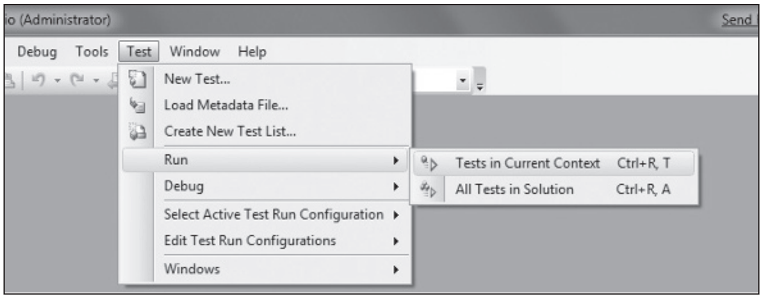
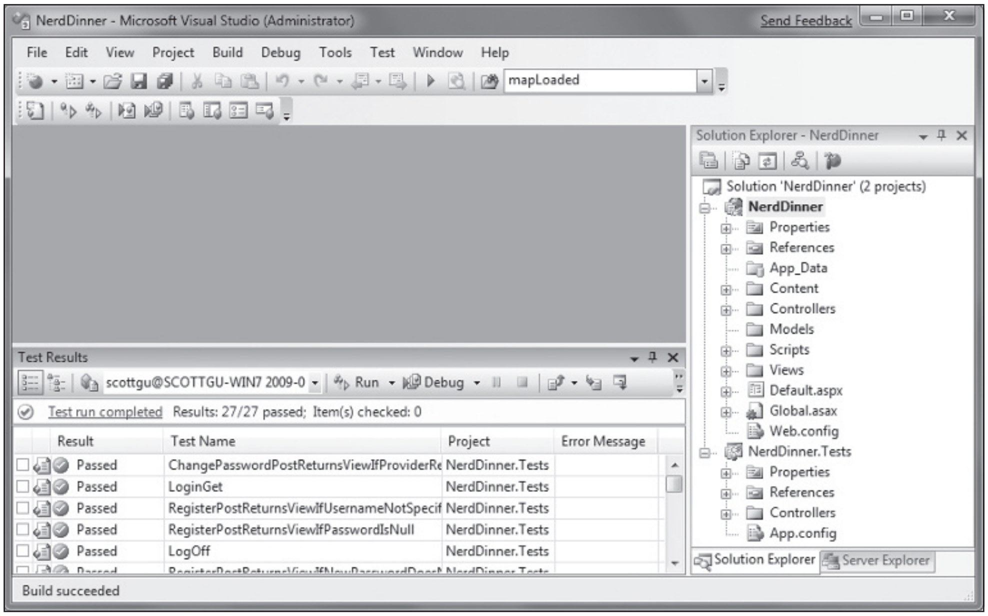

o (Administrator)
Send
Debug
Tools
Test
Window
Help
New Test...
Load Metadata File...
Create New Test List...
Run
Debug
Select Active Test Run Configuration
Edit Test Run Configurations
Windows
Tests in Current Context Ctrl+R, T
All Tests in Solution Ctrl+R, A

Figure 1-22



Test run completed Results: 27/27 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	ChangePasswordPostReturnsViewIfProviderRe	NerdDinner.Tests	
Passed	LoginGet	NerdDinner.Tests	
Passed	RegisterPostReturnsViewIfUsernameNotSpecif	NerdDinner.Tests	
Passed	RegisterPostReturnsViewIfPasswordIsNull	NerdDinner.Tests	
Passed	LogOff	NerdDinner.Tests	

Figure 1-23

Creating the Database

We'll be using a database to store all of the Dinner and RSVP data for our NerdDinner application.

The steps below show creating the database using the free SQL Server Express edition. All of the code we'll write works with both SQL Server Express and the full SQL Server.

Creating a New SQL Server Express Database

We'll begin by right-clicking on our web project, and then selecting the Add ➪ New Item menu command (Figure 1-24).

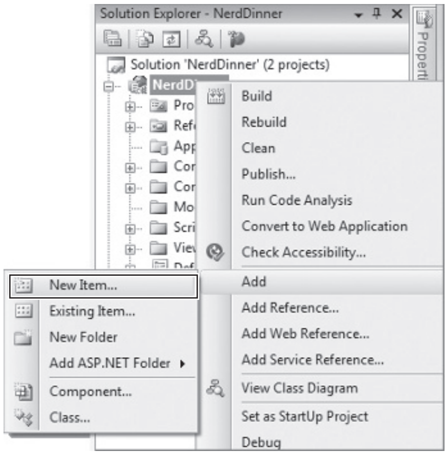
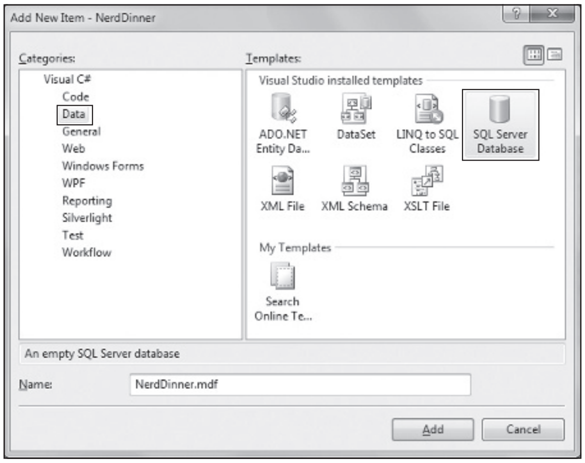


Figure 1-24

This will bring up the Add New Item dialog (Figure 1-25). We'll filter by the Data category and select the SQL Server Database item template.



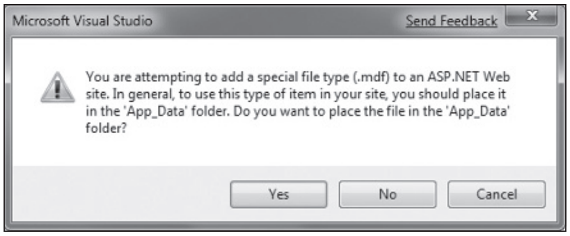
An empty SQL Server database

Name: NerdDinner.mdf

Figure 1-25

We'll name the SQL Server Express database we want to create NerdDinner.mdf and hit OK. Visual Studio will then ask us if we want to add this file to our `\App_Data` directory (Figure 1-26), which is a directory already set up with both read and write security ACLs.

We'll click Yes and our new database will be created and added to our Solution Explorer (Figure 1-27).



Microsoft Visual Studio

You are attempting to add a special file type (.mdf) to an ASP.NET Web site. In general, to use this type of item in your site, you should place it in the 'App_Data' folder. Do you want to place the file in the 'App_Data' folder?

Figure 1-26

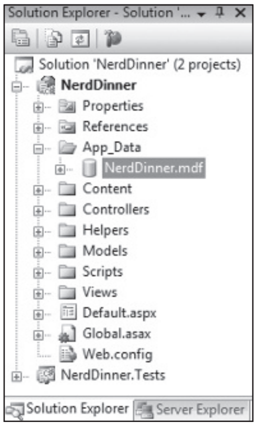


Figure 1-27

Creating Tables within Our Database

We now have a new empty database. Let's add some tables to it.

To do this we'll navigate to the Server Explorer tab window within Visual Studio, which enables us to manage databases and servers. SQL Server Express databases stored in the `\App_Data` folder of our application will automatically show up within the Server Explorer. We can optionally use the Connect to Database icon on the top of the Server Explorer window to add additional SQL Server databases (both local and remote) to the list as well (Figure 1-28).

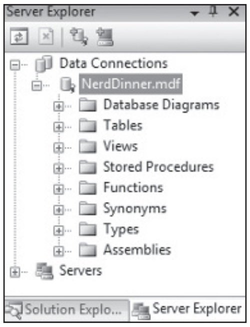


Figure 1-28

We will add two tables to our NerdDinner database — one to store our Dinners, and the other to track RSVP acceptances to them. We can create new tables by right-clicking on the Tables folder within our database and choosing the Add New Table menu command (Figure 1-29).

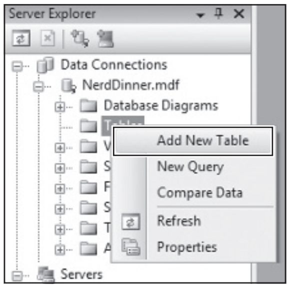
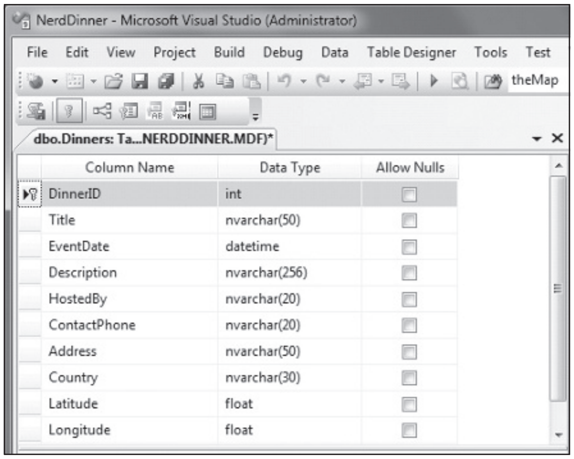


Figure 1-29

This will open up a table designer that allows us to configure the schema of our table. For our Dinners table, we will add 10 columns of data (Figure 1-30).


NerdDinner - Microsoft Visual Studio (Administrator)
File
Edit
View
Project
Build
Debug
Data
Table Designer
Tools
Test
theMap
dbo.Dinners: Ta...NERDDINNER.MDF)*

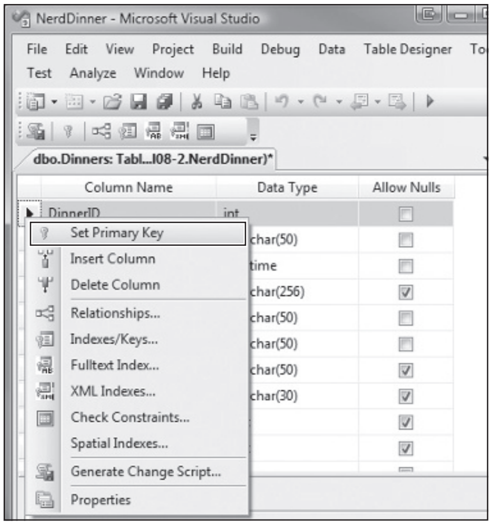
Column Name	Data Type	Allow Nulls
DinnerID	int	□
Title	nvarchar(50)	□
EventDate	datetime	□
Description	nvarchar(256)	□
HostedBy	nvarchar(20)	□
ContactPhone	nvarchar(20)	□
Address	nvarchar(50)	□
Country	nvarchar(30)	□
Latitude	float	□
Longitude	float	□

Figure 1-30

We want the DinnerID column to be a unique primary key for the table. We can configure this by right-clicking on the DinnerID column and choosing the Set Primary Key menu item (Figure 1-31).

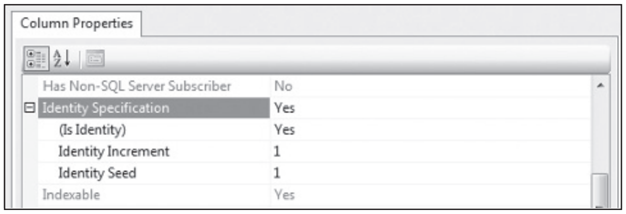
In addition to making DinnerID a primary key, we also want configure it as an *identity* column whose value is automatically incremented as new rows of data are added to the table (meaning the first inserted Dinner row will have a DinnerID of 1, the second inserted row will have a DinnerID of 2, etc.).

We can do this by selecting the DinnerID column and then using the Column Properties editor to set the "(Is Identity)" property on the column to Yes (Figure 1-32). We will use the standard identity defaults (start at 1 and increment 1 on each new Dinner row).



Column Name	Data Type	Allow Nulls
DinnerID	int	☐
	char(50)	☐
	time	☐
	char(256)	☑
	char(50)	☐
	char(50)	☐
	char(50)	☑
	char(30)	☑
		☑
		☑

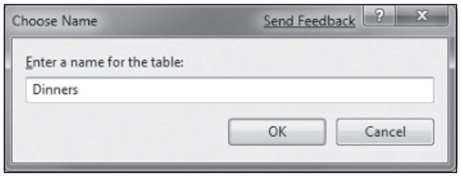
Figure 1-31



Has Non-SQL Server Subscriber	No
⊟ Identity Specification	Yes
(Is Identity)	Yes
Identity Increment	1
Identity Seed	1
Indexable	Yes

Figure 1-32

We'll then save our table by pressing Ctrl-S or by clicking the File ➪ Save menu command. This will prompt us to name the table. We'll name it Dinners (Figure 1-33).



Enter a name for the table:

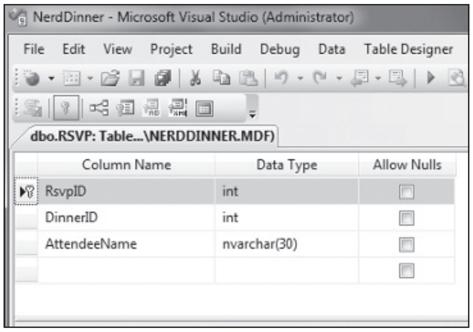
Dinners

Figure 1-33

Our new Dinners table will then show up in our database in the Server Explorer.

We'll then repeat the above steps and create a RSVP table. This table will have three columns. We will set up the RsvpID column as the primary key, and also make it an identity column (Figure 1-34).

We'll save it and give it the name **RSVP**.


NerdDinner - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Table Designer
dbo.RSVP: Table...\NERDDINNER.MDF)

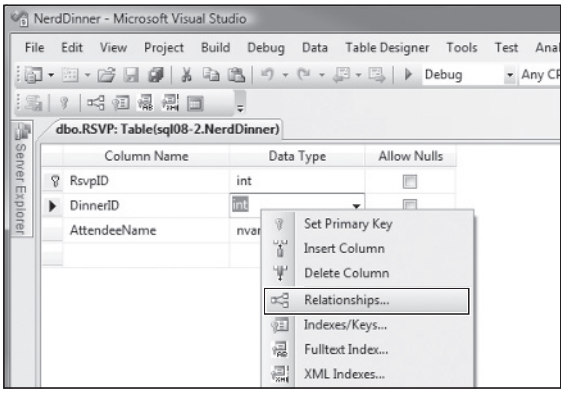
Column Name	Data Type	Allow Nulls
RsvpID	int	□
DinnerID	int	□
AttendeeName	nvarchar(30)	□
		□

Figure 1-34

Setting Up a Foreign Key Relationship Between Tables

We now have two tables within our database. Our last schema design step will be to set up a "one-to-many" relationship between these two tables — so that we can associate each Dinner row with zero or more RSVP rows that apply to it. We will do this by configuring the RSVP table's DinnerID column to have a foreign-key relationship to the DinnerID column in the Dinners table.

To do this we'll open up the RSVP table within the table designer by double-clicking it in the Server Explorer. We'll then select the DinnerID column within it, right-click, and choose the Relationships… context menu command (Figure 1-35):


NerdDinner - Microsoft Visual Studio
File Edit View Project Build Debug Data Table Designer Tools Test Ana
Debug
Any CP
Server Explorer
dbo.RSVP: Table(sql08-2.NerdDinner)
Set Primary Key
Insert Column
Delete Column
Relationships...
Indexes/Keys...
Fulltext Index...
XML Indexes...

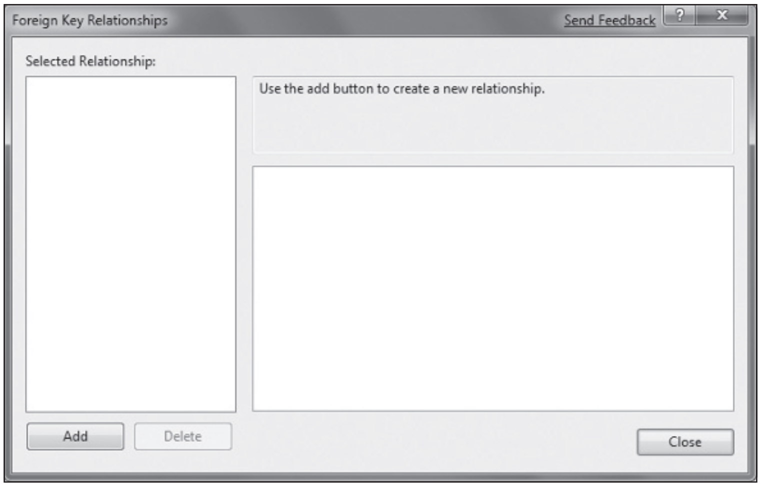
Column Name	Data Type	Allow Nulls
RsvpID	int	□
DinnerID	int	□
AttendeeName	nvar	

Figure 1-35

This will bring up a dialog that we can use to set up relationships between tables (Figure 1-36).

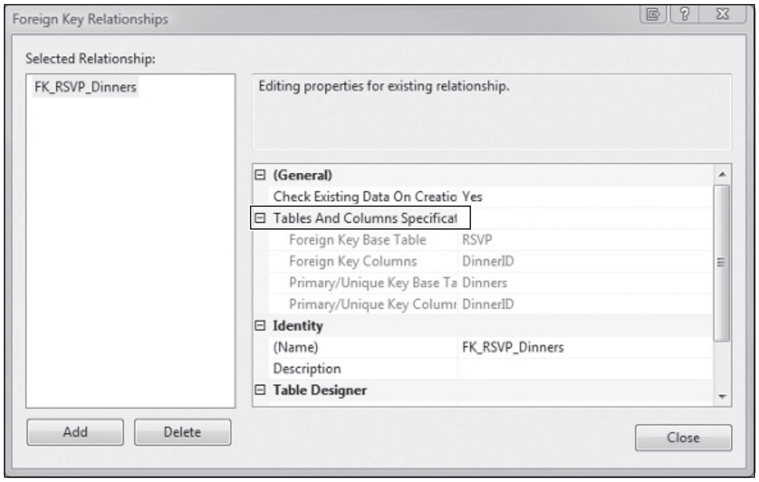
We'll click the Add button to add a new relationship to the dialog. Once a relationship has been added, we'll expand the Tables and Column Specification tree-view node within the property grid to the right of the dialog, and then click the "…" button to the right of it (Figure 1-37).

Clicking the "…" button will bring up another dialog that allows us to specify which tables and columns are involved in the relationship, as well as allow us to name the relationship.



Use the add button to create a new relationship.

Figure 1-36



FK_RSVP_Dinners

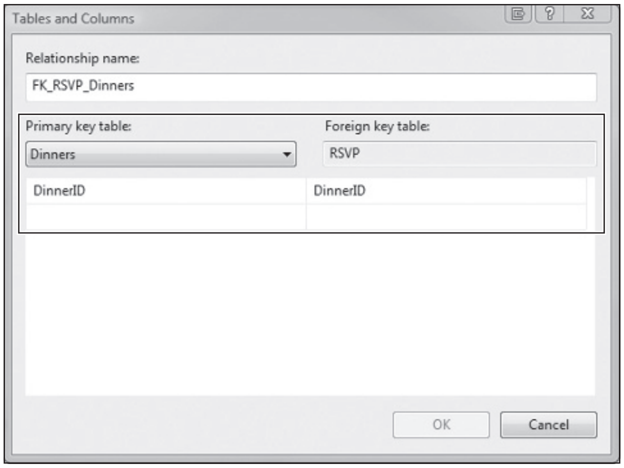
Editing properties for existing relationship.

⊟ (General)	
Check Existing Data On Creatio	Yes
⊟ Tables And Columns Specificat	
Foreign Key Base Table	RSVP
Foreign Key Columns	DinnerID
Primary/Unique Key Base Ta	Dinners
Primary/Unique Key Columi	DinnerID
⊟ Identity	
(Name)	FK_RSVP_Dinners
Description	
⊟ Table Designer	

Figure 1-37

We will change the Primary Key Table to be Dinners, and select the DinnerID column within the Dinners table as the primary key. Our RSVP table will be the foreign-key table, and the RSVP.DinnerID column will be associated as the foreign-key (Figure 1-38).

Now each row in the RSVP table will be associated with a row in the Dinner table. SQL Server will maintain referential integrity for us — and prevent us from adding a new RSVP row if it does not point to a valid Dinner row. It will also prevent us from deleting a Dinner row if there are still RSVP rows referring to it.



Tables and Columns

Relationship name: FK_RSVP_Dinners

Primary key table: Dinners

Foreign key table: RSVP

Primary key table	Foreign key table
DinnerID	DinnerID

Figure 1-38

Adding Data to Our Tables

Let's finish by adding some sample data to our Dinners table. We can add data to a table by right-clicking on it in the Server Explorer and choosing the Show Table Data command (Figure 1-39):

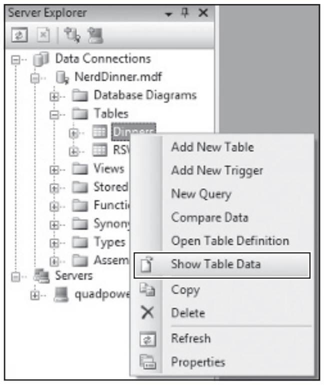
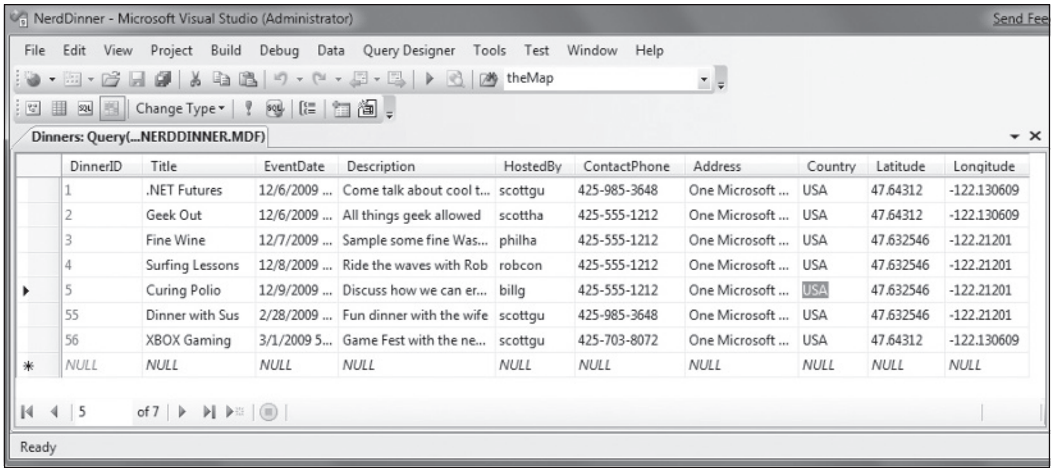


Figure 1-39

Let's add a few rows of Dinner data that we can use later as we start implementing the application (Figure 1-40).

Building the Model

In a Model-View-Controller framework the term *Model* refers to the objects that represent the data of the application, as well as the corresponding domain logic that integrates validation and business rules with it. The Model is in many ways the "heart" of an MVC-based application, and as we'll see later, it fundamentally drives the behavior of the application.


NerdDinner - Microsoft Visual Studio (Administrator)
Send Fee
File Edit View Project Build Debug Data Query Designer Tools Test Window Help
theMap
Change Type
Dinners: Query(...NERDDINNER.MDF)

	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

Figure 1-40

The ASP.NET MVC framework supports using any data access technology. Developers can choose from a variety of rich .NET data options to implement their models including: LINQ to Entities, LINQ to SQL, NHibernate, LLBLGen Pro, SubSonic, WilsonORM, or just raw ADO.NET DataReaders or DataSets.

For our NerdDinner application, we are going to use LINQ to SQL to create a simple domain model that corresponds fairly closely to our database design, and add some custom validation logic and business rules. We will then implement a repository class that helps abstract away the data persistence implementation from the rest of the application, and enables us to easily unit test it.

LINQ to SQL

LINQ to SQL is an ORM (object relational mapper) that ships as part of .NET 3.5.

LINQ to SQL provides an easy way to map database tables to .NET classes we can code against. For our NerdDinner application, we'll use it to map the Dinners and RSVP tables within our database to `Dinner` and `RSVP` model classes. The columns of the Dinners and RSVP tables will correspond to properties on the `Dinner` and `RSVP` classes. Each Dinner and RSVP object will represent a separate row within the Dinners or RSVP tables in the database.

LINQ to SQL allows us to avoid having to manually construct SQL statements to retrieve and update Dinner and RSVP objects with database data. Instead, we'll define the Dinner and RSVP classes, how they map to/from the database, and the relationships between them. LINQ to SQL will then take care of generating the appropriate SQL execution logic to use at runtime when we interact and use them.

We can use the LINQ language support within VB and C# to write expressive queries that retrieve Dinner and RSVP objects. This minimizes the amount of data code we need to write, and allows us to build really clean applications.

Adding LINQ to SQL Classes to Our Project

We'll begin by right-clicking on the Models folder in our project, and select the Add ➪ New Item menu command (Figure 1-41).

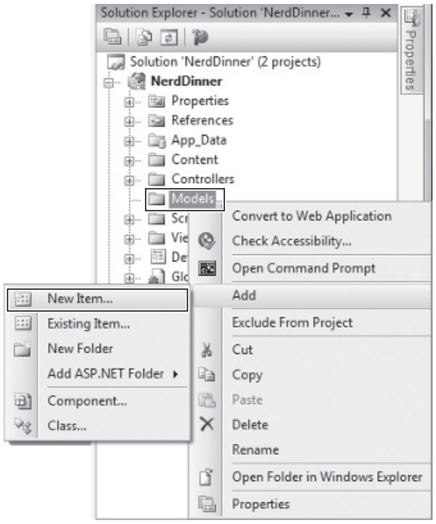
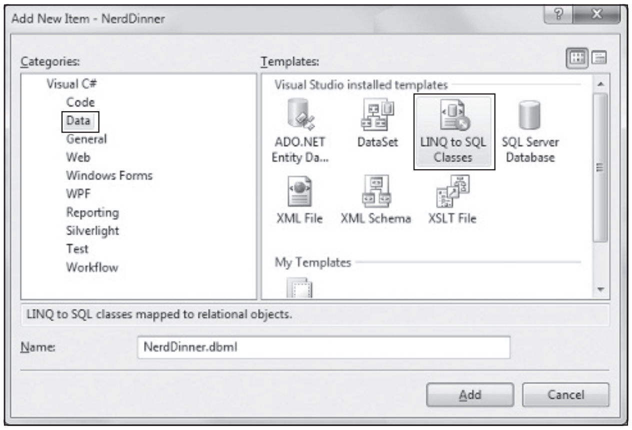


Figure 1-41

This will bring up the Add New Item dialog (Figure 1-42). We'll filter by the Data category and select the LINQ to SQL Classes template within it.



LINQ to SQL classes mapped to relational objects.

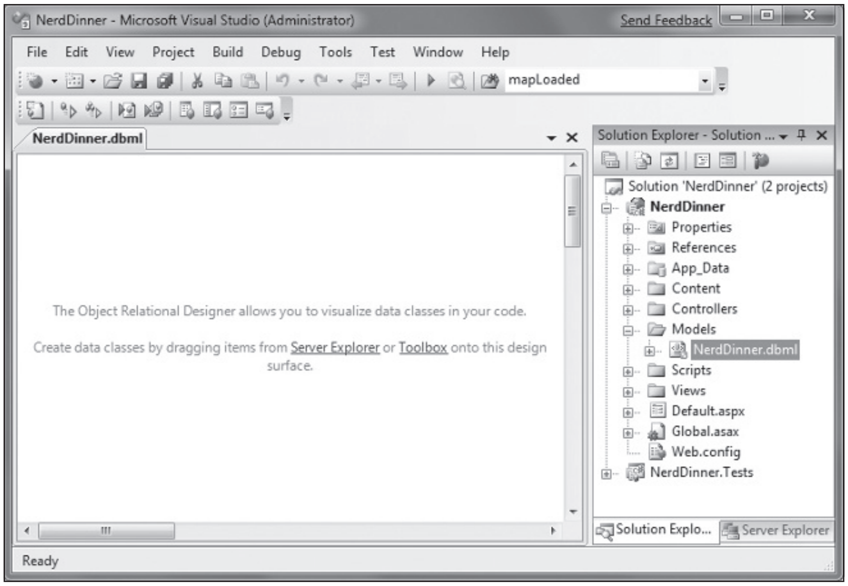
Name: NerdDinner.dbml

Figure 1-42

We'll name the item **NerdDinner** and click the Add button. Visual Studio will add a `NerdDinner.dbml` file under our `\Models` directory, and then open the LINQ to SQL object relational designer (Figure 1-43).

Creating Data Model Classes with LINQ to SQL

LINQ to SQL enables us to quickly create data model classes from an existing database schema. To do this we'll open the NerdDinner database in the Server Explorer, and select the Tables we want to model in it (Figure 1-44).



The Object Relational Designer allows you to visualize data classes in your code.

Create data classes by dragging items from Server Explorer or Toolbox onto this design surface.

Figure 1-43

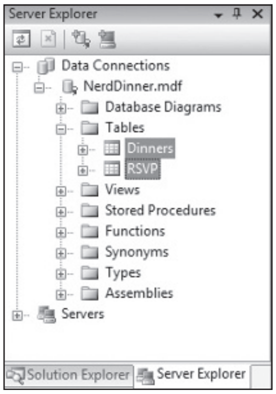


Figure 1-44

We can then drag the tables onto the LINQ to SQL designer surface. When we do this, LINQ to SQL will automatically create Dinner and RSVP classes using the schema of the tables (with class properties that map to the database table columns as shown in Figure 1-45).

By default the LINQ to SQL designer automatically *pluralizes* table and column names when it creates classes based on a database schema. For example: the "Dinners" table in our example above resulted in a `Dinner` class. This class naming helps make our models consistent with .NET naming conventions, and I usually find that having the designer fix this up is convenient (especially when adding lots of tables). If you don't like the name of a class or property that the designer generates, though, you can always override it and change it to any name you want. You can do this either by editing the entity/property name in-line within the designer or by modifying it via the property grid.

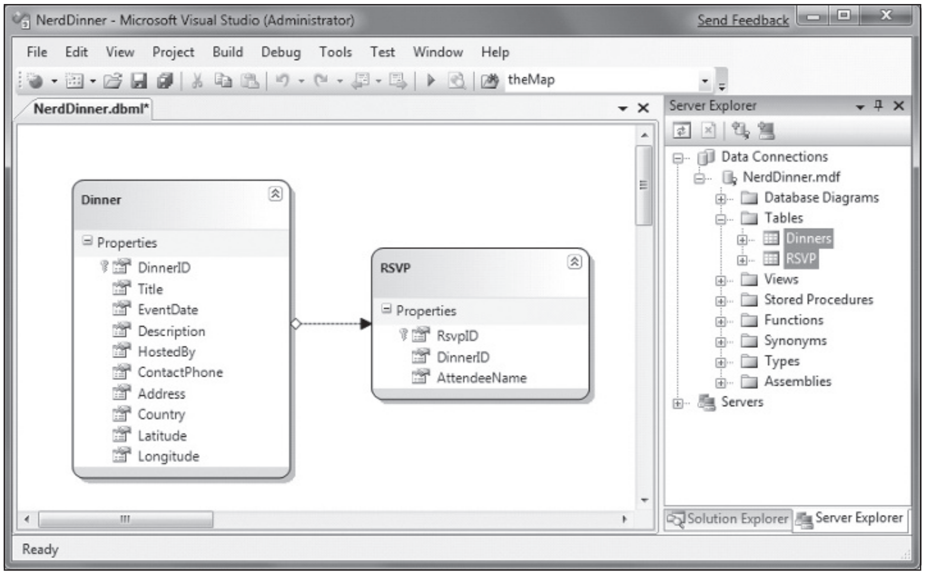


Figure 1-45

By default the LINQ to SQL designer also inspects the primary key/foreign key relationships of the tables, and based on them automatically creates default *relationship associations* between the different model classes it creates. For example, when we modeled the Dinners and RSVP tables onto the LINQ to SQL designer, a one-to-many relationship association between the two was inferred based on the fact that the RSVP table had a foreign key to the Dinners table (this is indicated by the arrow in the designer in Figure 1-46).

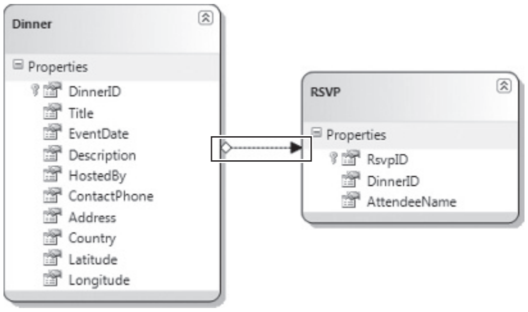
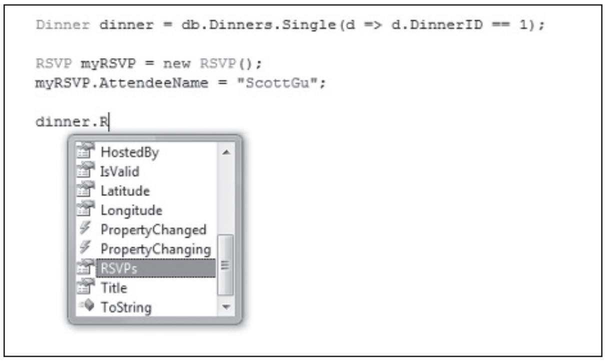


Figure 1-46

The association in Figure 1-46 will cause LINQ to SQL to add a strongly typed `Dinner` property to the RSVP class that developers can use to access the Dinner entity associated with a given RSVP. It will also cause the `Dinner` class to have a strongly typed `RSVPs` collection property that enables developers to retrieve and update `RSVP` objects associated with that Dinner.

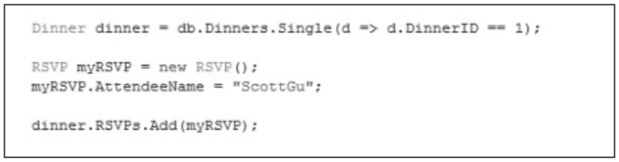
In Figure 1-47, you can see an example of IntelliSense within Visual Studio when we create a new `RSVP` object and add it to a Dinner's RSVPs collection.



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

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

dinner.R
```

Figure 1-47

Notice how LINQ to SQL created a "RSVPs" collection on the `Dinner` object. We can use this to associate a foreign-key relationship between a Dinner and a RSVP row in our database (Figure 1-48):



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

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

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

Figure 1-48

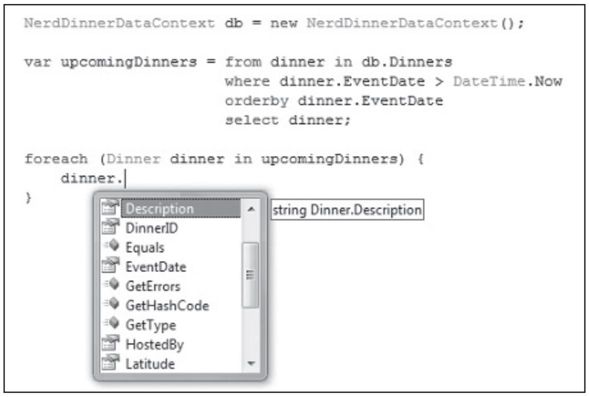
If you don't like how the designer has modeled or named a table association, you can override it. Just click on the association arrow within the designer and access its properties via the property grid to rename, delete, or modify it. For our NerdDinner application, though, the default association rules work well for the data model classes we are building and we can just use the default behavior.

NerdDinnerDataContext Class

Visual Studio automatically generates .NET classes that represent the models and database relationships defined using the LINQ to SQL designer. A LINQ to SQL `DataContext` class is also generated for each LINQ to SQL designer file added to the solution. Because we named our LINQ to SQL class item "NerdDinner," the `DataContext` class created will be called `NerdDinnerDataContext`. This `NerdDinnerDataContext` class is the primary way we will interact with the database.

Our `NerdDinnerDataContext` class exposes two properties — `Dinners` and `RSVP` — that represent the two tables we modeled within the database. We can use C# to write LINQ queries against those properties to query and retrieve `Dinner` and `RSVP` objects from the database.

The following code (Figure 1-49) demonstrates how to instantiate a `NerdDinnerDataContext` object and perform a LINQ query against it to obtain a sequence of Dinners that occur in the future.



```
NerdDinnerDataContext db = new NerdDinnerDataContext();

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

foreach (Dinner dinner in upcomingDinners) {
    dinner.
}
```

Figure 1-49

A `NerdDinnerDataContext` object tracks any changes made to `Dinner` and `RSVP` objects retrieved using it, and enable us to easily save the changes back to the database. The code that follows demonstrates how we can use a LINQ query to retrieve a single Dinner object from the database, update two of its properties, and then save the changes back to the database:

```
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();
```

The `NerdDinnerDataContext` object in the code automatically tracked the property changes made to the `Dinner` object we retrieved from it. When we called the `SubmitChanges` method, it executed an appropriate SQL "UPDATE" statement to the database to persist the updated values back.

Creating a DinnerRepository Class

For small applications, it is sometimes fine to have Controllers work directly against a LINQ to SQL `DataContext` class, and embed LINQ queries within the Controllers. As applications get larger, though, this approach becomes cumbersome to maintain and test. It can also lead to us duplicating the same LINQ queries in multiple places.

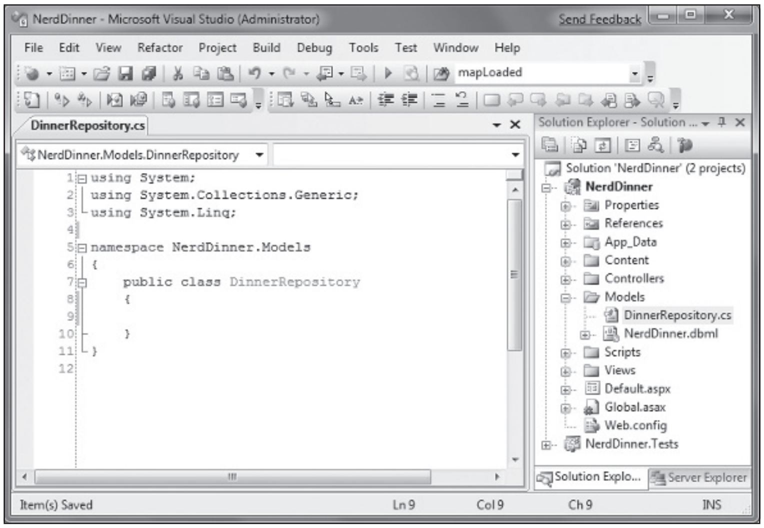
One approach that can make applications easier to maintain and test is to use a *repository* pattern. A repository class helps encapsulate data querying and persistence logic, and abstracts away the implementation details of the data persistence from the application. In addition to making application code cleaner, using a repository pattern can make it easier to change data storage implementations in the future, and it can help facilitate unit testing an application without requiring a real database.

For our NerdDinner application we'll define a `DinnerRepository` class with the following signature:

```
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();
}
```

Later in this chapter, we'll extract an IDinnerRepository interface from this class and enable dependency injection with it on our Controllers. To begin with, though, we are going to start simple and just work directly with the `DinnerRepository` *class.*

To implement this class we'll right-click on our Models folder and choose the Add ➪ New Item menu command. Within the Add New Item dialog, we'll select the Class template and name the file `DinnerRepository.cs` (Figure 1-50).



```
using System;
using System.Collections.Generic;
using System.Linq;

namespace NerdDinner.Models
{
    public class DinnerRepository
    {

    }
}
```

Figure 1-50

We can then implement our `DinnerRespository` class using the code that follows:

```
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();
        }
    }
```

Retrieving, Updating, Inserting, and Deleting Using the DinnerRepository Class

Now that we've created our `DinnerRepository` class, let's look at a few code examples that demonstrate common tasks we can do with it.

Querying Examples

The code that follows retrieves a single Dinner using the `DinnerID` value:

```
DinnerRepository dinnerRepository = new DinnerRepository();

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

The code that follows retrieves all upcoming dinners and loops over them:

```
DinnerRepository dinnerRepository = new DinnerRepository();

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

// Loop over each upcoming Dinner
foreach (Dinner dinner in upcomingDinners) {

}
```

Insert and Update Examples

The code that follows demonstrates adding two new dinners. Additions/modifications to the repository aren't committed to the database until the `Save` method is called on it. LINQ to SQL automatically wraps all changes in a database transaction — so either all changes happen or none of them does when our repository saves:

```
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();
```

The code that follows retrieves an existing Dinner object and modifies two properties on it. The changes are committed back to the database when the `Save` method is called on our repository:

```
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();
```

The code that follows retrieves a dinner and then adds an RSVP to it. It does this using the RSVPs collection on the `Dinner` object that LINQ to SQL created for us (because there is a primary-key/foreign-key relationship between the two in the database). This change is persisted back to the database as a new RSVP table row when the `Save` method is called on the repository:

```
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();
```

Delete Example

The code that follows retrieves an existing Dinner object, and then marks it to be deleted. When the `Save` method is called on the repository, it will commit the delete back to the database:

```
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();
```

Integrating Validation and Business Rule Logic with Model Classes

Integrating validation and business rule logic is a key part of any application that works with data.

Schema Validation

When model classes are defined using the LINQ to SQL designer, the datatypes of the properties in the data model classes will correspond to the datatypes of the database table. For example: if the EventDate column in the Dinners table is a datetime, the data model class created by LINQ to SQL will be of type `DateTime` (which is a built-in .NET datatype). This means you will get compile errors if you attempt to assign an integer or boolean to it from code, and it will raise an error automatically if you attempt to implicitly convert a non-valid string type to it at runtime.

LINQ to SQL will also automatically handle escaping SQL values for you when you use strings — so you don't need to worry about SQL injection attacks when using it.

Validation and Business Rule Logic

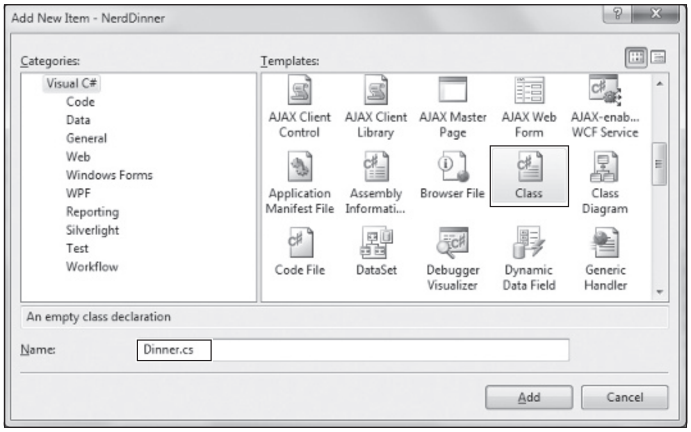
Datatype validation is useful as a first step but is rarely sufficient. Most real-world scenarios require the ability to specify richer validation logic that can span multiple properties, execute code, and often have awareness of a model's state (for example: is it being created /updated/deleted, or within a domain-specific state like "archived").

There are a variety of different patterns and frameworks that can be used to define and apply validation rules to model classes, and there are several .NET based frameworks out there that can be used to help with this. You can use pretty much any of them within ASP.NET MVC applications.

For the purposes of our NerdDinner application, we'll use a relatively simple and straightforward pattern where we expose an `IsValid` property and a `GetRuleViolations` method on our Dinner model object. The `IsValid` property will return true or false depending on whether the validation and business rules are all valid. The `GetRuleViolations` method will return a list of any rule errors.

We'll implement `IsValid` and `GetRuleViolations` by adding a *partial class* to our project. Partial classes can be used to add methods/properties/events to classes maintained by a VS designer (like the Dinner class generated by the LINQ to SQL designer) and help avoid having the tool from messing with our code.

We can add a new partial class to our project by right-clicking on the `\Models` folder, and then selecting the Add New Item menu command. We can then choose the Class template within the Add New Item dialog (Figure 1-51) and name it `Dinner.cs`.



An empty class declaration

Name: Dinner.cs

Figure 1-51

Clicking the Add button will add a `Dinner.cs` file to our project and open it within the IDE. We can then implement a basic rule/validation enforcement framework using the following code:

```
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) {
            ErrorMessage = errorMessage;
        }

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

A few notes about this code:

- The Dinner class is prefaced with a *partial* keyword — which means the code contained within it will be combined with the class generated/maintained by the LINQ to SQL designer and compiled into a single class.
- Invoking the `GetRuleViolations` method will cause our validation and business rules to be evaluated (we'll implement them shortly). The `GetRuleViolations` method returns back a sequence of `RuleViolation` objects that provide more details about each rule error.
- The `IsValid` property provides a convenient helper property that indicates whether the Dinner object has any active `RuleViolations`. It can be proactively checked by a developer using the Dinner object at any time (and does not raise an exception).
- The `OnValidate` partial method is a hook that LINQ to SQL provides that allows us to be notified any time the Dinner object is about to be persisted within the database. Our `OnValidate` implementation in the previous code ensures that the Dinner has no `RuleViolations` before it is saved. If it is in an invalid state, it raises an exception, which will cause LINQ to SQL to abort the transaction.

This approach provides a simple framework that we can integrate validation and business rules into. For now let's add the below rules to our `GetRuleViolations` method:

```
    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;
    }
```

We are using the *yield return* feature of C# to return a sequence of any `RuleViolations`. The first six rule checks in the previous code simply enforce that string properties on our Dinner cannot be null or empty. The last rule is a little more interesting and calls a `PhoneValidator.IsValidNumber` helper method that we can add to our project to verify that the `ContactPhone` number format matches the Dinner's country.

We can use .NET's regular expression support to implement this phone validation support. The code that follows is a simple `PhoneValidator` implementation that we can add to our project that enables us to add country-specific Regex pattern checks:

```
    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;
            }
        }
    }
```

Now when we try to create or update a Dinner, our validation logic rules will be enforced. Developers can proactively determine if a Dinner object is valid, and retrieve a list of all violations in it without raising any exceptions:

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

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

if (!dinner.IsValid) {

    var errors = dinner.GetRuleViolations();

    // do something to fix errors
}
```

If we attempt to save a Dinner in an invalid state, an exception will be raised when we call the `Save` method on the `DinnerRepository`. This occurs because our `Dinner.OnValidate` partial method raises an exception if any rule violations exist in the Dinner. We can catch this exception and reactively retrieve a list of the violations to fix:

```
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
}
```

Because our validation and business rules are implemented within our domain model layer, and not within the UI layer, they will be applied and used across all scenarios within our application. We can later change or add business rules and have all code that works with our Dinner objects honor them. Having the flexibility to change business rules in one place, without having these changes ripple throughout the application and UI logic, is a sign of a well-written application, and a benefit that an MVC framework helps encourage.

Controllers and Views

With traditional web frameworks (classic ASP, PHP, ASP.NET Web Forms, etc.), incoming URLs are typically mapped to files on disk. For example: a request for a URL like `/Products.aspx` or `/Products.php` might be processed by a `Products.aspx` or `Products.php` file.

Web-based MVC frameworks map URLs to server code in a slightly different way. Instead of mapping incoming URLs to files, they instead map URLs to methods on classes. These classes are called *Controllers* and they are responsible for processing incoming HTTP requests, handling user input, retrieving and saving data, and determining the response to send back to the client (display HTML, download a file, redirect to a different URL, etc.).

Now that we have built up a basic model for our NerdDinner application, our next step will be to add a Controller to the application that takes advantage of it to provide users with a data listing/details navigation experience for dinners on our site.

Adding a DinnersController Controller

We'll begin by right-clicking on the Controllers folder within our web project, and then selecting the Add ➪ Controller menu command (Figure 1-52).

You can also execute this command by typing Ctrl-M, Ctrl-C.

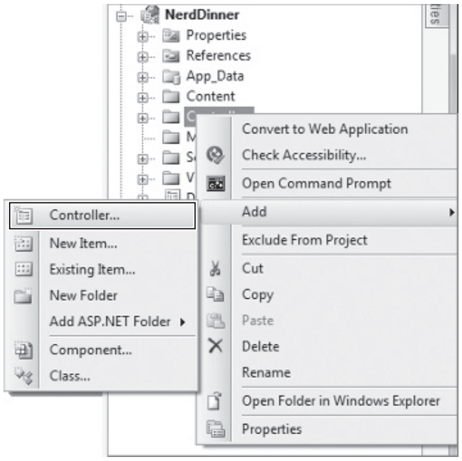
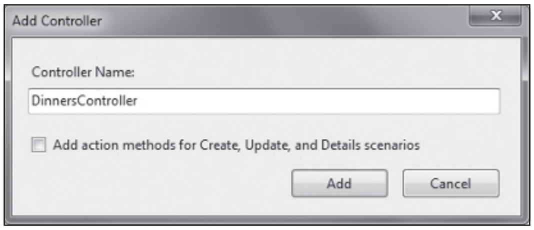


Figure 1-52

This will bring up the Add Controller dialog (Figure 1-53):


Add Controller

Controller Name: DinnersController

□ Add action methods for Create, Update, and Details scenarios

Figure 1-53

We'll name the new controller **DinnersController** and click the Add button. Visual Studio will then add a `DinnersController.cs` file under our `\Controllers` directory (Figure 1-54).

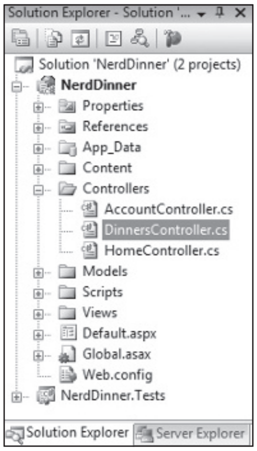


Figure 1-54

It will also open up the new `DinnersController` class within the code-editor.

Adding Index and Details Action Methods to the DinnersController Class

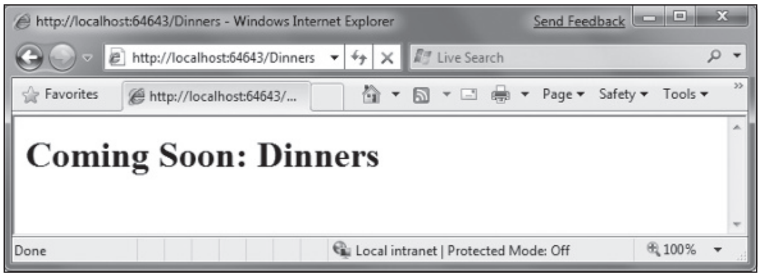
We want to enable visitors using our application to browse the list of upcoming dinners, and enable them to click on any dinner in the list to see specific details about it. We'll do this by publishing the following URLs from our application:

URL	Purpose
`/Dinners/`	Display an HTML list of upcoming dinners.
`/Dinners/Details/[id]`	Display details about a specific dinner indicated by an "id" parameter embedded within the URL — which will match the `DinnerID` of the dinner in the database. For example: `/Dinners/Details/2` would display an HTML page with details about the Dinner whose `DinnerID` value is 2.

We can publish initial implementations of these URLs by adding two public "action methods" to our `DinnersController` class:

```
public class DinnersController : Controller {

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

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

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

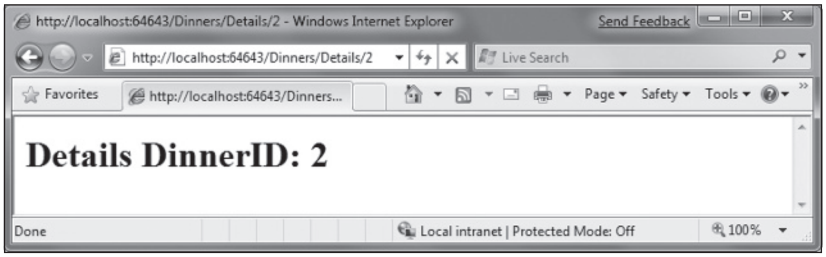
We can then run the application and use our browser to invoke them. Typing in the **`/Dinners/`** URL will cause our *`Index`* method to run, and it will send back the following response (Figure 1-55):


http://localhost:64643/Dinners - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners
Live Search
Favorites
http://localhost:64643/...
Page
Safety
Tools
Done
Local intranet | Protected Mode: Off
100%

Coming Soon: Dinners

Figure 1-55

Typing in the **`/Dinners/Details/2`** URL will cause our `Details` method to run, and send back the response in Figure 1-56.


http://localhost:64643/Dinners/Details/2 - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Details/2
Live Search
Favorites
http://localhost:64643/Dinners...
Page
Safety
Tools
Done
Local intranet | Protected Mode: Off
100%

Details DinnerID: 2

Figure 1-56

You might be wondering — how did ASP.NET MVC know to create our `DinnersController` class and invoke those methods? To understand that let's take a quick look at how routing works.

Understanding ASP.NET MVC Routing

ASP.NET MVC includes a powerful URL routing engine that provides a lot of flexibility in controlling how URLs are mapped to controller classes. It allows us to completely customize how ASP.NET MVC chooses which controller class to create, which method to invoke on it, as well as configure different ways that variables can be automatically parsed from the URL/querystring and passed to the method as parameter arguments. It delivers the flexibility to totally optimize a site for SEO (search engine optimization) as well as publish any URL structure we want from an application.

By default, new ASP.NET MVC projects come with a preconfigured set of URL routing rules already registered. This enables us to easily get started on an application without having to explicitly configure anything. The default routing rule registrations can be found within the Application class of our projects — which we can open by double-clicking the Global.asax file in the root of our project (Figure 1-57).

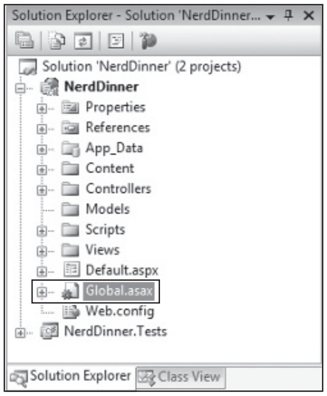


Figure 1-57

The default ASP.NET MVC routing rules are registered within the RegisterRoutes method of this class:

```
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
    );
}
```

The routes.MapRoute method call in the previous code registers a default routing rule that maps incoming URLs to controller classes using the URL format: /{*controller*}/{*action*}/{*id*} — where *controller* is the name of the controller class to instantiate, *action* is the name of a public method to invoke on it, and *id* is an optional parameter embedded within the URL that can be passed as an argument to the method. The third parameter passed to the MapRoute method call is a set of default values to use for the controller/action/id values in the event that they are not present in the URL (controller = "Home", action="Index", id="").

The following table demonstrates how a variety of URLs are mapped using the default /{*controllers*}/{*action*}/{*id*} route rule:

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

The last three rows show the default values (`Controller = Home, Action = Index, Id = ""`) being used. Because the `Index` method is registered as the default action name if one isn't specified, the `/Dinners` and `/Home` URLs cause the `Index` action method to be invoked on their Controller classes. Because the "Home" controller is registered as the default controller if one isn't specified, the `/` URL causes the `HomeController` to be created, and the `Index` action method on it to be invoked.

If you don't like these default URL routing rules, the good news is that they are easy to change — just edit them within the `RegisterRoutes` method in the previous code. For our NerdDinner application, though, we aren't going to change any of the default URL routing rules — instead we'll just use them as-is.

Using the DinnerRepository from Our DinnersController

Let's now replace the current implementation of our `Index` and `Details` action methods with implementations that use our model.

We'll use the `DinnerRepository` class we built earlier to implement the behavior. We'll begin by adding a *using* statement that references the `NerdDinner.Models` namespace, and then declare an instance of our `DinnerRepository` as a field on our `DinnerController` class.

Later in this chapter, we'll introduce the concept of *Dependency Injection* and show another way for our Controllers to obtain a reference to a `DinnerRepository` that enables better unit testing — but for right now we'll just create an instance of our `DinnerRepository` inline like the code that follows.

```
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);
            }
        }
    }
```

Now we are ready to generate a HTML response back using our retrieved data model objects.

Using Views with Our Controller

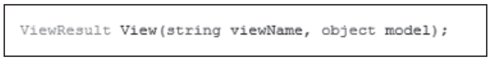
While it is possible to write code within our action methods to assemble HTML and then use the `Response.Write` helper method to send it back to the client, that approach becomes fairly unwieldy quickly. A much better approach is for us to only perform application and data logic inside our `DinnersController` action methods, and to then pass the data needed to render a HTML response to a separate view template that is responsible for outputting the HTML representation of it. As we'll see in a moment, a *view* template is a text file that typically contains a combination of HTML markup and embedded rendering code.

Separating our controller logic from our view rendering brings several big benefits. In particular it helps enforce a clear *separation of concerns* between the application code and UI formatting/rendering code. This makes it much easier to unit test application logic in isolation from UI rendering logic. It makes it easier to later modify the UI rendering templates without having to make application code changes. And it can make it easier for developers and designers to collaborate together on projects.

We can update our `DinnersController` class to indicate that we want to use a view template to send back an HTML UI response by changing the method signatures of our two action methods from having a return type of "void" to instead have a return type of `ActionResult`. We can then call the `View` helper method on the Controller base class to return back a `ViewResult` object:

```
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);
    }
}
```

The signature of the `View` helper method we are using in the previous code looks like Figure 1-58.



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

Figure 1-58

The first parameter to the `View` helper method is the name of the view template file we want to use to render the HTML response. The second parameter is a model object that contains the data that the view template needs in order to render the HTML response.

Within our `Index` action method we are calling the `View` helper method and indicating that we want to render an HTML listing of dinners using an "Index" view template. We are passing the view template a sequence of Dinner objects to generate the list from:

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

public ActionResult Index() {

    var dinners = dinnerRepository.FindUpcomingDinners().ToList();

    return View("Index", dinners);
}
```

Within our `Details` action method, we attempt to retrieve a Dinner object using the id provided within the URL. If a valid Dinner is found we call the `View` helper method, indicating we want to use a `"Details"` view template to render the retrieved Dinner object. If an invalid dinner is requested, we render a helpful error message that indicates that the dinner doesn't exist using a `"NotFound"` view template (and an overloaded version of the `View()` helper method that just takes the template name):

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

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.FindDinner(id);

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

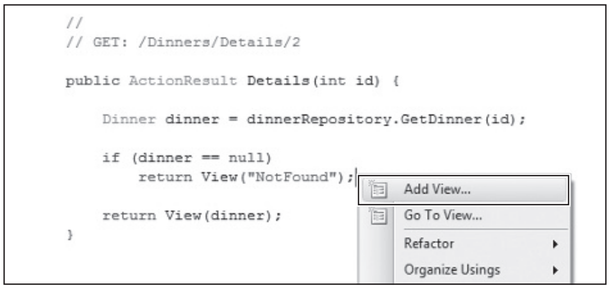
```
                return View("Details", dinner);
        }
```

Let's now implement the "NotFound", "Details", and "Index" view templates.

Implementing the "NotFound" View Template

We'll begin by implementing the "NotFound" view template — which displays a friendly error message indicating that the requested dinner can't be found.

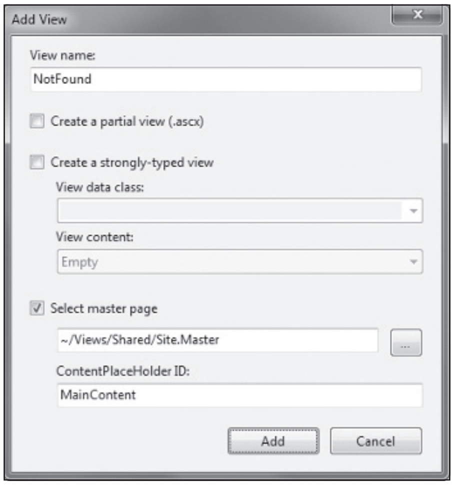
We'll create a new view template by positioning our text cursor within a controller action method, and then by right clicking and choosing the Add View menu command (Figure 1-59; we can also execute this command by pressing Ctrl-M, Ctrl-V):



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

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

    return View(dinner);
}
```

Figure 1-59

This will bring up an Add View dialog shown in Figure 1-60. By default, the dialog will pre-populate the name of the view to create to match the name of the action method the cursor was in when the dialog was launched (in this case "Details"). Because we want to first implement the "NotFound" template, we'll override this view name and set it instead to be **NotFound**:



View name: NotFound

□ Create a partial view (.ascx)

□ Create a strongly-typed view

View data class:

View content: Empty

☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-60

When we click the Add button, Visual Studio will create a new `NotFound.aspx` (Figure 1-61) view template for us within the `\Views\Dinners` directory (which it will also create if the directory doesn't already exist):

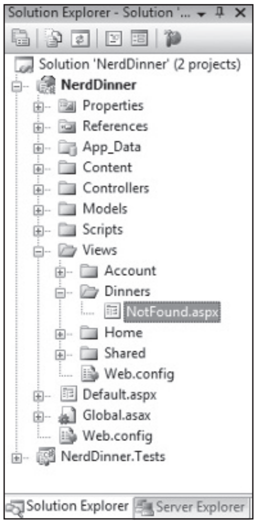
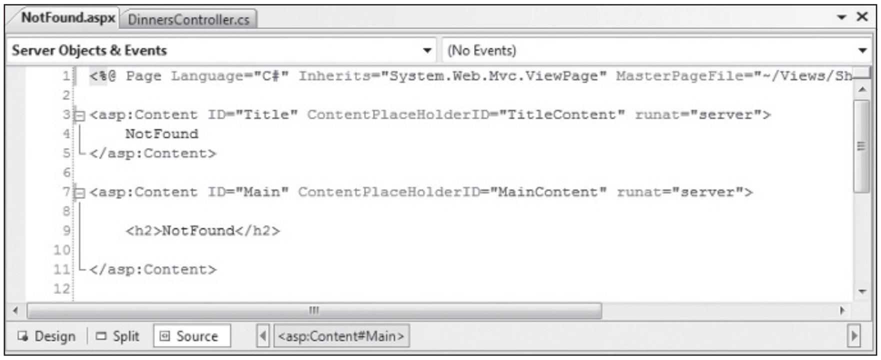


Figure 1-61

It will also open up our new `NotFound.aspx` view template within the code-editor (Figure 1-62):



```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" MasterPageFile="~/Views/Sh

<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    NotFound
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>NotFound</h2>

</asp:Content>

```

Figure 1-62

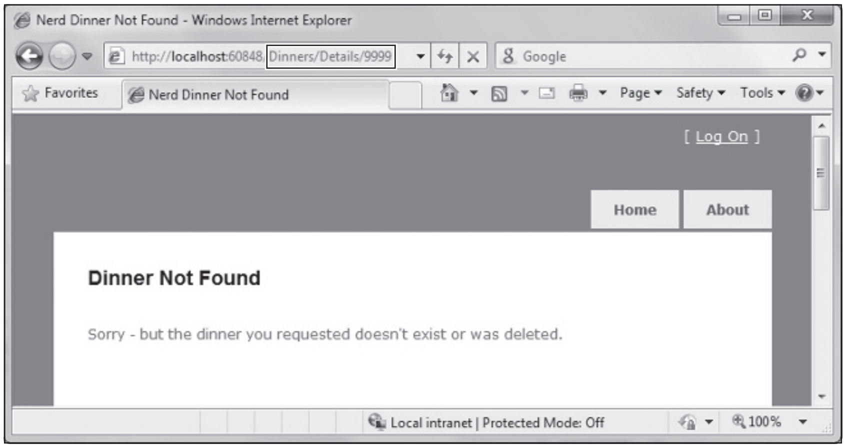
View templates by default have two *content regions* where we can add content and code. The first allows us to customize the "title" of the HTML page sent back. The second allows us to customize the "main content" of the HTML page sent back.

To implement our `"NotFound"` view template, we'll add some basic content:

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

```
</asp:Content>

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

We can then try it out within the browser. To do this let's request the `/Dinners/Details/9999` URL. This will refer to a dinner that doesn't currently exist in the database, and will cause our `DinnersController.Details` action method to render our `"NotFound"` view template (Figure 1-63).


Nerd Dinner Not Found - Windows Internet Explorer
http://localhost:60848/Dinners/Details/9999
Google
Favorites
Nerd Dinner Not Found
Page
Safety
Tools
[Log On]
Home
About
Local intranet | Protected Mode: Off
100%

Dinner Not Found

Sorry - but the dinner you requested doesn't exist or was deleted.

Figure 1-63

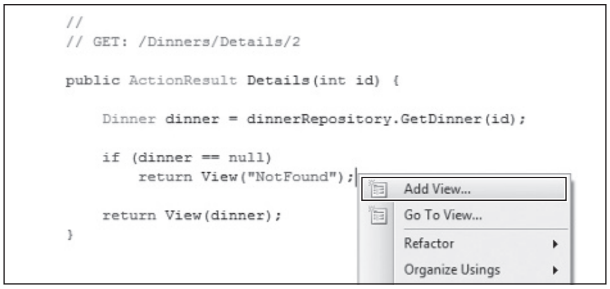
One thing you'll notice in Figure 1-63 is that our basic view template has inherited a bunch of HTML that surrounds the main content on the screen. This is because our view template is using a *master page* template that enables us to apply a consistent layout across all views on the site. We'll discuss how master pages work more in a later part of this chapter.

Implementing the "Details" View Template

Let's now implement the `"Details"` view template — which will generate HTML for a single Dinner model.

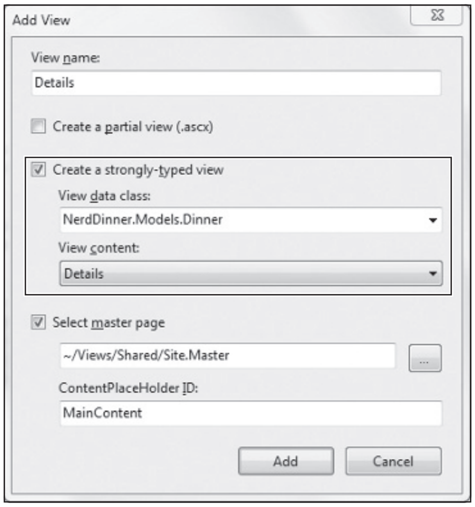
We'll do this by positioning our text cursor within the `Details` action method, and then right-clicking and choosing the Add View menu command — Figure 1-64 — or pressing Ctrl-M, Ctrl-V.

This will bring up the Add View dialog. We'll keep the default view name (Details). We'll also select the "Create a strongly typed view" checkbox in the dialog and select (using the combobox drop-down) the name of the model type we are passing from the Controller to the View. For this view we are passing a Dinner object (the fully qualified name for this type is: `NerdDinner.Models.Dinner`) as shown in Figure 1-65.



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

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

    return View(dinner);
}
```

Figure 1-64



Add View

View name:
Details

☐ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class:
NerdDinner.Models.Dinner

View content:
Details

☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID:
MainContent

Figure 1-65

Unlike the previous template, where we chose to create an "Empty View," this time we will choose to automatically *scaffold* the view using a `"Details"` template. We can indicate this by changing the View content drop-down in the dialog above.

Scaffolding will generate an initial implementation of our details view template based on the Dinner model we are passing to it. This provides an easy way for us to quickly get started on our view template implementation.

When we click the Add button, Visual Studio will create a new Details.aspx view template file for us within our \Views\Dinners directory (Figure 1-66).

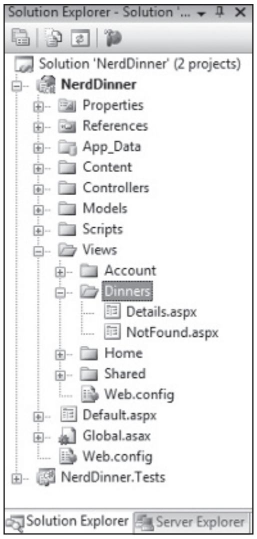


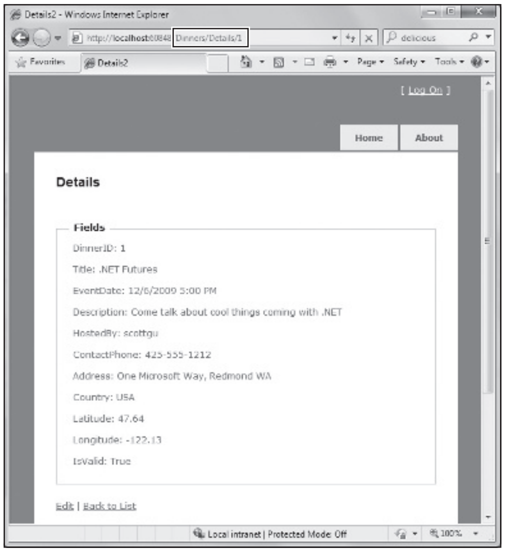
Figure 1-66

It will also open up our new Details.aspx view template within the code-editor. It will contain an initial scaffold implementation of a details view based on a Dinner model. The scaffolding engine uses .NET reflection to look at the public properties exposed on the class passed to it, and will add appropriate content based on each type it finds:

```
<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>
```

We can request the `/Dinners/Details/1` URL to see what this "details" scaffold implementation looks like in the browser. Using this URL will display one of the dinners we manually added to our database when we first created it (Figure 1-67).


Details2 - Windows Internet Explorer
http://localhost:60848/Dinners/Details/1
delicious
Favorites
Details2
Page
Safety
Tools
[Log On]
Home
About
Local intranet | Protected Mode: Off
100%

Details

Fields

DinnerID: 1

Title: .NET Futures

EventDate: 12/6/2009 5:00 PM

Description: Come talk about cool things coming with .NET

HostedBy: scottgu

ContactPhone: 425-555-1212

Address: One Microsoft Way, Redmond WA

Country: USA

Latitude: 47.64

Longitude: -122.13

IsValid: True

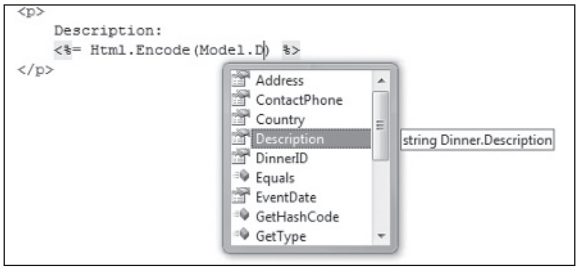
Edit | Back to List

Figure 1-67

This gets us up and running quickly, and provides us with an initial implementation of our Details.aspx view. We can then go and tweak it to customize the UI to our satisfaction.

When we look at the Details.aspx template more closely, we'll find that it contains static HTML as well as embedded rendering code. `<% %>` code nuggets execute code when the view template renders, and `<%= %>` code nuggets execute the code contained within them and then render the result to the output stream of the template.

We can write code within our View that accesses the Dinner model object that was passed from our controller using a strongly typed `Model` property. Visual Studio provides us with full code-IntelliSense when accessing this `Model` property within the editor (Figure 1-68).



```
<p>
    Description:
    <%= Html.Encode(Model.D) %>
</p>
```

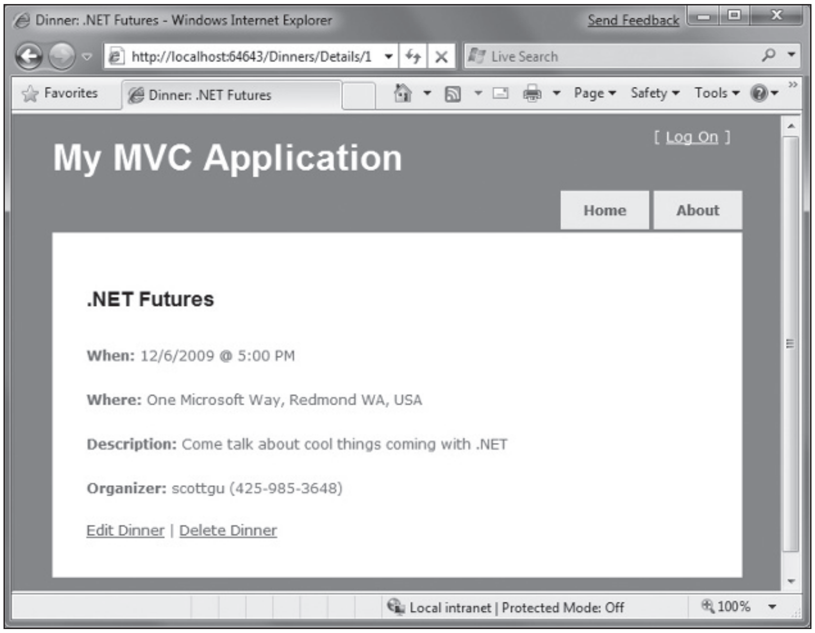
Figure 1-68

Let's make some tweaks so that the source for our final Details view template looks like that below:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
 Dinner: <%= 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 Dinner", "Edit", new { id=Model.DinnerID })%> |
        <%= Html.ActionLink("Delete Dinner","Delete", new { id=Model.DinnerID})%>

    </asp:Content>
```

When we access the /Dinners/Details/1 URL again, it will render like so (Figure 1-69):


Dinner: .NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Details/1
Live Search
Favorites
Dinner: .NET Futures
Page
Safety
Tools
[Log On]
My MVC Application
Home
About
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

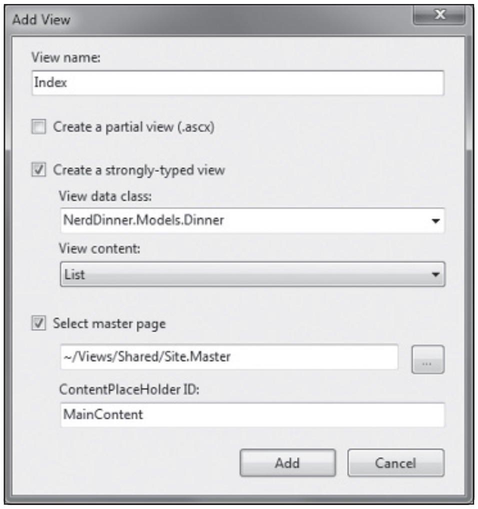
Edit Dinner | Delete Dinner

Figure 1-69

Implementing the "Index" View Template

Let's now implement the "Index" view template — which will generate a listing of upcoming dinners. To do this we'll position our text cursor within the Index action method, and then right-click and choose the Add View menu command (or press Ctrl-M, Ctrl-V).

Within the Add View dialog (Figure 1-70), we'll keep the view template named **Index** and select the "Create a strongly-typed view" checkbox. This time we will choose to automatically generate a List view template, and select NerdDinner.Models.Dinner as the model type passed to the view (which because we have indicated we are creating a List scaffold will cause the Add View dialog to assume we are passing a sequence of Dinner objects from our Controller to the View):



View name: Index

☐ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class: NerdDinner.Models.Dinner

View content: List

☑ Select master page

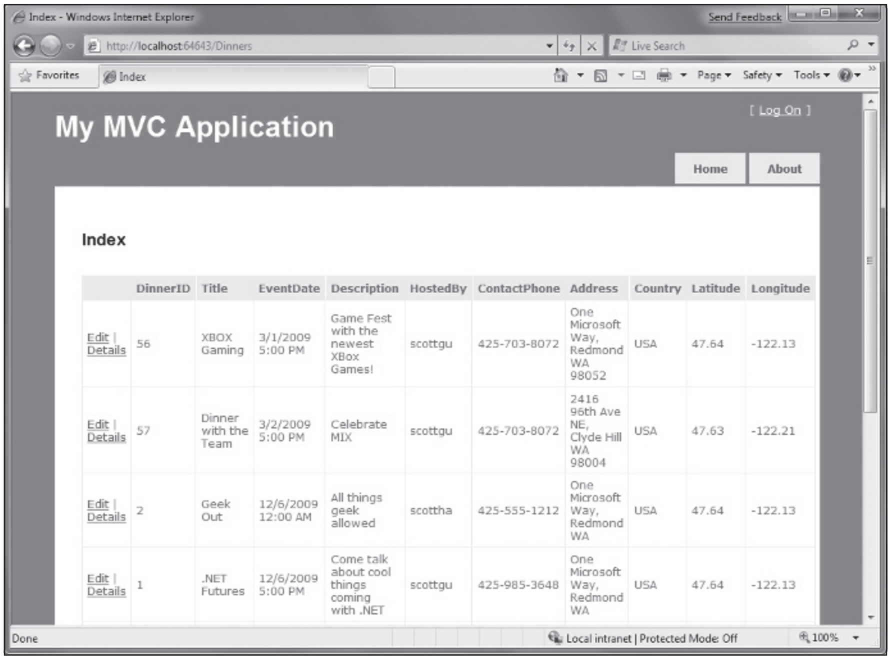
~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-70

When we click the Add button, Visual Studio will create a new Index.aspx view template file for us within our `\Views\Dinners` directory. It will *scaffold* an initial implementation within it that provides an HTML table listing of the Dinners we pass to the view.

When we run the application and access the `/Dinners/` URL, it will render our list of dinners like so (Figure 1-71):



Index

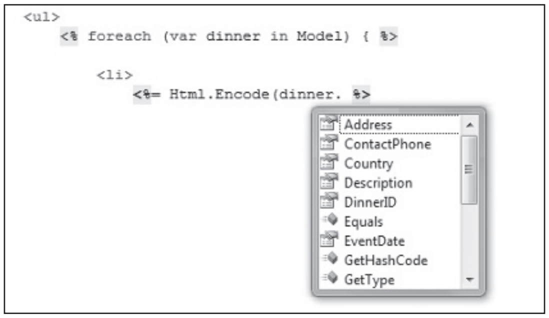
	DinnerID	Title	EventDate	Description	HostedBy	ContactPhone	Address	Country	Latitude	Longitude
Edit | Details	56	XBOX Gaming	3/1/2009 5:00 PM	Game Fest with the newest XBox Games!	scottgu	425-703-8072	One Microsoft Way, Redmond WA 98052	USA	47.64	-122.13
Edit | Details	57	Dinner with the Team	3/2/2009 5:00 PM	Celebrate MIX	scottgu	425-703-8072	2416 96th Ave NE, Clyde Hill WA 98004	USA	47.63	-122.21
Edit | Details	2	Geek Out	12/6/2009 12:00 AM	All things geek allowed	scottha	425-555-1212	One Microsoft Way, Redmond WA	USA	47.64	-122.13
Edit | Details	1	.NET Futures	12/6/2009 5:00 PM	Come talk about cool things coming with .NET	scottgu	425-985-3648	One Microsoft Way, Redmond WA	USA	47.64	-122.13

Figure 1-71

The table solution in Figure 1-71 gives us a grid-like layout of our Dinner data — which isn’t quite what we want for our consumer-facing Dinner listing. We can update the Index.aspx view template and modify it to list fewer columns of data, and use a `<ul>` element to render them instead of a table using the code that follows:

```
<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>
```

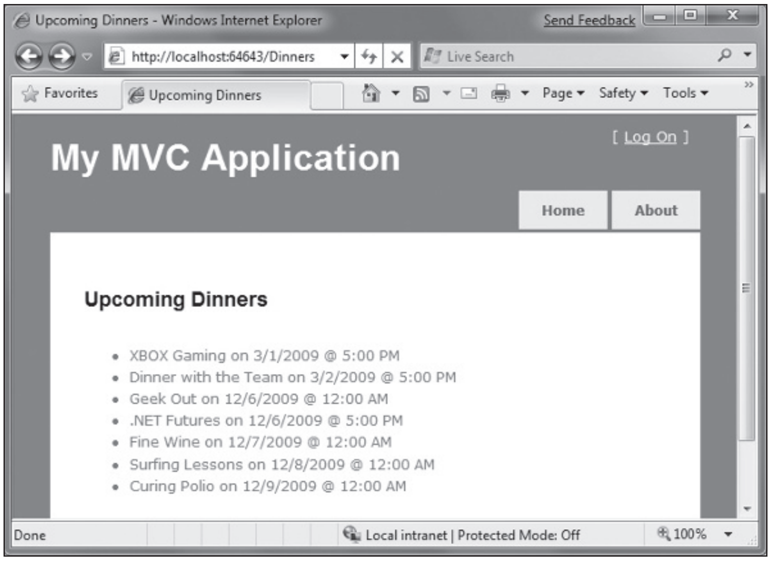
We are using the `var` keyword within the `foreach` statement as we loop over each dinner in our model. Those unfamiliar with C# 3.0 might think that using `var` means that the Dinner object is late-bound. It, instead, means that the compiler is using type-inference against the strongly typed `Model` property (which is of type `IEnumerable<Dinner>`) and compiling the local “dinner” variable as a Dinner type — which means we get full IntelliSense and compile-time checking for it within code blocks (Figure 1-72).



```
<ul>
    <% foreach (var dinner in Model) { %>

        <li>
            <%= Html.Encode(dinner. %>
```

Figure 1-72

When we press the Refresh button on the `/Dinners` URL in our browser, our updated view now looks like Figure 1-73.


Upcoming Dinners - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners
Live Search
Favorites
Upcoming Dinners
Page
Safety
Tools
[Log On]
Home
About
Done
Local intranet | Protected Mode: Off
100%

My MVC Application

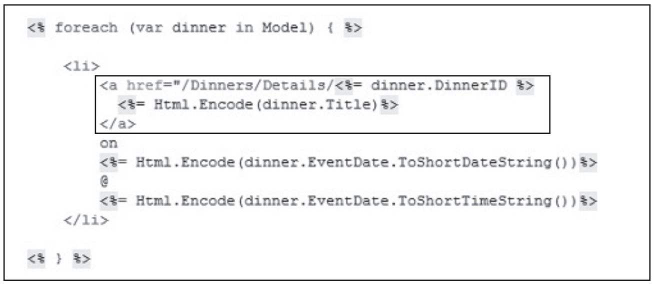
Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- .NET Futures on 12/6/2009 @ 5:00 PM
- Fine Wine on 12/7/2009 @ 12:00 AM
- Surfing Lessons on 12/8/2009 @ 12:00 AM
- Curing Polio on 12/9/2009 @ 12:00 AM

Figure 1-73

This is looking better — but isn't entirely there yet. Our last step is to enable end users to click individual dinners in the list and see details about them. We'll implement this by rendering HTML hyperlink elements that link to the Details action method on our `DinnersController`.

We can generate these hyperlinks within our Index view in one of two ways. The first is to manually create HTML `<a>` elements like Figure 1-74, where we embed `<% %>` blocks within the `<a>` HTML element:



```
<% 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>

<% } %>
```

Figure 1-74

An alternative approach we can use is to take advantage of the built-in `Html.ActionLink` helper method within ASP.NET MVC that supports programmatically creating an HTML `<a>` element that links to another action method on a Controller:

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

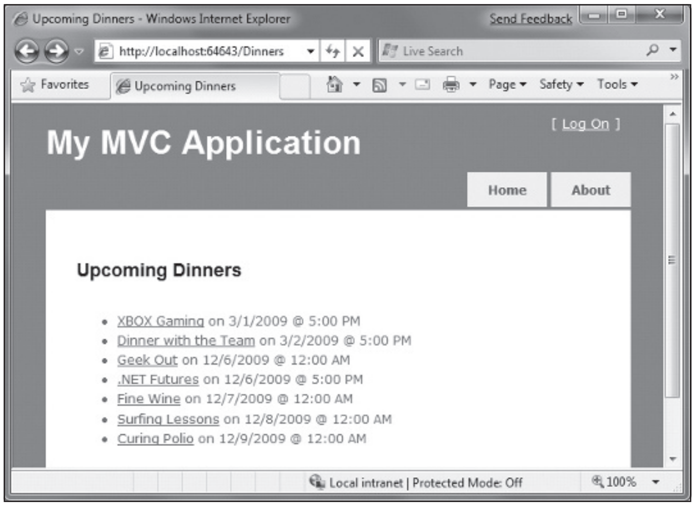
The first parameter to the `Html.ActionLink` helper method is the link-text to display (in this case the title of the dinner), the second parameter is the Controller action name we want to generate the link to (in this case the "Details" method), and the third parameter is a set of parameters to send to the action (implemented as an anonymous type with property name/values). In this case we are specifying the id parameter of the dinner we want to link to, and because the default URL routing rule in ASP.NET MVC is {*Controller*}/{*Action*}/{*id*} the Html.ActionLink helper method will generate the following output:

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

For our Index.aspx view we'll use the Html.ActionLink helper method approach and have each dinner in the list link to the appropriate details URL:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Upcoming Dinners
</asp:Content>

<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>
```

And now when we hit the /Dinners URL, our dinner list looks like Figure 1-75:

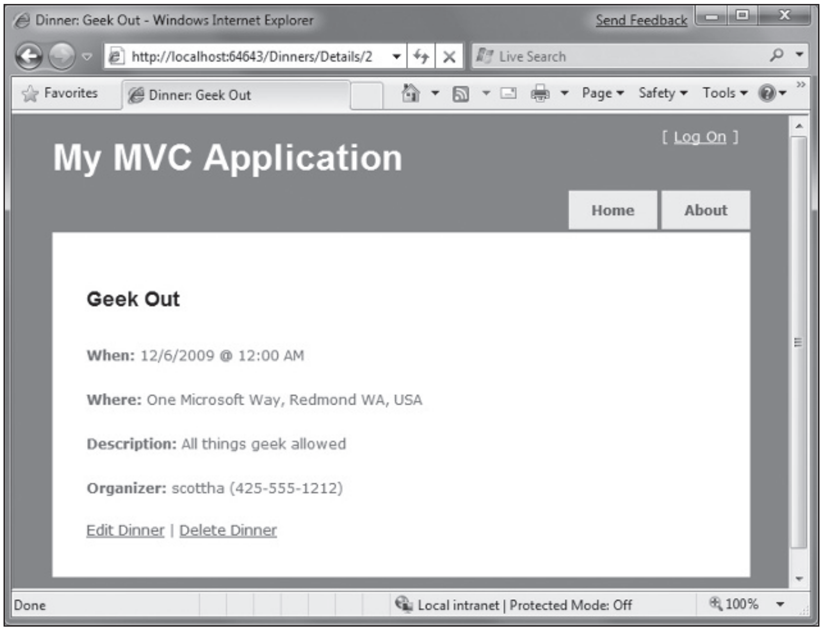

Upcoming Dinners - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners
Live Search
Favorites
Upcoming Dinners
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Local intranet | Protected Mode: Off
100%

Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- .NET Futures on 12/6/2009 @ 5:00 PM
- Fine Wine on 12/7/2009 @ 12:00 AM
- Surfing Lessons on 12/8/2009 @ 12:00 AM
- Curing Polio on 12/9/2009 @ 12:00 AM

Figure 1-75

When we click any of the dinners in the list, we'll navigate to see details about it (Figure 1-76):


Dinner: Geek Out - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Details/2
Live Search
Favorites
Dinner: Geek Out
Page
Safety
Tools
[Log On]
My MVC Application
Home
About
Edit Dinner | Delete Dinner
Done
Local intranet | Protected Mode: Off
100%

Geek Out

When: 12/6/2009 @ 12:00 AM

Where: One Microsoft Way, Redmond WA, USA

Description: All things geek allowed

Organizer: scottha (425-555-1212)

Figure 1-76

Convention-Based Naming and the \Views Directory Structure

ASP.NET MVC applications, by default, use a convention-based directory naming structure when resolving view templates. This allows developers to avoid having to fully qualify a location path when referencing views from within a Controller class. By default ASP.NET MVC will look for the view template file within the \Views\*[ControllerName]*\\ directory underneath the application.

For example, we've been working on the `DinnersController` class — which explicitly references three view templates: `"Index"`, `"Details"`, and `"NotFound"`. ASP.NET MVC will, by default, look for these views within the \Views\Dinners directory underneath our application root directory (Figure 1-77).

Notice in Figure 1-77 how there are currently three controller classes within the project (`DinnersController`, `HomeController`, and `AccountController` — the last two were added by default when we created the project), and there are three subdirectories (one for each controller) within the `\Views` directory.

Views referenced from the Home and Accounts controllers will automatically resolve their view templates from the respective \Views\Home and \Views\Account directories. The \Views\Shared subdirectory provides a way to store view templates that are reused across multiple controllers within the application. When ASP.NET MVC attempts to resolve a view template, it will first check within the \Views\*[Controller]* specific directory, and if it can't find the view template there it will look within the \Views\Shared directory.

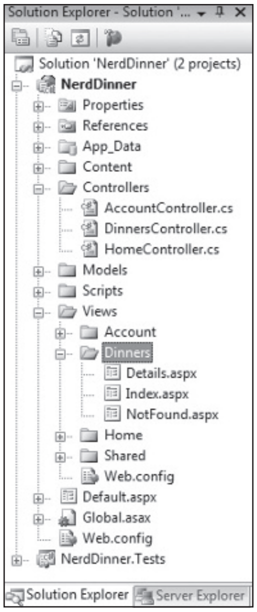


Figure 1-77

When it comes to naming individual view templates, the recommended guidance is to have the view template share the same name as the action method that caused it to render. For example, above our `Index` action method is using the `"Index"` view to render the view result, and the `Details` action method is using the `"Details"` view to render its results. This makes it easy to quickly see which template is associated with each action.

Developers do not need to explicitly specify the view template name when the view template has the same name as the action method being invoked on the controller. We can instead just pass the model object to the `View` helper method (without specifying the view name), and ASP.NET MVC will automatically infer that we want to use the *\Views\[ControllerName]\[ActionName]* view template on disk to render it.

This allows us to clean up our controller code a little, and avoid duplicating the name twice in our code:

```
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);
        }
    }
```

The previous code is all that is needed to implement a nice Dinner listing/details experience for the site.

Create, Update, Delete Form Scenarios

We've introduced controllers and views, and covered how to use them to implement a listing/details experience for dinners on the site. Our next step will be to take our `DinnersController` class further and enable support for editing, creating, and deleting dinners with it as well.

URLs Handled by DinnersController

We previously added action methods to `DinnersController` that implemented support for two URLs: `/Dinners` and `/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.

We will now add action methods to implement three additional URLs: *`/Dinners/Edit/[id]`*, *`/Dinners/Create`*, and *`/Dinners/Delete/[id]`*. These URLs will enable support for editing existing dinners, creating new dinners, and deleting dinners.

We will support both HTTP `GET` and HTTP `POST` verb interactions with these new URLs. HTTP `GET` requests to these URLs will display the initial HTML view of the data (a form populated with the Dinner data in the case of "edit," a blank form in the case of "create," and a delete confirmation screen in the case of "delete"). HTTP POST requests to these URLs will save/update/delete the Dinner data in our DinnerRepository (and from there to the database).

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.

Let's begin by implementing the "edit" scenario.

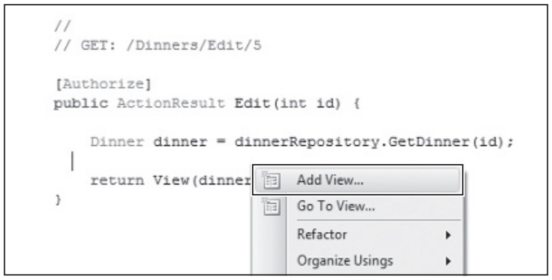
Implementing the HTTP-GET Edit Action Method

We'll start by implementing the HTTP `GET` behavior of our edit action method. This method will be invoked when the `/Dinners/Edit/[id]` URL is requested. Our implementation will look like:

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

public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    return View(dinner);
}
```

The code above uses the `DinnerRepository` to retrieve a `Dinner` object. It then renders a view template using the `Dinner` object. Because we haven't explicitly passed a template name to the `View` helper method, it will use the convention based default path to resolve the view template: `/Views/Dinners/Edit.aspx`.

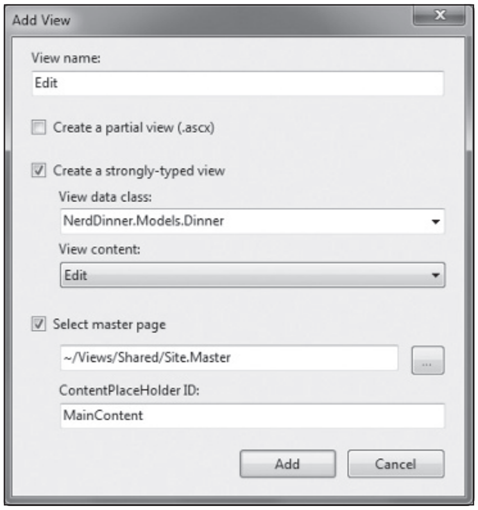
Let's now create this view template. We will do this by right-clicking within the `Edit` method and selecting the Add View context menu command (Figure 1-78).



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

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    return View(dinner
}
```

Figure 1-78

Within the Add View dialog, we'll indicate that we are passing a `Dinner` object to our view template as its model, and choose to auto-scaffold an Edit template (Figure 1-79).



View name: Edit

☐ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class: NerdDinner.Models.Dinner

View content: Edit

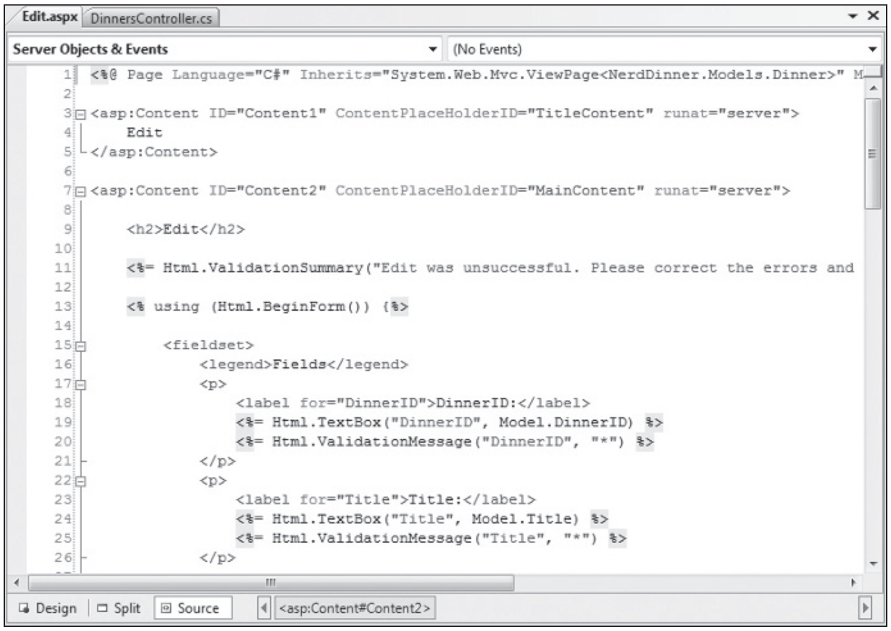
☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-79

When we click the Add button, Visual Studio will add a new `Edit.aspx` view template file for us within the *\Views\Dinners* directory. It will also open up the new `Edit.aspx` view template within the code-editor — populated with an initial "Edit" scaffold implementation like that in Figure 1-80.



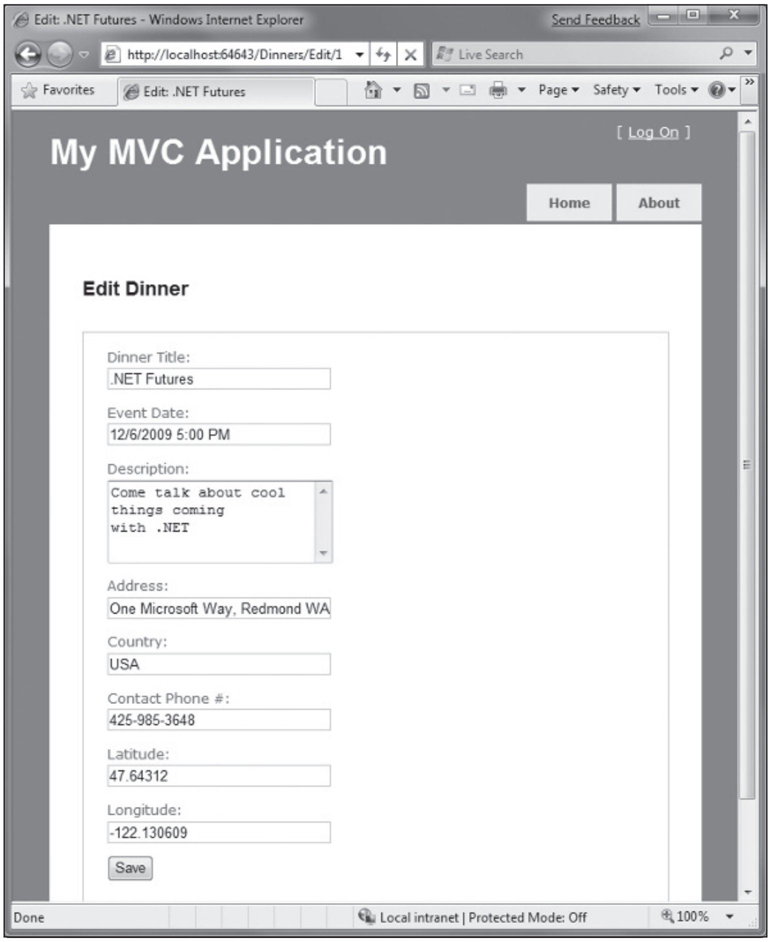
```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<NerdDinner.Models.Dinner>" M

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Edit
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit</h2>

    <%= Html.ValidationSummary("Edit was unsuccessful. Please correct the errors and

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

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="DinnerID">DinnerID:</label>
                <%= Html.TextBox("DinnerID", Model.DinnerID) %>
                <%= Html.ValidationMessage("DinnerID", "*") %>
            </p>
            <p>
                <label for="Title">Title:</label>
                <%= Html.TextBox("Title", Model.Title) %>
                <%= Html.ValidationMessage("Title", "*") %>
            </p>
```

Figure 1-80

Let’s make a few changes to the default “Edit” scaffold generated, and update the Edit view template to have the content below (which removes a few of the properties we don’t want to expose):

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

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Dinner</h2>

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

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

        <fieldset>
            <p>
                <label for="Title">Dinner Title:</label>
                <%= Html.TextBox("Title") %>
                <%= Html.ValidationMessage("Title", "*") %>
            </p>
            <p>
                <label for="EventDate">Event Date:</label>
                <%= Html.TextBox("EventDate", String.Format("{0:g}",
                                                            Model.EventDate)) %>
                <%= Html.ValidationMessage("EventDate", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextArea("Description") %>
                <%= Html.ValidationMessage("Description", "*")%>
            </p>
            <p>
                <label for="Address">Address:</label>
                <%= Html.TextBox("Address") %>
                <%= Html.ValidationMessage("Address", "*") %>
            </p>
            <p>
                <label for="Country">Country:</label>
                <%= Html.TextBox("Country") %>
                <%= Html.ValidationMessage("Country", "*") %>
            </p>
            <p>
                <label for="ContactPhone">Contact Phone #:</label>
                <%= Html.TextBox("ContactPhone") %>
                <%= Html.ValidationMessage("ContactPhone", "*") %>
            </p>

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

    <% } %>

</asp:Content>
```

When we run the application and request the `/Dinners/Edit/1` URL we will see the page in Figure 1-81:


Edit: .NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Edit/1
Live Search
Favorites
Edit: .NET Futures
Page
Safety
Tools
[Log On]
Home
About
Done
Local intranet | Protected Mode: Off
100%

My MVC Application

Edit Dinner

Dinner Title:
.NET Futures

Event Date:
12/6/2009 5:00 PM

Description:
Come talk about cool things coming with .NET

Address:
One Microsoft Way, Redmond WA

Country:
USA

Contact Phone #:
425-985-3648

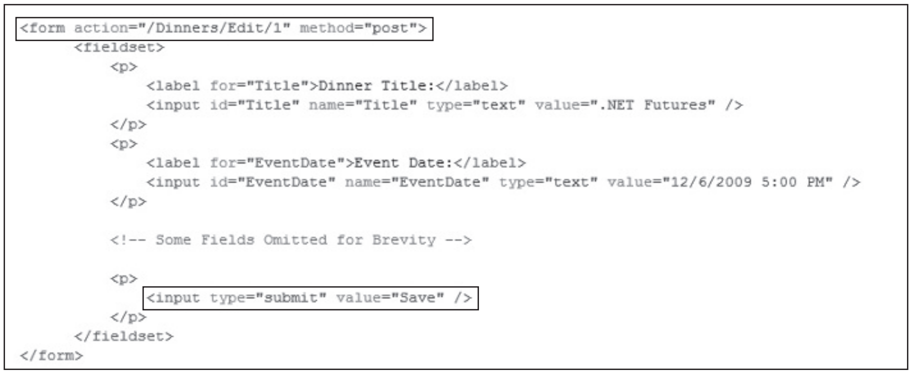
Latitude:
47.64312

Longitude:
-122.130609

Save

Figure 1-81

The HTML markup generated by our view looks like that below. It is standard HTML — with a `<form>` element that performs an HTTP `POST` to the `/Dinners/Edit/1` URL when the Save `<input type="submit"/>` button is pushed. A HTML `<input type="text"/>` element has been output for each editable property (Figure 1-82).



```
<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>
```

Figure 1-82

Html.BeginForm and Html.TextBox Html Helper Methods

Our `Edit.aspx` view template is using several “Html Helper” methods: `Html.ValidationSummary`, `Html.BeginForm`, `Html.TextBox`, and `Html.ValidationMessage`. In addition to generating HTML markup for us, these helper methods provide built-in error handling and validation support.

Html.BeginForm Helper Method

The `Html.BeginForm` helper method is what output the HTML `<form>` element in our markup. In our `Edit.aspx` view template, you’ll notice that we are applying a C# “using” statement when using this method. The open curly brace indicates the beginning of the `<form>` content, and the closing curly brace is what indicates the end of the `</form>` element:

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

   <fieldset>

         <!-- Fields Omitted for Brevity -->

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

   </fieldset>

<% } %>
```

Alternatively, if you find the “using” statement approach unnatural for a scenario like this, you can use a `Html.BeginForm` and `Html.EndForm` combination (which does the same thing):

```
<% Html.BeginForm();  %>
```

```
        <fieldset>
             <!-- Fields Omitted for Brevity -->

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

    <% Html.EndForm(); %>
```

Calling `Html.BeginForm` without any parameters will cause it to output a form element that does an HTTP-POST to the current request's URL. That is why our Edit view generates a `<form action="/Dinners/Edit/1" method="post">` element. We could have alternatively passed explicit parameters to `Html.BeginForm` if we wanted to post to a different URL.

Html.TextBox Helper Method

Our `Edit.aspx` view uses the `Html.TextBox` helper method to output `<input type="text"/>` elements:

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

The `Html.TextBox` method above takes a single parameter — which is being used to specify both the id/name attributes of the `<input type="text"/>` element to output, as well as the model property to populate the textbox value from. For example, the Dinner object we passed to the Edit view had a `"Title"` property value of `.NET Futures`, and so our `Html.TextBox("Title")` method call output is: `<input id="Title" name="Title" type="text" value=".NET Futures" />`.

Alternatively, we can use the first `Html.TextBox` parameter to specify the id/name of the element, and then explicitly pass in the value to use as a second parameter:

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

Often we'll want to perform custom formatting on the value that is output. The `String.Format` static method built into .NET is useful for these scenarios. Our `Edit.aspx` view template is using this to format the `EventDate` value (which is of type `DateTime`) so that it doesn't show seconds for the time:

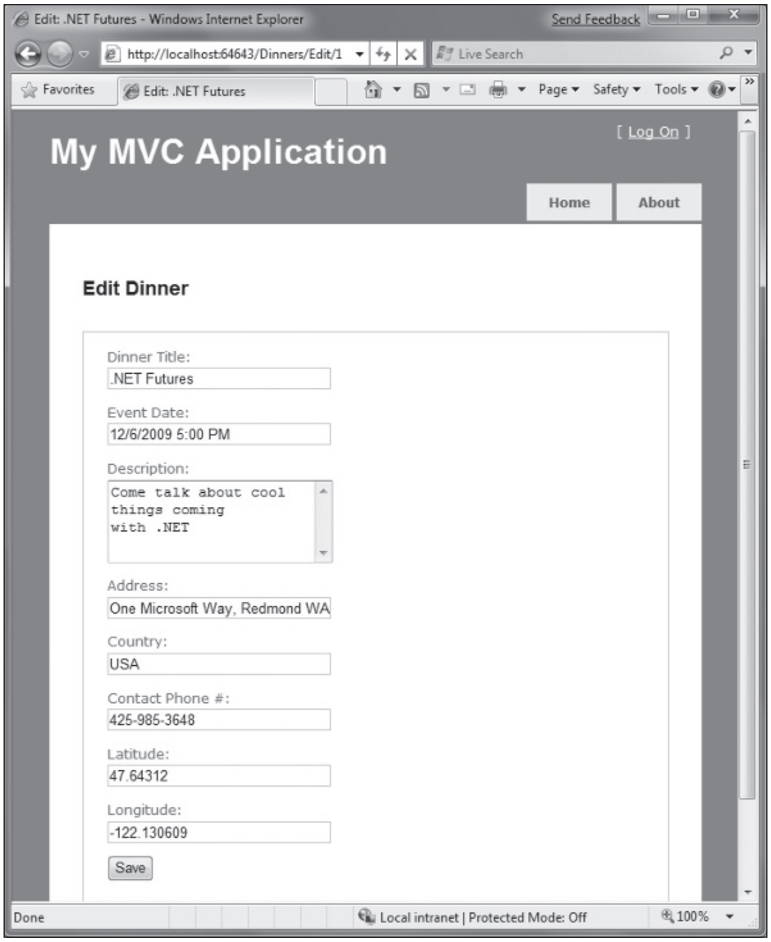
```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

A third parameter to `Html.TextBox` can optionally be used to output additional HTML attributes. The code-snippet below demonstrates how to render an additional `size="30"` attribute and a `class="mycssclass"` attribute on the `<input type="text"/>` element. Note how we are escaping the name of the class attribute using a `@` character because `class` is a reserved keyword in C#:

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

Implementing the HTTP-POST Edit Action Method

We now have the HTTP-GET version of our Edit action method implemented. When a user requests the `/Dinners/Edit/1` URL they receive an HTML page like the one in Figure 1-83:


Edit: .NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Edit/1
Live Search
Favorites
Edit: .NET Futures
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Save
Done
Local intranet | Protected Mode: Off
100%

Edit Dinner

Dinner Title: .NET Futures

Event Date: 12/6/2009 5:00 PM

Description:

```
Come talk about cool
things coming
with .NET
```

Address: One Microsoft Way, Redmond WA

Country: USA

Contact Phone #: 425-985-3648

Latitude: 47.64312

Longitude: -122.130609

Figure 1-83

Pressing the Save button causes a form post to the `/Dinners/Edit/1` URL, and submits the HTML `<input>` form values using the HTTP `POST` verb. Let's now implement the HTTP `POST` behavior of our edit action method — which will handle saving the dinner.

We'll begin by adding an overloaded `Edit` action method to our `DinnersController` that has an "AcceptVerbs" attribute on it that indicates it handles HTTP `POST` scenarios:

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

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

When the [AcceptVerbs] attribute is applied to overloaded action methods, ASP.NET MVC automatically handles dispatching requests to the appropriate action method depending on the incoming HTTP verb. HTTP POST requests to /Dinners/Edit/[id] URLs will go to the above Edit method, while all other HTTP verb requests to /Dinners/Edit/[id] URLs will go to the first Edit method we implemented (which did not have an [AcceptVerbs] attribute).

Why Differentiate via HTTP Verbs?

You might ask — why are we using a single URL and differentiating its behavior via the HTTP verb? Why not just have two separate URLs to handle loading and saving edit changes? For example: /Dinners/Edit/[id] to display the initial form and /Dinners/Save/[id] to handle the form post to save it?

The downside with publishing two separate URLs is that in cases where we post to /Dinners/Save/2, and then need to redisplay the HTML form because of an input error, the end user will end up having the /Dinners/Save/2 URL in their browser's address bar (since that was the URL the form posted to). If the end user bookmarks this redisplayed page to their browser favorites list, or copy/pastes the URL and emails it to a friend, they will end up saving a URL that won't work in the future (since that URL depends on post values).

By exposing a single URL (like: /Dinners/Edit/[id]) and differentiating the processing of it by HTTP verb, it is safe for end users to bookmark the edit page and/or send the URL to others.

Retrieving Form Post Values

There are a variety of ways we can access posted form parameters within our HTTP POST Edit method. One simple approach is to just use the Request property on the Controller base class to access the form collection and retrieve the posted values directly:

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

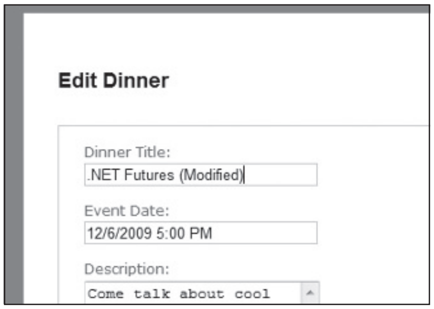
The approach in the previous code is a little verbose, though, especially once we add error handling logic.

A better approach for this scenario is to leverage the built-in *UpdateModel* helper method on the Controller base class. It supports updating the properties of an object we pass it using the incoming form parameters. It uses reflection to determine the property names on the object, and then automatically converts and assigns values to them based on the input values submitted by the client.

We could use the UpdateModel method to implement our HTTP-POST Edit action using this code:

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

    [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 });
    }
```

We can now visit the /Dinners/Edit/1 URL, and change the title of our dinner (Figure 1-84).


Edit Dinner

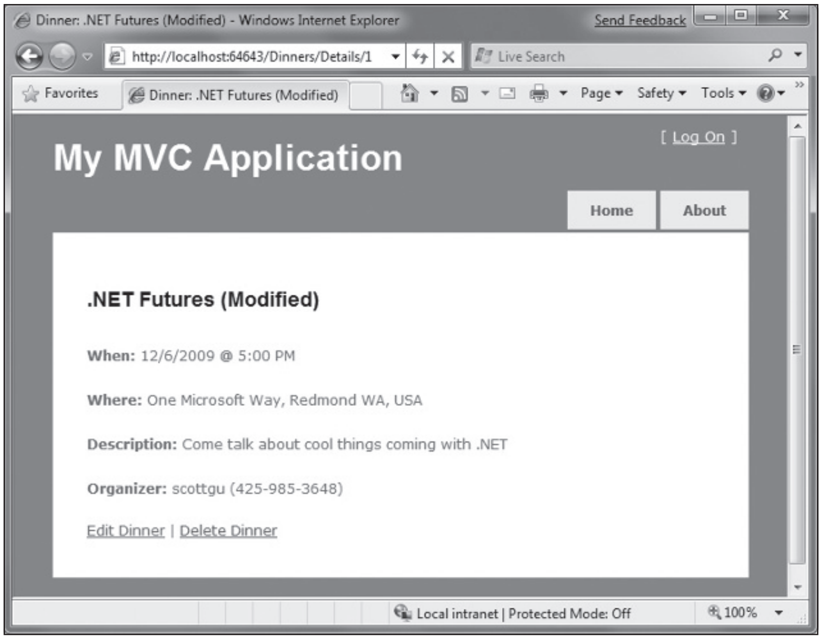
Dinner Title:
.NET Futures (Modified)

Event Date:
12/6/2009 5:00 PM

Description:
Come talk about cool

Figure 1-84

When we click the Save button, we'll perform a form post to our Edit action, and the updated values will be persisted in the database. We will then be redirected to the Details URL for the dinner (which will display the newly saved values like those in Figure 1-85).


Dinner: .NET Futures (Modified) - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Details/1
Live Search
Favorites
Dinner: .NET Futures (Modified)
Page
Safety
Tools
[Log On]
Home
About
Edit Dinner | Delete Dinner
Local intranet | Protected Mode: Off
100%

My MVC Application

.NET Futures (Modified)

When: 12/6/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

Figure 1-85

Handling Edit Errors

Our current HTTP-POST implementation works fine — except when there are errors.

When a user makes a mistake editing a form, we need to make sure that the form is redisplayed with an informative error message that guides them to fix it. This includes cases where an end-user posts incorrect input (for example: a malformed date string), as well as cases where the input format is valid but there is a business rule violation. When errors occur, the form should preserve the input data the user originally entered so that they don't have to refill their changes manually. This process should repeat as many times as necessary until the form successfully completes.

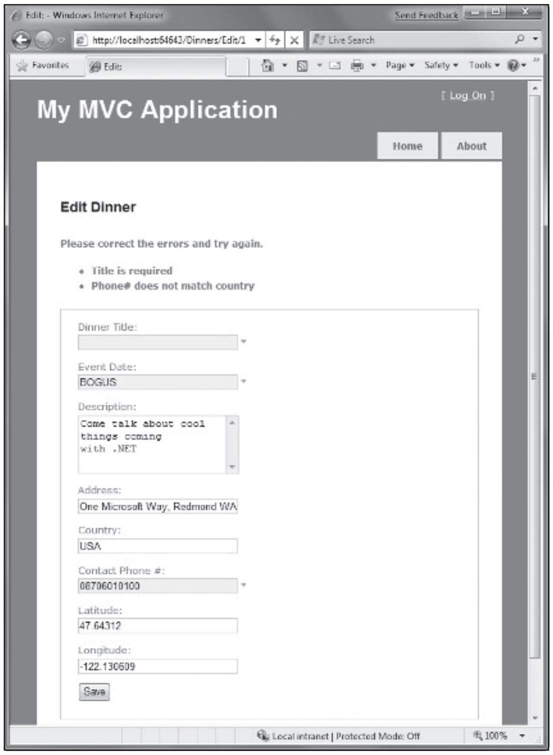
ASP.NET MVC includes some nice built-in features that make error handling and form redisplay easy. To see these features in action, let's update our `Edit` action method with the following code:

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

The previous code is similar to our previous implementation — except that we are now wrapping a try/catch error handling block around our work. If an exception occurs either when calling `UpdateModel`, or when we try and save the `DinnerRepository` (which will raise an exception if the `Dinner` object we are trying to save is invalid because of a rule violation), our catch error handling block will execute. Within it, we loop over any rule violations that exist in the `Dinner` object and add them to a `ModelState` object (which we'll discuss shortly). We then redisplay the view.

To see this working let's re-run the application, edit a dinner, and change it to have an empty Title, an Event Date of **BOGUS**, and use a UK phone number with a country value of USA. When we press the Save button our HTTP POST Edit method will not be able to save the dinner (because there are errors) and will redisplay the form in Figure 1-86.


Edit: - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Edit/1
Live Search
Favorites
Edit:
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Save
Local intranet | Protected Mode: Off
100%

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 with .NET

Address: One Microsoft Way, Redmond WA

Country: USA

Contact Phone #: 08706010100

Latitude: 47.64312

Longitude: -122.130609

Figure 1-86

Our application has a decent error experience. The text elements with the invalid input are highlighted in red, and validation error messages are displayed to the end user about them. The form is also preserving the input data the user originally entered — so that they don't have to refill anything.

How, you might ask, did this occur? How did the Title, Event Date, and Contact Phone textboxes highlight themselves in red and know to output the originally entered user values? And how did error messages get displayed in the list at the top? The good news is that this didn't occur by magic — rather it was because we used some of the built-in ASP.NET MVC features that make input validation and error handling scenarios easy.

Understanding ModelState and the Validation HTML Helper Methods

Controller classes have a `ModelState` property collection that provides a way to indicate that errors exist with a model object being passed to a View. Error entries within the `ModelState` collection identify the name of the model property with the issue (for example: "Title", "EventDate", or "ContactPhone"), and allow a human-friendly error message to be specified (for example: "Title is required").

The `UpdateModel()` helper method automatically populates the `ModelState` collection when it encounters errors while trying to assign form values to properties on the model object. For example, our Dinner object's `EventDate` property is of type `DateTime`. When the `UpdateModel` method was unable to assign the string value `BOGUS` to it in the previous scenario, the `UpdateModel` method added an entry to the `ModelState` collection indicating an assignment error had occurred with that property.

Developers can also write code to explicitly add error entries into the `ModelState` collection as we are doing below within our "catch" error handling block, which is populating the `ModelState` collection with entries based on the active Rule Violations in the Dinner object:

```
[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 Helper Integration with ModelState

HTML helper methods — like Html.TextBox — check the ModelState collection when rendering output. If an error for the item exists, they render the user-entered value and a CSS error class.

For example, in our "Edit" view we are using the Html.TextBox helper method to render the EventDate of our Dinner object:

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

When the view was rendered in the error scenario, the Html.TextBox method checked the ModelState collection to see if there were any errors associated with the "EventDate" property of our Dinner object. When it determined that there was an error, it rendered the submitted user input ("BOGUS") as the value, and added a CSS error class to the <input type="textbox"/> markup it generated:

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

You can customize the appearance of the CSS error class to look however you want. The default CSS error class — input-validation-error — is defined in the \content\site.css stylesheet and looks like the code below:

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

This CSS rule is what caused our invalid input elements to be highlighted, as in Figure 1-87.

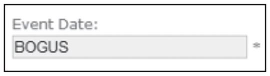

Event Date:
BOGUS *

Figure 1-87

Html .ValidationMessage Helper Method

The Html.ValidationMessage helper method can be used to output the ModelState error message associated with a particular model property:

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

The previous code outputs: <span class="field-validation-error"> The value 'BOGUS' is invalid</span>

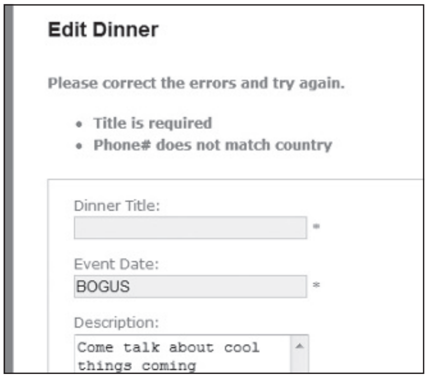
The Html.ValidationMessage helper method also supports a second parameter that allows developers to override the error text message that is displayed:

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

The previous code outputs: <span class="field-validation-error">*</span> instead of the default error text when an error is present for the EventDate property.

Html.ValidationSummary() Helper Method

The `Html.ValidationSummary` helper method can be used to render a summary error message, accompanied by a `<ul><li/></ul>` list of all detailed error messages in the `ModelState` collection (Figure 1-88):



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

Figure 1-88

The `Html.ValidationSummary` helper method takes an optional string parameter — which defines a summary error message to display above the list of detailed errors:

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

You can optionally use CSS to override what the error list looks like.

Using a AddRuleViolations Helper Method

Our initial HTTP-POST Edit implementation used a `foreach` statement within its catch block to loop over the `Dinner` object's Rule Violations and add them to the controller's `ModelState` collection:

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

        return View(dinner);
    }
```

We can make this code a little cleaner by adding a `ControllerHelpers` class to the NerdDinner project, and implement an `AddRuleViolations` extension method within it that adds a helper method to the ASP.NET MVC `ModelStateDictionary` class. This extension method can encapsulate the logic necessary to populate the `ModelStateDictionary` with a list of `RuleViolation` errors:

```
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);
       }
   }
}
```

We can then update our HTTP-POST Edit action method to use this extension method to populate the `ModelState` collection with our Dinner Rule Violations.

Complete Edit Action Method Implementations

The following code implements all of the controller logic necessary for our Edit scenario:

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

public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    return View(dinner);
}
//
// POST: /Dinners/Edit/2

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

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

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

        return View(dinner);
    }
}
```

The nice thing about our Edit implementation is that neither our Controller class nor our view template has to know anything about the specific validation or business rules being enforced by our Dinner model. We can add additional rules to our model in the future and *do not have to make any code changes* to our controller or view in order for them to be supported. This provides us with the flexibility to easily evolve our application requirements in the future with a minimum of code changes.

Implementing the HTTP-GET Create Action Method

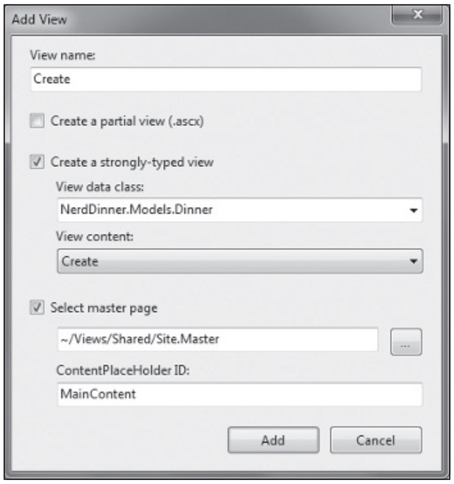
We've finished implementing the `Edit` behavior of our `DinnersController` class. Let's now move on to implement the `Create` support on it — which will enable users to add new dinners.

We'll begin by implementing the HTTP `GET` behavior of our create action method. This method will be called when someone visits the `/Dinners/Create` URL. Our implementation looks like:

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

public ActionResult Create() {

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

    return View(dinner);
}
```

The previous code creates a new Dinner object, and assigns its `EventDate` property to be one week in the future. It then renders a View that is based on the new Dinner object. Because we haven't explicitly passed a name to the `View` helper method, it will use the convention based default path to resolve the view template: `/Views/Dinners/Create.aspx`.

Let's now create this view template. We can do this by right-clicking within the Create action method and selecting the Add View context menu command. Within the Add View dialog we'll indicate that we are passing a `Dinner` object to the view template, and choose to auto-scaffold a Create template (Figure 1-89).



View name: Create

☐ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class: NerdDinner.Models.Dinner

View content: Create

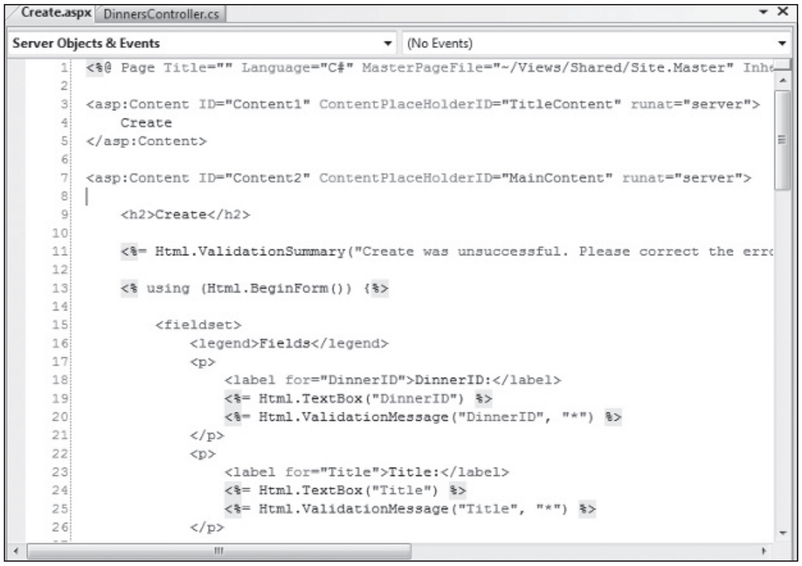
☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-89

When we click the Add button, Visual Studio will save a new scaffold-based `Create.aspx` view to the `\Views\Dinners` directory, and open it up within the IDE (Figure 1-90).



```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inhe

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Create
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errc

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

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="DinnerID">DinnerID:</label>
                <%= Html.TextBox("DinnerID") %>
                <%= Html.ValidationMessage("DinnerID", "*") %>
            </p>
            <p>
                <label for="Title">Title:</label>
                <%= Html.TextBox("Title") %>
                <%= Html.ValidationMessage("Title", "*") %>
            </p>
```

Figure 1-90

Let's make a few changes to the default "create" scaffold file that was generated for us, and modify it up to look like the code below:

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

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Host a Dinner</h2>

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

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

        <fieldset>
            <p>
                <label for="Title">Title:</label>
                <%= Html.TextBox("Title") %>
                <%= Html.ValidationMessage("Title", "*") %>
            </p>
            <p>
                <label for="EventDate">Event Date:</label>
                <%= Html.TextBox("EventDate") %>
                <%= Html.ValidationMessage("EventDate", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextArea("Description") %>
                <%= Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="Address">Address:</label>
                <%= Html.TextBox("Address") %>
                <%= Html.ValidationMessage("Address", "*") %>
            </p>
            <p>
                <label for="Country">Country:</label>
                <%= Html.TextBox("Country") %>
                <%= Html.ValidationMessage("Country", "*") %>
            </p>
            <p>
                <label for="ContactPhone">ContactPhone:</label>
                <%= Html.TextBox("ContactPhone") %>
                <%= Html.ValidationMessage("ContactPhone", "*") %>
            </p>
            <p>
                <label for="Latitude">Latitude:</label>
                <%= Html.TextBox("Latitude") %>
                <%= Html.ValidationMessage("Latitude", "*") %>
            </p>
            <p>
                <label for="Longitude">Longitude:</label>
                <%= Html.TextBox("Longitude") %>
                <%= Html.ValidationMessage("Longitude", "*") %>
            </p>
            <p>
                <input type="submit" value="Save" />
            </p>
        </fieldset>

    <% } %>

</asp:Content>
```

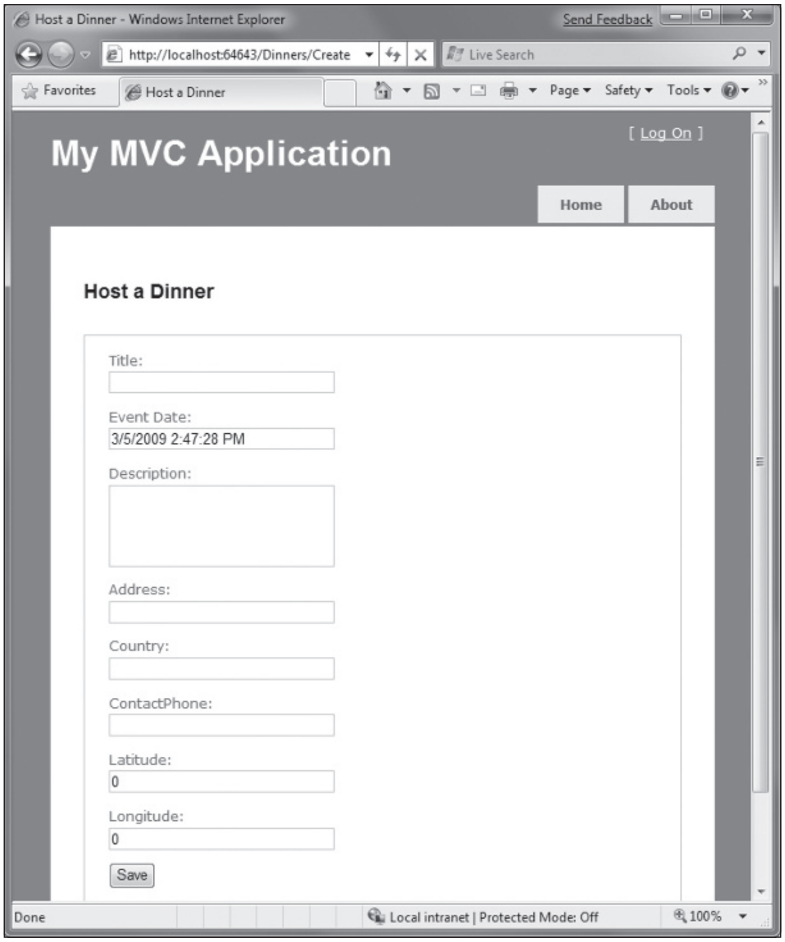
And now when we run our application and access the /Dinners/Create URL within the browser, it will render the UI as in Figure 1-91 from our Create action implementation.

Implementing the HTTP-POST Create Action Method

We have the HTTP-GET version of our Create action method implemented. When a user clicks the Save button, it performs a form post to the /Dinners/Create URL, and submits the HTML <input> form values using the HTTP POST verb.

Let's now implement the HTTP POST behavior of our create action method. We'll begin by adding an overloaded Create action method to our DinnersController that has an AcceptVerbs attribute on it that indicates it handles HTTP POST scenarios:

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

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection formValues) {
    ...
}
```


Host a Dinner - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Create
Live Search
Favorites
Host a Dinner
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Save
Done
Local intranet | Protected Mode: Off
100%

Host a Dinner

Title:

Event Date: 3/5/2009 2:47:28 PM

Description:

Address:

Country:

ContactPhone:

Latitude: 0

Longitude: 0

Figure 1-91

There are a variety of ways we can access the posted form parameters within our HTTP-POST-enabled `Create` method.

One approach is to create a new Dinner object and then use the `UpdateModel` helper method (as we did with the Edit action) to populate it with the posted form values. We can then add it to our `DinnerRepository`, persist it to the database, and redirect the user to our `Details` action to show the newly created dinner, using the following code:

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

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection formValues) {

    Dinner dinner = new Dinner();
```

```
        try {
            UpdateModel(dinner);

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

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

            return View(dinner);
        }
    }
```

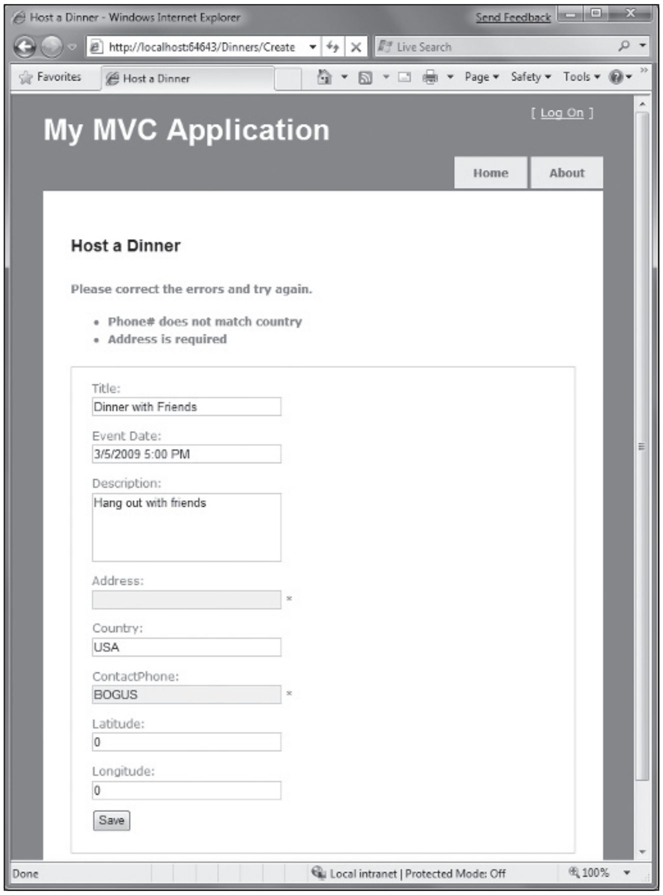
Alternatively, we can use an approach where we have our `Create` action method take a `Dinner` object as a method parameter. ASP.NET MVC will then automatically instantiate a new `Dinner` object for us, populate its properties using the form inputs, and pass it to our action method:

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

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

        if (ModelState.IsValid) {

            try {
                dinner.HostedBy = "SomeUser";

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

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

        return View(dinner);
    }
```

Our action method in the previous code verifies that the `Dinner` object has been successfully populated with the form post values by checking the `ModelState.IsValid` property. This will return false if there are input conversion issues (for example: a string of `"BOGUS"` for the `EventDate` property), and if there are any issues, our action method redisplays the form.

If the input values are valid, then the action method attempts to add and save the new dinner to the `DinnerRepository`. It wraps this work within a try/catch block and redisplays the form if there are any business rule violations (which would cause the `dinnerRepository.Save` method to raise an exception).

To see this error handling behavior in action, we can request the `/Dinners/Create` URL and fill out details about a new dinner. Incorrect input or values will cause the create form to be redisplayed with the errors highlighted in Figure 1-92.


Host a Dinner - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Create
Live Search
Favorites
Host a Dinner
Page ▾
Safety ▾
Tools ▾
[Log On]
Home
About
Save
Done
Local intranet | Protected Mode: Off
100%

My MVC Application

Host a Dinner

Please correct the errors and try again.

- Phone# does not match country
- Address is required

Title:
Dinner with Friends

Event Date:
3/5/2009 5:00 PM

Description:
Hang out with friends

Address:
*

Country:
USA

ContactPhone:
BOGUS *

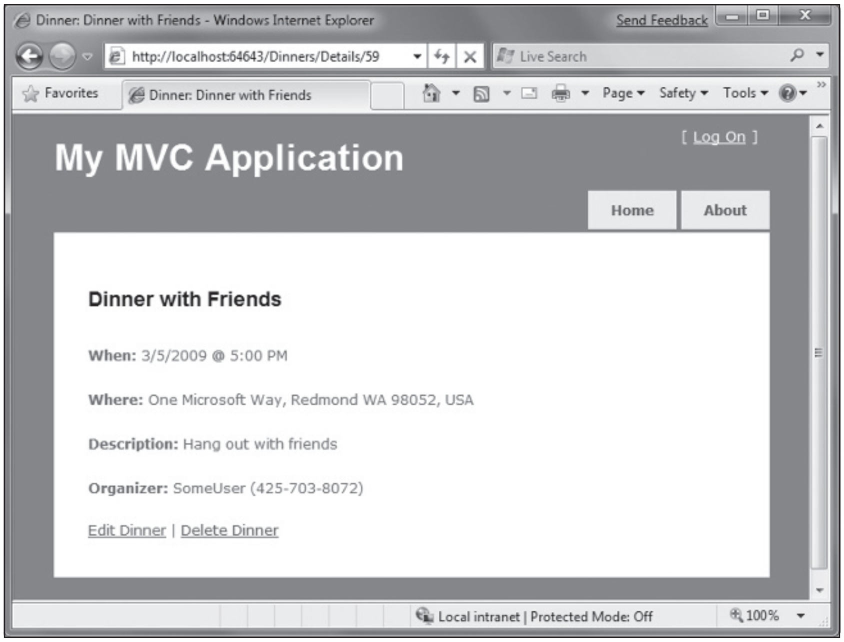
Latitude:
0

Longitude:
0

Figure 1-92

Notice how our Create form is honoring the exact same validation and business rules as our Edit form. This is because our validation and business rules were defined in the model, and were not embedded within the UI or controller of the application. This means we can later change/evolve our validation or business rules in a single place and have them apply throughout our application. We will not have to change any code within either our Edit or Create action methods to automatically honor any new rules or modifications to existing ones.

When we fix the input values and click the Save button again, our addition to the `DinnerRepository` will succeed, and a new dinner will be added to the database. We will then be redirected to the /*`Dinners/Details/[id]`* URL — where we will be presented with details about the newly created dinner (Figure 1-93):


Dinner: Dinner with Friends - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Details/59
Live Search
Favorites
Dinner: Dinner with Friends
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Local intranet | Protected Mode: Off
100%

Dinner with Friends

When: 3/5/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA 98052, USA

Description: Hang out with friends

Organizer: SomeUser (425-703-8072)

Edit Dinner | Delete Dinner

Figure 1-93

Implementing the HTTP-GET Delete Action Method

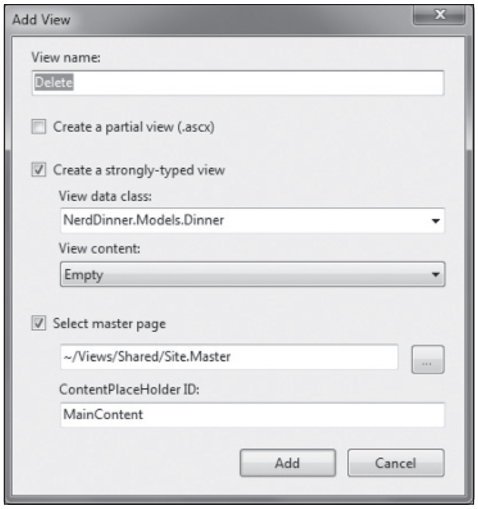
Let's now add "Delete" support to our `DinnersController`.

We'll begin by implementing the HTTP `GET` behavior of our delete action method. This method will get called when someone visits the `/Dinners/Delete/[id]` URL . Below is the implementation:

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

The action method attempts to retrieve the dinner to be deleted. If the dinner exists it renders a View based on the Dinner object. If the object doesn't exist (or has already been deleted) it returns a View that renders the "NotFound" view template we created earlier for our "Details" action method.

We can create the "Delete" view template by right-clicking within the Delete action method and selecting the "Add View" context menu command. Within the "Add View" dialog we'll indicate that we are passing a Dinner object to our view template as its model, and choose to create an empty template (Figure 1-94):



Add View

View name:
Delete

☐ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class:
NerdDinner.Models.Dinner

View content:
Empty

☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID:
MainContent

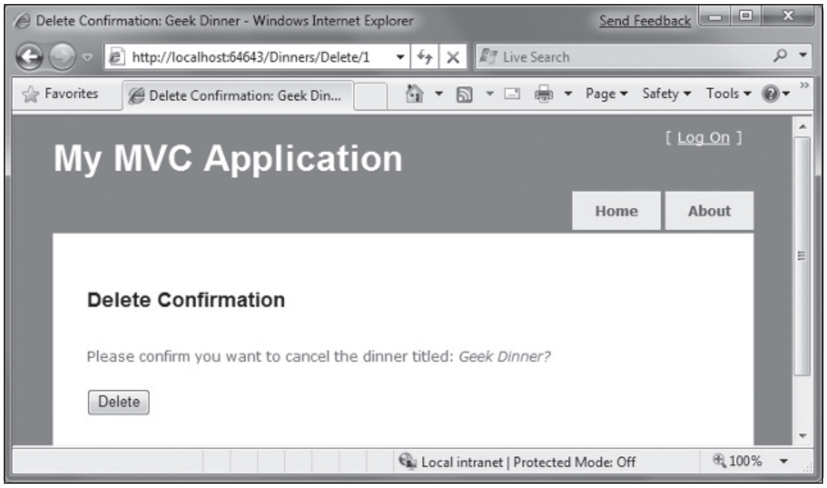
Figure 1-94

When we click the Add button, Visual Studio will add a new Delete.aspx view template file for us within our \Views\Dinners directory. We'll add some HTML and code to the template to implement a delete confirmation screen as shown below:

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

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>
        Delete Confirmation
    </h2>

    <div>
        <p>Please confirm you want to cancel the dinner titled:
        <i> <%=Html.Encode(Model.Title) %>? </i> </p>
    </div>

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

        <input name="confirmButton" type="submit" value="Delete" />

    <% } %>

</asp:Content>
```

The code above displays the title of the dinner to be deleted, and outputs a <form> element that does a POST to the `/Dinners/Delete/[id]` URL if the end user clicks the Delete button within it.

When we run our application and access the `/Dinners/Delete/[id]` URL for a valid Dinner object, it renders the UI as in Figure 1-95.


Delete Confirmation: Geek Dinner - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Delete/1
Live Search
Favorites
Delete Confirmation: Geek Din...
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Delete
Local intranet | Protected Mode: Off
100%

Delete Confirmation

Please confirm you want to cancel the dinner titled: *Geek Dinner?*

Figure 1-95

Why Are We Doing a `POST`

You might ask — why did we go through the effort of creating a `<form>` within our Delete confirmation screen? Why not just use a standard hyperlink to link to an action method that does the actual delete operation?

The reason is because we want to be careful to guard against web-crawlers and search engines discovering our URLs and inadvertently causing data to be deleted when they follow the links. HTTP-GET-based URLs are considered "safe" for them to access/crawl, and they are supposed to not follow HTTP-POST ones.

A good rule is to make sure you always put destructive or data modifying operations behind HTTP-POST requests.

Implementing the HTTP-POST Delete Action Method

We now have the HTTP-GET version of our Delete action method implemented that displays a delete confirmation screen. When an end user clicks the Delete button, it will perform a form post to the `/Dinners/Dinner/[id]` URL.

Let's now implement the HTTP `POST` behavior of the delete action method using the code that follows:

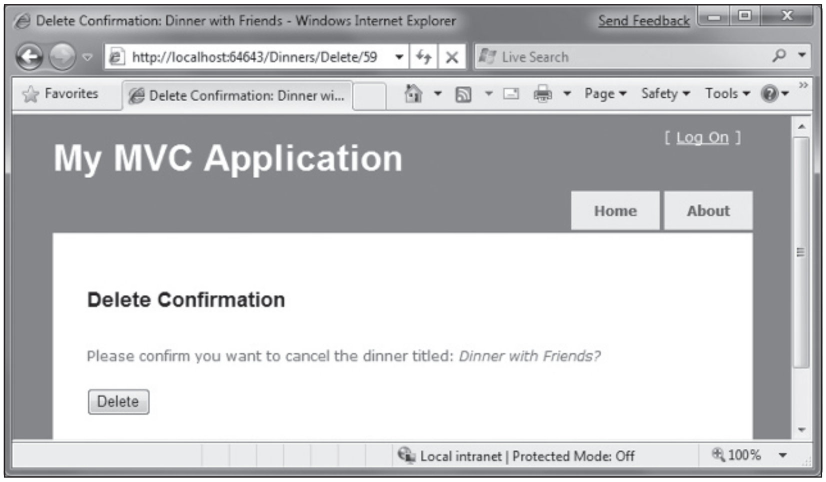
```
//
// HTTP POST: /Dinners/Delete/1

[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");
}
```

The HTTP-POST version of our `Delete` action method attempts to retrieve the Dinner object to delete. If it can't find it (because it has already been deleted) it renders our `"NotFound"` template. If it finds the dinner, it deletes it from the `DinnerRepository`. It then renders a "Deleted" template.

To implement the "Deleted" template, we'll right-click in the action method and choose the Add View context menu. We'll name our view **Deleted** and have it be an empty template (and not take a strongly typed model object). We'll then add some HTML content to it:

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

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Dinner Deleted</h2>

    <div>
        <p>Your dinner was successfully deleted.</p>
    </div>
    <div>
        <p><a href="/dinners">Click for Upcoming Dinners</a></p>
    </div>
</asp:Content>
```

And now when we run our application and access the `/Dinners/Delete/[id]` URL for a valid `Dinner` object, it will render our Dinner delete confirmation screen as in Figure 1-96.

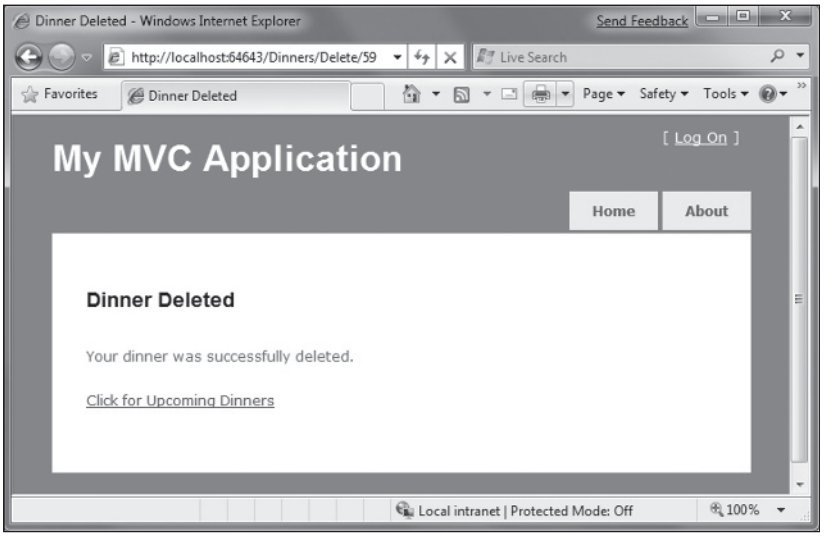

Delete Confirmation: Dinner with Friends - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Delete/59
Live Search
Favorites
Delete Confirmation: Dinner wi...
Page
Safety
Tools
[Log On]
My MVC Application
Home
About
Delete
Local intranet | Protected Mode: Off
100%

Delete Confirmation

Please confirm you want to cancel the dinner titled: *Dinner with Friends?*

Figure 1-96

When we click the Delete button, it will perform an HTTP-POST to the `/Dinners/Delete/[id]` URL, which will delete the dinner from our database, and display our "Deleted" view template (Figure 1-97).


Dinner Deleted - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Delete/59
Live Search
Favorites
Dinner Deleted
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Local intranet | Protected Mode: Off
100%

Dinner Deleted

Your dinner was successfully deleted.

Click for Upcoming Dinners

Figure 1-97

Model Binding Security

We've discussed two different ways to use the built-in model-binding features of ASP.NET MVC. The first using the `UpdateModel` method to update properties on an existing model object, and the second using ASP.NET MVC's support for passing model objects in as action method parameters. Both of these techniques are very powerful and extremely useful.

This power also brings with it responsibility. It is important to always be paranoid about security when accepting any user input, and this is also true when binding objects to form input. You should be careful to always HTML encode any user-entered values to avoid HTML and JavaScript injection attacks, and be careful of SQL injection attacks (note: we are using LINQ to SQL for our application, which automatically encodes parameters to prevent these types of attacks). You should never rely on client-side validation alone, and always employ server-side validation to guard against hackers attempting to send you bogus values.

One additional security item to make sure you think about when using the binding features of ASP.NET MVC is the scope of the objects you are binding. Specifically, you want to make sure you understand the security implications of the properties you are allowing to be bound, and make sure you only allow those properties that really should be updatable by an end user to be updated.

By default, the `UpdateModel` method will attempt to update all properties on the model object that match incoming form parameter values. Likewise, objects passed as action method parameters also, by default, can have all of their properties set via form parameters.

Locking Down Binding on a Per-Usage Basis

You can lock down the binding policy on a per-usage basis by providing an explicit *include list* of properties that can be updated. This can be done by passing an extra string array parameter to the UpdateModel method like the following code:

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

UpdateModel(dinner, allowedProperties);
```

Objects passed as action method parameters also support a [Bind] attribute that enables an include list of allowed properties to be specified like the code that follows:

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

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

Locking Down Binding on a Type Basis

You can also lock down the binding rules on a per-type basis. This allows you to specify the binding rules once and then have them apply in all scenarios (including both UpdateModel and action method parameter scenarios) across all controllers and action methods.

You can customize the per-type binding rules by adding a [Bind] attribute onto a type, or by registering it within the Global.asax file of the application (useful for scenarios where you don't own the type). You can then use the Bind attribute's Include and Exclude properties to control which properties are bindable for the particular class or interface.

We'll use this technique for the Dinner class in our NerdDinner application, and add a [Bind] attribute to it that restricts the list of bindable properties to the following:

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

Notice we are not allowing the RSVPs collection to be manipulated via binding, nor are we allowing the DinnerID or HostedBy properties to be set via binding. For security reasons we'll instead only manipulate these particular properties using explicit code within our action methods.

CRUD Wrap-Up

ASP.NET MVC includes a number of built-in features that help with implementing form posting scenarios. We used a variety of these features to provide CRUD UI support on top of our DinnerRepository.

We are using a model-focused approach to implement our application. This means that all our validation and business rule logic is defined within our model layer — and not within our controllers or views. Neither our Controller class nor our view templates know anything about the specific business rules being enforced by our Dinner model class.

This will keep our application architecture clean and make it easier to test. We can add additional business rules to our model layer in the future and *not have to make any code changes* to our Controller or View in order for them to be supported. This is going to provide us with a great deal of agility to evolve and change our application in the future.

Our `DinnersController` now enables dinner listings/details, as well as create, edit, and delete support. The complete code for the class can be found below:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // GET: /Dinners/

    public ActionResult Index() {

        var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        return View(dinners);
    }

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

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

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

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

    public ActionResult Edit(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);
        return View(dinner);
    }

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

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

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

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

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

        return View(dinner);
    }
}


//
// GET: /Dinners/Create

public ActionResult Create() {

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

//
// POST: /Dinners/Create

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

    if (ModelState.IsValid) {

        try {
            dinner.HostedBy = "SomeUser";

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

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

We've covered a number of form post scenarios, and discussed how to implement create, update and delete (CRUD) support. We'll now take our `DinnersController` implementation further and enable support for richer form editing scenarios. While doing this we'll discuss two approaches that can be used to pass data from controllers to views: `ViewData` and `ViewModel`.

Passing Data from Controllers to View Templates

One of the defining characteristics of the MVC pattern is the strict *separation of concerns* it helps enforce between the different components of an application. Models, Controllers, and Views each have well defined roles and responsibilities, and they communicate amongst each other in well-defined ways. This helps promote testability and code reuse.

When a Controller class decides to render an HTML response back to a client, it is responsible for explicitly passing to the view template all of the data needed to render the response. View templates ***should never*** perform any data retrieval or application logic — and should instead limit themselves to only having rendering code that is driven off of the model/data passed to it by the controller.

Right now the model data being passed by our DinnersController class to our view templates is simple and straightforward — a list of Dinner objects in the case of Index, and a single Dinner object in the case of Details, Edit, Create, and Delete. As we add more UI capabilities to our application, we are often going to need to pass more than just this data to render HTML responses within our view templates. For example, we might want to change the Country field within our Edit and Create views from being an HTML textbox to a dropdownlist. Rather than hard-code the dropdownlist of country names in the view template, we might want to generate it from a list of supported countries that we populate dynamically. We will need a way to pass both the Dinner object *and* the list of supported countries from our controller to our view templates.

Let's look at two ways we can accomplish this.

Using the ViewData Dictionary

The Controller base class exposes a ViewData dictionary property that can be used to pass additional data items from Controllers to Views.

For example, to support the scenario where we want to change the Country textbox within our Edit view from being an HTML textbox to a dropdownlist, we can update our Edit action method to pass (in addition to a Dinner object) a SelectList object that can be used as the model of a countries dropdownlist.

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

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    ViewData["Countries"] = new SelectList(PhoneValidator.Countries,
                                           dinner.Country);

    return View(dinner);
}
```

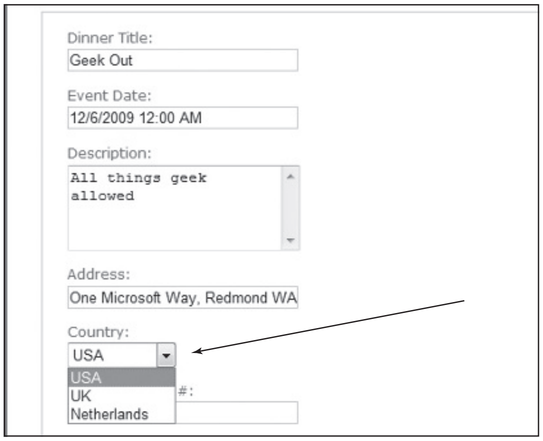
The constructor of the SelectList from the previous code is accepting a list of countries to populate the dropdownlist with, as well as the currently selected value.

We can then update our Edit.aspx view template to use the Html.DropDownList helper method instead of the Html.TextBox helper method we used previously:

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

The Html.DropDownList helper method in the previous line of code takes two parameters. The first is the name of the HTML form element to output. The second is the SelectList model we passed via the ViewData dictionary. We are using the C# "as" keyword to cast the type within the dictionary as a SelectList.

And now when we run our application and access the `/Dinners/Edit/1` URL within our browser, we'll see that our edit UI has been updated to display a drop-down list of countries instead of a textbox (Figure 1-98):



Dinner Title:
Geek Out

Event Date:
12/6/2009 12:00 AM

Description:
All things geek allowed

Address:
One Microsoft Way, Redmond WA

Country:
USA

USA
UK
Netherlands

#:

Figure 1-98

Because we also render the Edit view template from the HTTP-POST `Edit` method (in scenarios when errors occur), we'll want to make sure that we also update this method to add the `SelectList` to `ViewData` when the view template is rendered in error scenarios:

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

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

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

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

        ViewData["countries"] = new SelectList(PhoneValidator.Countries,
                                               dinner.Country);

        return View(dinner);
    }
}
```

And now our `DinnersController` edit scenario supports a drop-down list.

Using a ViewModel Pattern

The ViewData dictionary approach has the benefit of being fairly fast and easy to implement. Some developers don't like using string-based dictionaries, though, since typos can lead to errors that will not be caught at compile-time. The un-typed ViewData dictionary also requires using the "as" operator or casting when using a strongly typed language like C# in a view template.

An alternative approach that we could use is one often referred to as the *ViewModel* pattern. When using this pattern, we create strongly typed classes that are optimized for our specific view scenarios, and that expose properties for the dynamic values/content needed by our view templates. Our controller classes can then populate and pass these view-optimized classes to our view template to use. This enables type-safety, compile-time checking, and editor IntelliSense within view templates.

For example, to enable dinner form editing scenarios, we can create a DinnerFormViewModel class like the following code that exposes two strongly typed properties: a Dinner object and the SelectList model needed to populate the countries drop-down list:

```
public class DinnerFormViewModel {

    // Properties
    public Dinner     Dinner    { get; private set; }
    public SelectList Countries { get; private set; }

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

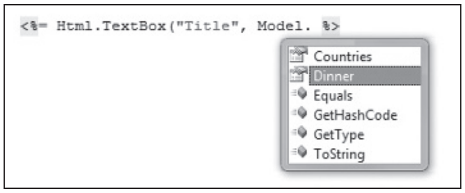
We can then update our Edit action method to create the DinnerFormViewModel using the Dinner object we retrieve from our repository, and then pass it to our view template:

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

public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

We'll then update our view template so that it expects a DinnerFormViewModel instead of a Dinner object by changing the Inherits attribute at the top of the edit.aspx page like so:

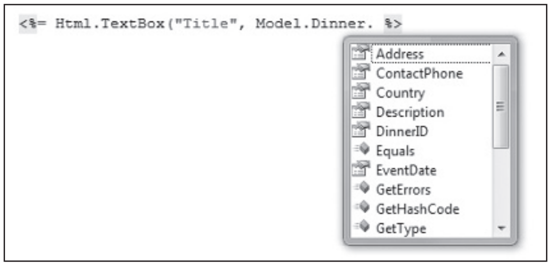
```
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerFormViewModel>
```

Once we do this, the IntelliSense of the Model property within our view template will be updated to reflect the object model of the DinnerFormViewModel type we are passing it (see Figures 1-99 and 1-100):



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

Figure 1-99



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

Figure 1-100

We can then update our view code to work off of it. Notice in the following code how we are not changing the names of the input elements we are creating (the form elements will still be named "Title", "Country") — but we are updating the HTML Helper methods to retrieve the values using the DinnerFormViewModel class:

```
<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>
```

We'll also update our Edit post method to use the DinnerFormViewModel class when rendering errors:

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

We can also update our `Create` action methods to reuse the exact same `DinnerFormViewModel` class to enable the countries dropdownlist within those as well. The following code is the HTTP-GET implementation:

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

    public ActionResult Create() {

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

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

The following code is the implementation of the HTTP-POST `Create` method:

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

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

        if (ModelState.IsValid) {

            try {
                dinner.HostedBy = "SomeUser";

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

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

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

And now both our Edit and Create screens support drop-down lists for picking the country.

Custom-Shaped ViewModel Classes

In the scenario above, our `DinnerFormViewModel` class directly exposes the `Dinner` model object as a property, along with a supporting `SelectList` model property. This approach works fine for scenarios where the HTML UI we want to create within our view template corresponds relatively closely to our domain model objects.

For scenarios where this isn't the case, one option that you can use is to create a custom-shaped `ViewModel` class whose object model is more optimized for consumption by the view — and which might look completely different from the underlying domain model object. For example, it could potentially expose different property names and/or aggregate properties collected from multiple model objects.

Custom-shaped `ViewModel` classes can be used both to pass data from controllers to views to render and to help handle form data posted back to a controller's action method. For this later scenario, you might have the action method update a `ViewModel` object with the form-posted data, and then use the `ViewModel` instance to map or retrieve an actual domain model object.

Custom-shaped `ViewModel` classes can provide a great deal of flexibility, and are something to investigate any time you find the rendering code within your view templates or the form-posting code inside your action methods starting to get too complicated. This is often a sign that your domain models don't cleanly correspond to the UI you are generating, and that an intermediate custom-shaped `ViewModel` class can help.

Partials and Master Pages

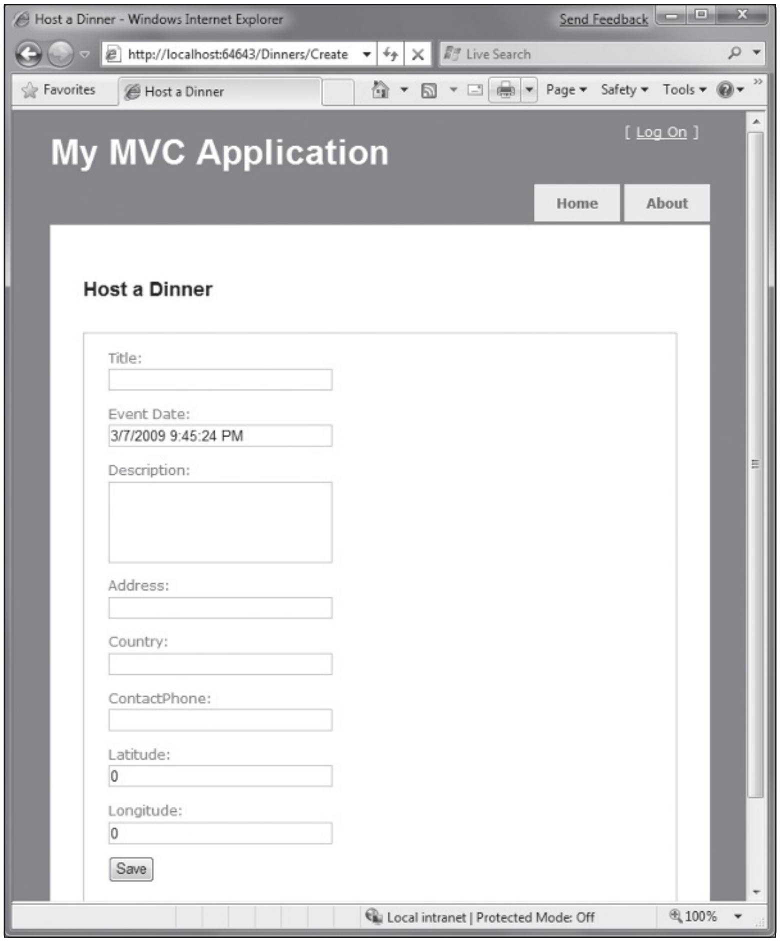
One of the design philosophies ASP.NET MVC embraces is the *Do Not Repeat Yourself* principle (commonly referred to as *DRY*). A DRY design helps eliminate the duplication of code and logic, which ultimately makes applications faster to build and easier to maintain.

We've already seen the DRY principle applied in several of our NerdDinner scenarios. A few examples: our validation logic is implemented within our model layer, which enables it to be enforced across both edit and create scenarios in our controller; we are reusing the `"NotFound"` view template across the `Edit`, `Details` and `Delete` action methods; we are using a convention-naming pattern with our view templates, which eliminates the need to explicitly specify the name when we call the `View` helper method; and we are reusing the `DinnerFormViewModel` class for both `Edit` and `Create` action scenarios.

Let's now look at ways we can apply the DRY Principle within our view templates to eliminate code duplication there as well.

Revisiting Our Edit and Create View Templates

Currently we are using two different view templates — `Edit.aspx` and `Create.aspx` — to display our Dinner form UI. A quick visual comparison of them highlights how similar they are. Figure 1-101 shows what the create form looks like:


Host a Dinner - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Create
Live Search
Favorites
Host a Dinner
Page
Safety
Tools
My MVC Application
[Log On]
Home
About
Save
Local intranet | Protected Mode: Off
100%

Host a Dinner

Title:

Event Date: 3/7/2009 9:45:24 PM

Description:

Address:

Country:

ContactPhone:

Latitude: 0

Longitude: 0

Figure 1-101

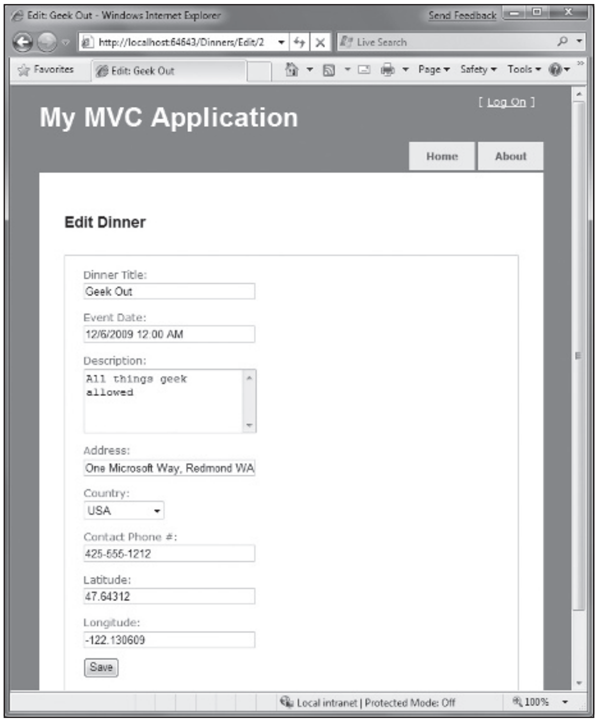
And Figure 1-102 is what our "Edit" form looks like.

Not much of a difference is there? Other than the title and header text, the form layout and input controls are identical.

If we open up the `Edit.aspx` and `Create.aspx` view templates, we'll find that they contain identical form layout and input control code. This duplication means we end up having to make changes twice anytime we introduce or change a new Dinner property — which is not good.

Using Partial View Templates

ASP.NET MVC supports the ability to define *partial view* templates that can be used to encapsulate view rendering logic for a sub-portion of a page. Partials provide a useful way to define view rendering logic once, and then reuse it in multiple places across an application.


Edit: Geek Out - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Edit/2
Live Search
Favorites
Edit: Geek Out
Page ▾
Safety ▾
Tools ▾
My MVC Application
[Log On]
Home
About
Save
Local intranet | Protected Mode: Off
100%

Edit Dinner

Dinner Title:
Geek Out

Event Date:
12/6/2009 12:00 AM

Description:
All things geek allowed

Address:
One Microsoft Way, Redmond WA

Country:
USA

Contact Phone #:
425-555-1212

Latitude:
47.64312

Longitude:
-122.130609

Figure 1-102

To help "DRY-up" our `Edit.aspx` and `Create.aspx` View template duplication, we can create a partial View template named `DinnerForm.ascx` that encapsulates the form layout and input elements common to both. We'll do this by right-clicking on our *\Views\Dinners* directory and choosing the Add ➪ View menu command shown in Figure 1-103:

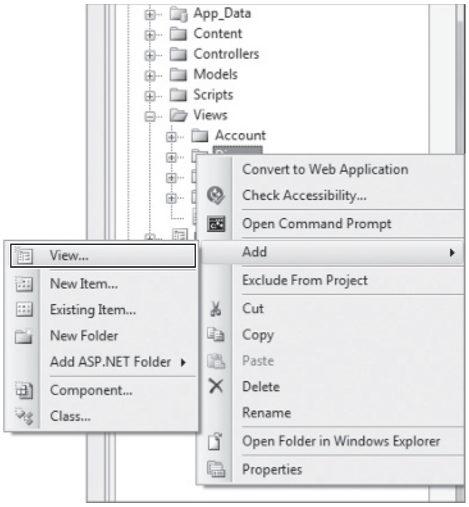
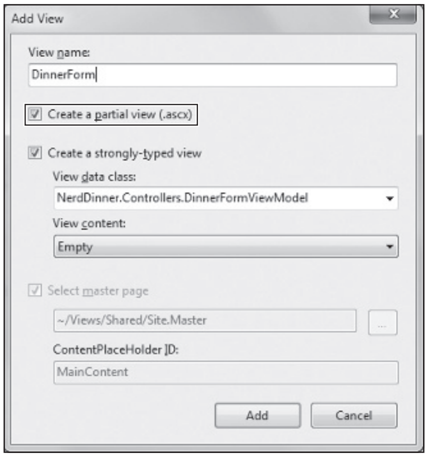


Figure 1-103

This will display the Add View dialog. We'll name the new view we want to create **DinnerForm**, select the "Create a partial view" checkbox on the dialog, and indicate that we will pass it a `DinnerFormViewModel` class (see Figure 1-104).



Add View

View name: DinnerForm

☑ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class: NerdDinner.Controllers.DinnerFormViewModel

View content: Empty

☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-104

When we click the Add button, Visual Studio will create a new `DinnerForm.ascx` view template for us within the `\Views\Dinners` directory.

We can then copy/paste the duplicate form layout/input control code from our `Edit.aspx/ Create.aspx` view templates into our new `DinnerForm.ascx` partial view template:

```
<%= 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>

<% } %>
```

We can then update our `Edit` and `Create` view templates to call the `DinnerForm` partial template and eliminate the form duplication. We can do this by calling `Html.RenderPartial("DinnerForm")` within our view templates:

Create.aspx

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

<asp:Content ID="Create" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Host a Dinner</h2>

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

</asp:Content>
```

Edit.aspx

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

<asp:Content ID="Edit" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Dinner</h2>

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

</asp:Content>
```

You can explicitly qualify the path of the partial template you want when calling `Html.RenderPartial` (for example: `~/Views/Dinners/DinnerForm.ascx`). In our previous code, though, we are taking advantage of the convention-based naming pattern within ASP.NET MVC, and just specifying DinnerForm as the name of the partial to render. When we do this, ASP.NET MVC will look first in the convention-based views directory (for DinnersController this would be /Views/Dinners). If it doesn't find the partial template there, it will then look for it in the /Views/Shared directory.

When Html.RenderPartial is called with just the name of the partial view, ASP.NET MVC will pass to the partial view the same Model and ViewData dictionary objects used by the calling view template. Alternatively, there are overloaded versions of Html.RenderPartial that enable you to pass an alternate Model object and/or ViewData dictionary for the partial view to use. This is useful for scenarios where you only want to pass a subset of the full Model/ViewModel.

Why <% %> Instead of <%= %>?

One of the subtle things you might have noticed with the previous code is that we are using a <% %> block instead of a <%= %> block when calling Html.RenderPartial.

<%= %> blocks in ASP.NET indicate that a developer wants to render a specified value (for example: <%= "Hello" %> would render "Hello"). <% %> blocks instead indicate that the developer wants to execute code, and that any rendered output within them must be done explicitly (for example: <% Response.Write("Hello"); %>).

The reason we are using a <% %> block with our previous Html.RenderPartial code is because the Html.RenderPartial method doesn't return a string, and instead outputs the content directly to the calling the View template's output stream. It does this for performance efficiency reasons, and by doing so, it avoids the need to create a (potentially very large) temporary string object. This reduces memory usage and improves overall application throughput.

One common mistake when using Html.RenderPartial is to forget to add a semicolon at the end of the call when it is within a <% %> block. For example, this code will cause a compiler error:

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

You instead need to write:

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

This is because <% %> blocks are self-contained code statements, and when using C# code statements, need to be terminated with a semicolon.

Using Partial View Templates to Clarify Code

We created the DinnerForm partial view template to avoid duplicating view rendering logic in multiple places. This is the most common reason to create partial view templates.

Sometimes it still makes sense to create partial views even when they are only being called in a single place. Very complicated view templates can often become much easier to read when their view rendering logic is extracted and partitioned into one or more well-named partial templates.

For example, consider the below code-snippet from the e file in our project (which we will be looking at shortly). The code is relatively straightforward to read — partly because the logic to display a login/logout link at the top right of the screen is encapsulated within the `LogOnUserControl` partial:

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

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

    <div id="menucontainer">

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

    </div>
</div>
```

Whenever you find yourself getting confused trying to understand the HTML/code markup within a view template, consider whether it wouldn't be clearer if some of it was extracted and refactored into well-named partial views.

Master Pages

In addition to supporting partial views, ASP.NET MVC also supports the ability to create *master page* templates that can be used to define the common layout and top-level HTML of a site. Content placeholder controls can then be added to the master page to identify replaceable regions that can be overridden or *filled in* by views. This provides a very effective (and DRY) way to apply a common layout across an application.

By default, new ASP.NET MVC projects have a master page template automatically added to them. This master page is named `Site.master` and lives within the `\Views\Shared\` folder as shown in Figure 1-105.

The default `Site.master` file looks like the following code. It defines the outer HTML of the site, along with a menu for navigation at the top. It contains two replaceable content placeholder controls — one for the title, and the other for where the primary content of a page should be replaced:

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">
   <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
   <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
</head>
```

```
<body>
    <div class="page">

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

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

            <div id="menucontainer">

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

            </div>
        </div>

        <div id="main">
            <asp:ContentPlaceHolder ID="MainContent" runat="server" />
        </div>
    </div>
</body>
</html>
```

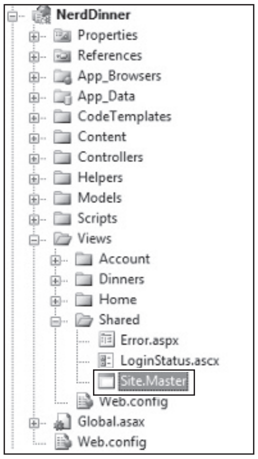


Figure 1-105

All of the view templates we've created for our NerdDinner application (`"List"`, `"Details"`, `"Edit"`, `"Create"`, `"NotFound"`, etc.) have been based on this `Site.master` template. This is indicated via the

MasterPageFile attribute that was added by default to the top <% @ Page %> directive when we created our views using the Add View dialog:

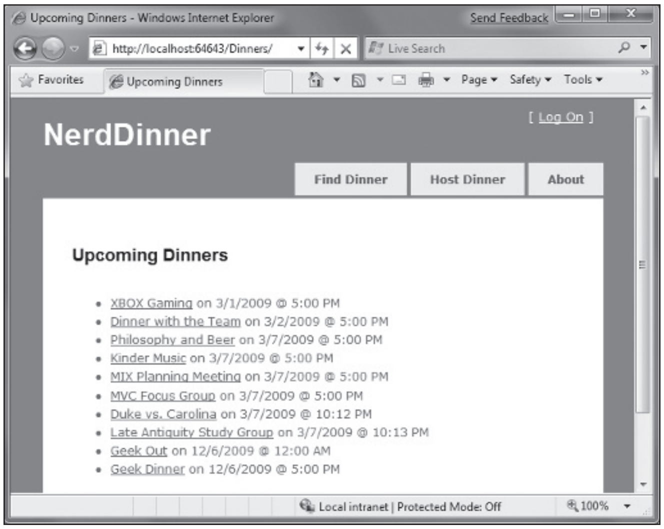
```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers
.DinnerViewModel>" MasterPageFile="~/Views/Shared/Site.Master" %>
```

What this means is that we can change the Site.master content, and have the changes automatically be applied and used when we render any of our view templates.

Let's update our Site.master's header section so that the header of our application is "NerdDinner" instead of "My MVC Application." Let's also update our navigation menu so that the first tab is "Find a Dinner" (handled by the HomeController's Index action method), and let's add a new tab called "Host a Dinner" (handled by the DinnersController's Create action method):

```
<div id="header">
    <div id="title">
        <h1>NerdDinner</h1>
    </div>

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

    <div id="menucontainer">
        <ul id="menu">
           <li><%= Html.ActionLink("Find Dinner", "Index", "Dinners")%></li>
           <li><%= Html.ActionLink("Host Dinner", "Create", "Dinners")%></li>
           <li><%= Html.ActionLink("About", "About", "Home")%></li>
        </ul>
    </div>
</div>
```

When we save the Site.master file and refresh our browser, we'll see our header changes show up across all views within our application. For example, see Figure 1-106.

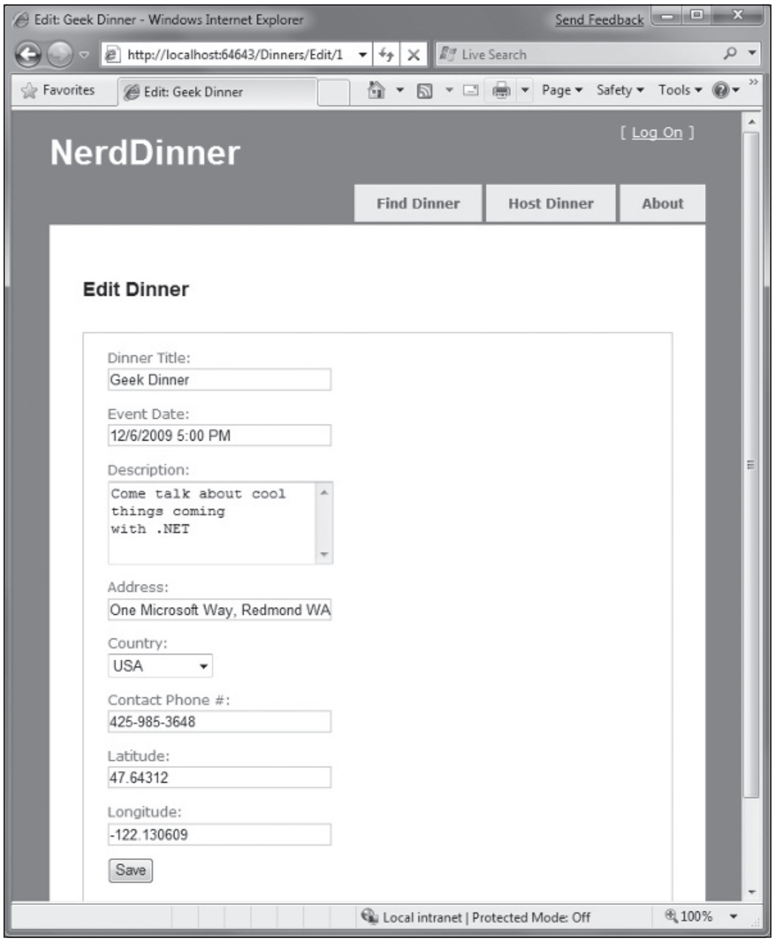


Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Philosophy and Beer on 3/7/2009 @ 5:00 PM
- Kinder Music on 3/7/2009 @ 5:00 PM
- MIX Planning Meeting on 3/7/2009 @ 5:00 PM
- MVC Focus Group on 3/7/2009 @ 5:00 PM
- Duke vs. Carolina on 3/7/2009 @ 10:12 PM
- Late Antiquity Study Group on 3/7/2009 @ 10:13 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- Geek Dinner on 12/6/2009 @ 5:00 PM

Figure 1-106

And with the `/Dinners/Edit/[id]` URL (Figure 1-107):



Edit Dinner

Dinner Title: Geek Dinner

Event Date: 12/6/2009 5:00 PM

Description: Come talk about cool things coming with .NET

Address: One Microsoft Way, Redmond WA

Country: USA

Contact Phone #: 425-985-3648

Latitude: 47.64312

Longitude: -122.130609

Figure 1-107

Partials and master pages provide very flexible options that enable you to cleanly organize views. You'll find that they help you avoid duplicating view content/code, and make your view templates easier to read and maintain.

Paging Support

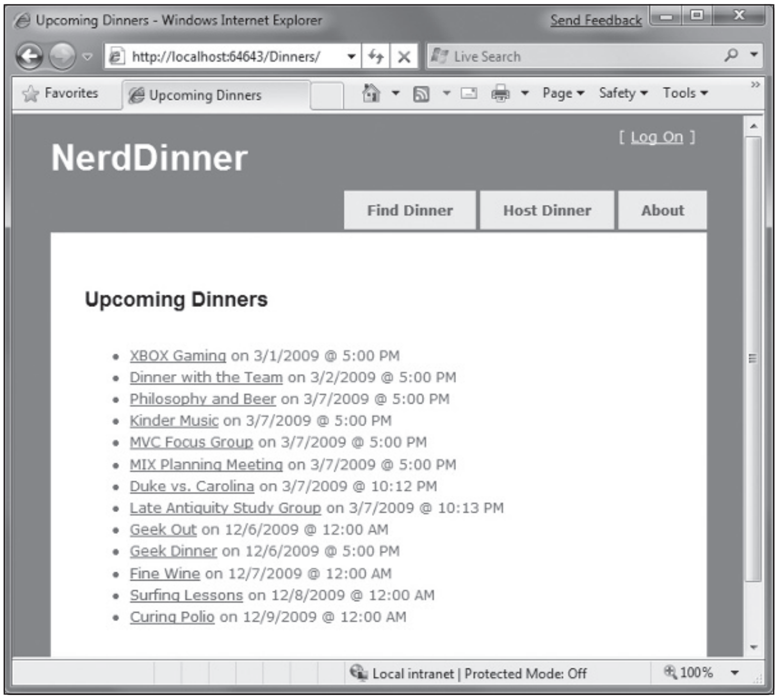
If our site is successful, it will have thousands of upcoming dinners. We need to make sure that our UI scales to handle all of these dinners and allows users to browse them. To enable this, we'll add paging support to our `/Dinners` URL so that instead of displaying thousands of dinners at once, we'll only display 10 upcoming dinners at a time — and allow end users to page back and forward through the entire list in an SEO friendly way.

Index() Action Method Recap

The `Index` action method within our `DinnersController` class currently looks like the following code:

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

public ActionResult Index() {

    var dinners = dinnerRepository.FindUpcomingDinners().ToList();

    return View(dinners);
}
```

When a request is made to the `/Dinners` URL, it retrieves a list of all upcoming dinners and then renders a listing of all of them (Figure 1-108):



Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Philosophy and Beer on 3/7/2009 @ 5:00 PM
- Kinder Music on 3/7/2009 @ 5:00 PM
- MVC Focus Group on 3/7/2009 @ 5:00 PM
- MIX Planning Meeting on 3/7/2009 @ 5:00 PM
- Duke vs. Carolina on 3/7/2009 @ 10:12 PM
- Late Antiquity Study Group on 3/7/2009 @ 10:13 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- Geek Dinner on 12/6/2009 @ 5:00 PM
- Fine Wine on 12/7/2009 @ 12:00 AM
- Surfing Lessons on 12/8/2009 @ 12:00 AM
- Curing Polio on 12/9/2009 @ 12:00 AM

Figure 1-108

Understanding IQueryable<T>

`IQueryable<T>` is an interface that was introduced with LINQ in .NET 3.5. It enables powerful *deferred execution* scenarios that we can take advantage of to implement paging support.

In our DinnerRepository in the following code we are returning an IQueryable<Dinner> sequence from our FindUpcomingDinners method:

```
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;
    }
```

The IQueryable<Dinner> object returned by our FindUpcomingDinners method encapsulates a query to retrieve Dinner objects from our database using LINQ to SQL. Importantly, it won't execute the query against the database until we attempt to access/iterate over the data in the query, or until we call the ToListmethod on it. The code calling our FindUpcomingDinners method can optionally choose to add additional "chained" operations/filters to the IQueryable<Dinner> object before executing the query. LINQ to SQL is then smart enough to execute the combined query against the database when the data is requested.

To implement paging logic, we can update our Index action method so that it applies additional Skip and Take operators to the returned IQueryable<Dinner> sequence before calling ToList on it:

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

public ActionResult Index() {

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

    return View(paginatedDinners);
}
```

The above code skips over the first 10 upcoming dinners in the database, and then returns 20 dinners. LINQ to SQL is smart enough to construct an optimized SQL query that performs this skipping logic in the SQL database — and not in the web server. This means that even if we have millions of upcoming dinners in the database, only the 10 we want will be retrieved as part of this request (making it efficient and scalable).

Adding a "page" Value to the URL

Instead of hard-coding a specific page range, we'll want our URLs to include a *page* parameter that indicates which Dinner range a user is requesting.

Using a Querystring Value

The code that follows demonstrates how we can update our `Index` action method to support a querystring parameter and enable URLs like `/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);
}
```

The `Index` action method in the previous code has a parameter named `page`. The parameter is declared as a nullable integer. This means that the `/Dinners?page=2` URL will cause a value of "2" to be passed as the parameter value. The `/Dinners` URL (without a querystring value) will cause a null value to be passed.

We are multiplying the page value by the page size (in this case 10 rows) to determine how many dinners to skip over. We are using the C# "coalescing" operator (??) which is useful when dealing with nullable types. The previous code assigns page the value of 0 if the page parameter is null.

Using Embedded URL Values

An alternative to using a querystring value would be to embed the page parameter within the actual URL itself. For example: `/Dinners/Page/2` or `/Dinners/2`. ASP.NET MVC includes a powerful URL routing engine that makes it easy to support scenarios like this.

We can register custom routing rules that map any incoming URL or URL format to any controller class or action method we want. All we need to do is to open the `Global.asax` file within our project (Figure 1-109).

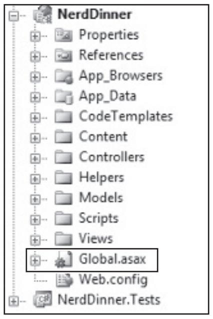


Figure 1-109

And then register a new mapping rule using the `MapRoute` helper method as in the first call to `routes.MapRoute` that follows:

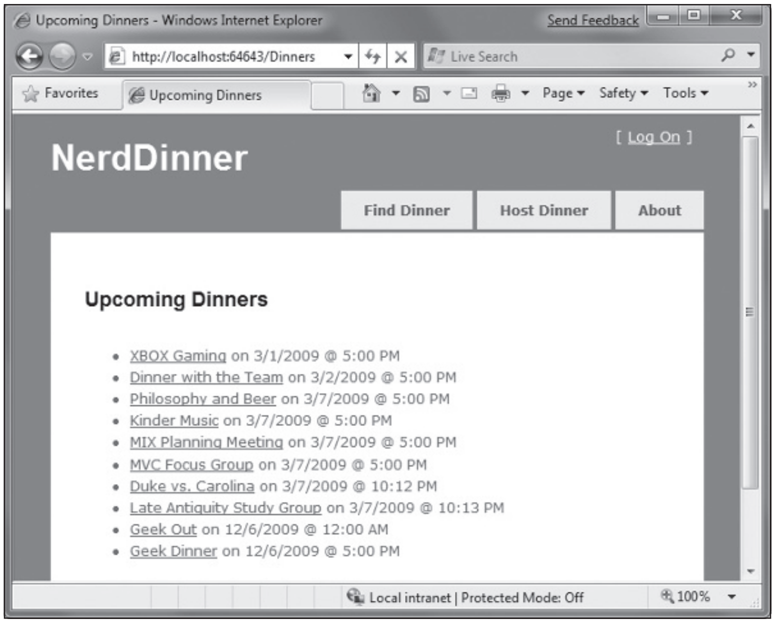
```
public void RegisterRoutes(RouteCollection routes) {

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

    routes.MapRoute(
        "UpcomingDinners",
        "Dinners/Page/{page}",
        new { controller = "Dinners", action = "Index" }
    );

    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);
}
```

In the previous code, we are registering a new routing rule named `"UpcomingDinners"`. We are indicating it has the URL format `"Dinners/Page/{page}"` — where *`{page}`* is a parameter value embedded within the URL. The third parameter to the `MapRoute` method indicates that we should map URLs that match this format to the `Index` action method on the `DinnersController` class.

We can use the exact same `Index` code we had before with our Querystring scenario — except now our `page` parameter will come from the URL and not the 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);
}
```

And now when we run the application and type in **`/Dinners`**, we'll see the first 10 upcoming dinners, as shown in Figure 1-110.

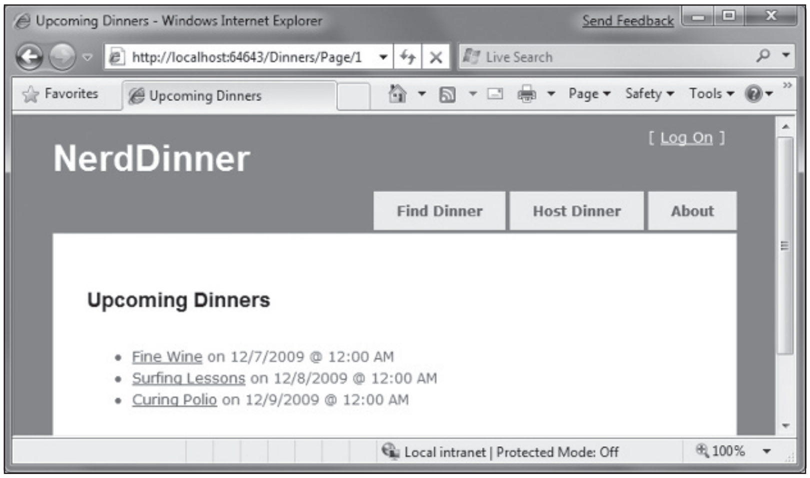


Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Philosophy and Beer on 3/7/2009 @ 5:00 PM
- Kinder Music on 3/7/2009 @ 5:00 PM
- MIX Planning Meeting on 3/7/2009 @ 5:00 PM
- MVC Focus Group on 3/7/2009 @ 5:00 PM
- Duke vs. Carolina on 3/7/2009 @ 10:12 PM
- Late Antiquity Study Group on 3/7/2009 @ 10:13 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- Geek Dinner on 12/6/2009 @ 5:00 PM

Figure 1-110

And when we type in **`/Dinners/Page/1`**, we'll see the next page of dinners (Figure 1-111):


Upcoming Dinners - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Page/1
Live Search
Favorites
Upcoming Dinners
Page
Safety
Tools
[Log On]
NerdDinner
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

Upcoming Dinners

- Fine Wine on 12/7/2009 @ 12:00 AM
- Surfing Lessons on 12/8/2009 @ 12:00 AM
- Curing Polio on 12/9/2009 @ 12:00 AM

Figure 1-111

Adding Page Navigation UI

The last step to complete our paging scenario will be to implement "next" and "previous" navigation UI within our view template to enable users to easily skip over the Dinner data.

To implement this correctly, we'll need to know the total number of Dinners in the database, as well as how many pages of data this translates to. We'll then need to calculate whether the currently requested "page" value is at the beginning or end of the data, and show or hide the "previous" and "next" UI accordingly. We could implement this logic within our Index action method. Alternatively, we can add a helper class to our project that encapsulates this logic in a more reusable way.

The following code is a simple PaginatedList helper class that derives from the List<T> collection class built into the .NET Framework. It implements a reusable collection class that can be used to paginate any sequence of IQueryable data. In our NerdDinner application we'll have it work over IQueryable<Dinner> results, but it could just as easily be used against IQueryable<Product> or IQueryable<Customer> results in other application scenarios:

```
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);
        }
    }
}
```

Notice in the previous code how it calculates and then exposes properties like PageIndex, PaegeSize, TotalCount, and TotalPages. It also then exposes two helper properties HasPreviousPage and HasNextPage that indicate whether the page of data in the collection is at the beginning or end of the original sequence. The above code will cause two SQL queries to be run — the first to retrieve the count of the total number of Dinner objects (this doesn't return the objects — rather it performs a SELECT COUNT statement that returns an integer), and the second to retrieve just the rows of data we need from our database for the current page of data.

We can then update our DinnersController.Index helper method to create a PaginatedList<Dinner> from our DinnerRepository.FindUpcomingDinners result, and pass it to our view template:

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

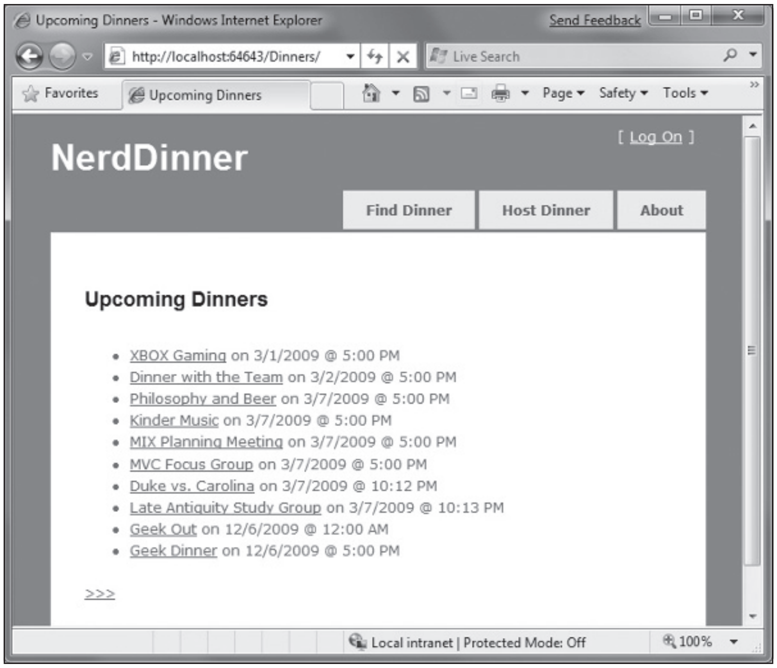
We can then update the `\Views\Dinners\Index.aspx` view template to inherit from `ViewPage<NerdDinner.Helpers.PaginatedList<Dinner>>` instead of `ViewPage<IEnumerable<Dinner>>`, and then add the following code to the bottom of our view template to show or hide next and previous navigation UI:

```
<% if (Model.HasPreviousPage) { %>

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

<% } %>

<% if (Model.HasNextPage) { %>

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

<% } %>
```

Notice, in the previous code, how we are using the `Html.RouteLink` helper method to generate our hyperlinks. This method is similar to the `Html.ActionLink` helper method we've used previously. The difference is that we are generating the URL using the `"UpcomingDinners"` routing rule we set up within our `Global.asax` file. This ensures that we'll generate URLs to our `Index` action method that have the format: *`/Dinners/Page/{page}`* — where the *`{page}`* value is a variable we are providing above based on the current `PageIndex`.

And now when we run our application again, we'll see 10 dinners at a time in our browser, as shown in Figure 1-112.

We also have <<< and >>> navigation UI at the bottom of the page that allows us to skip forwards and backwards over our data using search-engine-accessible URLs (Figure 1-113).

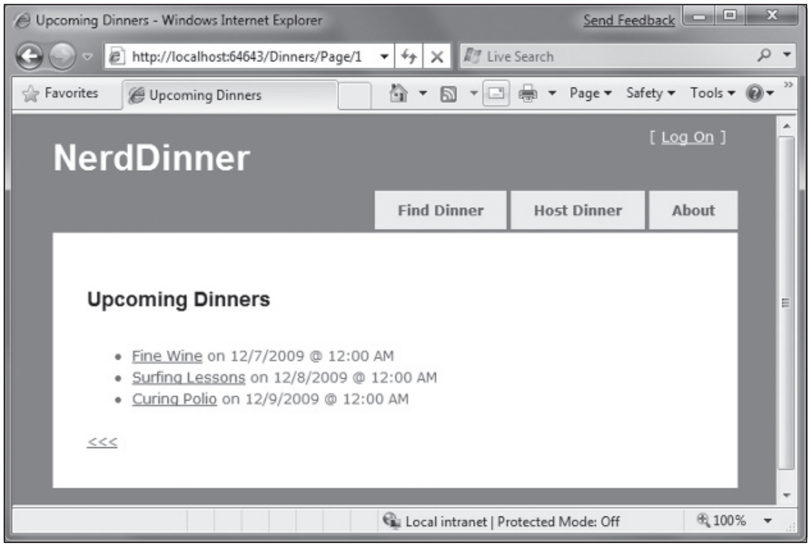


Upcoming Dinners

- XBOX Gaming on 3/1/2009 @ 5:00 PM
- Dinner with the Team on 3/2/2009 @ 5:00 PM
- Philosophy and Beer on 3/7/2009 @ 5:00 PM
- Kinder Music on 3/7/2009 @ 5:00 PM
- MIX Planning Meeting on 3/7/2009 @ 5:00 PM
- MVC Focus Group on 3/7/2009 @ 5:00 PM
- Duke vs. Carolina on 3/7/2009 @ 10:12 PM
- Late Antiquity Study Group on 3/7/2009 @ 10:13 PM
- Geek Out on 12/6/2009 @ 12:00 AM
- Geek Dinner on 12/6/2009 @ 5:00 PM

>>>

Figure 1-112


Upcoming Dinners - Windows Internet Explorer
Send Feedback
http://localhost:64643/Dinners/Page/1
Live Search
Favorites
Upcoming Dinners
Page
Safety
Tools
[Log On]
NerdDinner
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

Upcoming Dinners

- Fine Wine on 12/7/2009 @ 12:00 AM
- Surfing Lessons on 12/8/2009 @ 12:00 AM
- Curing Polio on 12/9/2009 @ 12:00 AM

<<<

Figure 1-113

Understanding the Implications of IQueryable <T>

IQueryable<T> is a very powerful feature that enables a variety of interesting deferred execution scenarios (like paging and composition-based queries). As with all powerful features, you want to be careful with how you use it and make sure it is not abused.

It is important to recognize that returning an IQueryable<T> result from your repository enables calling code to append on chained operator methods to it and so participate in the ultimate query execution. If you do not want to provide calling code this ability, then you should return back IList<T>, List<T> or IEnumerable<T> results — which contain the results of a query that has already executed.

For pagination scenarios, this would require you to push the actual data pagination logic into the repository method being called. In this scenario, we might update our FindUpcomingDinners finder method to have a signature that either returned a PaginatedList:

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

or returned an IList<Dinner>, and use a totalCount out param to return the total count of Dinners:

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

Authentication and Authorization

Right now our NerdDinner application grants anyone visiting the site the ability to create and edit the details of any dinner. Let's change this so that users need to register and log in to the site to create new dinners, and add a restriction so that only the user who is hosting a dinner can edit it later.

To enable this we'll use authentication and authorization to secure our application.

Understanding Authentication and Authorization

Authentication is the process of identifying and validating the identity of a client accessing an application. Put more simply, it is about identifying *who* the end user is when they visit a website.

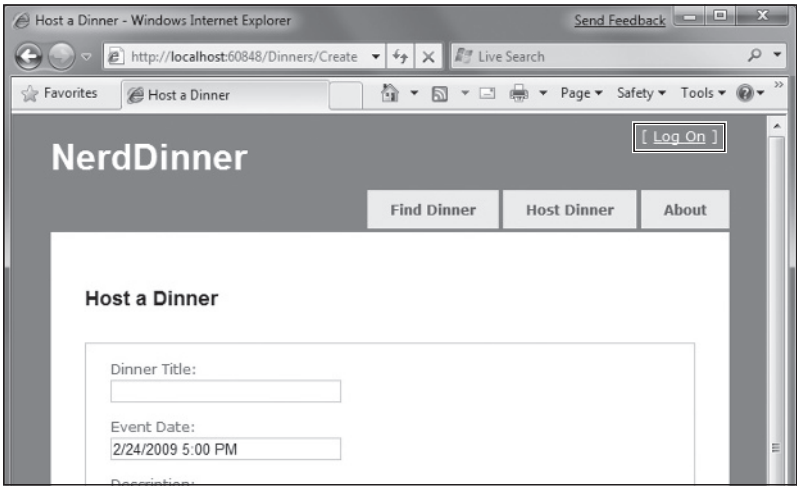
ASP.NET supports multiple ways to authenticate browser users. For Internet web applications, the most common authentication approach used is called *Forms Authentication*. Forms Authentication enables a developer to author an HTML login form within their application and then validate the username/password an end user submits against a database or other password credential store. If the username/password combination is correct, the developer can then ask ASP.NET to issue an encrypted HTTP cookie to identify the user across future requests. We'll be using forms authentication with our NerdDinner application.

Authorization is the process of determining whether an authenticated user has permission to access a particular URL/resource or to perform some action. For example, within our NerdDinner application we'll want to authorize only users who are logged in to access the `/Dinners/Create` URL and create new dinners. We'll also want to add authorization logic so that only the user who is hosting a dinner can edit it — and deny edit access to all other users.

Forms Authentication and the AccountController

The default Visual Studio project template for ASP.NET MVC automatically enables forms authentication when new ASP.NET MVC applications are created. It also automatically adds a pre-built account login implementation to the project — which makes it really easy to integrate security within a site.

The default `Site.master` master page displays a [Log On] link (shown in Figure 1-114) at the top right of the site when the user accessing it is not authenticated:



Dinner Title:

Event Date: 2/24/2009 5:00 PM

Description:

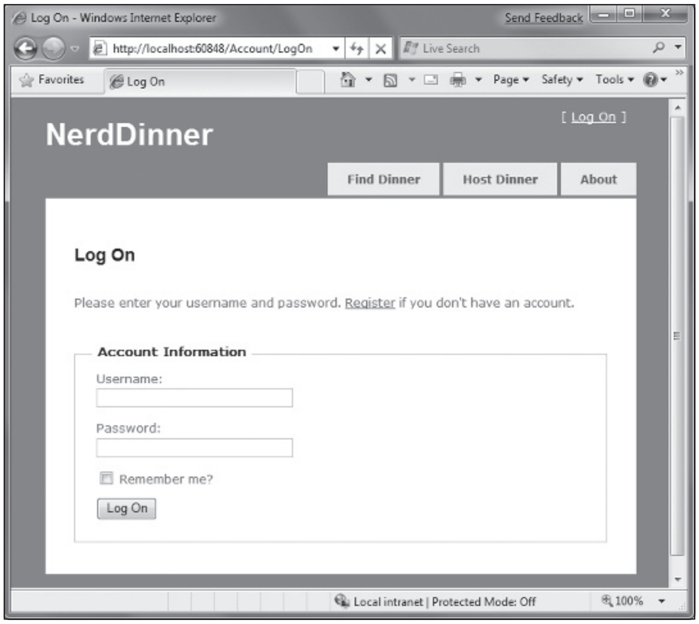
Figure 1-114

Clicking the [Log On] link takes a user to the `/Account/LogOn` URL (Figure 1-115)

Visitors who haven't registered can do so by clicking the Register link — which will take them to the `/Account/Register` URL and allow them to enter account details (Figure 1-116).

Clicking the Register button will create a new user within the ASP.NET Membership system, and authenticate the user onto the site using forms authentication.

When a user is logged in, the `Site.master` changes the top right of the page to output a "Welcome [username]!" message and renders a [Log Off] link instead of a [Log On] one. Clicking the [Log Off] link logs out the user (Figure 1-117).



Log On

Please enter your username and password. Register if you don't have an account.

Account Information

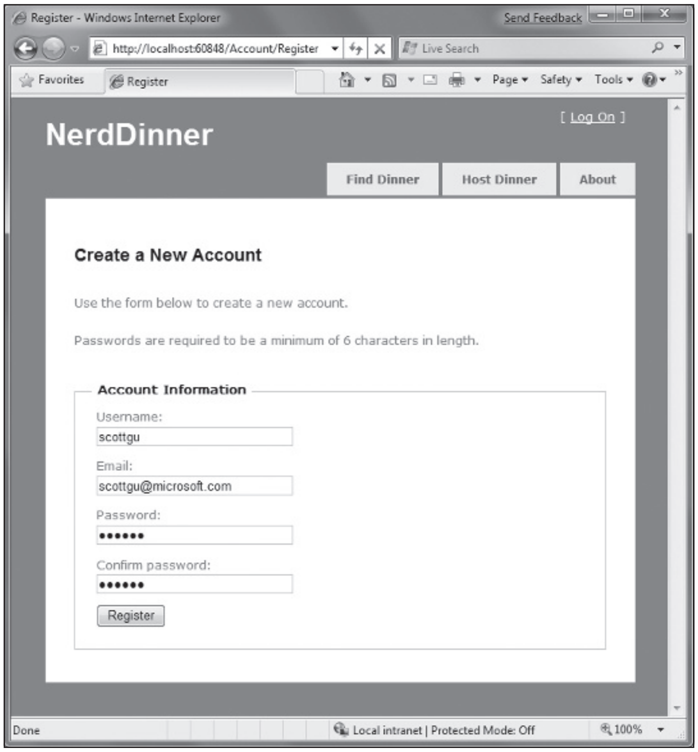
Username:

Password:

□ Remember me?

Log On

Figure 1-115



Create a New Account

Use the form below to create a new account.

Passwords are required to be a minimum of 6 characters in length.

Account Information

Username: scottgu

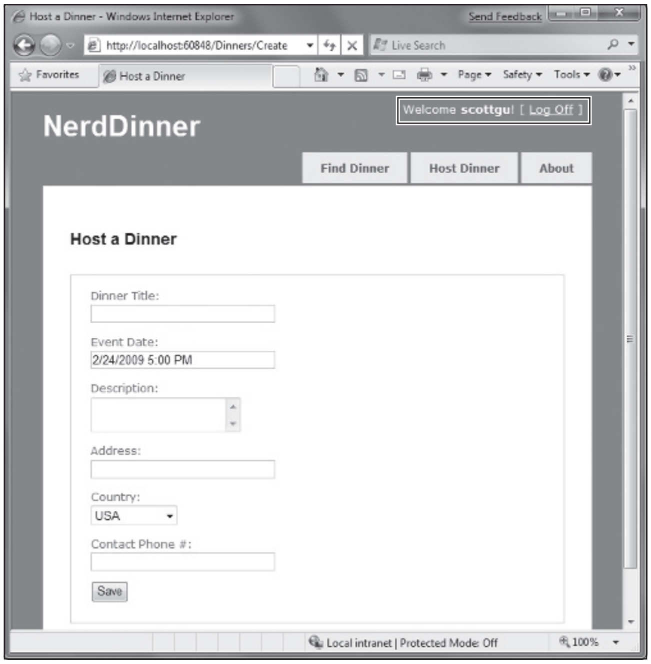
Email: scottgu@microsoft.com

Password: ••••••

Confirm password: ••••••

Register

Figure 1-116


Host a Dinner - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Create
Live Search
Favorites
Host a Dinner
Page
Safety
Tools
Welcome scottgu! [Log Off]
NerdDinner
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

Host a Dinner

Dinner Title:

Event Date: 2/24/2009 5:00 PM

Description:

Address:

Country: USA

Contact Phone #:

Save

Figure 1-117

The above login, logout, and registration functionality is implemented within the AccountController class that was added to our project by VS when it created it. The UI for the AccountController is implemented using view templates within the \Views\Account directory (Figure 1-118).

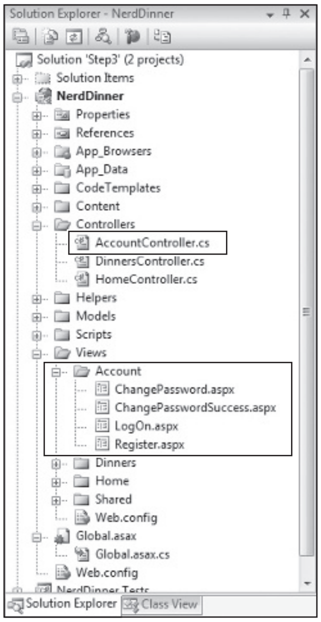


Figure 1-118

The AccountController class uses the ASP.NET Forms Authentication system to issue encrypted authentication cookies, and the ASP.NET Membership API to store and validate usernames/passwords. The ASP.NET Membership API is extensible and enables any password credential store to be used. ASP.NET ships with built-in membership provider implementations that store username/passwords within a SQL database, or within Active Directory.

We can configure which membership provider our NerdDinner application should use by opening the web.config file at the root of the project and looking for the <membership> section within it. The default web.config, added when the project was created, registers the SQL membership provider, and configures it to use a connection-string named ApplicationServices to specify the database location.

The default ApplicationServices connection string (which is specified within the <connectionStrings> section of the web.config file) is configured to use SQL Express. It points to a SQL Express database named ASPNETDB.MDF under the application's App_Data directory. If this database doesn't exist the first time the Membership API is used within the application, ASP.NET will automatically create the database and provision the appropriate membership database schema within it (Figure 1-119).

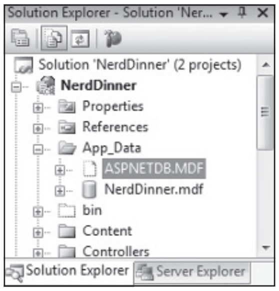


Figure 1-119

If instead of using SQL Express we wanted to use a full SQL Server instance (or connect to a remote database), all we'd need to do is to update the ApplicationServices connection string within the web.config file and make sure that the appropriate membership schema has been added to the database it points at. You can run the aspnet_regsql.exe utility within the \Windows\Microsoft.NET\Framework\v2.0.50727\ directory to add the appropriate schema for membership and the other ASP.NET application services to a database.

Authorizing the /Dinners/Create URL Using the [Authorize] Filter

We didn't have to write any code to enable a secure authentication and account management implementation for the NerdDinner application. Users can register new accounts with our application, and log in/log out of the site. And now we can add authorization logic to the application, and use the authentication status and username of visitors to control what they can and can't do within the site.

Let's begin by adding authorization logic to the Create action methods of our DinnersController class. Specifically, we will require that users accessing the /Dinners/Create URL must be logged in. If they aren't logged in, we'll redirect them to the login page so that they can sign in.

Implementing this logic is pretty easy. All we need to do is to add an `[Authorize]` filter attribute to our `Create` action methods like so:

```
//
// GET: /Dinners/Create
[Authorize]
public ActionResult Create() {
   ...
}

//
// POST: /Dinners/Create

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

ASP.NET MVC supports the ability to create *action filters* that can be used to implement reusable logic that can be declaratively applied to action methods. The `[Authorize]` filter is one of the built-in action filters provided by ASP.NET MVC, and it enables a developer to declaratively apply authorization rules to action methods and controller classes.

When applied without any parameters (as in the previous code), the `[Authorize]` filter enforces that the user making the action method request must be logged in — and it will automatically redirect the browser to the login URL if they aren't. When doing this redirect, the originally requested URL is passed as a querystring argument (for example: `/Account/LogOn?ReturnUrl=%2fDinners%2fCreate`). The `AccountController` will then redirect the user back to the originally requested URL once they log in.

The `[Authorize]` filter optionally supports the ability to specify a `Users` or `Roles` property that can be used to require that the user is both logged in and within a list of allowed users or a member of an allowed security role. For example, the code below only allows two specific users, `scottgu` and `billg`, to access the `/Dinners/Create` URL:

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

Embedding specific user names within code tends to be pretty unmaintainable though. A better approach is to define higher-level *roles* that the code checks against, and then to map users into the role using either a database or active directory system (enabling the actual user mapping list to be stored externally from the code). ASP.NET includes a built-in role management API as well as a built-in set of role providers (including ones for SQL and Active Directory) that can help perform this user/role mapping. We could then update the code to only allow users within a specific `"admin"` role to access the `/Dinners/Create` URL:

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

Using the User.Identity.Name Property When Creating Dinners

We can retrieve the username of the currently logged-in user of a request using the `User.Identity.Name` property exposed on the Controller base class.

Earlier, when we implemented the HTTP-POST version of our `Create` action method, we had hard-coded the `HostedBy` property of the dinner to a static string. We can now update this code to instead use the `User.Identity.Name` property, as well as automatically add an RSVP for the host creating the dinner:

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

Because we have added an `[Authorize]` attribute to the `Create` method, ASP.NET MVC ensures that the action method only executes if the user visiting the `/Dinners/Create` URL is logged in on the site. As such, the `User.Identity.Name` property value will always contain a valid username.

Using the User.Identity.Name Property When Editing Dinners

Let's now add some authorization logic that restricts users so that they can only edit the properties of dinners they themselves are hosting.

To help with this, we'll first add an `IsHostedBy(`*`username`*`)` helper method to our `Dinner` object (within the `Dinner.cs` partial class we built earlier). This helper method returns true or false, depending on whether a supplied username matches the Dinner HostedBy property, and encapsulates the logic necessary to perform a case-insensitive string comparison of them:

```
public partial class Dinner {

    public bool IsHostedBy(string userName) {

        return HostedBy.Equals(userName,
                               StringComparison.InvariantCultureIgnoreCase);
    }
}
```

We'll then add an [Authorize] attribute to the Edit action methods within our DinnersController class. This will ensure that users must be logged in to request a /Dinners/Edit/[id] URL.

We can then add code to our Edit methods that uses the Dinner.IsHostedBy(*username*) helper method to verify that the logged-in user matches the dinner host. If the user is not the host, we'll display an "InvalidOwner" view and terminate the request. The code to do this looks like the following:

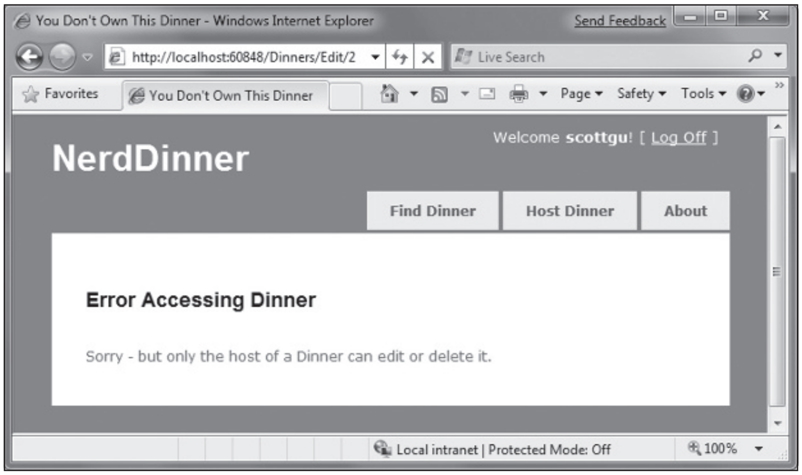
```
//
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    return View(new DinnerFormViewModel(dinner));
}

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

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

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

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

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

We can then right-click on the `\Views\Dinners` directory and choose the Add ➪ View menu command to create a new `"InvalidOwner"` view. We'll populate it with the following error message:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
      You Don't Own This Dinner
</asp:Content>

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

And now when a user attempts to edit a dinner they don't own, they'll get the error message shown in Figure 1-120.


You Don't Own This Dinner - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Edit/2
Live Search
Favorites
You Don't Own This Dinner
Page
Safety
Tools
NerdDinner
Welcome scottgu! [Log Off]
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

Error Accessing Dinner

Sorry - but only the host of a Dinner can edit or delete it.

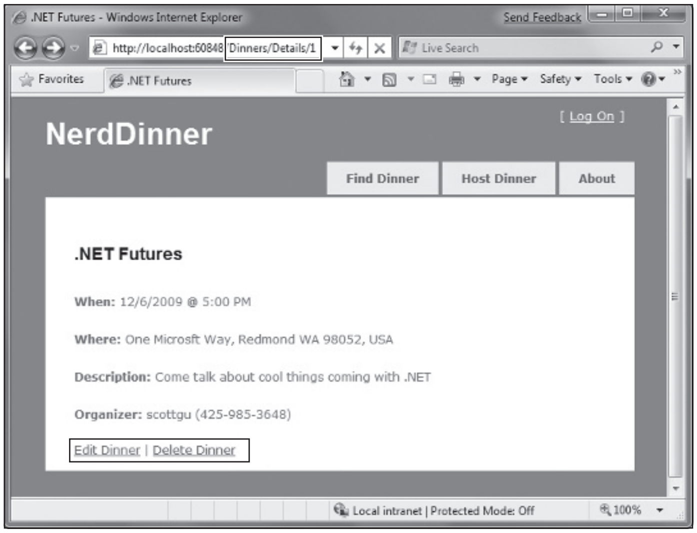
Figure 1-120

We can repeat the same steps for the `Delete` action methods within our controller to lock down permission to delete dinners as well, and ensure that only the host of a dinner can delete it.

Showing/Hiding Edit and Delete Links

We are linking to the `Edit` and `Delete` action method of our `DinnersController` class from our `/Details` URL (Figure 1-121).

Currently we are showing the Edit and Delete action links regardless of whether the visitor to the details URL is the host of the dinner. Let's change this so that the links are only displayed if the visiting user is the owner of the dinner.


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page ▾
Safety ▾
Tools ▾
NerdDinner
[Log On]
Find Dinner
Host Dinner
About
Edit Dinner | Delete Dinner
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

Figure 1-121

The `Details` action method within our `DinnersController` retrieves a Dinner object and then passes it as the model object to our view template:

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

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

    return View(dinner);
}
```

We can update our view template to conditionally show/hide the Edit and Delete links by using the `Dinner.IsHostedBy` helper method as in the code that follows:

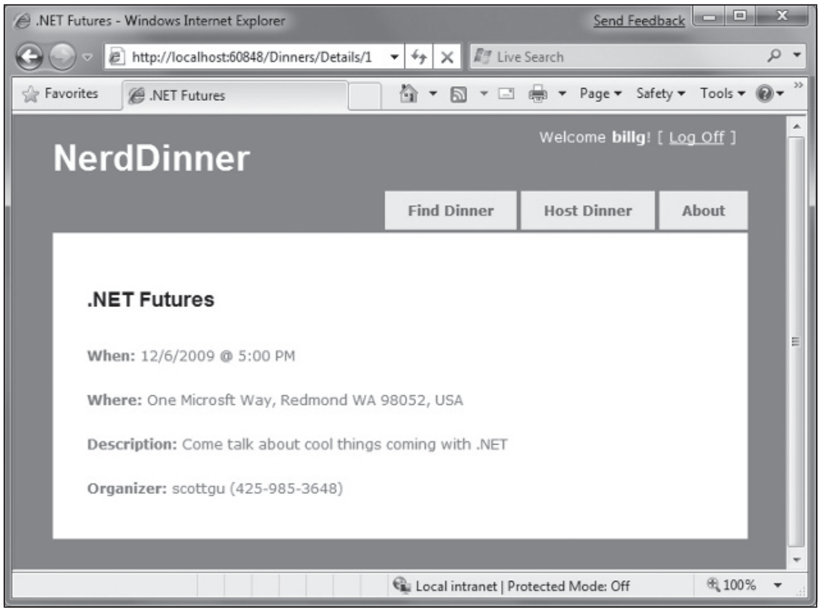
```
<% if (Model.IsHostedBy(Context.User.Identity.Name)) { %>

   <%= Html.ActionLink("Edit Dinner", "Edit", new { id=Model.DinnerID })%> |
   <%= Html.ActionLink("Delete Dinner", "Delete", new {id=Model.DinnerID})%>

<% } %>
```

AJAX Enabling RSVPs Accepts

Let's now add support for logged-in users to RSVP their interest in attending a dinner. We'll implement this using an AJAX-based approach integrated within the dinner details page.

Indicating Whether the User Is RSVP'ed

Users can visit the `/Dinners/Details/[id]` URL to see details about a particular dinner (Figure 1-122).


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page ▾ Safety ▾ Tools ▾
NerdDinner
Welcome billg! [Log Off]
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

Figure 1-122

The `Details` action method is implemented like so:

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

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

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

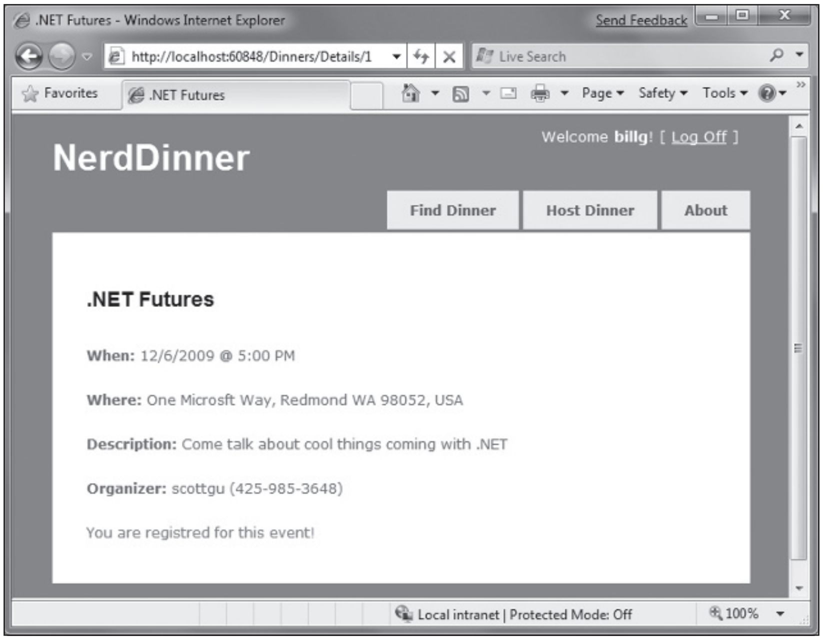
Our first step to implement RSVP support will be to add an `IsUserRegistered(`*`username`*`)` helper method to our `Dinner` object (within the `Dinner.cs` partial class we built earlier). This helper method returns true or false, depending on whether the user is currently RSVP'd for the dinner:

```
public partial class Dinner {

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

We can then add the following code to our `Details.aspx` view template to display an appropriate message indicating whether the user is registered or not for the event:

```
<% if (Request.IsAuthenticated) { %>

    <% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>

        <p>You are registered 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.

<% } %>
```

And now when a user visits a dinner they are registered for they'll see the message in Figure 1-123.


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page ▾
Safety ▾
Tools ▾
Welcome billg! [Log Off]
NerdDinner
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

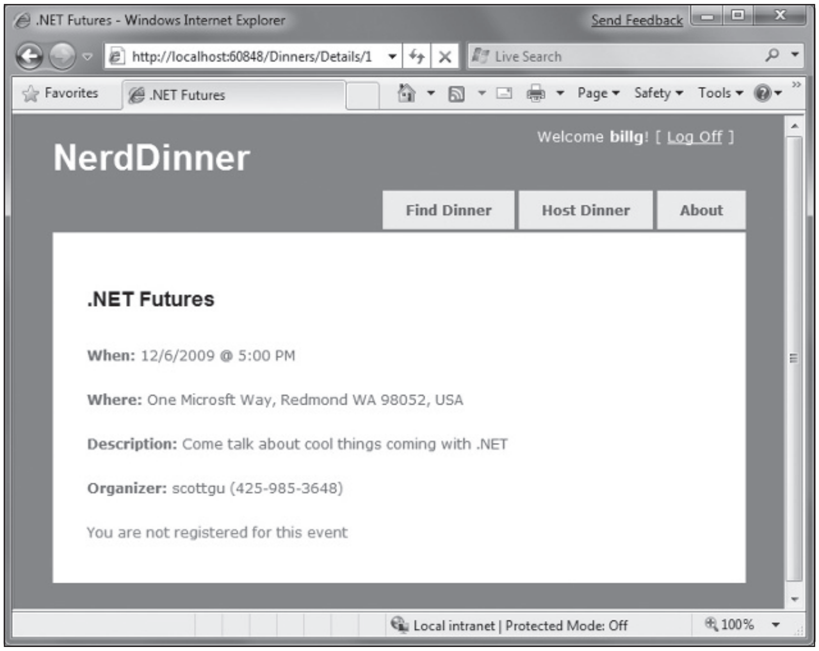
Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

You are registred for this event!

Figure 1-123

And when they visit a dinner they are not registered for, they'll see the message in Figure 1-124.


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page
Safety
Tools
Local intranet | Protected Mode: Off
100%

NerdDinner

Welcome **billg**! [Log Off]

Find Dinner | Host Dinner | About

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

You are not registered for this event

Figure 1-124

Implementing the Register Action Method

Let's now add the functionality necessary to enable users to RSVP for a dinner from the details page.

To implement this, we'll create a new `RSVPController` class by right-clicking on the `\Controllers` directory and choosing the Add ➪ Controller menu command.

We'll implement a `Register` action method within the new `RSVPController` class that takes an ID for a dinner as an argument, retrieves the appropriate `Dinner` object, checks to see if the logged-in user is currently in the list of users who have registered for it, and if not adds an RSVP object for them:

```
public class RSVPController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // AJAX: /Dinners/Register/1

    [Authorize, AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Register(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (!dinner.IsUserRegistered(User.Identity.Name)) {
            RSVP rsvp = new RSVP();
            rsvp.AttendeeName = User.Identity.Name;

            dinner.RSVPs.Add(rsvp);
            dinnerRepository.Save();
        }

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

Notice, in the previous code, how we are returning a simple string as the output of the action method. We could have embedded this message within a view template — but since it is so small we'll just use the `Content` helper method on the controller base class and return a string message like that above.

Calling the Register Action Method Using AJAX

We'll use AJAX to invoke the Register action method from our `Details` view. Implementing this is pretty easy. First we'll add two script library references:

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

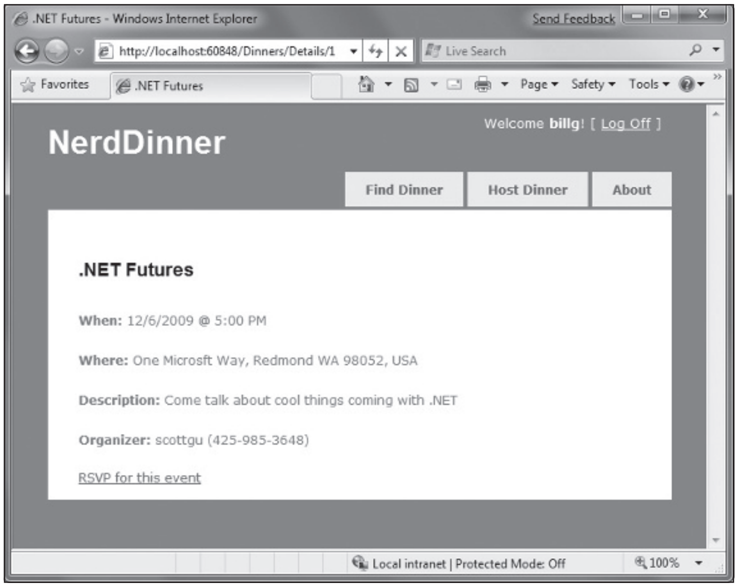
The first library references the core ASP.NET AJAX client-side script library. This file is approximately 24k in size (compressed) and contains core client-side AJAX functionality. The second library contains utility functions that integrate with ASP.NET MVC's built-in AJAX helper methods (which we'll use shortly).

We can then update the view template code we added earlier so that, instead of outputing a "You are not registered for this event" message, we render a link that when pushed performs an AJAX call that invokes our `Register` action method on our `RSVP` controller and RSVPs the user:

```
<div id="rsvpmsg">

<% if (Request.IsAuthenticated) { %>

    <% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>

        <p>You are registered 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>
```

The `Ajax.ActionLink` helper method in the previous code is built into ASP.NET MVC and is similar to the `Html.ActionLink` helper method except that instead of performing a standard navigation, it makes an AJAX call to the action method. Above we are calling the `"Register"` action method on the `"RSVP"` controller and passing the DinnerID as the `id` parameter to it. The final `AjaxOptions` parameter we are passing indicates that we want to take the content returned from the action method and update the HTML `<div>` element on the page whose id is `"rsvpmsg"`.

And now when a user browses to a dinner they aren't registered for yet, they'll see a link to RSVP for it (Figure 1-125).


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page
Safety
Tools
NerdDinner
Welcome billg! [Log Off]
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

RSVP for this event

Figure 1-125

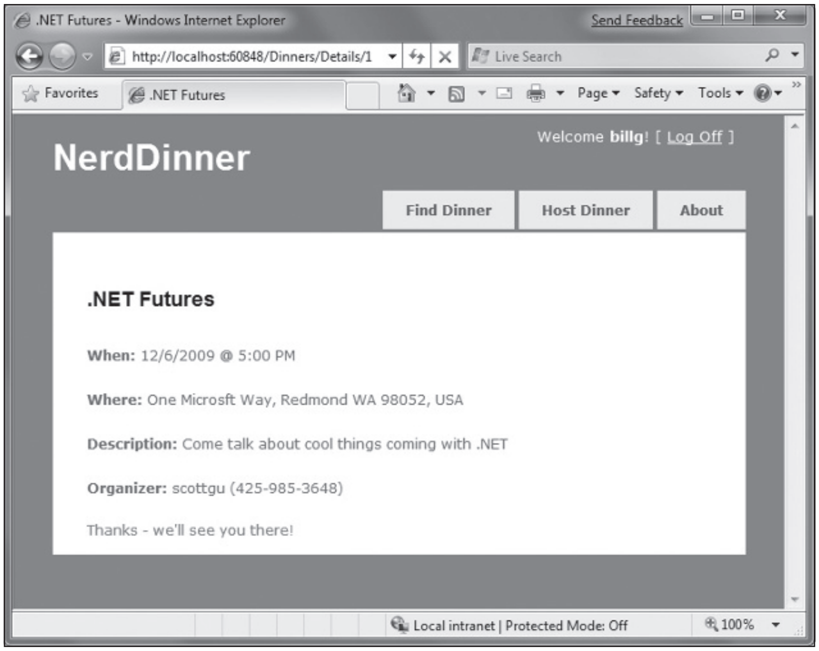
If they click the "RSVP for this event" link, they'll make an AJAX call to the `Register` action method on the `RSVP` controller, and when it completes they'll see an updated message like that in Figure 1-126.

The network bandwidth and traffic involved when making this AJAX call is really lightweight. When the user clicks on the "RSVP for this event" link, a small HTTP POST network request is made to the `/Dinners/Register/1` URL that looks like the following on the wire:

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

And the response from our `Register` action method is simply:

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


.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page
Safety
Tools
NerdDinner
Welcome billg! [Log Off]
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

Thanks - we'll see you there!

Figure 1-126

This lightweight call is fast and will work even over a slow network.

Adding a jQuery Animation

The AJAX functionality we implemented works well and fast. Sometimes it can happen so fast, though, that a user might not notice that the RSVP link has been replaced with new text. To make the outcome a little more obvious, we can add a simple animation to draw attention to the updates message.

The default ASP.NET MVC project template includes jQuery — an excellent (and very popular) open source JavaScript library that is also supported by Microsoft. jQuery provides a number of features, including a nice HTML DOM selection and effects library.

To use jQuery, we'll first add a script reference to it. Because we are going to be using jQuery within a variety of places within our site, we'll add the script reference within our `Site.master` master page file so that all pages can use it.

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

> *Make sure you have installed the JavaScript IntelliSense hotfix for VS 2008 SP1 that enables richer intellisense support for JavaScript files (including jQuery). You can download it from:* `http://tinyurl.com/vs2008javascripthotfix`

Code written using JQuery often uses a global `$()` JavaScript method that retrieves one or more HTML elements using a CSS selector. For example, `$("#`*`rsvpmsg`*`")` selects any HTML element with the ID of `rsvpmsg`, while `$(".`*`something`*`")` would select all elements with the `"something"` CSS class name.

You can also write more advanced queries like “return all of the checked radio buttons” using a selector query like: `$("input[@type=radio][@checked]")`.

Once you’ve selected elements, you can call methods on them to take action, such as hiding them: `$("#rsvpmsg").hide();`

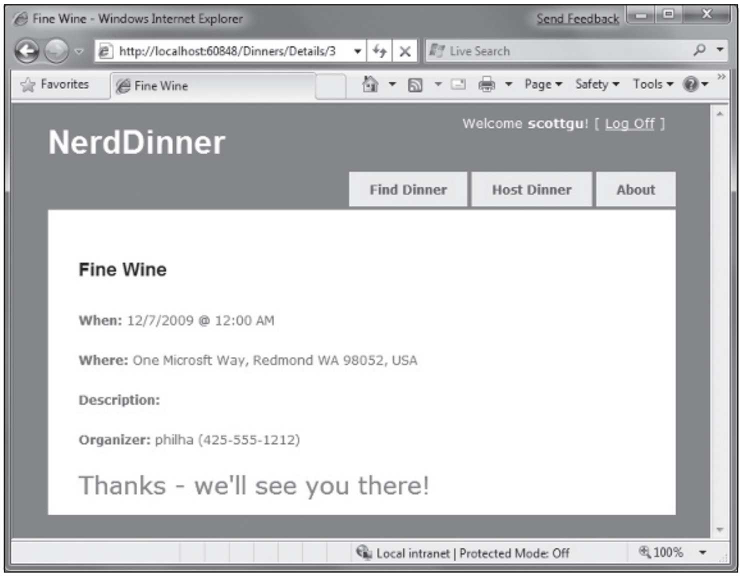
For our RSVP scenario, we’ll define a simple JavaScript function named `AnimateRSVPMessage` that selects the “rsvpmsg” <div> and animates the size of its text content. The code below starts the text small and then causes it to increase over a 400 milliseconds timeframe:

```
<script type="text/javascript">

    function AnimateRSVPMessage() {
        $("#rsvpmsg").animate({fontSize: "1.5em"}, 400);
    }

</script>
```

We can then wire up this JavaScript function to be called after our AJAX call successfully completes by passing its name to our `Ajax.ActionLink` helper method (via the `AjaxOptions OnSuccess` event property):

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

And now when the “RSVP for this event” link is clicked and our AJAX call completes successfully, the content message sent back will animate and grow large (Figure 1-127).


Fine Wine - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/3
Live Search
Favorites
Fine Wine
Page
Safety
Tools
NerdDinner
Welcome scottgu! [Log Off]
Find Dinner
Host Dinner
About
Local intranet | Protected Mode: Off
100%

Fine Wine

When: 12/7/2009 @ 12:00 AM

Where: One Microsft Way, Redmond WA 98052, USA

Description:

Organizer: philha (425-555-1212)

Thanks - we'll see you there!

Figure 1-127

In addition to providing an OnSuccess event, the AjaxOptions object exposes OnBegin, OnFailure, and OnComplete events that you can handle (along with a variety of other properties and useful options).

Cleanup — Refactor Out a RSVP Partial View

Our details view template is starting to get a little long, which over time will make it a little harder to understand. To help improve the code readability, let's finish up by creating a partial view — RSVPStatus.ascx — that encapsulates all of the RSVP view code for our Details page.

We can do this by right-clicking on the \Views\Dinners folder and then choosing the Add ➪ View menu command. We'll have it take a Dinner object as its strongly typed ViewModel. We can then copy/paste the RSVP content from our Details.aspx view into it.

Once we've done that, let's also create another partial view — EditAndDeleteLinks.ascx — that encapsulates our Edit and Delete link view code. We'll also have it take a Dinner object as its strongly typed ViewModel, and copy/paste the Edit and Delete logic from our Details.aspx view into it.

Our details view template can then just include two Html.RenderPartial method calls at the bottom:

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

This makes the code cleaner to read and maintain.

Integrating an AJAX Map

We'll now make our application a little more visually exciting by integrating AJAX mapping support. This will enable users who are creating, editing, or viewing dinners to see the location of the dinner graphically.

Creating a Map Partial View

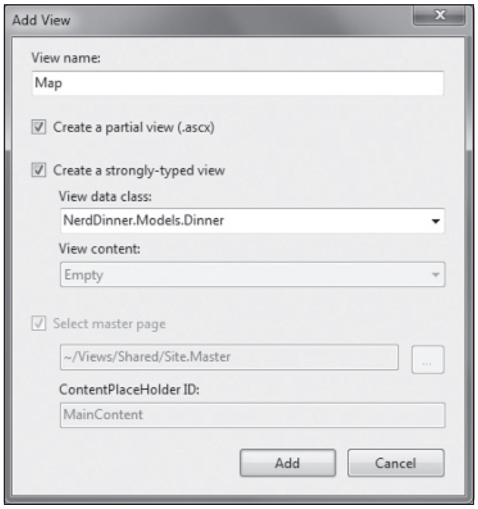
We are going to use mapping functionality in several places within our application. To keep our code DRY, we'll encapsulate the common map functionality within a single partial template that we can reuse across multiple controller actions and views. We'll name this partial view map.ascx and create it within the \Views\Dinners directory.

We can create the map.ascx partial by right-clicking on the \Views\Dinners directory and choosing the Add ➪ View menu command. We'll name the view Map.ascx, check it as a partial view, and indicate that we are going to pass it a strongly typed Dinner model class (Figure 1-128):

When we click the "Add" button our partial template will be created. We'll then update the Map.ascx file to have the following content:

```
<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>
```



View name: Map

☑ Create a partial view (.ascx)

☑ Create a strongly-typed view

View data class: NerdDinner.Models.Dinner

View content: Empty

☑ Select master page

~/Views/Shared/Site.Master

ContentPlaceHolder ID: MainContent

Figure 1-128

The first `<script>` reference points to the Microsoft Virtual Earth 6.2 mapping library. The second `<script>` reference points to a `map.js` file that we will shortly create, which will encapsulate our common JavaScript mapping logic. The `<div id="theMap">` element is the HTML container that Virtual Earth will use to host the map.

We then have an embedded `<script>` block that contains two JavaScript functions specific to this view. The first function uses jQuery to wire up a function that executes when the page is ready to run client-side script. It calls a LoadMap helper function that we'll define within our Map.js script file to load the Virtual Earth map control. The second function is a callback event handler that adds a pin to the map that identifies a location.

Notice how we are using a server-side <%= %> block within the client-side script block to embed the latitude and longitude of the dinner we want to map into the JavaScript. This is a useful technique to output dynamic values that can be used by client-side script (without requiring a separate AJAX call back to the server to retrieve the values — which makes it faster). The <%= %> blocks will execute when the view is rendering on the server — and so the output of the HTML will just end up with embedded JavaScript values (for example: var latitude = 47.64312;).

Creating a Map.js Utility Library

Let's now create the Map.js file that we can use to encapsulate the JavaScript functionality for our map (and implement the LoadMap and LoadPin methods above). We can do this by right-clicking on the \Scripts directory within our project, and then choose the Add ➪ New Item menu command, select the JScript item, and name it Map.js.

Below is the JavaScript code we'll add to the Map.js file that will interact with Virtual Earth to display our map and add locations pins to it for our dinners:

```
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 obtrusive.
    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) {
            var 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 = [];
    }
```

Integrating the Map with Create and Edit Forms

We'll now integrate the Map support with our existing Create and Edit scenarios. The good news is that this is pretty easy to do, and doesn't require us to change any of our Controller code. Because our `Create` and `Edit` views share a common `DinnerForm` partial view used to implement the dinner form UI, we can add the map in one place and have both our Create and Edit scenarios use it.

All we need to do is to open the `\Views\Dinners\DinnerForm.ascx` partial view and update it to include our new map partial. Below is what the updated `DinnerForm` will look like once the map is added (the HTML form elements are omitted from the code snippet below for brevity):

```
<%= 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>

<% } %>
```

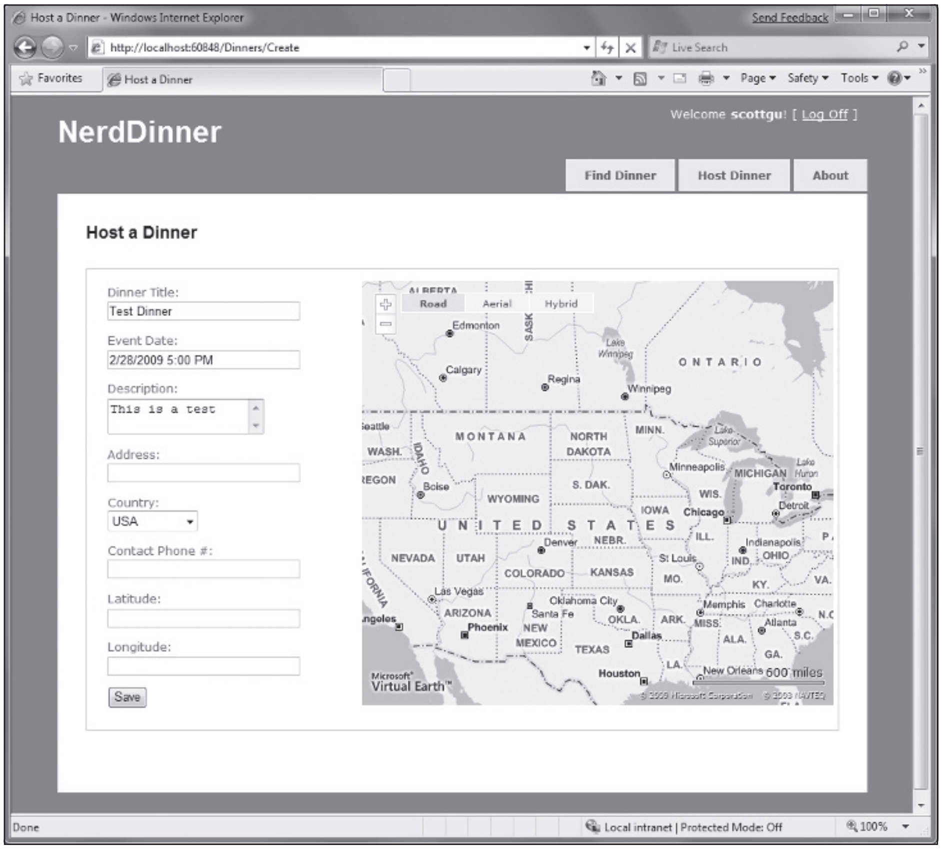
The `DinnerForm` partial above takes an object of type `DinnerFormViewModel` as its model type (because it needs both a `Dinner` object and a `SelectList` to populate the drop-down list of countries).

Our map partial just needs an object of type `Dinner` as its model type, and so when we render the map partial we are passing just the Dinner sub-property of `DinnerFormViewModel` to it:

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

The JavaScript function we've added to the partial uses jQuery to attach a *blur* event to the Address HTML textbox. You've probably heard of *focus* events that fire when a user clicks or tabs into a textbox. The opposite is a blur event that fires when a user exits a textbox. The event handler in the previous code clears the latitude and longitude textbox values when this happens, and then plots the new address location on our map. A callback event handler that we defined within the `map.js` file will then update the longitude and latitude textboxes on our form using values returned by Virtual Earth based on the address we gave it.

And now when we run our application again and click the Host Dinner tab, we'll see a default map displayed along with our standard Dinner form elements (Figure 1-129).



Host a Dinner - Windows Internet Explorer

http://localhost:60848/Dinners/Create

NerdDinner

Welcome **scottgu**! [Log Off]

Host a Dinner

Dinner Title: Test Dinner

Event Date: 2/28/2009 5:00 PM

Description: This is a test

Address:

Country: USA

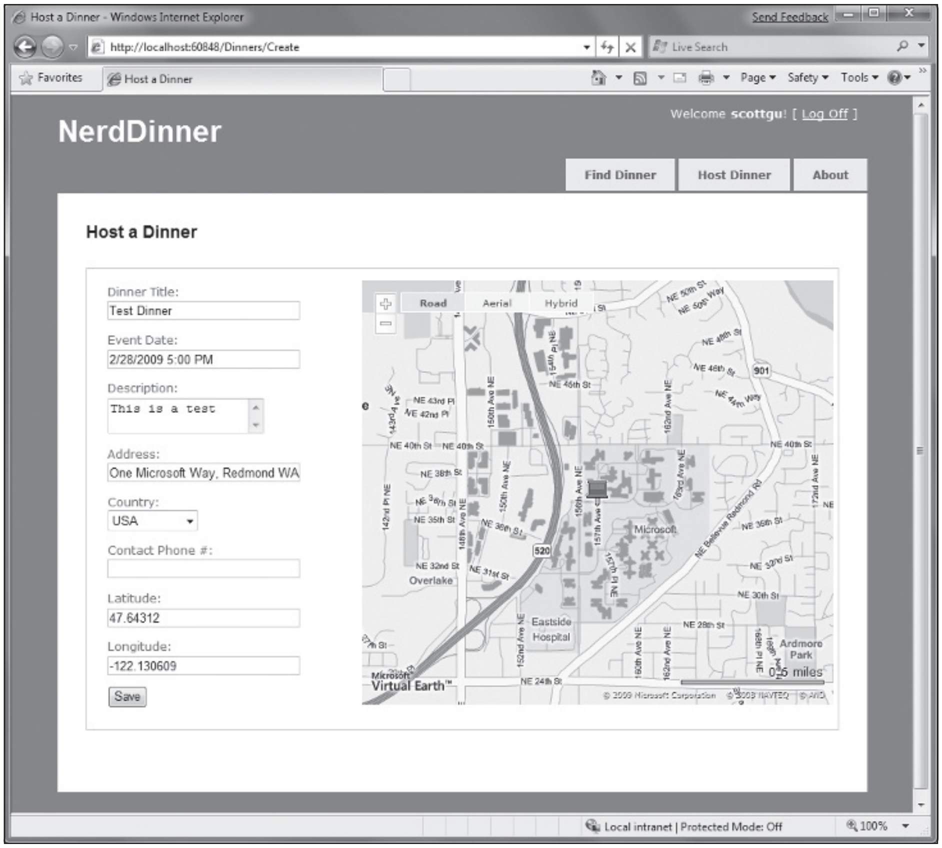
Contact Phone #:

Latitude:

Longitude:

Figure 1-129

When we type in an address, and then tab away, the map will dynamically update to display the location, and our event handler will populate the latitude/longitude textboxes with the location values (Figure 1-130).



Host a Dinner

Dinner Title: Test Dinner

Event Date: 2/28/2009 5:00 PM

Description: This is a test

Address: One Microsoft Way, Redmond WA

Country: USA

Contact Phone #:

Latitude: 47.64312

Longitude: -122.130609

Save

Figure 1-130

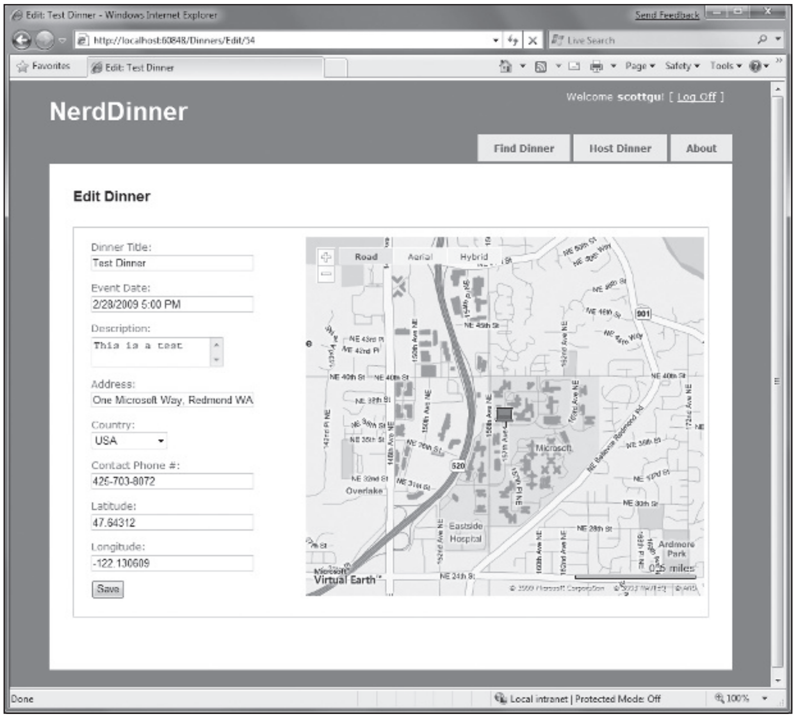
If we save the new dinner and then open it again for editing, we'll find that the map location is displayed when the page loads (Figure 1-131).

Every time the address field is changed, the map and the latitude/longitude coordinates will update.

Now that the map displays the dinner location, we can also change the Latitude and Longitude form fields from being visible textboxes to instead be hidden elements (since the map is automatically updating them each time an address is entered). To do this, we'll switch from using the `Html.TextBox` HTML helper to using the `Html.Hidden` helper method:

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

And now our forms are a little more user-friendly (Figure 1-132) and avoid displaying the raw latitude/longitude (while still storing them with each dinner in the database).



http://localhost:60848/Dinners/Edit/54

NerdDinner

Welcome **scottgu**! [Log Off]

Edit Dinner

Dinner Title:
Test Dinner

Event Date:
2/28/2009 5:00 PM

Description:
This is a test

Address:
One Microsoft Way, Redmond WA

Country:
USA

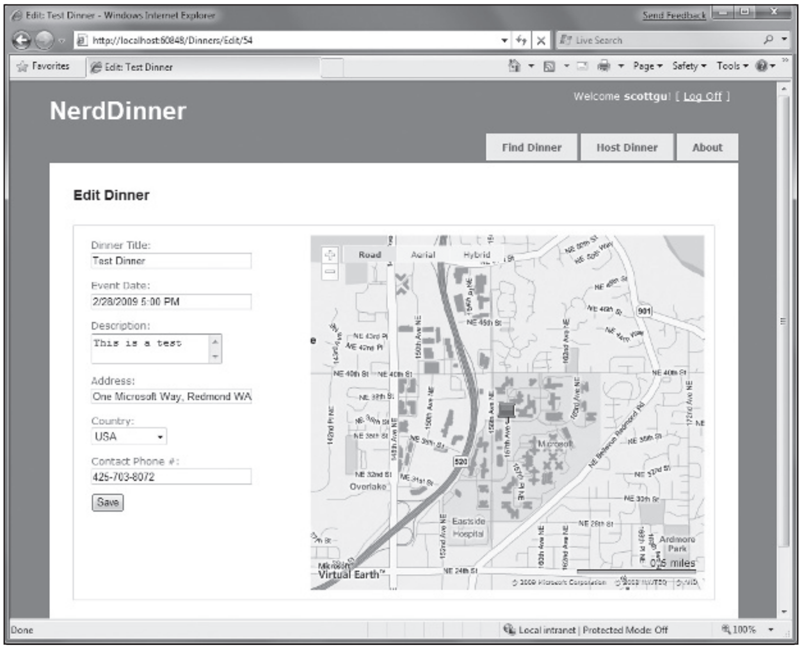
Contact Phone #:
425-703-8072

Latitude:
47.64312

Longitude:
-122.130609

Save

Figure 1-131



http://localhost:60848/Dinners/Edit/54

NerdDinner

Welcome **scottgu**! [Log Off]

Edit Dinner

Dinner Title:
Test Dinner

Event Date:
2/28/2009 5:00 PM

Description:
This is a test

Address:
One Microsoft Way, Redmond WA

Country:
USA

Contact Phone #:
425-703-8072

Save

Figure 1-132

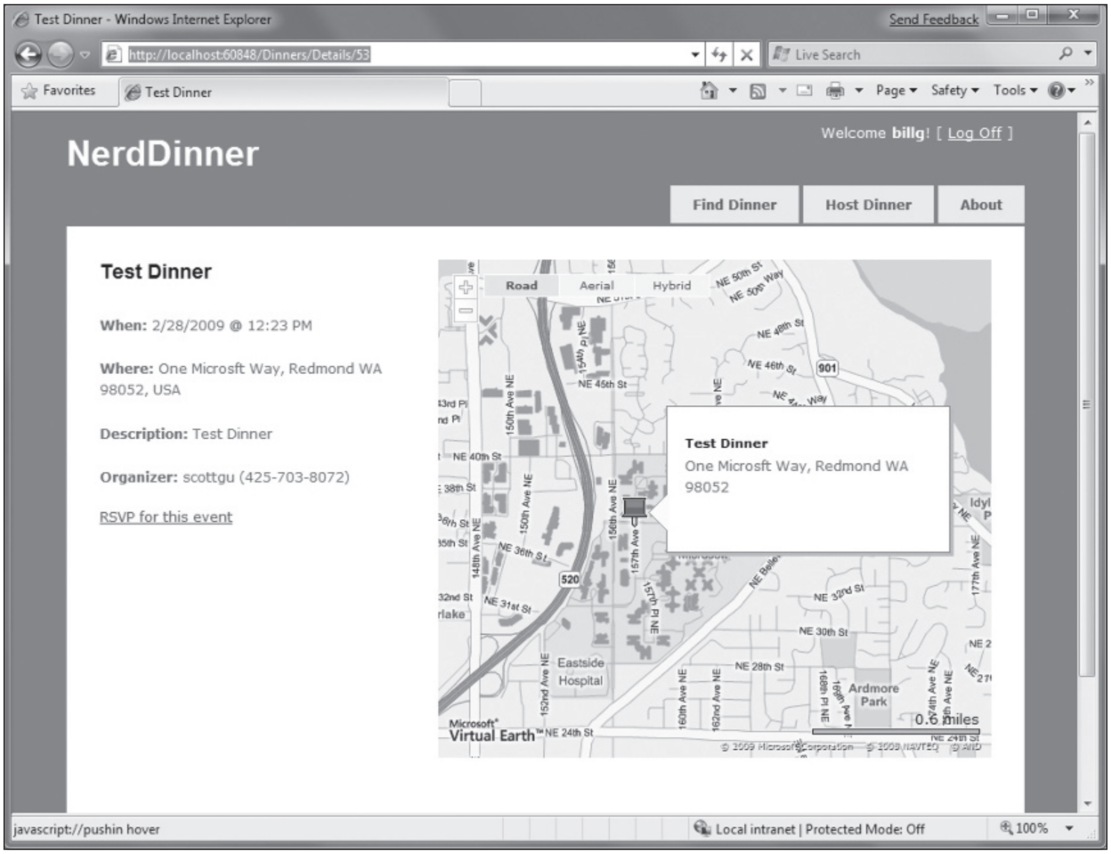
Integrating the Map with the Details View

Now that we have the map integrated with our Create and Edit scenarios, let's also integrate it with our Details scenario. All we need to do is to call `<% Html.RenderPartial("map"); %>` within the `Details` view.

Below is what the source code to the complete `Details` view (with map integration) looks like:

```
<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>

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

</asp:Content>
```

And now when a user navigates to a `/Dinners/Details/[id]` URL, they'll see details about the dinner, the location of the dinner on the map (complete with a pushpin that when hovered over displays the title of the dinner and the address of it), and have an AJAX link to RSVP for it (Figure 1-133).



Test Dinner

When: 2/28/2009 @ 12:23 PM

Where: One Microsft Way, Redmond WA 98052, USA

Description: Test Dinner

Organizer: scottgu (425-703-8072)

RSVP for this event

Test Dinner
One Microsft Way, Redmond WA 98052

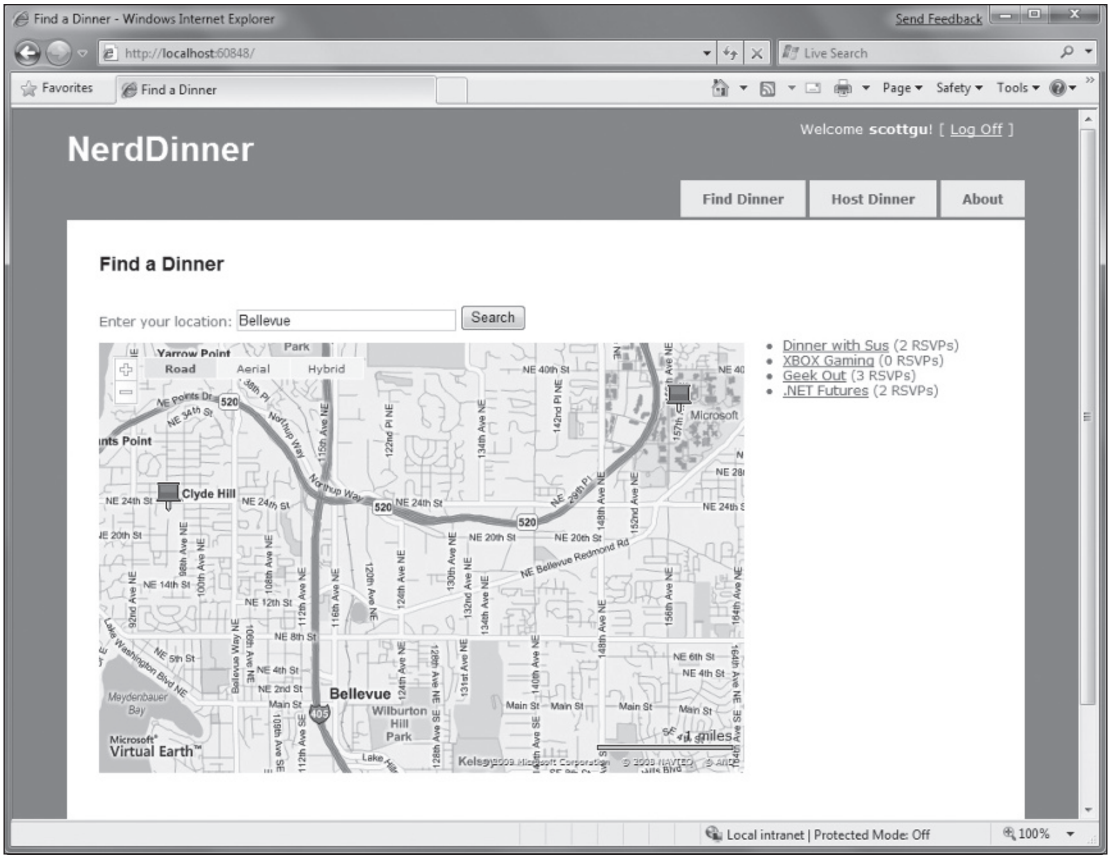
Figure 1-133

Implementing Location Search in Our Database and Repository

To finish off our AJAX implementation, let's add a map to the home page of the application that allows users to graphically search for dinners near them (Figure 1-134).

We'll begin by implementing support within our database and data repository layer to efficiently perform a location-based radius search for dinners. We could use the new geospatial features of SQL 2008 (`www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx`) to implement this, or alternatively we can use a SQL function approach that Gary Dryden discussed in article here: `www.codeproject.com/KB/cs/distancebetweenlocations.aspx` and Rob Conery blogged about using with LINQ to SQL here: `http://blog.wekeroad.com/2007/08/30/linq-and-geocoding/`.

To implement this technique, we will open the Server Explorer within Visual Studio, select the NerdDinner database, and then right-click on the functions sub-node under it and choose to create a new Scalar-valued function (Figure 1-135).



Find a Dinner

Enter your location: Bellevue

- Dinner with Sus (2 RSVPs)
- XBOX Gaming (0 RSVPs)
- Geek Out (3 RSVPs)
- .NET Futures (2 RSVPs)

Figure 1-134

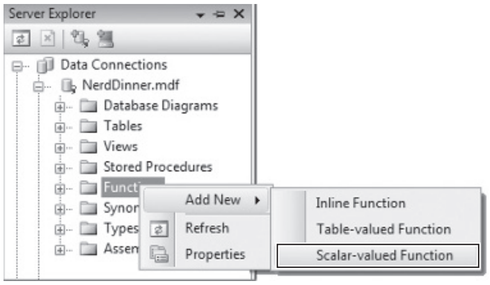


Figure 1-135

We'll then paste in the following `DistanceBetween` function:

```
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
```

We'll then create a new table-valued function in SQL Server that we'll call `NearestDinners` (Figure 1-136):

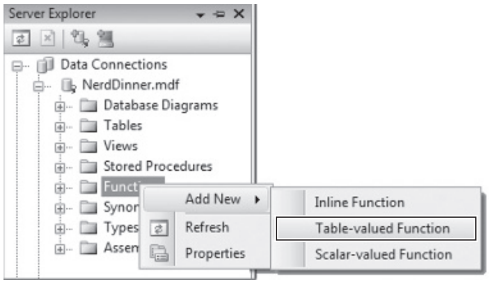


Figure 1-136

This `NearestDinners` table function uses the `DistanceBetween` helper function to return all dinners within 100 miles of the latitude and longitude we supply it:

```
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
```

To call this function, we'll first open up the LINQ to SQL designer by double-clicking on the `NerdDinner.dbml` file within our `\Models` directory (Figure 1-137).

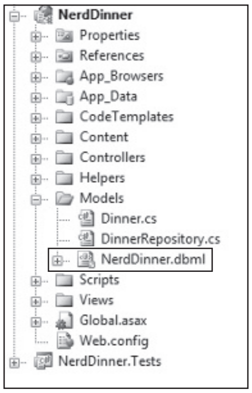


Figure 1-137

We'll then drag the `NearestDinners` and `DistanceBetween` functions onto the LINQ to SQL designer, which will cause them to be added as methods on our LINQ to SQL `NerdDinnerDataContext` class (Figure 1-138).

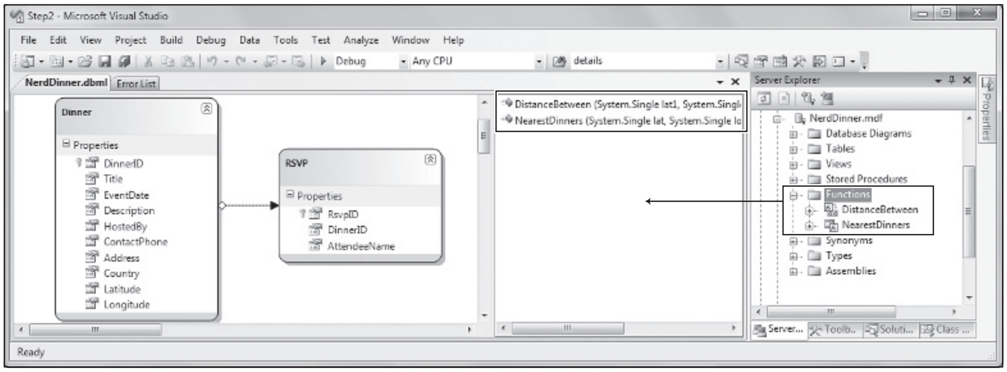


Figure 1-138

We can then expose a FindByLocation query method on our DinnerRepository class that uses the NearestDinner function to return upcoming dinners that are within 100 miles of the specified location:

```
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;
}
```

Implementing a JSON-Based AJAX Search Action Method

We'll now implement a controller action method that takes advantage of the new FindByLocation repository method to return a list of Dinner data that can be used to populate a map. We'll have this action method return the Dinner data in a JSON (JavaScript Object Notation) format so that it can be easily manipulated using JavaScript on the client.

To implement this, we'll create a new SearchController class by right-clicking on the \Controllers directory and choosing the Add ➪ >Controller menu command. We'll then implement a SearchByLocation action method within the new SearchController class like the one that follows:

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

public class SearchController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // AJAX: /Search/SearchByLocation

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult SearchByLocation(float longitude, float latitude) {

        var dinners = dinnerRepository.FindByLocation(latitude, longitude);

        var jsonDinners = from dinner in dinners
                          select new JsonDinner {
                              DinnerID = dinner.DinnerID,
                              Latitude = dinner.Latitude,
                              Longitude = dinner.Longitude,
                              Title = dinner.Title,
                              Description = dinner.Description,
                              RSVPCount = dinner.RSVPs.Count
                          };

        return Json(jsonDinners.ToList());
    }
}
```

The `SearchController`'s `SearchByLocation` action method internally calls the `FindByLocation` method on `DinnerRespository` to get a list of nearby dinners. Rather than return the `Dinner` objects directly to the client, though, it instead returns `JsonDinner` objects. The `JsonDinner` class exposes a subset of Dinner properties (for example: for security reasons it doesn't disclose the names of the people who have RSVP'ed for a dinner). It also includes an `RSVPCount` property that doesn't exist in Dinner — and that is dynamically calculated by counting the number of `RSVP` objects associated with a particular dinner.

We are then using the `Json` helper method on the Controller base class to return the sequence of dinners using a JSON-based wire format. JSON is a standard text format for representing simple data structures. The following is an example of what a JSON-formatted list of two JsonDinner objects looks like when returned from our action method:

```
 [{"DinnerID":53,"Title":"Dinner with the Family","Latitude":47.6431
2,"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}]
```

Calling the JSON-Based AJAX Method Using jQuery

We are now ready to update the home page of the NerdDinner application to use the SearchController's `SearchByLocation` action method. To do this, we'll open the /Views/Home/Index.aspx view template and update it to have a textbox, search button, our map, and a `<div>` element named `dinnerList`:

```
<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>
```

We can then add two JavaScript functions to the page:

```
    <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>
```

The first JavaScript function loads the map when the page first loads. The second JavaScript function wires up a JavaScript click event handler on the search button. When the button is pressed, it calls the `FindDinnersGivenLocation` JavaScript function which we'll add to our `Map.js` file:

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

This `FindDinnersGivenLocation` function calls `map.Find` on the Virtual Earth Control to center it on the entered location. When the Virtual Earth map service returns, the `map.Find` method invokes the `callbackUpdateMapDinners` callback method we passed it as the final argument.

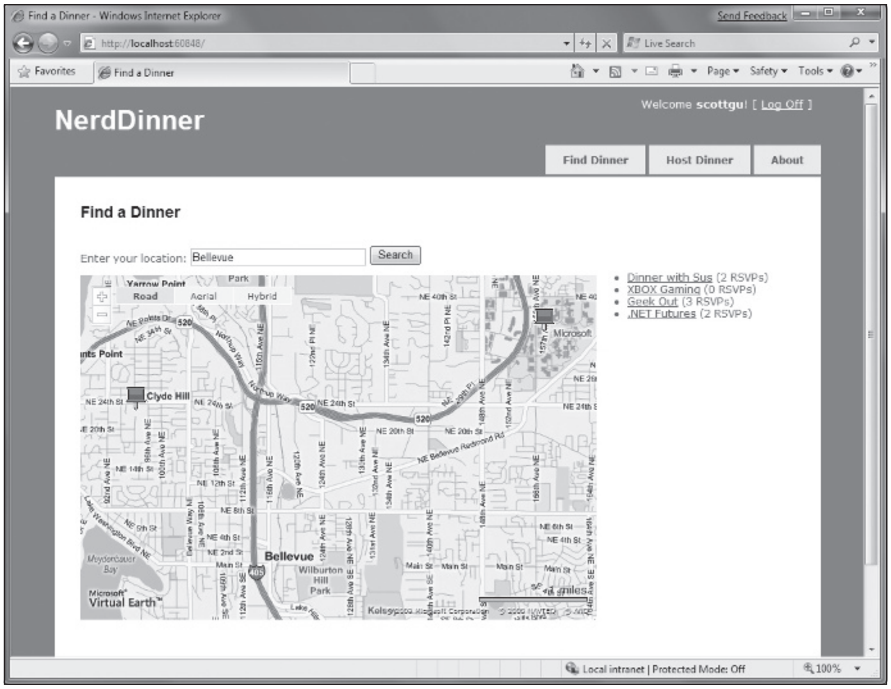
The `callbackUpdateMapDinners` method is where the real work is done. It uses jQuery's `$.post` helper method to perform an AJAX call to our SearchController's `SearchByLocation` action method — passing it the latitude and longitude of the newly centered map. It defines an inline function that will be called when the `$.post` helper method completes, and the JSON-formatted dinner results returned from the `SearchByLocation` action method will be passed it using a variable called `dinners`. It then does a `foreach` over each returned dinner, and uses the dinner's latitude and longitude and other properties to add a new pin on the map. It also adds a dinner entry to the HTML list of dinners to the right of the map. It then wires up a hover event for both the pushpins and the HTML list so that details about the dinner are displayed when a user hovers over them:

```
    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.

        if (points.length > 1) {
                 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");
```

And now when we run the application and visit the home page, we'll be presented with a map. When we enter the name of a city the map will display the upcoming dinners near it (Figure 1-139).

Hovering over a dinner will display details about it (Figure 1-140).

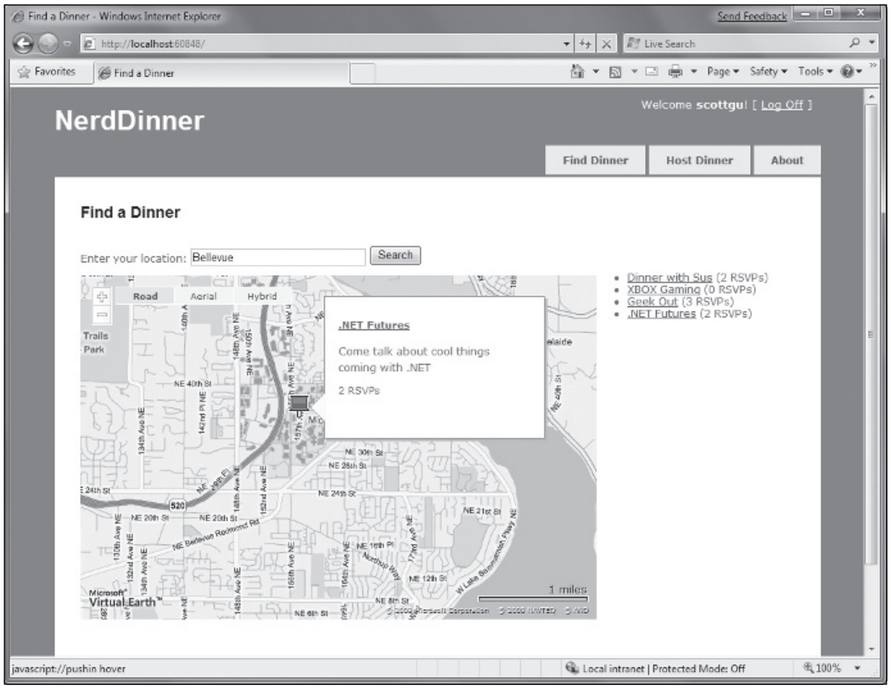


Find a Dinner

Enter your location: Bellevue

- Dinner with Sus (2 RSVPs)
- XBOX Gaming (0 RSVPs)
- Geek Out (3 RSVPs)
- .NET Futures (2 RSVPs)

Figure 1-139



Find a Dinner

Enter your location: Bellevue

- Dinner with Sus (2 RSVPs)
- XBOX Gaming (0 RSVPs)
- Geek Out (3 RSVPs)
- .NET Futures (2 RSVPs)

.NET Futures

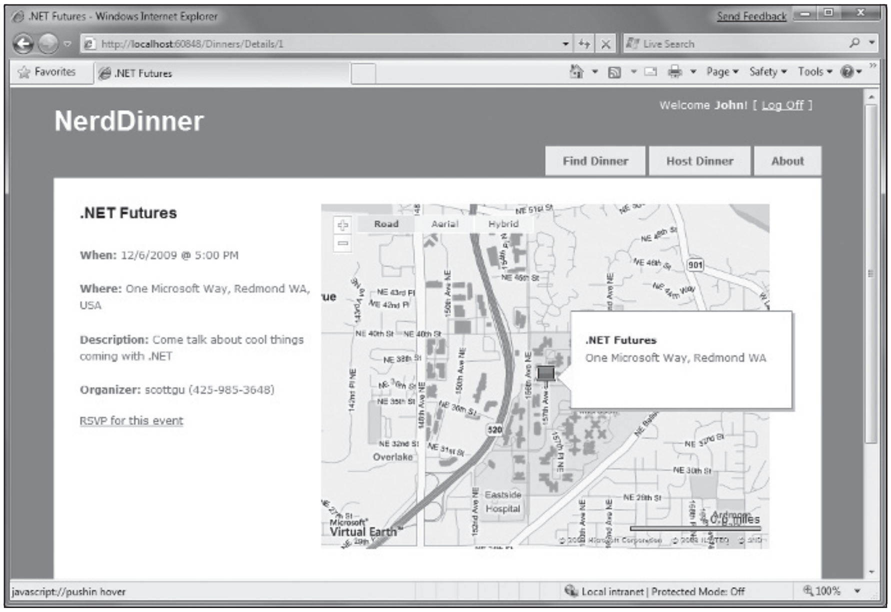
Come talk about cool things coming with .NET

2 RSVPs

javascript://pushin hover

Figure 1-140

Clicking the Dinner title either in the bubble or on the right-hand side in the HTML list will navigate us to the dinner — which we can then optionally RSVP for (Figure 1-141).



.NET Futures - Windows Internet Explorer
Send Feedback
http://localhost:60848/Dinners/Details/1
Live Search
Favorites
.NET Futures
Page ▾ Safety ▾ Tools ▾
NerdDinner
Welcome John! [Log Off]
Find Dinner
Host Dinner
About
Road
Aerial
Hybrid
.NET Futures
One Microsoft Way, Redmond WA
Overlake
Eastside Hospital
Microsoft Virtual Earth
0.6 miles
javascript://pushin hover
Local intranet | Protected Mode: Off
100%

.NET Futures

When: 12/6/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

Description: Come talk about cool things coming with .NET

Organizer: scottgu (425-985-3648)

RSVP for this event

Figure 1-141

Unit Testing

Let's develop a suite of automated unit tests that verify our NerdDinner functionality, and that will give us the confidence to make changes and improvements to the application in the future.

Why Unit Test?

On the drive into work one morning you have a sudden flash of inspiration about an application you are working on. You realize there is a change you can implement that will make the application dramatically better. It might be a refactoring that cleans up the code, adds a new feature, or fixes a bug.

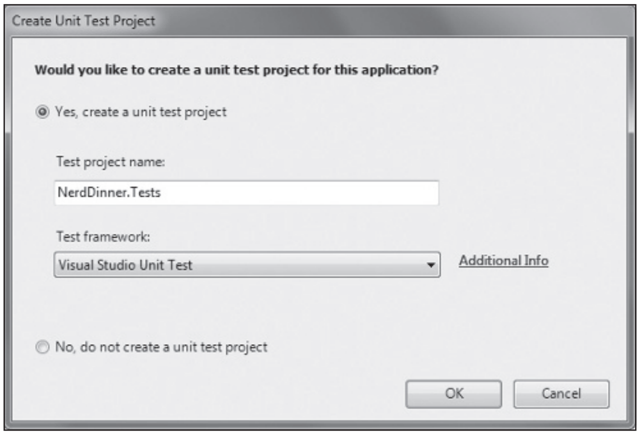
The question that confronts you when you arrive at your computer is — "how safe is it to make this improvement?" What if making the change has side effects or breaks something? The change might be simple and only take a few minutes to implement, but what if it takes hours to manually test out all of the application scenarios? What if you forget to cover a scenario and a broken application goes into production? Is making this improvement really worth all the effort?

Automated unit tests can provide a safety net that enables you to continually enhance your applications, and avoid being afraid of the code you are working on. Having automated tests that quickly verify functionality enables you to code with confidence — and empowers you to make improvements you might otherwise not have felt comfortable doing. They also help create solutions that are more maintainable and have a longer lifetime — which leads to a much higher return on investment.

The ASP.NET MVC Framework makes it easy and natural to unit test application functionality. It also enables a Test Driven Development (TDD) workflow that enables test-first-based development.

NerdDinner.Tests Project

When we created our NerdDinner application at the beginning of this tutorial, we were prompted with a dialog asking whether we wanted to create a unit test project to go along with the application project (Figure 1-142).



Create Unit Test Project

Would you like to create a unit test project for this application?

● Yes, create a unit test project

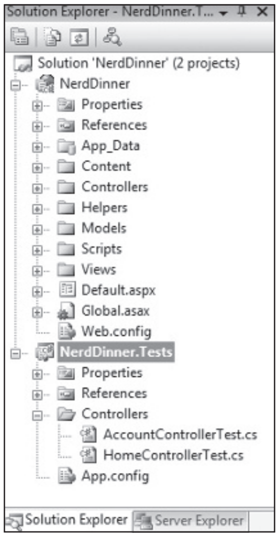
Test project name: NerdDinner.Tests

Test framework: Visual Studio Unit Test

○ No, do not create a unit test project

Figure 1-142

We kept the "Yes, create a unit test project" radio button selected — which resulted in a `NerdDinner.Tests` project being added to our solution (Figure 1-143).



Solution Explorer - NerdDinner.T...

```
Solution 'NerdDinner' (2 projects)
  NerdDinner
    Properties
    References
    App_Data
    Content
    Controllers
    Helpers
    Models
    Scripts
    Views
    Default.aspx
    Global.asax
    Web.config
  NerdDinner.Tests
    Properties
    References
    Controllers
      AccountControllerTest.cs
      HomeControllerTest.cs
    App.config
```

Figure 1-143

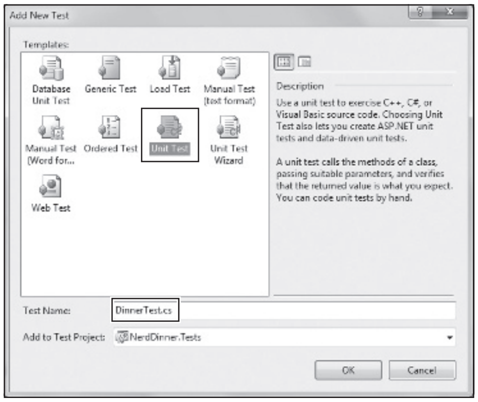
The `NerdDinner.Tests` project references the NerdDinner application project assembly, and enables us to easily add automated tests to it that verify the application.

Creating Unit Tests for Our Dinner Model Class

Let's add some tests to our `NerdDinner.Tests` project that verify the `Dinner` class we created when we built our model layer.

We'll start by creating a new folder within our test project called "Models" where we'll place our model-related tests. We'll then right-click on the folder and choose the Add ➪ New Test menu command. This will bring up the Add New Test dialog.

We'll choose to create a Unit Test and name it **DinnerTest.cs** (Figure 1-144).



Description

Use a unit test to exercise C++, C#, or Visual Basic source code. Choosing Unit Test also lets you create ASP.NET unit tests and data-driven unit tests.

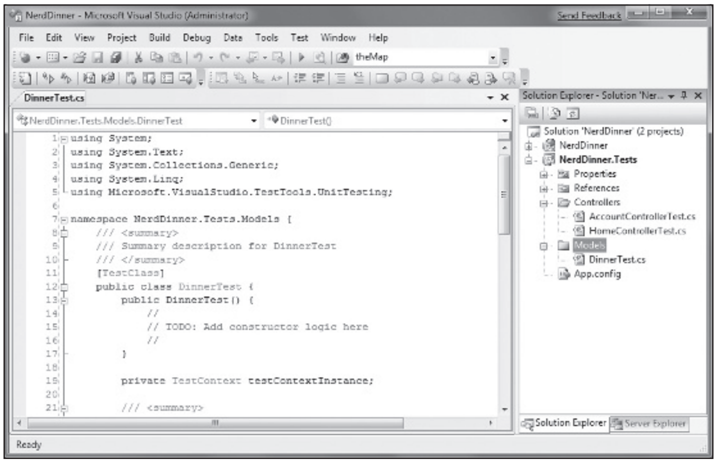
A unit test calls the methods of a class, passing suitable parameters, and verifies that the returned value is what you expect. You can code unit tests by hand.

Test Name: DinnerTest.cs

Add to Test Project: NerdDinner.Tests

Figure 1-144

When we click the OK button, Visual Studio will add (and open) a `DinnerTest.cs` file to the project (Figure 1-145).



```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace NerdDinner.Tests.Models {
    /// <summary>
    /// Summary description for DinnerTest
    /// </summary>
    [TestClass]
    public class DinnerTest {
        public DinnerTest() {
            //
            // TODO: Add constructor logic here
            //
        }

        private TestContext testContextInstance;

        /// <summary>
```

Figure 1-145

The default Visual Studio unit test template has a bunch of boilerplate code within it that I find a little messy. Let's clean it up to just contain the code that follows:

```
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 {

    }
}
```

The [TestClass] attribute on the DinnerTest class above identifies it as a class that will contain tests, as well as optional test initialization and teardown code. We can define tests within it by adding public methods that have a [TestMethod] attribute on them.

In the following code is the first of two tests we'll add that exercise our Dinner class. The first test verifies that our Dinner is invalid if a new Dinner is created without all properties being set correctly. The second test verifies that our Dinner is valid when a Dinner has all properties set with valid values:

```
[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);
    }
}
```

You'll notice above that our test names are very explicit (and somewhat verbose). We are doing this because we might end up creating hundreds or thousands of small tests, and we want to make it easy to quickly determine the intent and behavior of each of them (especially when we are looking through a list of failures in a test runner). The test names should always be named after the functionality they are testing. Above we are using a *Noun_Should_Verb* naming pattern.

We are structuring the tests using the *AAA* testing pattern — which stands for *Arrange, Act, Assert*:

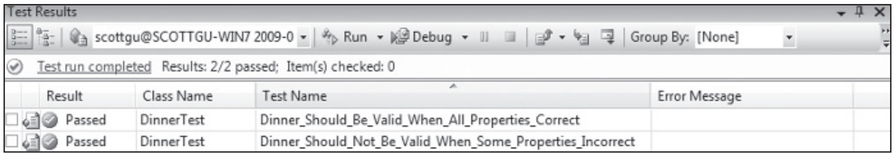
- ❑ **Arrange**: Set up the unit being tested
- ❑ **Act**: Exercise the unit under test and capture results
- ❑ **Assert**: Verify the behavior

When we write tests, we want to avoid having the individual tests do too much. Instead each test should verify only a single concept (which will make it much easier to pinpoint the cause of failures). A good guideline is to try to only have a single assert statement for each test. If you have more than one assert statement in a test method, make sure they are all being used to test the same concept. When in doubt, make another test.

Running Tests

Visual Studio 2008 Professional (and higher editions) includes a built-in test runner that can be used to run Visual Studio Unit Test projects within the IDE. We can select the Test ➪ Run ➪ All Tests in Solution menu command (or press Ctrl-R, A) to run all of our unit tests. Or alternatively we can position our cursor within a specific test class or test method and use the Test ➪ Run ➪ Tests in Current Context menu command (or press Ctrl-R, T) to run a subset of the unit tests.

Let's position our cursor within the `DinnerTest` class and press Ctrl-R, T to run the two tests we just defined. When we do this, a Test Results window will appear within Visual Studio and we'll see the results of our test run listed within it (Figure 1-146).



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

Result	Class Name	Test Name	Error Message
□ Passed	DinnerTest	Dinner_Should_Be_Valid_When_All_Properties_Correct	
□ Passed	DinnerTest	Dinner_Should_Not_Be_Valid_When_Some_Properties_Incorrect	

Figure 1-146

The VS test results window does not show the Class Name column by default. You can add this by right-clicking within the Test Results window and using the Add/Remove Columns menu command.

Our two tests took only a fraction of a second to run — and as you can see they both passed. We can now go on and augment them by creating additional tests that verify specific rule validations, as well as cover the two helper methods — `IsUserHost` and `IsUserRegistered` — that we added to the `Dinner` class. Having all these tests in place for the `Dinner` class will make it much easier and safer to add new business rules and validations to it in the future. We can add our new rule logic to Dinner, and then within seconds verify that it hasn't broken any of our previous logic functionality.

Notice how using a descriptive test name makes it easy to quickly understand what each test is verifying. I recommend using the Tools ➪ Options menu command, opening the Test Tools/Test Execution configuration screen, and checking the "Double-clicking a failed or inconclusive unit test result displays the point of failure in the test" checkbox. This will allow you to double-click on a failure in the test results window and jump immediately to the assert failure.

Creating DinnersController Unit Tests

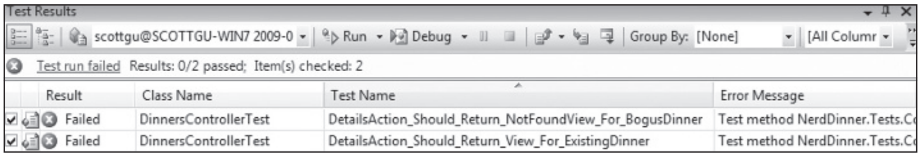
Let's now create some unit tests that verify our `DinnersController` functionality. We'll start by right-clicking on the Controllers folder within our Test project and then choose the Add ➪ New Test menu command. We'll create a Unit Test and name it **DinnersControllerTest.cs**.

We'll create two test methods that verify the `Details` action method on the `DinnersController`. The first will verify that a view is returned when an existing dinner is requested. The second will verify that a `"NotFound"` view is returned when a nonexistent dinner is requested:

```
[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);
        }
    }
```

The previous code compiles cleanly. When we run the tests, though, they both fail (Figure 1-147).



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.C
☑	Failed	DinnersControllerTest	DetailsAction_Should_Return_View_For_ExistingDinner	Test method NerdDinner.Tests.C

Figure 1-147

If we look at the error messages, we'll see that the reason the tests failed was because our `DinnersRepository` class was unable to connect to a database. Our NerdDinner application is using a connection string to a local SQL Server Express file which lives under the `\App_Data` directory of the NerdDinner application project. Because our `NerdDinner.Tests` project compiles and runs in a different directory than the application project, the relative path location of our connection string is incorrect.

We *could* fix this by copying the SQL Express database file to our test project, and then add an appropriate test connection string to it in the `App.config` of our test project. This would get the above tests unblocked and running.

Unit testing code using a real database, though, brings with it a number of challenges. Specifically:

- ❑ It significantly slows down the execution time of unit tests. The longer it takes to run tests, the less likely you are to execute them frequently. Ideally, you want your unit tests to be able to be run in seconds — and have it be something you do as naturally as compiling the project.
- ❑ It complicates the setup and cleanup logic within tests. You want each unit test to be isolated and independent of others (with no side effects or dependencies). When working against a real database you have to be mindful of state and reset it between tests.

Let's look at a design pattern called *dependency injection* that can help us work around these issues and avoid the need to use a real database with our tests.

Dependency Injection

Right now DinnersController is tightly *coupled* to the DinnerRepository class. Coupling refers to a situation where a class explicitly relies on another class in order to work:

```
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);
    }
```

Because the DinnerRepository class requires access to a database, the tightly coupled dependency the DinnersController class has on the DinnerRepository ends up requiring us to have a database in order for the DinnersController action methods to be tested.

We can get around this by employing a design pattern called "dependency injection" — which is an approach where dependencies (like repository classes that provide data access) are no longer implicitly created within classes that use them. Instead, dependencies can be explicitly passed to the class that uses them, using constructor arguments. If the dependencies are defined using interfaces, we then have the flexibility to pass in *fake* dependency implementations for unit test scenarios. This enables us to create test-specific dependency implementations that do not actually require access to a database.

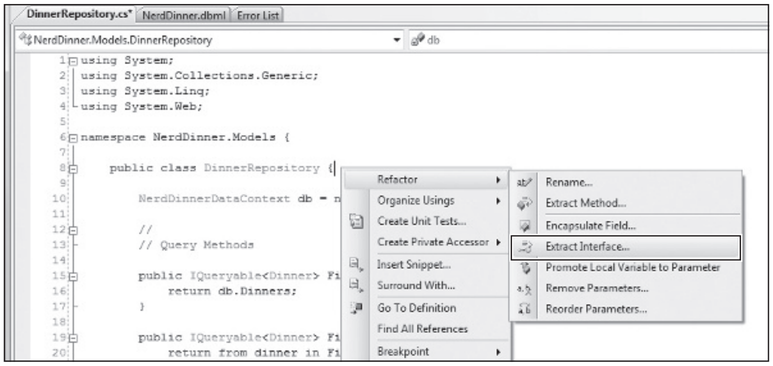
To see this in action, let's implement dependency injection with our DinnersController.

Extracting an IDinnerRepository Interface

Our first step will be to create a new IDinnerRepository interface that encapsulates the repository contract our controllers require to retrieve and update dinners.

We can define this interface contract manually by right-clicking on the \Models folder, and then choosing the Add ➪ New Item menu command and creating a new interface named IDinnerRepository.cs.

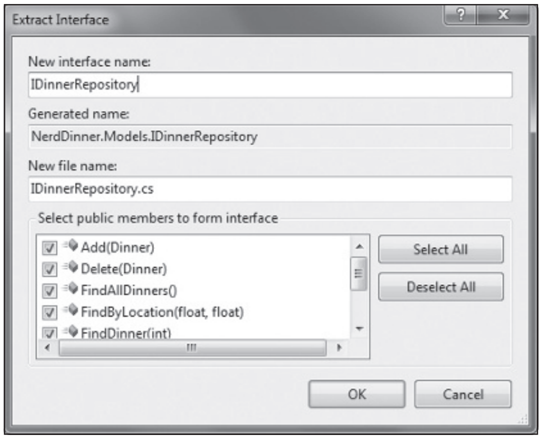
Alternatively, we can use the refactoring tools built into Visual Studio Professional (and higher editions) to automatically extract and create an interface for us from our existing DinnerRepository class. To extract this interface using VS, simply position the cursor in the text editor on the DinnerRepository class, and then right-click and choose the Refactor ➪ Extract Interface menu command (Figure 1-148).



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace NerdDinner.Models {

    public class DinnerRepository {

        NerdDinnerDataContext db = n

        //
        // Query Methods

        public IQueryable<Dinner> Fi
            return db.Dinners;
        }

        public IQueryable<Dinner> Fi
            return from dinner in Fi
```

Figure 1-148

This will launch the Extract Interface dialog and prompt us for the name of the interface to create. It will default to `IDinnerRepository` and automatically select all public methods on the existing `DinnerRepository` class to add to the interface (Figure 1-149).



New interface name: IDinnerRepository

Generated name: NerdDinner.Models.IDinnerRepository

New file name: IDinnerRepository.cs

Select public members to form interface

- ☑ Add(Dinner)
- ☑ Delete(Dinner)
- ☑ FindAllDinners()
- ☑ FindByLocation(float, float)
- ☑ FindDinner(int)

Figure 1-149

When we click the OK button, Visual Studio will add a new `IDinnerRepository` interface to our application:

```
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();
}
```

And our existing DinnerRepository class will be updated so that it implements the interface:

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

Updating DinnersController to Support Constructor Injection

We'll now update the DinnersController class to use the new interface.

Currently DinnersController is hard-coded such that its dinnerRepository field is always a DinnerRepository instance:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    ...
}
```

We'll change it so that the dinnerRepository field is of type IDinnerRepository instead of DinnerRepository. We'll then add two public DinnersController constructors. One of the constructors allows an IDinnerRepository to be passed as an argument. The other is a default constructor that uses our existing DinnerRepository implementation:

```
public class DinnersController : Controller {

    IDinnerRepository dinnerRepository;

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

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

Because ASP.NET MVC, by default creates controller classes using default constructors, our DinnersController at runtime will continue to use the DinnerRepository class to perform data access.

We can now update our unit tests, though, to pass in a *fake* dinner repository implementation using the parameter constructor. This fake dinner repository will not require access to a real database, and instead will use in-memory sample data.

Creating the FakeDinnerRepository Class

Let's create a FakeDinnerRepository class.

We'll begin by creating a `Fakes` directory within our `NerdDinner.Tests` project and then add a new `FakeDinnerRepository` class to it (right-click on the folder and choose Add ➪ New Class, as shown in Figure 1-150):

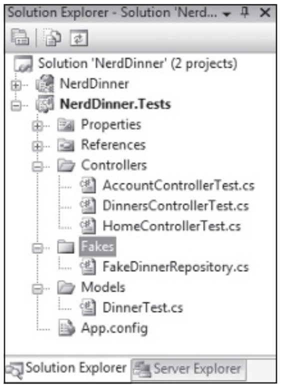
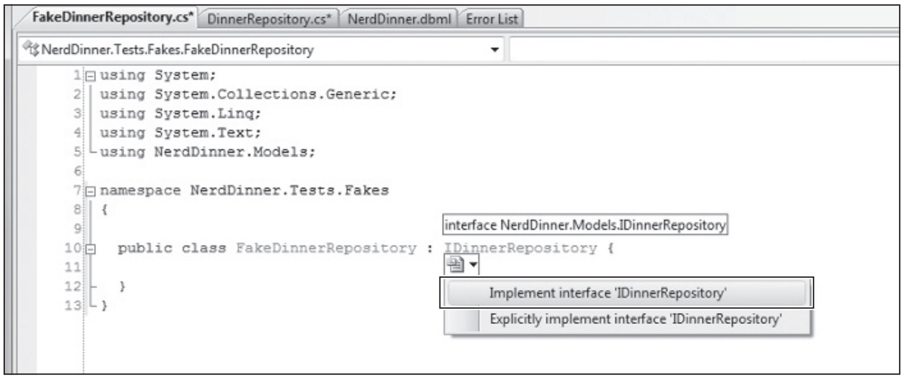


Figure 1-150

We'll update the code so that the `FakeDinnerRepository` class implements the `IDinnerRepository` interface. We can then right-click on it and choose the Implement interface IDinnerRepository context menu command (Figure 1-151).


FakeDinnerRepository.cs*
DinnerRepository.cs*
NerdDinner.dbml
Error List
NerdDinner.Tests.Fakes.FakeDinnerRepository
interface NerdDinner.Models.IDinnerRepository
Implement interface 'IDinnerRepository'
Explicitly implement interface 'IDinnerRepository'

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NerdDinner.Models;

namespace NerdDinner.Tests.Fakes
{

    public class FakeDinnerRepository : IDinnerRepository {

    }
}
```

Figure 1-151

This will cause Visual Studio to automatically add all of the `IDinnerRepository` interface members to our `FakeDinnerRepository` class with default *stub out* implementations:

```
public class FakeDinnerRepository : IDinnerRepository {

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

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

We can then update the `FakeDinnerRepository` implementation to work off of an in-memory `List<Dinner>` collection passed to it as a constructor argument:

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

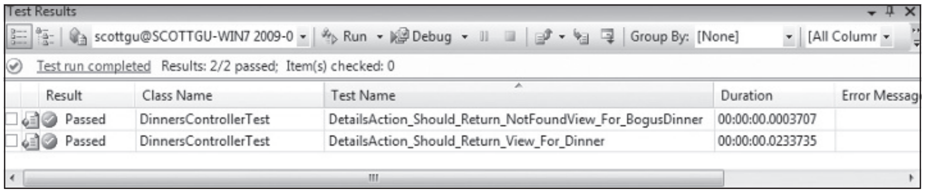
We now have a fake `IDinnerRepository` implementation that does not require a database and can instead work off an in-memory list of `Dinner` objects.

Using the FakeDinnerRepository with Unit Tests

Let's return to the `DinnersController` unit tests that failed earlier because the database wasn't available. We can update the test methods to use a `FakeDinnerRepository` populated with sample in-memory dinner data to the `DinnersController` using the code that follows:

```
    [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);
        }
    }
```

And now when we run these tests, they both pass (Figure 1-152).



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

Result	Class Name	Test Name	Duration	Error Messag
Passed	DinnersControllerTest	DetailsAction_Should_Return_NotFoundView_For_BogusDinner	00:00:00.0003707	
Passed	DinnersControllerTest	DetailsAction_Should_Return_View_For_Dinner	00:00:00.0233735	

Figure 1-152

Best of all, they take only a fraction of a second to run, and do not require any complicated setup/cleanup logic. We can now unit test all of our `DinnersController` action method code (including listing, paging, details, create, update, and delete) without ever needing to connect to a real database.

Dependency Injection Frameworks

Performing manual dependency injection (like we are above) works fine, but does become harder to maintain as the number of dependencies and components in an application increases.

Several dependency injection frameworks exist for .NET that can help provide even more dependency management flexibility. These frameworks, also sometimes called *Inversion of Control* (*IoC*) containers, provide mechanisms that enable an additional level of configuration support for specifying and passing dependencies to objects at runtime (most often using constructor injection). Some of the more popular OSS Dependency Injection/IOC frameworks in .NET include: AutoFac, Ninject, Spring.NET, StructureMap, and Windsor.

> ASP.NET MVC exposes extensibility APIs that enable developers to participate in the resolution and instantiation of controllers, and that enables Dependency Injection/IoC frameworks to be cleanly integrated within this process. Using a DI/IOC framework would also enable us to remove the default constructor from our DinnersController — which would completely remove the coupling between it and the DinnerRepositorys.
>
> We won't be using a dependency injection/IOC framework with our NerdDinner application. But it is something we could consider for the future if the NerdDinner code-base and capabilities grew.

Creating Edit Action Unit Tests

Let's now create some unit tests that verify the Edit functionality of the DinnersController. We'll start by testing the HTTP-GET version of our Edit action:

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

We'll create a test that verifies that a View backed by a DinnerFormViewModel object is rendered back when a valid dinner is requested:

```
[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));
}
```

When we run the test, though, we'll find that it fails because a null reference exception is thrown when the Edit method accesses the User.Identity.Name property to perform the Dinner.IsHostedBy check.

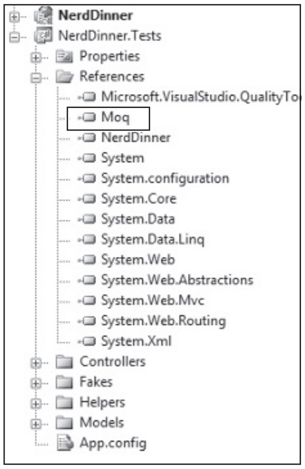
The User object on the Controller base class encapsulates details about the logged-in user, and is populated by ASP.NET MVC when it creates the controller at runtime. Because we are testing the DinnersController outside of a web-server environment, the User object isn't set (hence the null reference exception).

Mocking the User.Identity.Name Property

Mocking frameworks make testing easier by enabling us to dynamically create fake versions of dependent objects that support our tests. For example, we can use a mocking framework in our `Edit` action test to dynamically create a `User` object that our `DinnersController` can use to look up a simulated username. This will avoid a null reference from being thrown when we run our test.

There are many .NET mocking frameworks that can be used with ASP.NET MVC (you can see a list of them here: `www.mockframeworks.com/`). For testing our NerdDinner application, we'll use an open source mocking framework called *Moq*, which can be downloaded for free from `www.mockframeworks.com/moq`.

Once it is downloaded, we'll add a reference in our `NerdDinner.Tests` project to the `Moq.dll` assembly (Figure 1-153).



```
NerdDinner
NerdDinner.Tests
    Properties
    References
        Microsoft.VisualStudio.QualityTo
        Moq
        NerdDinner
        System
        System.configuration
        System.Core
        System.Data
        System.Data.Linq
        System.Web
        System.Web.Abstractions
        System.Web.Mvc
        System.Web.Routing
        System.Xml
    Controllers
    Fakes
    Helpers
    Models
    App.config
```

Figure 1-153

We'll then add an overloaded `CreateDinnersControllerAs(`*`username`*`)` helper method to the test class that takes a username as a parameter, and which then *mocks* the `User.Identity.Name` property on the `DinnersController` instance:

```
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;
}
```

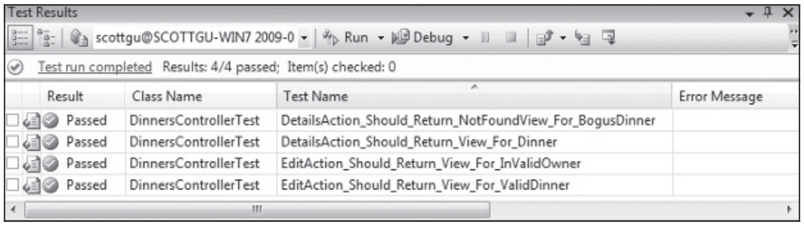
Above, we are using Moq to create a Mock object that fakes a ControllerContext object (which is what ASP.NET MVC passes to Controller classes to expose runtime objects like User, Request, Response, and Session). We are calling the SetupGet method on the Mock to indicate that the HttpContext.User.Identity.Name property on ControllerContext should return the username string we passed to the helper method.

We can mock any number of ControllerContext properties and methods. To illustrate this, I've also added a SetupGet call for the Request.IsAuthenticated property (which isn't actually needed for the tests below — but which helps illustrate how you can mock Request properties). When we are done we assign an instance of the ControllerContext mock to the DinnersController our helper method returns.

We can now write unit tests that use this helper method to test Edit scenarios involving different users:

```
[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");
}
```

And now when we run the tests, they pass (Figure 1-154).



Test run completed Results: 4/4 passed; Item(s) checked: 0

	Result	Class Name	Test Name	Error Message
□	Passed	DinnersControllerTest	DetailsAction_Should_Return_NotFoundView_For_BogusDinner	
□	Passed	DinnersControllerTest	DetailsAction_Should_Return_View_For_Dinner	
□	Passed	DinnersControllerTest	EditAction_Should_Return_View_For_InValidOwner	
□	Passed	DinnersControllerTest	EditAction_Should_Return_View_For_ValidDinner	

Figure 1-154

Testing UpdateModel() Scenarios

We've created tests that cover the HTTP-GET version of the `Edit` action. Let's now create some tests that verify the HTTP-POST version of the `Edit` action:

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

The interesting new testing scenario for us to support with this action method is its usage of the `UpdateModel` helper method on the Controller base class. We are using this helper method to bind form-post values to our `Dinner` object instance.

The following code has two tests that demonstrates how we can supply form posted values for the `UpdateModel` helper method to use. We'll do this by creating and populating a `FormCollection` object, and then assign it to the `ValueProvider` property on the Controller.

The first test verifies that on a successful save the browser is redirected to the details action. The second test verifies that when invalid input is posted the action redisplays the Edit view again with an error message.

```
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");
    }
```

Testing Wrap-Up

We've covered the core concepts involved in unit testing controller classes. We can use these techniques to easily create hundreds of simple tests that verify the behavior of our application.

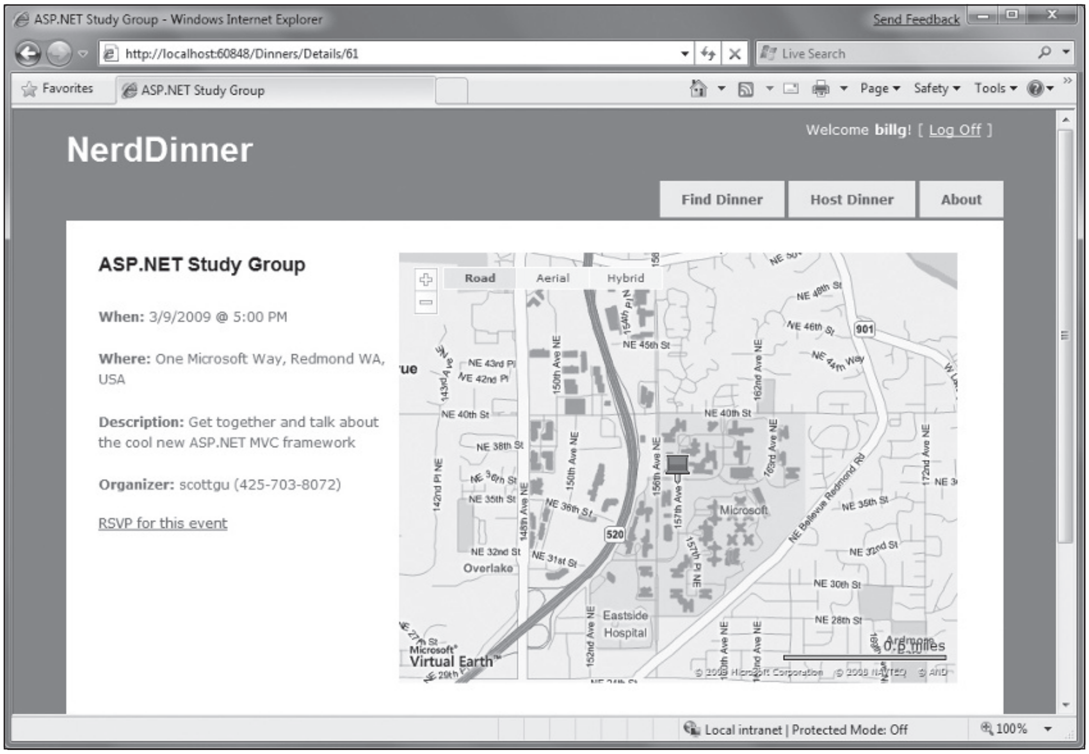
Because our controller and model tests do not require a real database, they are extremely fast and easy to run. We'll be able to execute hundreds of automated tests in seconds, and immediately get feedback as to whether a change we made broke something. This will help provide us the confidence to continually improve, refactor, and refine our application.

We covered testing as the last topic in this chapter — but not because testing is something you should do at the end of a development process! On the contrary, you should write automated tests as early as possible in your development process. Doing so enables you to get immediate feedback as you develop, helps you think thoughtfully about your application's use case scenarios, and guides you to design your application with clean layering and coupling in mind.

A later chapter in this book will discuss Test Driven Development (TDD) and how to use it with ASP.NET MVC. TDD is an iterative coding practice where you first write the tests that your resulting code will satisfy. With TDD you begin each feature by creating a test that verifies the functionality you are about to implement. Writing the unit test first helps ensure that you clearly understand the feature and how it is supposed to work. Only after the test is written (and you have verified that it fails) do you then implement the actual functionality the test verifies. Because you've already spent time thinking about the use case of how the feature is supposed to work, you will have a better understanding of the requirements and how best to implement them. When you are done with the implementation you can re-run the test — and get immediate feedback as to whether the feature works correctly. We'll cover TDD more in Chapter 10.

NerdDinner Wrap-Up

Our initial version of our NerdDinner application is now complete and ready to deploy on the Web (Figure 1-155).



ASP.NET Study Group

When: 3/9/2009 @ 5:00 PM

Where: One Microsoft Way, Redmond WA, USA

Description: Get together and talk about the cool new ASP.NET MVC framework

Organizer: scottgu (425-703-8072)

RSVP for this event

Figure 1-155

We used a broad set of ASP.NET MVC features to build NerdDinner. Hopefully the process of developing it shed some light on how the core ASP.NET MVC features work, and provided context on how these features integrate together within an application.

The following chapters will go into more depth on ASP.NET MVC and discuss its features in detail.