

Microsoft

Moving to
Microsoft[®]
Visual Studio[®]
2010



Patrice Pelland,
Pascal Paré, and Ken Haines

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2011 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2010934433

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editor: Devon Musgrave

Project Editors: Roger LeBlanc and John Pierce

Editorial Production: MPS Limited, a Macmillan Company

Technical Reviewer: Todd Meister; Technical Review Services provided
by Content Master, a member of CM Group, Ltd.

Cover: Tom Draper Design

Body Part No. X17-13257

Table of Contents

Introduction	vii
Part I Moving from Microsoft Visual Studio 2003 to Visual Studio 2010	
1 From 2003 to 2010: Business Logic and Data.....	3
Application Architecture	3
Plan My Night Data in Microsoft Visual Studio 2003	5
Data with the Entity Framework in Visual Studio 2010	6
EF: Importing an Existing Database	7
EF: Model First	16
POCO Templates	22
Putting It All Together	27
Getting Data from the Database	27
Getting Data from the Bing Maps Web Services	32
Parallel Programming	35
AppFabric Caching	36
Summary	38
2 From 2003 to 2010: Designing the Look and Feel.....	39
Introducing the PlanMyNight.Web Project	39
Running the Project	42
Creating the Account Controller	43
Implementing the Functionality	44
Creating the Account View	59
Using the Designer View to Create a Web Form	66
Extending the Application with MEF	74
Print Itinerary Add-in Explained	77
Summary	79

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

3	From 2003 to 2010: Debugging an Application	81
	Visual Studio 2010 Debugging Features.....	81
	Managing Your Debugging Session.....	82
	New Threads Window	100
	Summary.....	101
4	From 2003 to 2010: Deploying an Application	103
	Visual Studio 2010 Web Deployment Packages	103
	Visual Studio 2010 and Web Deployment Packages.....	104
	Summary.....	113

Part II **Moving from Microsoft Visual Studio 2005 to Visual Studio 2010**

5	From 2005 to 2010: Business Logic and Data	117
	Application Architecture	117
	Plan My Night Data in Microsoft Visual Studio 2005.....	119
	Data with the Entity Framework in Visual Studio 2010	121
	EF: Importing an Existing Database.....	122
	EF: Model First	131
	POCO Templates.....	138
	Putting It All Together.....	142
	Getting Data from the Database	142
	Getting Data from the Bing Maps Web Services.....	146
	Parallel Programming.....	149
	AppFabric Caching	150
	Summary.....	152
6	From 2005 to 2010: Designing the Look and Feel	153
	Introducing the PlanMyNight.Web Project	153
	Running the Project	156
	Creating the Account Controller	157
	Implementing the Functionality.....	158
	Creating the Account View	173
	Using the Designer View to Create a Web Form.....	180
	Extending the Application with MEF.....	188
	Print Itinerary Add-in Explained.....	190
	Summary.....	193
7	From 2005 to 2010: Debugging an Application	195
	Visual Studio 2010 Debugging Features.....	195

Managing Your Debugging Session.	196
New Threads Window	213
Summary.	214

Part III **Moving from Microsoft Visual Studio 2008 to Visual Studio 2010**

8	From 2008 to 2010: Business Logic and Data.	217
	Application Architecture	217
	Plan My Night Data in Microsoft Visual Studio 2008.	219
	Data with the Entity Framework in Visual Studio 2010	222
	EF: Importing an Existing Database.	222
	EF: Model First	232
	POCO Templates	239
	Putting It All Together	243
	Getting Data from the Database	243
	Parallel Programming.	247
	AppFabric Caching	248
	Summary.	250
9	From 2008 to 2010: Designing the Look and Feel.	251
	Introducing the PlanMyNight.Web Project	251
	Running the Project	254
	Creating the Account Controller	255
	Implementing the Functionality.	256
	Creating the Account View	270
	Using the Designer View to Create a Web Form.	278
	Extending the Application with MEF.	286
	Print Itinerary Add-in Explained.	288
	Summary.	291
10	From 2008 to 2010: Debugging an Application.	293
	Visual Studio 2010 Debugging Features.	293
	Managing Your Debugging Session.	294
	New Threads Window	311
	Summary.	312
	Index	315
	About the Authors.	321

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

Every time we get close to a new release of Microsoft Visual Studio, we can feel the excitement in the developer community. This release of Visual Studio is certainly no different, but at the same time we can feel a different vibe. In November 2009, at the Microsoft Professional Developer Conference in Los Angeles, participants had the chance to get their hands on the latest beta of this Visual Studio incarnation. The developer community started to see how different this release is compared to any of its predecessors. This might sound familiar, but Visual Studio 2010 constitutes, in our opinion, a big leap and is a true game changer in that it has been designed and developed from the core out.

Looking at posts in the MSDN forums and many other popular developer communities also reveals that many of you—professional developers—are still working in previous versions of Visual Studio. This book will show you how to move to Visual Studio 2010 and try to explain why it's a great time to make this move.

Who Is This Book For?

This book is for professional developers who are working with previous versions of Visual Studio and are looking to make the move to Visual Studio 2010 Professional.

What Is the Book About?

The book is not a language primer, language reference, or single-technology book. It's a book that will help professional developers move from previous versions of Visual Studio (starting with 2003 and moving on up). It will cover the features of Visual Studio 2010 through a sample application. It will go through a lot of the exciting new language features and new versions of the most popular technologies without focusing on the technologies themselves. It will instead put the emphasis on how you get to those new tools and features from Visual Studio 2010. If you are expecting this book to thoroughly cover the new Entity Framework or ASP.NET MVC 2, this is not the book for you. If you want to read a book where the focus is on Visual Studio 2010 and on the reasons for moving to Visual Studio 2010, this is the book for you.

How Will This Book Help Me Move to Visual Studio 2010?

This book will try to answer that question by using a practical approach and by going through the new features and characteristics of Visual Studio 2010 from your point of view—that is, from the view of someone using Visual Studio 2005, for example. To be consistent for all points of view and to cover the same topics from all points of view, we decided to build and use a real application that covers many areas of the product rather than show you many disjointed little samples. This application is named *Plan My Night*, and we'll describe it in detail in this Introduction.

To help as many developers as we can, we decided to divide this book into three parts:

- Part I is for developers moving from Visual Studio 2003
- Part II is for developers moving from Visual Studio 2005
- Part III is for developers moving from Visual Studio 2008

Each part will help developers understand how to use Visual Studio 2010 to create many different types of applications and unlock their creativity independently of the version they are using today. This book will focus on Visual Studio, but we'll also cover many language features that make the move even more interesting.

Each part will follow a similar approach and will include these chapters:

- "Business Logic and Data"
- "Designing the Look and Feel"
- "Debugging the Application"

For example, Part I, "Moving from Microsoft Visual Studio 2003 to Visual Studio 2010," includes a chapter called "From 2003 to 2010: Debugging the Application." Likewise, Part II, "Moving from Microsoft Visual Studio 2005 to Visual Studio 2010," includes a chapter called "From 2005 to 2010: Debugging the Application."

Designing the Look and Feel

These chapters will focus on comparing how the creation of the user interface has evolved through the versions of Visual Studio. They pay attention to the design surface, code editor, tools, and various controls, as well as compare UI validation methods. These chapters also tackle the topic of application extensibility.

Business Logic and Data

These chapters tackle how the application is structured and demonstrate the evolution of the tools and language features available to manage data. They describe the different application layers. They also show how the middle tier is created across versions and how the application will manage caching the data, as well as how to manage getting data in and from the database.

Debugging the Application

These chapters showcase the evolution of all developer aids and debugger tools, as well as compare the different ways to improve the performance of an application. They also discuss the important task of unit-testing your code.

Deploying Plan My Night

Part I, for developers using Visual Studio 2003, also includes one extra chapter, "From 2003 to 2010: Deploying an Application." This chapter goes through the different ways to package, deploy, and deliver your application to your end users. The topic of updating and sending new bits to your customers is also discussed. We feel that Parts II and III, for developers using Visual Studio 2005 and Visual Studio 2008, respectively, didn't require a chapter on deployment.

What Is Plan My Night?

Plan My Night (PMN) is an application that is self-describing, but just to make sure we're on the same page, here's the elevator pitch about PMN:

Plan My Night is designed and developed to help its users plan and manage their evening activities. It allows the user to create events, search for activities and venues, gather information about the activities and the venues, and finally share or produce information about them.

As the saying goes, a picture is worth a thousand words, so take a look at the Plan My Night user interface in Figure I-1.

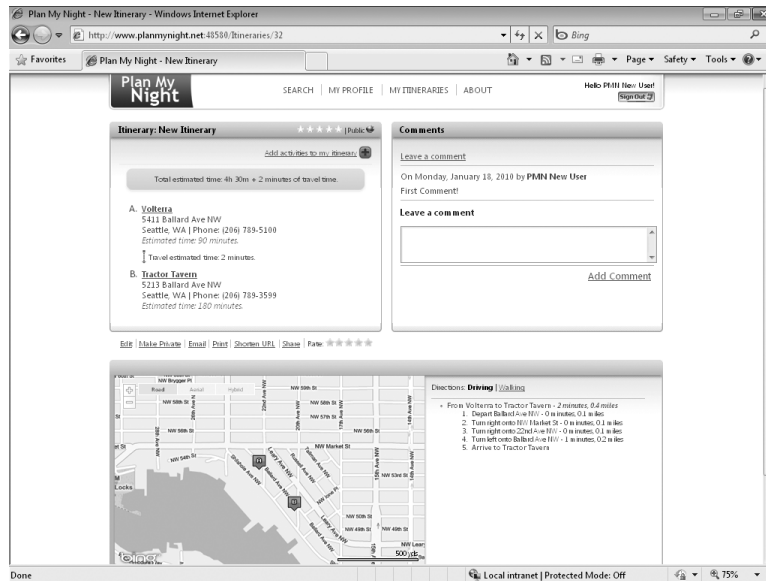


FIGURE I-1 PMN's user interface

In its Visual Studio 2010 version, Plan My Night is built with ASP.NET MVC 2.0 using jQuery and Ajax for UI validation and animation. It uses the Managed Extensibility Framework (MEF) for extending the capabilities of the application by building plug-ins: for sharing to social networks, printing, e-mailing, and so on. We have used the Entity Framework to create the data layer and the Windows Server AppFabric (formerly known as codename "Velocity") to cache data in memory sent to and obtained from the Microsoft SQL Server 2008 database.

We figure that three pictures are better than one, so take a look at Figure I-2 for a diagram displaying the different parts and how they interact with each other and at Figure I-3 to see the different technologies used in building Plan My Night.

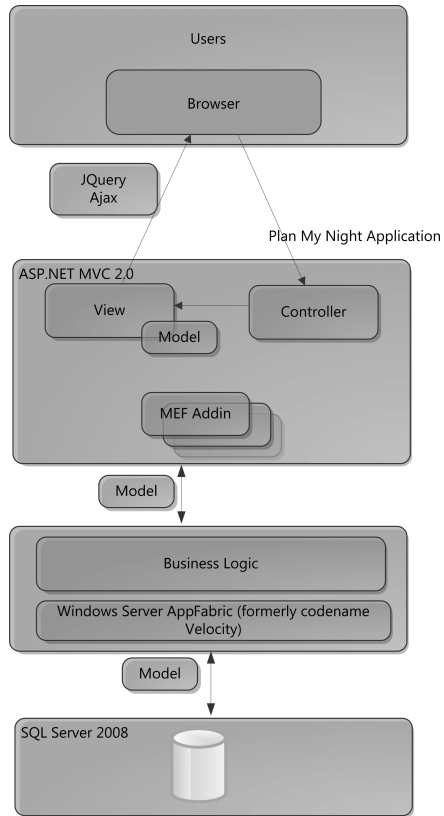


FIGURE I-2 Plan My Night components and interactions

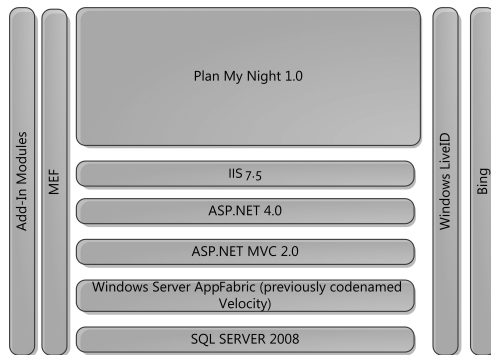


FIGURE I-3 PMN 1.0 and the different technologies used in building it

Why Should You Move to Visual Studio 2010?

There are numerous reasons to move to Visual Studio 2010 Professional, and before we dive into the book parts to examine them, we thought it would be good to list a few from a high-level perspective (presented without any priority ordering):

- Built-in tools for Windows 7, including multitouch and “ribbon” UI components.
- Rich, new editor with built-in Windows Presentation Foundation (WPF) that you can highly customize to suit how you work. Look at Figure I-4 for a sneak peek.
- Multimonitor support.
- New Quick Search, which helps to find relevant results just by quickly typing the first few letters of any method, class, or property.
- Great support for developing and deploying Microsoft Office 2010, SharePoint 2010, and Windows Azure applications.
- Multicore development support that allows you to parallelize your applications, and a new specialized debugger to help you track the tasks and threads.
- Improvements to the ASP.NET AJAX framework, core JavaScript IntelliSense support, and the inclusion in Visual Studio 2010 of jQuery, the open-source library for DOM interactions.
- Multitargeting/multiframe support. Read Scott Guthrie’s blog post to get an understanding of this great feature: <http://weblogs.asp.net/scottgu/archive/2009/08/27/multi-targeting-support-vs-2010-and-net-4-series.aspx>.
- Support for developing WPF and Silverlight applications with enhanced drag-and-drop support and data binding. This includes great new enhancements to the designers, enabling a higher fidelity in rendering your controls, which in turn enables you to discover bugs in rendering before they happen at run time (which is a great improvement from previous versions of Visual Studio). New WPF and Silverlight tools will help you to navigate the visual tree and inspect objects in your rich WPF and Silverlight applications.
- Great support for Team Foundation Server (TFS) 2010 (and previous versions) using Team Explorer. This enables you to use the data and reports that are automatically collected by Visual Studio 2010 and track and analyze the health of your projects with the integrated reports, as well as keep your bugs and tasks up to date.

- Integrated support for test-driven development. Automatic test stub generation and a rich unit test framework are two nice test features that developers can take advantage of for creating and executing unit tests. Visual Studio 2010 has great extensibility points that will enable you to also use common third-party or open-source unit test frameworks directly within Visual Studio 2010.

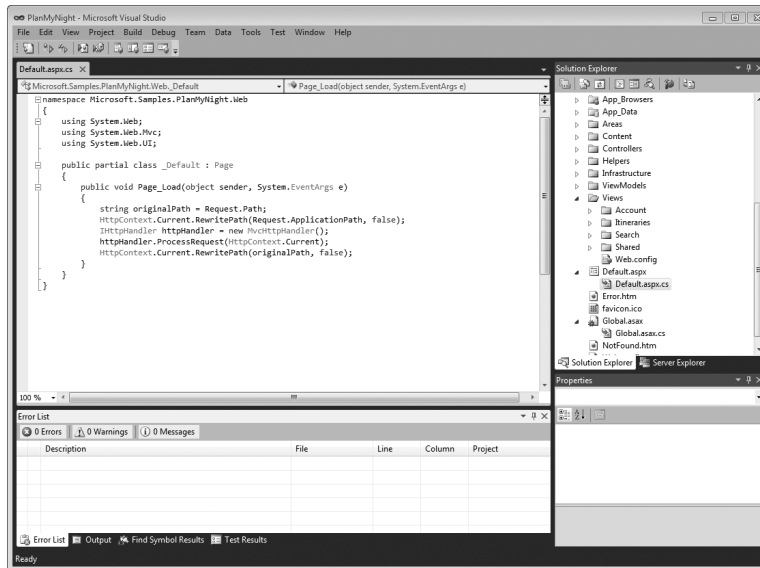


FIGURE I-4 Visual Studio new WPF code editor

This is just a short list of all the new features of Visual Studio 2010 Professional; you'll experience some of them firsthand in this book. You can get the complete list of new features by reading the information presented in the following two locations: [http://msdn.microsoft.com/en-us/library/dd547188\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd547188(VS.100).aspx) and [http://msdn.microsoft.com/en-us/library/bb386063\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb386063(VS.100).aspx).

But the most important reason for many developers and enterprises to make the move is to be able to concentrate on the real problems they're facing rather than spending their time interpreting code. You'll see that with Visual Studio 2010 you can solve those problems faster. Visual Studio 2010 provides you with new, powerful design surfaces and powerful tools that help you write less code, write it faster, and deliver it with higher quality.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site at [Oreilly.com](http://oreilly.com):

1. Go to <http://microsoftpress.oreilly.com>.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, under the cover image, you'll see a list of links.
5. Click View/Submit Errata.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

Part I

Moving from Microsoft Visual Studio 2003 to Visual Studio 2010

Authors Patrice Pelland, Ken Haines, and Pascal Pare

In this part:

From 2003 to 2010: Business Logic and Data (Pascal)	3
From 2003 to 2010: Designing the Look and Feel (Ken)	39
From 2003 to 2010: Debugging an Application (Patrice)	81
From 2003 to 2010: Deploying an Application (Patrice)	103

Chapter 1

From 2003 to 2010: Business Logic and Data

After reading this chapter, you will be able to

- Use the Entity Framework (EF) to build a data access layer using an existing database or with the Model-First approach
- Generate entity types from the Entity Data Model (EDM) Designer using the ADO.NET Entity Framework POCO templates
- Get data from Web services
- Learn about data caching using the Microsoft Windows Server AppFabric (formerly known by the codename "Velocity")

Application Architecture

The Plan My Night (PMN) application allows the user to manage his itinerary activities and share them with others. The data is stored in a Microsoft SQL Server database. Activities are gathered from searches to the Bing Maps Web services.

Let's have a look at the high-level block model of the data model for the application, which is shown in Figure 1-1.

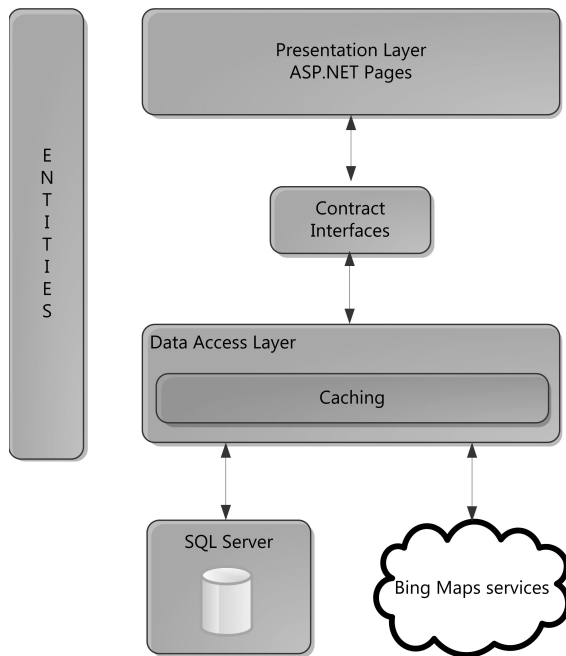


FIGURE 1-1 Plan My Night application architecture diagram

Defining contracts and entity classes that are cleared of any persistence-related code constraints allows us to put them in an assembly that has no persistence-aware code. This approach ensures a clean separation between the Presentation and Data layers.

Let's identify the contract interfaces for the major components of the PMN application:

- *ItinerariesRepository* is the interface to our data store (a Microsoft SQL Server database).
- *IActivitiesRepository* allows us to search for activities (using Bing Maps Web services).
- *ICachingProvider* provides us with our data-caching interface (ASP.NET caching or Windows Server AppFabric caching).



Note This is not an exhaustive list of the contracts implemented in the PMN application.

PMN stores the user's itineraries into an SQL database. Other users will be able to comment and rate each other's itineraries. Figure 1-2 shows the tables used by the PMN application.

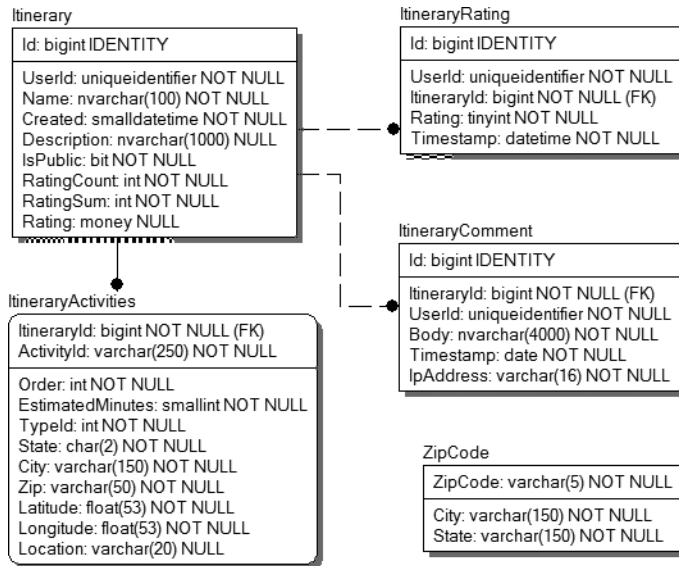


FIGURE 1-2 PlanMyNight database schema



Important The Plan My Night application uses the ASP.NET Membership feature to provide secure credential storage for the users. The user store tables are not shown in Figure 1-2. You can learn more about this feature on MSDN: *ASP.NET 4 - Introduction to Membership* ([http://msdn.microsoft.com/en-us/library/yh26yfzy\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/yh26yfzy(VS.100).aspx)).



Note The ZipCode table is used as a reference repository to provide a list of available Zip Codes and cities so that we can provide autocomplete functionality when the user is entering a search query in the application.

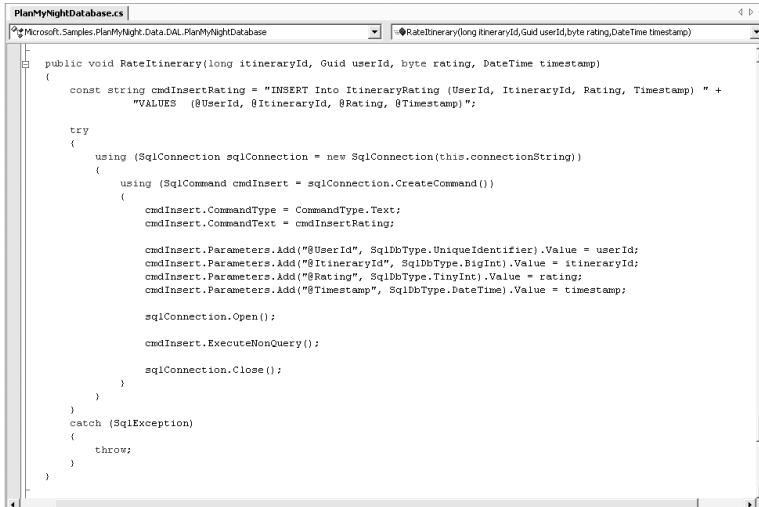
Plan My Night Data in Microsoft Visual Studio 2003

It would be straightforward to create the Plan My Night application in Visual Studio 2003 because it offers all the required tools to help you code the application. However, some of the technologies used back then required you to write a lot more code to achieve the same goals.

In Visual Studio 2003, you could create the required data layer using ADO.NET *DataSet* or *DataReader* to access your database. (See Figure 1-3.) This solution offers you great flexibility because you have complete control over access to the database. On the other hand, it also has some drawbacks:

- You need to know the SQL syntax.
- All queries are specialized. A change in requirement or in the tables will force you to update the queries affected by these changes.

- You need to map the properties of your entity classes using the column name, which is a tedious and error-prone process.
- You have to manage the relations between tables yourself.



```

PlanMyNightDatabase.cs
Microsoft.Samples.PlanMyNight.Data.DAL.PlanMyNightDatabase | RateItinerary(long itineraryId, Guid userId, byte rating, DateTime timestamp)
public void RateItinerary(long itineraryId, Guid userId, byte rating, DateTime timestamp)
{
    const string cmdInsertRating = "INSERT Into ItineraryRating (UserId, ItineraryId, Rating, Timestamp) " +
        "VALUES (@UserId, @ItineraryId, @Rating, @Timestamp)";

    try
    {
        using (SqlConnection sqlConnection = new SqlConnection(this.connectionString))
        {
            using (SqlCommand cmdInsert = sqlConnection.CreateCommand())
            {
                cmdInsert.CommandType = CommandType.Text;
                cmdInsert.CommandText = cmdInsertRating;

                cmdInsert.Parameters.Add("@UserId", SqlDbType.UniqueIdentifier).Value = userId;
                cmdInsert.Parameters.Add("@ItineraryId", SqlDbType.BigInt).Value = itineraryId;
                cmdInsert.Parameters.Add("@Rating", SqlDbType.TinyInt).Value = rating;
                cmdInsert.Parameters.Add("@Timestamp", SqlDbType.DateTime).Value = timestamp;

                sqlConnection.Open();

                cmdInsert.ExecuteNonQuery();

                sqlConnection.Close();
            }
        }
    }
    catch (SqlException)
    {
        throw;
    }
}

```

FIGURE 1-3 ADO.NET Insert query

In the next sections of this chapter, you'll explore some of the new features of Visual Studio 2010 that will help you create the PMN data layer with less code, give you more control of the generated code, and allow you to easily maintain and expand it.

Data with the Entity Framework in Visual Studio 2010

The ADO.NET Entity Framework (EF) allows you to easily create the data access layer for an application by abstracting the data from the database and exposing a model closer to business requirement of the application. The EF has been considerably enhanced in the .NET Framework 4 release.

See Also The MSDN Data Developer Center offers a lot of resources about the ADO.NET Entity Framework (<http://msdn.microsoft.com/en-us/data/aa937723.aspx>) in .NET 4.

You'll use the PlanMyNight project as an example of how to build an application using some of the features of the EF. The next two sections demonstrate two different approaches to generating the data model of PMN. In the first one, you let the EF generate the Entity Data Model (EDM) from an existing database. In the second part, you use a Model First approach, where you first create the entities from the EF designer and generate the Data Definition Language (DDL) scripts to create a database that can store your EDM.

EF: Importing an Existing Database

You'll start with an existing solution that already defines the main projects of the PMN application. If you installed the companion content at the default location, you'll find the solution at this location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\Code\ExistingDatabase. Double-click the PlanMyNight.sln file.

This solution includes all the projects in the following list, as shown in Figure 1-4:

- PlanMyNight.Data: Application data layer
- PlanMyNight.Contracts: Entities and contracts
- PlanMyNight.Bing: Bing Maps services
- PlanMyNight.Web: Presentation layer
- PlanMyNight.AppFabricCaching: AppFabric caching

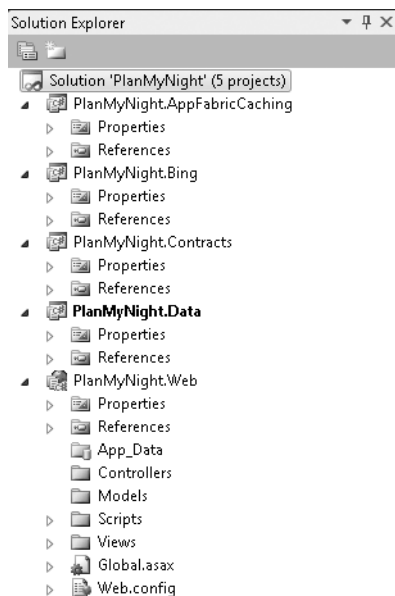


FIGURE 1-4 PlanMyNight solution

The EF allows you to easily import an existing database. Let's walk through this process.

The first step is to add an EDM to the PlanMyNight.Data project. Right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item, and change its name to **PlanMyNight.edmx**, as shown in Figure 1-5.

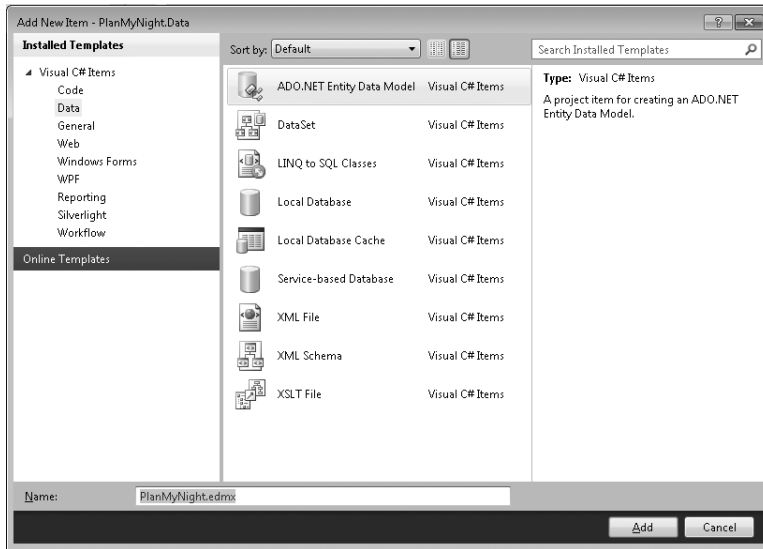


FIGURE 1-5 Add New Item dialog with ADO.NET Entity Data Model selected

The first dialog of the Entity Data Model Wizard allows you to choose the model content. You'll generate the model from an existing database. Select **Generate From Database** and then click **Next**.

You need to connect to an existing database file. Click **New Connection**. Select **Microsoft SQL Server Database File** from the **Choose Data Source** dialog, and click **Continue**. Select the `%userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\ExistingDatabase\PlanMyNight.Web\App_Data\PlanMyNight.mdf` file. (See Figure 1-6.)

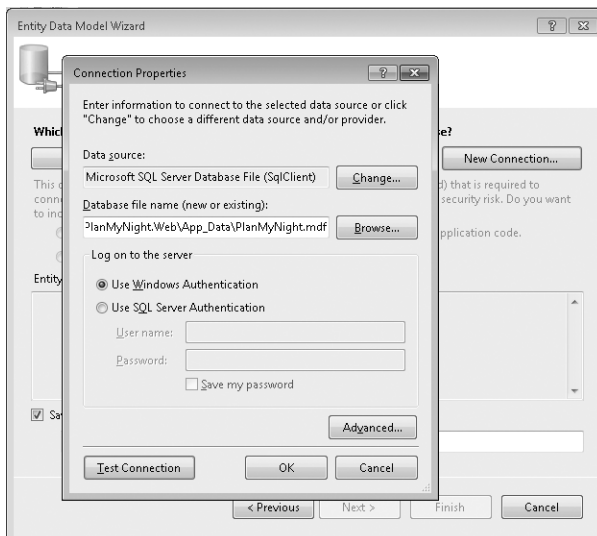


FIGURE 1-6 EDM Wizard database connection

Leave the other fields in the form as is for now, and click Next.



Note You'll get a warning stating that the local data file is not in the current project. Click No to close the dialog because you do not want to copy the database file to the current project.

From the Choose Your Database Objects dialog, select the *Itinerary*, *ItineraryActivities*, *ItineraryComment*, *ItineraryRating*, and *ZipCode* tables and the *UserProfile* view. Select the *RetrievalItinerariesWithinArea* stored procedure. Change the Model Namespace value to **Entities** as shown in Figure 1-7.

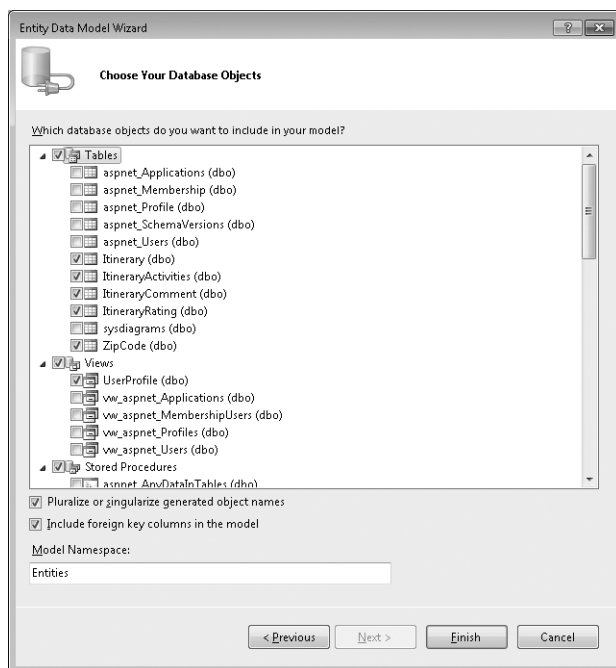


FIGURE 1-7 EDM Wizard: Choose Your Database Objects page

Click Finish to generate your EDM.

Fixing the Generated Data Model

You now have a model representing a set of entities matching your database. The wizard has generated all the navigation properties associated with the foreign keys from the database.

The PMN application requires only the navigation property *ItineraryActivities* from the *Itinerary* table, so you can go ahead and delete all the other navigation properties. You'll also need to rename the *ItineraryActivities* navigation property to **Activities**. Refer to Figure 1-8 for the updated model.

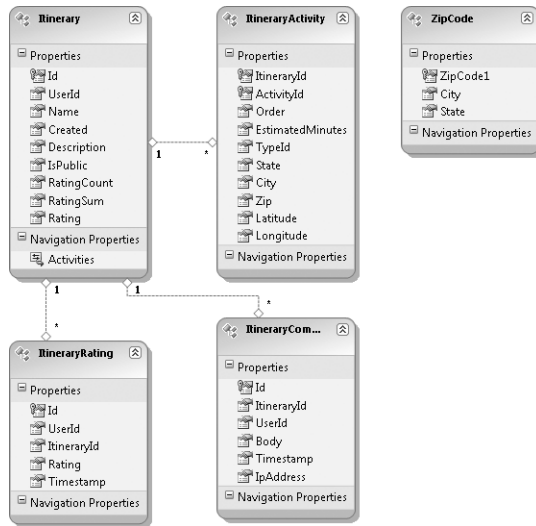


FIGURE 1-8 Model imported from the PlanMyNight database

Notice that one of the properties of the ZipCode entity has been generated with the name *ZipCode1* because the table itself is already named ZipCode and the name has to be unique. Let's fix the property name by double-clicking it. Change the name to **Code**, as shown in Figure 1-9.

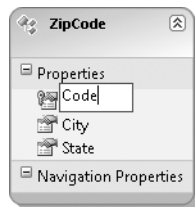


FIGURE 1-9 ZipCode entity

Build the solution by pressing Ctrl+Shift+B. When looking at the output window, you'll notice two messages from the generated EDM. You can discard the first one because the Location column is not required in PMN. The second message reads as follows:

The table/view 'dbo.UserProfile' does not have a primary key defined and no valid primary key could be inferred. This table/view has been excluded. To use the entity, you will need to review your schema, add the correct keys, and uncomment it.

When looking at the UserProfile view, you'll notice it does not explicitly define a primary key even though the UserName column is unique.

You need to modify the EDM manually to fix the UserProfile view mapping so that you can access the UserProfile data from the application.

From the project explorer, right-click the PlanMyNight.edmx file and then select Open With. Choose XML (Text) Editor from the Open With dialog as shown in Figure 1-10. Click OK to open the XML file associated with your model.

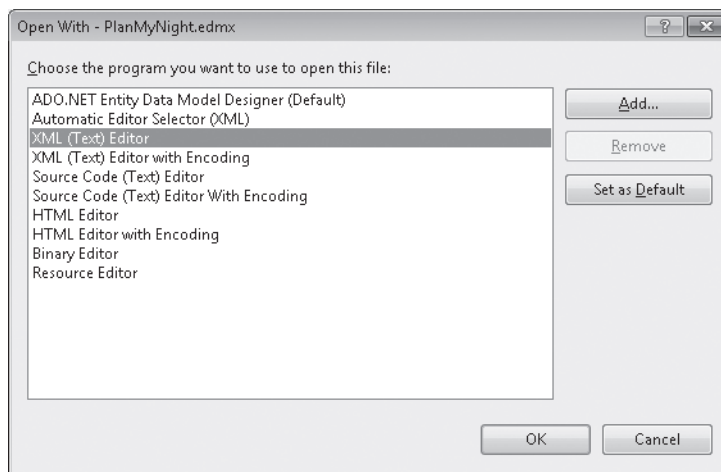


FIGURE 1-10 Open PlanMyNight.edmx in the XML Editor



Note You'll get a warning stating that the PlanMyNight.edmx file is already open. Click Yes to close it.

The generated code was commented out by the code-generation tool because there was no primary key defined. To be able to use the UserProfile view from the designer, you need to uncomment the UserProfile entity type and add the Key tag to it. Search for UserProfile in the file. Uncomment the entity type, add a Key tag and set its name to **UserName**, and make the *UserName* property not nullable. Refer to Listing 1-1 to see the updated entity type.

LISTING 1-1 UserProfile Entity Type XML Definition

```
<EntityType Name="UserProfile">
  <Key>
    <PropertyRef Name="UserName"/>
  </Key>
  <Property Name="UserName" Type="uniqueidentifier" Nullable="false" />
  <Property Name="FullName" Type="varchar" MaxLength="500" />
  <Property Name="City" Type="varchar" MaxLength="500" />
  <Property Name="State" Type="varchar" MaxLength="500" />
  <Property Name="PreferredActivityTypeId" Type="int" />
</EntityType>
```

If you close the XML file and try to open the EDM Designer, you'll get the following error message in the designer: "The Entity Data Model Designer is unable to display the file you requested. You can edit the model using the XML Editor."

There is a warning in the Error List pane that can give you a little more insight into what this error is all about:

Error 11002: Entity type 'UserProfile' has no entity set.

You need to define an entity set for the UserProfile type so that it can map the entity type to the store schema. Open the PlanMyNight.edmx file in the XML editor so that you can define an entity set for UserProfile. At the top of the file, just above the Itinerary entity set, add the XML code shown in Listing 1-2.

LISTING 1-2 UserProfile EntitySet XML Definition

```
<EntitySet Name="UserProfile" EntityType="Entities.Store.UserProfile"
  store:Type="Views" store:Schema="dbo" store:Name="UserProfile">
  <DefiningQuery>
    SELECT
      [UserProfile].[UserName] AS [UserName],
      [UserProfile].[FullName] AS [FullName],
      [UserProfile].[City] AS [City],
      [UserProfile].[State] AS [State],
      [UserProfile].[PreferredActivityTypeId] as [PreferredActivityTypeId]
    FROM [dbo].[UserProfile] AS [UserProfile]
  </DefiningQuery>
</EntitySet>
```

Save the EDM XML file, and reopen the EDM Designer. Figure 1-11 shows the UserProfile view in the Entities.Store section of the Model Browser.



Tip You can open the Model Browser from the View menu by clicking Other Windows and selecting the Entity Data Model Browser item.

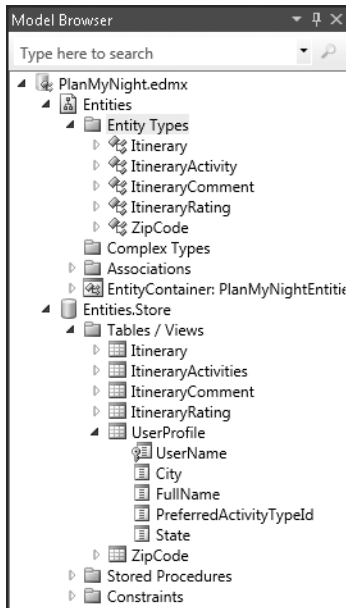


FIGURE 1-11 Model Browser with the UserProfile view

Now that the view is available in the store metadata, you add the UserProfile entity and map it to the UserProfile view. Right-click in the background of the EDM Designer, select Add, and then choose Entity. You'll see the dialog shown in Figure 1-12.

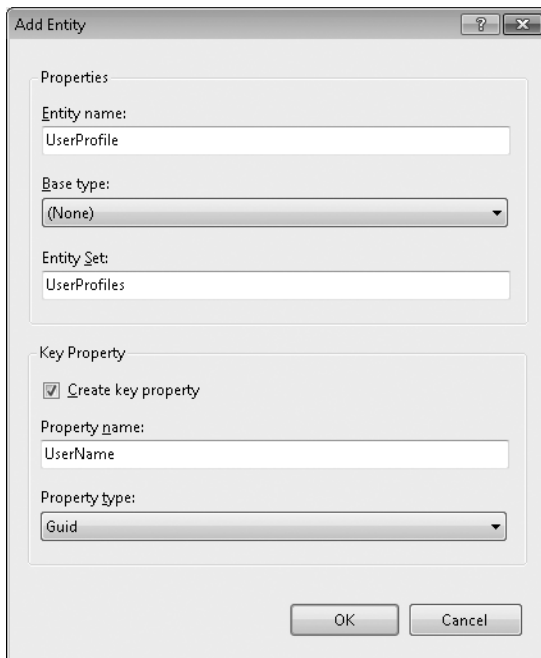


FIGURE 1-12 Add Entity dialog

Complete the dialog as shown in Figure 1-12, and click OK to generate the entity.

You need to add the remaining properties: *City*, *State*, and *PreferredActivityTypeId*. To do so, right-click the *UserProfile* entity, select *Add*, and then select *Scalar Property*. After the property is added, set the *Type*, *Max Length*, and *Unicode* field values. Table 1-1 shows the expected values for each of the fields.

TABLE 1-1 UserProfile Entity Properties

Name	Type	Max Length	Unicode
<i>FullName</i>	<i>String</i>	500	False
<i>City</i>	<i>String</i>	500	False
<i>State</i>	<i>String</i>	500	False
<i>PreferredActivityTypeId</i>	<i>Int32</i>	NA	NA

Now that you have created the *UserProfile* entity, you need to map it to the *UserProfile* view. Right-click the *UserProfile* entity, and select *Table Mapping* as shown in Figure 1-13.

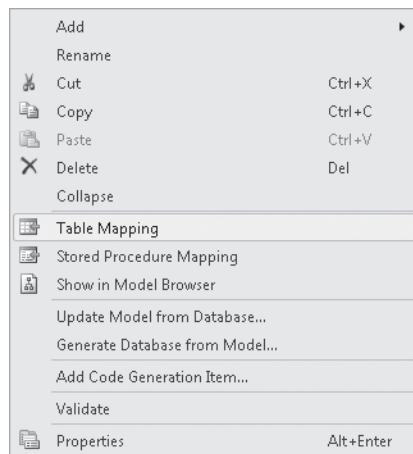


FIGURE 1-13 Table Mapping menu item

Then select the *UserProfile* view from the drop-down box as shown in Figure 1-14. Ensure that all the columns are correctly mapped to the entity properties. The *UserProfile* view of our store is now accessible from the code through the *UserProfile* entity.

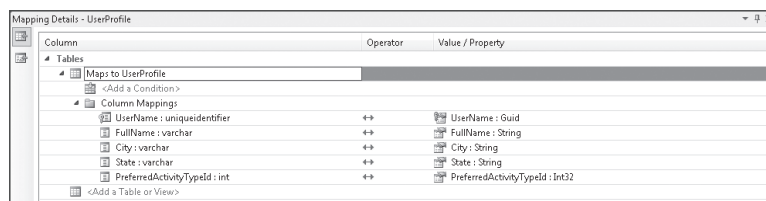


FIGURE 1-14 UserProfile mapping details

Stored Procedure and Function Imports

The Entity Data Model Wizard has created an entry in the storage model for the *RetrievalItinerariesWithinArea* stored procedure you selected in the last step of the wizard. You need to create a corresponding entry to the conceptual model by adding a Function Import entry.

From the Model Browser, open the Stored Procedures folder in the Entities.Store section. Right-click *RetrievalItineraryWithinArea*, and then select Add Function Import. The Add Function Import dialog appears as shown in Figure 1-15. Specify the return type by selecting Entities and then select the Itinerary item from the drop-down box. Click OK.

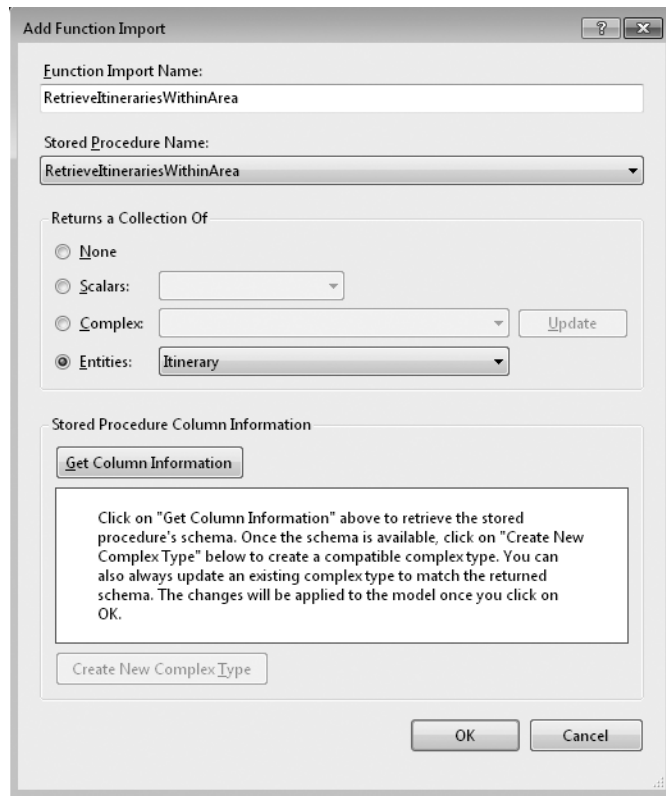


FIGURE 1-15 Add Function Import dialog

The *RetrievalItinerariesWithinArea* function import was added to the Model Browser as shown in Figure 1-16.

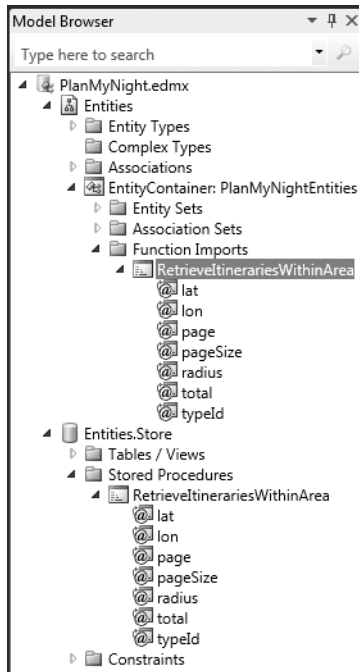


FIGURE 1-16 Function Imports in the Model Browser

You can now validate the EDM by right-clicking on the design surface and selecting Validate. There should be no error or warning.

EF: Model First

In the prior section, you saw how to use the EF designer to generate the model by importing an existing database. The EF designer in Visual Studio 2010 also supports the ability to generate the Data Definition Language (DDL) file that will allow you to create a database based on your entity model. In this section, you'll use a new solution to learn how to generate a database script from a model.

You can start from an empty model by selecting the Empty Model option from the Entity Data Model Wizard. (See Figure 1-17.)



Note To get the wizard, right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item.

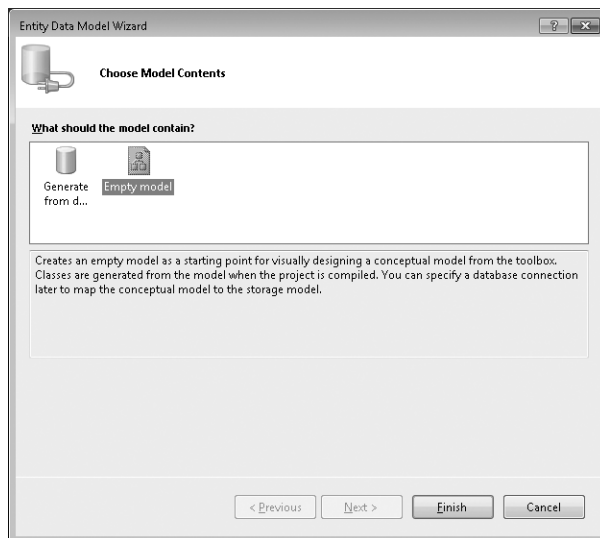


FIGURE 1-17 EDM Wizard: Empty model

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\Code\ModelFirst by double-clicking the PlanMyNight.sln file.

The PlanMyNight.Data project from this solution already contains an EDM file named PlanMyNight.edmx with some entities already created. These entities match the data schema you saw in Figure 1-2.

The Entity Model designer lets you easily add an entity to your data model. Let's add the missing ZipCode entity to the model. From the toolbox, drag an Entity item into the designer, as shown in Figure 1-18. Rename the entity as **ZipCode**. Rename the *Id* property as **Code**, and change its type to *String*.

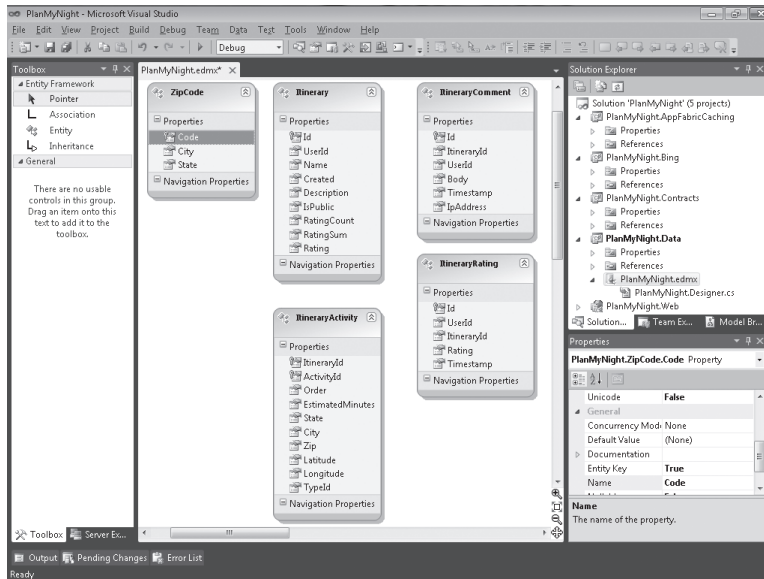


FIGURE 1-18 Entity Model designer

You need to add the *City* and *State* properties to the entity. Right-click the ZipCode entity, select Add, and then choose Scalar Property. Ensure that each property has the values shown in Table 1-2.

TABLE 1-2 ZipCode Entity Properties

Name	Type	Fixed Length	Max Length	Unicode
<i>Code</i>	<i>String</i>	False	5	False
<i>City</i>	<i>String</i>	False	150	False
<i>State</i>	<i>String</i>	False	150	False

Add the relations between the ItineraryComment and Itinerary entities. Right-click the designer background, select Add, and then choose Association. (See Figure 1-19.)

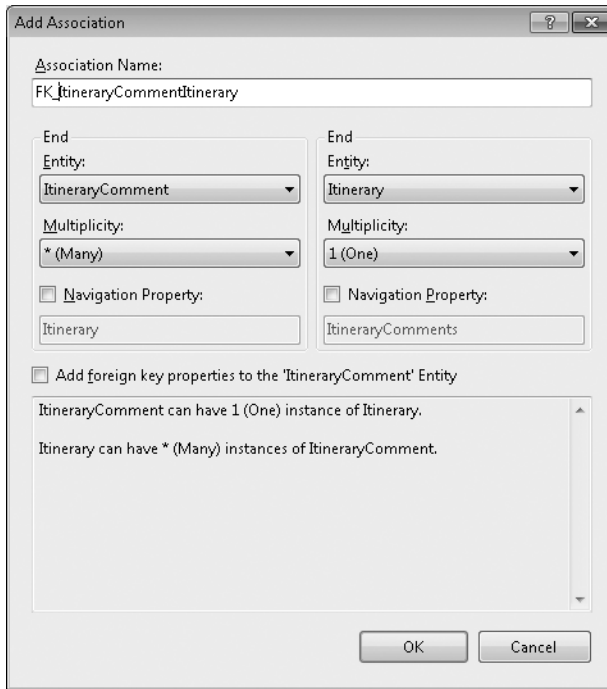


FIGURE 1-19 Add Association dialog for *FK_ItineraryCommentItinerary*

Set the association name to **FK_ItineraryCommentItinerary**, and then select the entity and the multiplicity for each end, as shown in Figure 1-19. After the association is created, double-click the association line to set the Referential Constraint as shown in Figure 1-20.

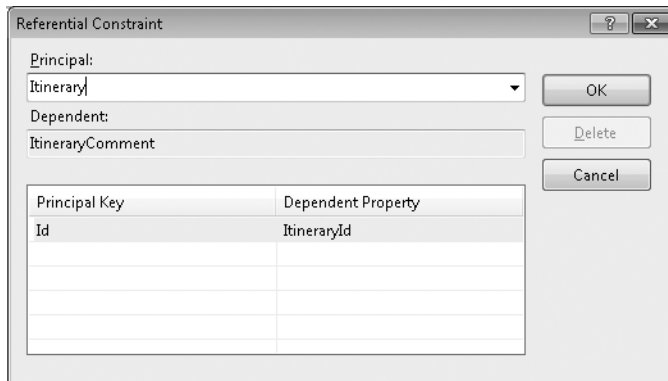


FIGURE 1-20 Association Referential Constraint dialog

Add the association between the *ItineraryRating* and *Itinerary* entities. Right-click the designer background, select **Add**, and then choose **Association**. Set the association name to **FK_ItineraryItineraryRating** and then select the entity and the multiplicity for each end as in the previous step, except set the first end to **ItineraryRating**. Double-click on the association line, and set the Referential Constraint as shown in Figure 1-20. Note that the Dependent field will read *ItineraryRating* instead of *ItineraryComment*.

Create a new association between the *ItineraryActivity* and *Itinerary* entities. For the *FK_ItineraryItineraryActivity* association, you also want to create a navigation property and name it **Activities**, as shown in Figure 1-21. After the association is created, set the Referential Constraint for this association by double-clicking on the association line.

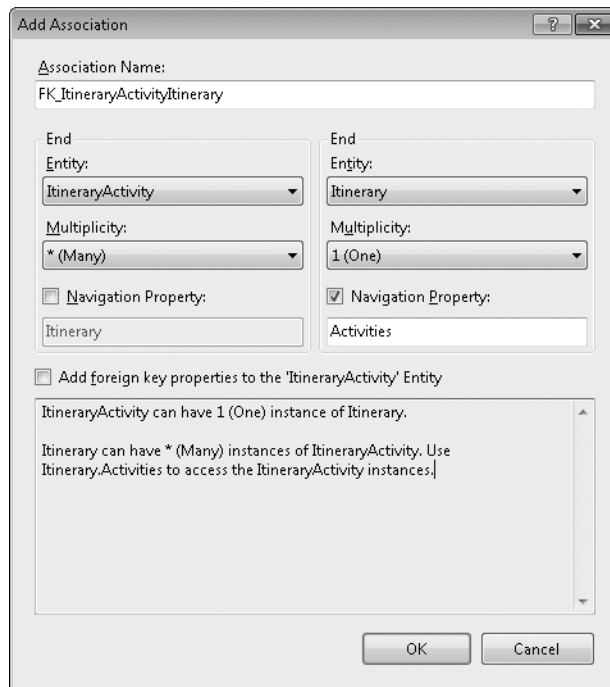


FIGURE 1-21 Add Association dialog for *FK_ItineraryActivityItinerary*

Generating the Database Script from the Model

Your data model is now complete, but there is no mapping or store associated with it. The EF designer offers you the possibility of generating a database script from our model.

Right-click on the designer surface, and choose **Generate Database From Model** as shown in Figure 1-22.

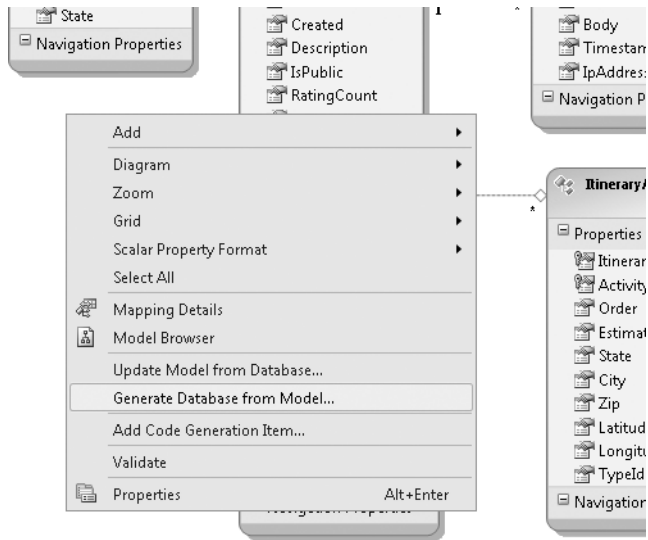


FIGURE 1-22 Generate Database From Model menu item

The Generate Database Wizard requires a data connection. The wizard uses the connection information to translate the model types to the database type and to generate a DDL script targeting this database.

Select New Connection, select Microsoft SQL Server Database File from the Choose Data Source dialog, and click Continue. Select the database file located at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\Code\ModelFirst\Data\PlanMyNight.mdf. (See Figure 1-23.)

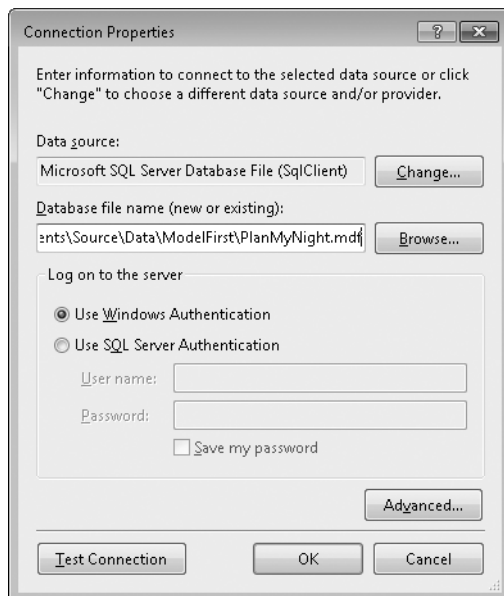


FIGURE 1-23 Generate a script database connection

After your connection is configured, click Next to get to the final page of the wizard as shown in Figure 1-24. When you click Finish, the generated T-SQL PlanMyNight.edmx.sql file is added to your project. The DDL script will generate the primary and foreign key constraints for your model.

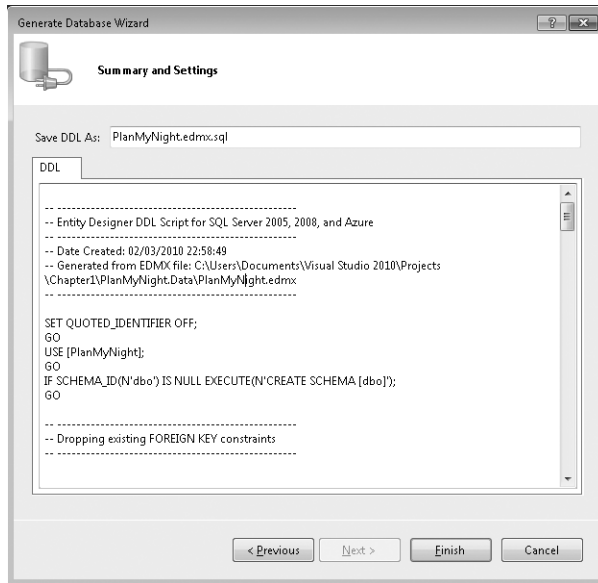


FIGURE 1-24 Generated T-SQL file

The EDM is also updated to ensure your newly created store is mapped to the entities. You can now use the generated DDL script to add the tables to the database. Also, you now have a data layer that exposes strongly typed entities that you can use in your application.



Important Generating the complete PMN database would require adding the remaining tables, stored procedures, and triggers used by the application. Instead of performing all these operations, we will go back to the solution we had at the end of the “EF: Importing an Existing Database” section.

POCO Templates

The EDM Designer uses T4 templates to generate the code for the entities. So far, we have let the designer create the entities using the default templates. You can take a look at the code generated by opening the PlanMyNight.Designer.cs file associated with PlanMyNight.edmx. The generated entities are based on the EntityObject type and decorated with attributes to allow the EF to manage them at run time.



Note T4 stands for *Text Template Transformation Toolkit*. T4 support in Visual Studio 2010 allows you to easily create your own templates and generate any type of text file (Web, resource, or source). To learn more about code generation in Visual Studio 2010, visit [Code Generation and Text Templates](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx) ([http://msdn.microsoft.com/en-us/library/bb126445\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx)).

The EF also supports POCO entity types. POCO classes are simple objects with no attributes or base class related to the framework. (Listing 1-3, in the next section, shows the POCO class for the ZipCode entity.) The EF uses the names of the types and the properties of these objects to map them to the model at run time.



Note POCO stands for *Plain-Old CLR Objects*.

ADO.NET POCO Entity Generator

Let's re-open the %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\Code\ExistingDatabase\PlanMyNight.sln file.

Open the PlanMyNight.edmx file, right-click on the design surface, and choose Add Code Generation Item. This opens a dialog like the one shown in Figure 1-25, where you can select the template you want to use. Select the ADO.NET POCO Entity Generator template, and name it **PlanMyNight.tt**. Then click the Add button.



Note You might get a security warning about running this text template. Click OK to close the dialog because the source for this template is trusted.

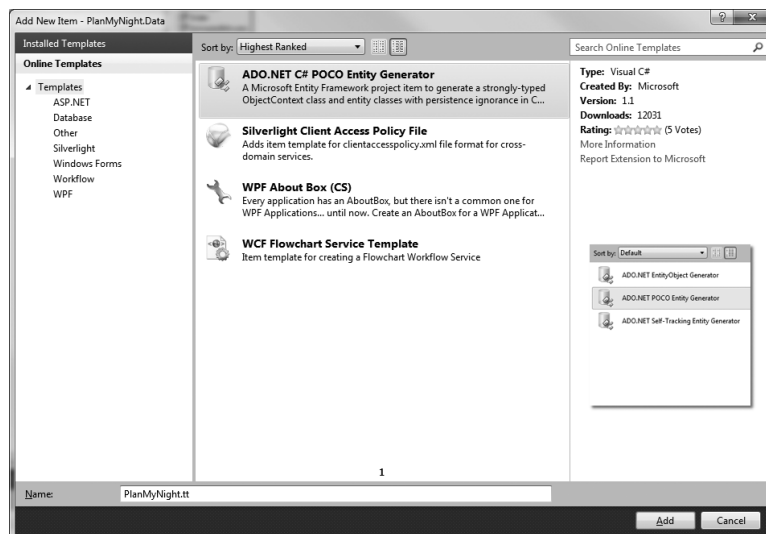


FIGURE 1-25 Add New Item dialog

Two files, `PlanMyNight.tt` and `PlanMyNight.Context.tt`, have been added to your project, as shown in Figure 1-26. These files replace the default code-generation template, and the code is no longer generated in the `PlanMyNight.Designer.cs` file.

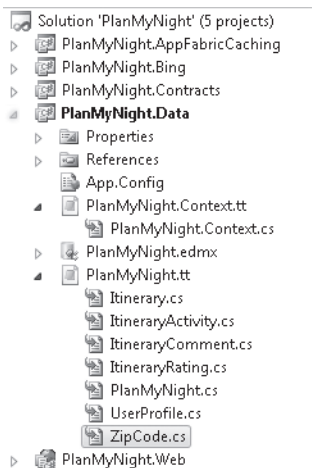


FIGURE 1-26 Added templates

The `PlanMyNight.tt` template produces a class file for each entity in the model. Listing 1-3 shows the POCO version of the `ZipCode` class.

LISTING 1-3 POCO Version of the `ZipCode` Class

```
namespace Microsoft.Samples.PlanMyNight.Data
{
    public partial class ZipCode
    {
        #region Primitive Properties
        public virtual string Code
        {
            get;
            set;
        }
        public virtual string City
        {
            get;
            set;
        }
        public virtual string State
        {
            get;
            set;
        }
        #endregion
    }
}
```



Tip Partial classes were added to C# 2.0. They allow splitting the implementation of a class over multiple files, where each file can contain one or more members and the files are combined when the application is compiled. Partial classes are really useful if you need to add code to automatically generated classes because the code is added outside of the generated file and it will not be overridden if the class is regenerated.



Tip C# 3.0 introduced a new feature called *automatic properties*. The backing field is created at compile time if the compiler finds empty *get* or *set* blocks.

The other file, `PlanMyNight.Context.cs`, generates the *ObjectContext* object for the `PlanMyNight.edmx` model. This is the object you'll use to interact with the database.



Tip The POCO templates will automatically update the generated classes to reflect the changes to your model when you save the `.edmx` file.

Moving the Entity Classes to the Contracts Project

We have designed the PMN application architecture to ensure that the presentation layer was persistence ignorant by moving the contracts and entity classes to an assembly that has no reference to the storage.

Visual Studio 2003 Even though it was possible to write add-ins in Visual Studio 2003 to generate code based on a database, it was not easy and you had to maintain these tools. The EF uses T4 templates to generate both the database schema and the code. These templates can easily be customized to your needs.

The ADO.NET POCO templates split the generation of the entity classes into a separate template, allowing you to easily move these entities to a different project.

You are going to move the `PlanMyNight.tt` file to the `PlanMyNight.Contracts` project. Right-click the `PlanMyNight.tt` file, and select `Cut`. Right-click the `Entities` folder in the `PlanMyNight.Contracts` project, and select `Paste`. The result is shown in Figure 1-27.

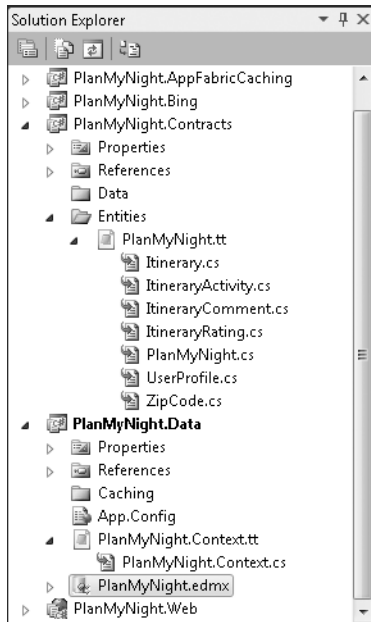


FIGURE 1-27 POCO template moved to the Contracts project

The `PlanMyNight.tt` template relies on the metadata from the EDM model to generate the entity type's code. You need to fix the relative path used by the template to access the EDMX file.

Open the `PlanMyNight.tt` template, and locate the following line:

```
string inputFile = @"PlanMyNight.edmx";
```

Fix the file location so that it points to the `PlanMyNight.edmx` file in the `PlanMyNight.Data` project:

```
string inputFile = @"..\..\PlanMyNight.Data\PlanMyNight.edmx";
```

The entity classes are regenerated when you save the template.

You also need to update the `PlanMyNight.Context.tt` template in the `PlanMyNight.Contracts` project because the entity classes are now in the `Microsoft.Samples.PlanMyNight.Entities` namespace instead of the `Microsoft.Samples.PlanMyNight.Data` namespace. Open the `PlanMyNight.Context.tt` file, and update the `using` section to include the new namespace:

```
using System;
using System.Data.Objects;
using System.Data.EntityClient;
using Microsoft.Samples.PlanMyNight.Entities;
```

Build the solution by pressing `Ctrl+Shift+B`. The project should now compile successfully.

Putting It All Together

Now that you have created the generic code layer to interact with your SQL database, you are ready to start implementing the functionalities specific to the PMN application. In the upcoming sections, you'll walk through this process, briefly look at getting the data from the Bing Maps services, and get a quick introduction to the Microsoft Windows Server AppFabric Caching feature used in PMN.

There is a lot of plumbing pieces of code required to get this all together. To simplify the process, you'll use an updated solution where the contracts, entities, and most of the connecting pieces to the Bing Maps services have been coded. The solution will also include the PlanMyNight.Data.Test project to help you validate the code from the PlanMyNight.Data project.



Note Testing in Visual Studio 2010 will be covered in Chapter 3.

Getting Data from the Database

At the beginning of this chapter, we decided to group the operations on the Itinerary entity in the *ItinerariesRepository* repository interface. Some of these operations are

- Searching for Itinerary by Activity
- Searching for Itinerary by ZipCode
- Searching for Itinerary by Radius
- Adding a new Itinerary

Let's take a look at the corresponding methods in the *ItinerariesRepository* interface:

- *SearchByActivity* allows searching for itineraries by activity and returning a page of data.
- *SearchByZipCode* allows searching for itineraries by Zip Code and returning a page of data.
- *SearchByRadius* allows searching for itineraries from a specific location and returning a page of data.
- *Add* allows you to add an itinerary to the database.

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 1\Code\Final by double-clicking the PlanMyNight.sln file.

Select the PlanMyNight.Data project, and open the ItinerariesRepository.cs file. This is the *ItinerariesRepository* interface implementation. Using the PlanMyNightEntities Object Context you generated earlier, you can write LINQ queries against your model, and the EF will translate these queries to native T-SQL that will be executed against the database.



Note LINQ stands for *Language Integrated Query* and was introduced in the .NET Framework 3.5. It adds native data-querying capability to the .NET Framework so that you don't have to worry about learning or maintaining custom SQL queries. LINQ allows you to use strongly typed objects, and Visual Studio IntelliSense lets you select the properties or methods that are in the current context as shown in Figure 1-28. To learn more about LINQ, visit the [.NET Framework Developer Center](http://msdn.microsoft.com/en-us/netframework/aa904594.aspx) (<http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>).

```
public PagingResult<Itinerary> SearchByActivity(string activityId, int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.
}
}
```

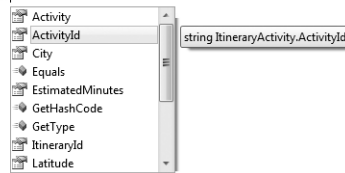


FIGURE 1-28 IntelliSense support for LINQ queries

Navigate to the *SearchByActivity* function definition. This method must return a set of itineraries where the *IsPublic* flag is set to true and where one of their activities has the same *activityId* that was passed in the argument to the function. You also need to order the result itinerary list by the rating field.

Visual Studio 2003 Implementing each method to retrieve the itinerary in Visual Studio 2003 would have required writing tailored SQL. With the EF and LINQ, any query becomes trivial and changes can be easily implemented at the code level!

Using standard LINQ operators, you can implement *SearchByActivity* as shown in Listing 1-4. Add the highlighted code to the *SearchByActivity* method body.

Visual Studio 2003 Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce the concept of type parameters, which make it possible to design classes and methods that defer the specification of types until the class or method is declared. They are often used with a collection, where the type parameter is used as a placeholder for the type of objects that it stores.

PMN uses generics to store results of different types. With Visual Studio 2003, you could have written such a class by using an *ArrayList*:

```
public class PagingResult
{
    private ArrayList items;
    ...
    public PagingResult(Array items)
    {
        this.items = new ArrayList(items);
    }
    ...
    public ArrayList Items
    {
        get { return this.items; }
    }
}
```

To use this class in your code, you always have to know the type of objects it contains and evaluate the cost or risk of runtime casts or boxing operations. Using a generic type parameter *T*, you can write a type-safe class so that the compiler will prevent any invalid use at build time:

```
public class PagingResult<T>
{
    public PagingResult(IEnumerable<T> items)
    {
        this.Items = new List<T>(items);
    }
    ...
    public ICollection<T> Items { get; }
}
```

To learn more about generics, visit [Generics in the .NET Framework](http://msdn.microsoft.com/en-us/library/ms172192.aspx) (<http://msdn.microsoft.com/en-us/library/ms172192.aspx>).

LISTING 1-4 *SearchByActivity* Implementation

```

public PagingResult<Itinerary> SearchByActivity(string activityId, int pageSize, int
pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.ActivityId == activityId)
                      && itinerary.IsPublic
                    orderby itinerary.Rating
                    select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}

```



Note The resulting paging is implemented in the *PageResults* method:

```

private static PagingResult<Itinerary> PageResults(IQueryable<Itinerary> query,
int page, int pageSize)
{
    int rowCount = query.Count();
    if (pageSize > 0)
    {
        query = query.Skip((page - 1) * pageSize)
                    .Take(pageSize);
    }
    var result = new PagingResult<Itinerary>(query.ToArray())
    {
        PageSize = pageSize,
        CurrentPage = page,
        TotalItems = rowCount
    };
    return result;
}

```

IQueryable<Itinerary> is passed to this function so that it can add the paging to the base query composition. Passing *IQueryable* instead of *IEnumerable* ensures that the T-SQL created for the query against the repository will be generated only when *query.ToArray* is called.

The *SearchByZipCode* function method is similar to the *SearchByActivity* method, but it also adds a filter on the Zip Code of the activity. Here again, LINQ support makes it easy to implement as shown in Listing 1-5. Add the highlighted code to the *SearchByZipCode* method body.

LISTING 1-5 *SearchByZipCode* Implementation

```

public PagingResult<Itinerary> SearchByZipCode(int activityTypeId, string zip,
    int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.TypeId == activityTypeId &&
                        t.Zip == zip)
                    && itinerary.IsPublic
                    orderby itinerary.Rating
                    select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}

```

The *SearchByRadius* function calls the *RetrieveItinerariesWithinArea* import function that was mapped to a stored procedure. It then loads the activities for each itinerary found. You can copy the highlighted code in Listing 1-6 to the *SearchByRadius* method body in the *ItinerariesRepository.cs* file.

LISTING 1-6 *SearchByRadius* Implementation

```

public PagingResult<Itinerary> SearchByRadius(int activityTypeId,
    double longitude, double latitude, double radius, int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        // Stored Procedure with output parameter
        var totalOutput = new ObjectParameter("total", typeof(int));
        var items = ctx.RetrieveItinerariesWithinArea(activityTypeId, latitude,
            longitude, radius, pageSize, pageNumber, totalOutput).ToArray();

        foreach (var item in items)
        {
            item.Activities.ToList().AddRange(this.Retrieve(item.Id).Activities);
        }

        int total = totalOutput.Value == DBNull.Value ? 0 : (int)totalOutput.Value;

        return new PagingResult<Itinerary>(items)
    }
}

```

```
    {  
        TotalItems = total,  
        PageSize = pageSize,  
        CurrentPage = pageNumber  
    };  
}  
}
```

The *Add* method allows you to add *Itinerary* to the data store. Implementing this functionality becomes trivial because your contract and context object use the same entity object. Copy and paste the highlighted code in Listing 1-7 to the *Add* method body.

LISTING 1-7 *Add* Implementation

```
public void Add(Itinerary itinerary)  
{  
    using (var ctx = new PlanMyNightEntities())  
    {  
        ctx.Itineraries.AddObject(itinerary);  
        ctx.SaveChanges();  
    }  
}
```

There you have it! You have completed the *ItinerariesRepository* implementation using the context object generated using the EF designer. Run all the tests in the solution by pressing Ctrl+R, A. The tests related to the *ItinerariesRepository* implementation should all succeed.

Getting Data from the Bing Maps Web Services

PMN relies on the Bing Maps services to allow the user to search for activities to add to her itineraries. To get a Bing Maps Key to use in the PMN application, you need to create a Bing Maps Developer Account. You can create a free developer account on the [Bing Maps Account Center](https://www.bingmapsportal.com/) (<https://www.bingmapsportal.com/>).

See Also Microsoft Bing Maps Web services is a set of programmable Simple Object Access Protocol (SOAP) services that allow you to match addresses to the map, search for points of interest, integrate maps and imagery, return driving directions, and incorporate other location intelligence into your Web application. You can learn more about these services by visiting the site for the [Bing Maps Web Services SDK](http://msdn.microsoft.com/en-us/library/cc980922.aspx) (<http://msdn.microsoft.com/en-us/library/cc980922.aspx>).

Visual Studio 2003 In Visual Studio 2003, if you had to add a reference to a Web service, you would have selected the Add Web Service Reference from the contextual menu to bring up the Add Web Reference dialog and then added a reference to a Web service to your project. (See Figure 1-29.)

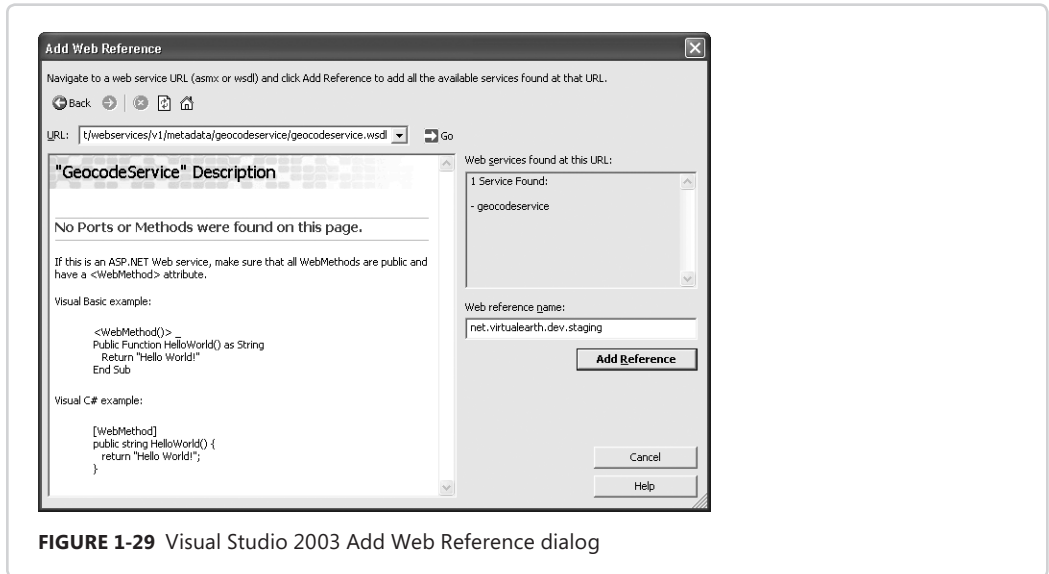


FIGURE 1-29 Visual Studio 2003 Add Web Reference dialog

Introduced in the .NET Framework 3.0, the Windows Communication Foundation (WCF) services brought the ASMX Web services and other communication technologies into a unified programming model.

Visual Studio 2010 provides tools for working with WCF services. You can bring up the new Add Service Reference dialog by right-clicking on a project node and selecting Add Service Reference as shown in Figure 1-30. In this dialog, you first need to specify the service metadata address in the Address field and then click Go to view the available service endpoints. You can then specify a namespace for the generated code and click OK to add the proxy to your project.

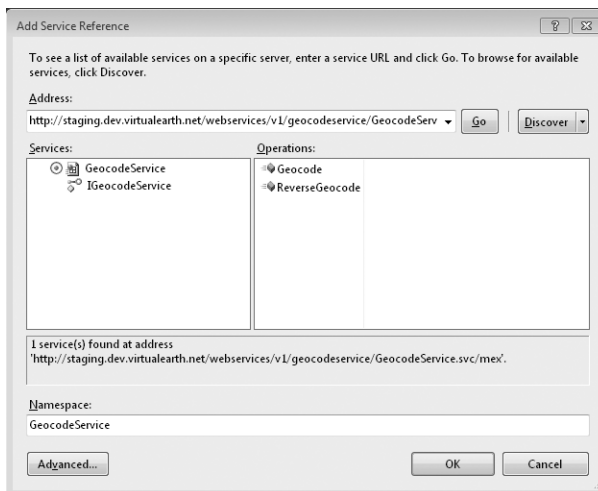


FIGURE 1-30 Add Service Reference dialog



Tip Click the Discover button to look for WCF services in the current solution.

See Also Click the Advanced button to access the Service Reference Settings dialog. This dialog lets you tweak the configuration of the WCF service proxy. You can add the .NET Framework 2.0 style reference by clicking the Add Web Service button. To learn more about these settings, visit the [MSDN - Configure Service Reference Dialog Box](http://msdn.microsoft.com/en-us/library/bb514724(VS.100).aspx) ([http://msdn.microsoft.com/en-us/library/bb514724\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb514724(VS.100).aspx)).

The generated WCF proxy can be used in the same way you used the ASMX-style proxy, as shown in Listing 1-8.

LISTING 1-8 Using a Web Service Proxy

```
public BingCoordinate GeocodeAddress(ActivityAddress address, string token)
{
    ...
    Microsoft.Samples.PlanMyNight.Bing.VEGeocodingService.GeocodeResponse geocodeResponse
    = null;
    // Make the geocode request
    using (var geocodeService = new
Microsoft.Samples.PlanMyNight.Bing.VEGeocodingService.GeocodeServiceClient())
    {
        try
        {
            geocodeResponse = geocodeService.Geocode(geocodeRequest);
            geocodeService.Close();
        }
        catch
        {
            geocodeService.Abort();
        }
    }

    if (geocodeResponse != null && geocodeResponse.Results != null && geocodeResponse.
Results.Length > 0)
    {
        var location = geocodeResponse.Results[0].Locations[0];
        return new BingCoordinate { Latitude = (float)location.Latitude, Longitude =
(float)location.Longitude };
    }

    return default(BingCoordinate);
}
```


Parallel Programming

With the advances in multicore computing, it is becoming more and more important for developers to be able to write parallel applications. Visual Studio 2010 and the .NET Framework 4.0 provide new ways to express concurrency in applications. The Task Parallel Library (TPL) is now part of the Base Class Library (BCL) for the .NET Framework. This means that every .NET application can now access the TPL without adding any assembly reference.

PMN stores only the Bing Activity ID for each *ItineraryActivity* to the database. When it's time to retrieve the entire Bing Activity object, a function that iterates through each of the *ItineraryActivity* instances for the current *Itinerary* is used to populate the Bing Activity entity from the Bing Maps Web services.

One way of performing this operation is to sequentially call the service for each activity in the *Itinerary* as shown in Listing 1-9. This function waits for each call to *RetrieveActivity* to complete before making another call, which has the effect of making its execution time linear.

LISTING 1-9 Activity Sequential Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    foreach (var item in itinerary.Activities.Where(i =>i.Activity == null))
    {
        item.Activity = this.RetrieveActivity(item.ActivityId);
    }
}
```

In the past, if you wanted to parallelize this task, you had to use threads and then hand off work to them. With the TPL, all you have to do now is use a *Parallel.ForEach* that will take care of the threading for you, as seen in Listing 1-10.

LISTING 1-10 Activity Parallel Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    Parallel.ForEach(itinerary.Activities.Where(i => i.Activity == null),
        item =>
        {
            item.Activity = this.RetrieveActivity(item.ActivityId);
        });
}
```

See Also *The .NET Framework 4.0 now includes the Parallel LINQ libraries (in System.Core.dll). PLINQ introduces the .AsParallel extension to perform parallel operations in LINQ queries. You can also easily enforce the treatment of a data source as if it was ordered by using the .AsOrdered extensions. Some new thread-safe collections have also been added in the System.Collections.Concurrent namespace. You can learn more about these new features from [Parallel Computing on MSDN](http://msdn.microsoft.com/en-us/concurrency/default.aspx) (<http://msdn.microsoft.com/en-us/concurrency/default.aspx>).*

AppFabric Caching

PMN is a data-driven application that gets its data from the application database and the Bing Maps Web services. One of the challenges you might face when building a Web application is managing the needs of a large number of users, including performance and response time. The operations that use the data store and the services used to search for activities can increase the usage of server resources dramatically for items that are shared across many users. For example, many users have access to the public itineraries, so displaying these will generate numerous calls to the database for the same items. Implementing caching at the Web tier will help reduce the usage of resources at the data store and help mitigate latency for recurring searches to the Bing Maps Web services. Figure 1-31 shows the architecture for an application implementing a caching solution at the front-end server.

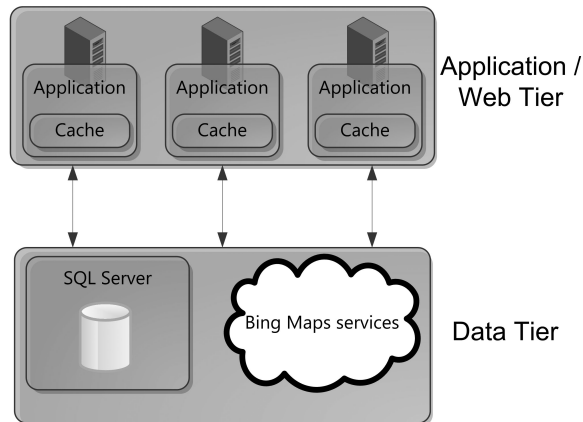


FIGURE 1-31 Typical Web application architecture

Using this approach reduces the pressure on the data layer, but the caching is still coupled to a specific server serving the request. Each Web tier server will have its own cache, but you can still end up with an uneven distribution of the processing to these servers.

Windows Server AppFabric caching offers a distributed, in-memory cache platform. The AppFabric client library allows the application to access the cache as a unified view event if the cache is distributed across multiple computers as shown in Figure 1-32. The API provides

simple *get* and *set* methods to retrieve and store any serializable common language runtime (CLR) objects easily. The AppFabric cache allows you to add a cache computer on demand, thus making it possible to scale in a manner that is transparent to the client. Another benefit is that the cache can also share copies of the data across the cluster, thereby protecting data against failure.

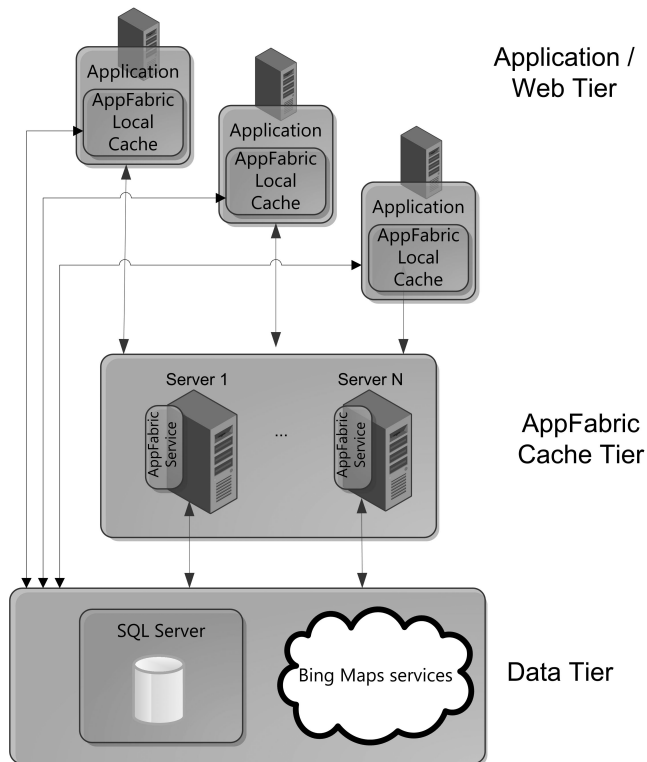


FIGURE 1-32 Web application using Windows Server AppFabric caching

See Also Windows Server AppFabric caching is available as a set of extensions to the .NET Framework 4.0. For more information about how to get, install, and configure Windows Server AppFabric, please visit [Windows Server AppFabric](http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx) (<http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx>).

See Also PMN can be configured to use either ASP.NET caching or Windows Server AppFabric caching. A complete walkthrough describing how to add Windows Server AppFabric caching to PMN is available here: [PMN: Adding Caching using Velocity](http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/) (<http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/>).

Summary

In this chapter, you used a few of the new Visual Studio 2010 features to structure the data layer of the Plan My Night application using the Entity Framework version 4.0 to access a database. You also were introduced to automated entity generation using the ADO.NET Entity Framework POCO templates and to the Windows Server AppFabric caching extensions.

In the next chapter, you will explore how the ASP.NET MVC framework and the Managed Extensibility Framework can help you build great Web applications.

Chapter 2

From 2003 to 2010: Designing the Look and Feel

After reading this chapter, you will be able to

- Create an ASP.NET MVC controller that interacts with the data model
- Create an ASP.NET MVC view that displays data from the controller and validates user input
- Extend the application with an external plug-in using the Managed Extensibility Framework

Web application development in Microsoft Visual Studio has certainly made significant improvements over the years since ASP.NET 1.0 was released. Visual Studio 2003 included features such as automatic input validation and mobile controls (plus many others) to help developers create efficient applications that were easy to manage.

The spirit of improvement to assist developers in creating world-class applications is very much alive in Visual Studio 2010. In this chapter, we'll explore some of the new features as we add functionality to the Plan My Night companion application.



Note The companion application is an ASP.NET MVC 2 project, but a Web developer has a choice in Visual Studio 2010 to use this new form of ASP.NET application or the more traditional ASP.NET (referred to in the community as *Web Forms for distinction*). ASP.NET 4.0 has many improvements to help developers and is still a very viable approach to creating Web applications.

We'll be using a modified version of the companion application's solution to work our way through this chapter. If you installed the companion content in the default location, the correct solution can be found at Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 2\ in a folder called UserInterface-Start.

Introducing the PlanMyNight.Web Project

The user interface portion of Plan My Night in Visual Studio 2010 was developed as an ASP.NET MVC application, the layout of which differs from what a developer might be accustomed to when developing an ASP.NET Web Forms application in Visual Studio 2003. Some items in the project (as seen in Figure 2-1) will look familiar (such as Global.asax), but others are completely new, and some of the structure is required by the ASP.NET MVC framework.

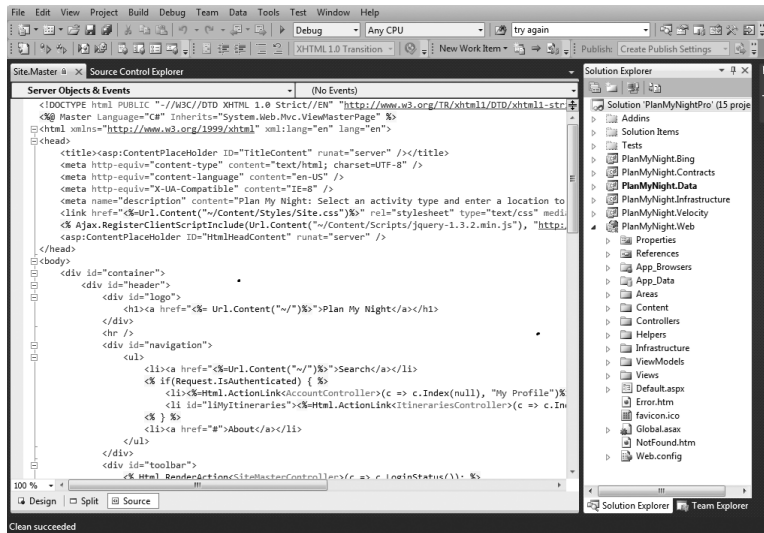


FIGURE 2-1 PlanMyNight.Web project view

Here are the items required by ASP.NET MVC:

- **Areas** This folder is used by the ASP.NET MVC framework to organize large Web applications into smaller components, without using separate solutions or projects. This feature is not used in the Plan My Night application but is called out because this folder is created by the MVC project template.
- **Controllers** During request processing, the ASP.NET MVC framework looks for controllers in this folder to handle the request.
- **Views** The Views folder is actually a structure of folders. The layer immediately inside the Views folder is named for each of the classes found in the Controllers folder, plus a Shared folder. The Shared subfolder is for common views, partial views, master pages, and anything else that will be available to all controllers.

See Also More information about ASP.NET MVC components, as well as how its request processing differs from ASP.NET Web Forms, can be found at <http://asp.net/mvc>.

In most cases, the web.config file is the last file in a project's root folder. However, it has received a much-needed update in Visual Studio 2010: Web.config Transformation. This feature allows for a base web.config file to be created but then to have build-specific web.config files override the settings of the base at build, deployment, and run times. These files appear under the base web.config file, as seen in Figure 2-2.

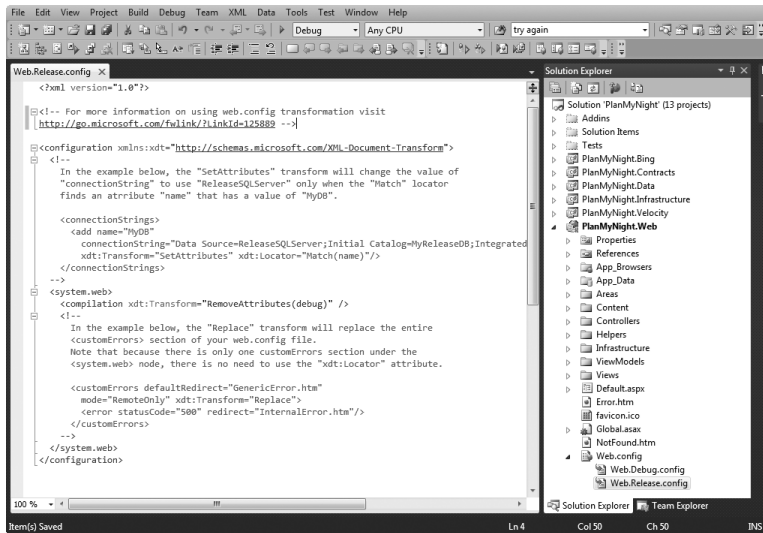


FIGURE 2-2 A web.config file with build-specific files expanded

Visual Studio 2003 When working on a project in Visual Studio 2003, do you recall needing to remember not to overwrite the web.config file with your debug settings? Or needing to remember to update web.config when it was published for a retail build with the correct settings? This is no longer an issue in Visual Studio 2010. The settings in the web.Release.config file will be used during release builds to override the values in web.config, and the same goes for the web.Debug.config in debug builds.

Other sections of the project include the following:

- **Content** A collection of folders containing images, scripts, and style files
- **Helpers** Includes miscellaneous classes, containing a number of extension methods, that add functionality to types used in the project
- **Infrastructure** Contains items related to dealing with the lower level infrastructure of ASP.NET MVC (for example: caching and controller factories)
- **ViewModels** Contains data entities filled out by controller classes and used by views to display data

Running the Project

If you compile and run the project, you should see a screen similar to Figure 2-3.

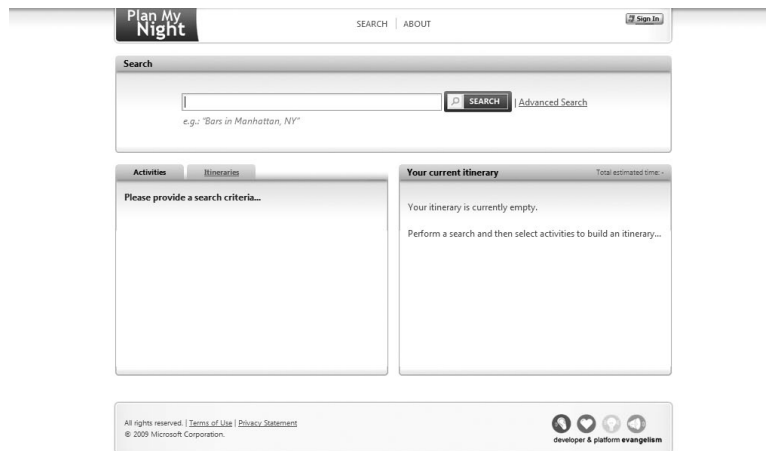


FIGURE 2-3 Default page of the Plan My Night application

The searching functionality and the ability to organize an initial list of itinerary items all works, but if you attempt to save the itinerary you are working on, or if you log in with Windows Live ID, the application will return a 404 Not Found error screen (as shown in Figure 2-4).

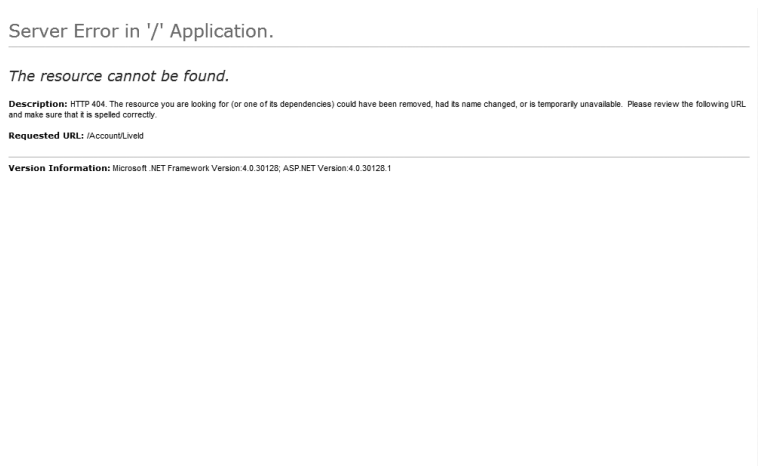


FIGURE 2-4 Error screen returned when logging in to the Plan My Night application

You get this error because currently the project does not include an account controller to handle these requests.

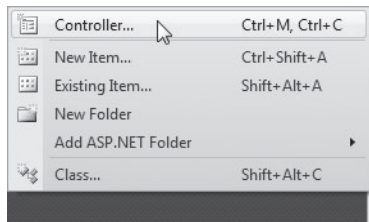
Creating the Account Controller

The *AccountController* class provides some critical functionality to the companion Plan My Night application:

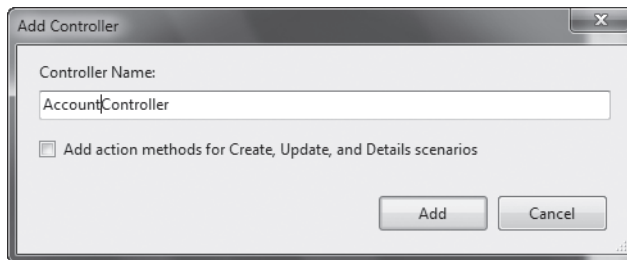
- It handles signing users in and out of the application (via Windows Live ID).
- It provides actions for displaying and updating user profile information.

To create a new ASP.NET MVC controller:

1. Use Solution Explorer to navigate to the Controllers folder in the PlanMyNight.Web project, and click the right mouse button.
2. Open the Add submenu, and select the Controller item.



3. Fill in the name of the controller as **AccountController**.



Note Leave the Add Action Methods For Create, Update And Details Scenarios check box blank. Selecting the box inserts some “starter” action methods, but because you will not be using the default methods, there is no reason to create them.

After you click the Add button in the Add Controller dialog box, you should have a basic *AccountController* class open, with a single *Index* method in its body:

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

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    public class AccountController : Controller
    {
        //
        // GET: /Account/

        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Visual Studio 2003 A difference to be noted from developing ASP.NET Web Forms applications in Visual Studio 2003, is that ASP.NET MVC applications do not have a companion code-behind file for each of their .aspx files. Controllers like the one you are currently creating perform the logic required to process input and prepare output. This approach allows for a clear separation of display and business logic, and it's a key aspect of ASP.NET MVC.

Implementing the Functionality

To communicate with any of the data layers and services (the Model), you'll need to add some instance fields and initialize them. Before that, you need to add some namespaces to your using block:

```
using System.IO;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using System.Collections.Specialized;
using WindowsLiveId;
```

Now, let's add the instance fields. These fields are interfaces to the various section of your Model:

```
public class AccountController : Controller
{
    private readonly IWindowsLiveLogin windowsLogin;
    private readonly IMembershipService membershipService;
    private readonly IFormsAuthentication formsAuthentication;
    private readonly IReferenceRepository referenceRepository;
    private readonly IActivitiesRepository activitiesRepository;
    .
    .
    .
}
```



Note Using interfaces to interact with all external dependencies allows for better portability of the code to various platforms. Also, during testing, dependencies can be mimicked much easier when using interfaces, making for more efficient isolation of a specific component.

As mentioned, these fields represent parts of the Model this controller will interact with to meet its functional needs. Here are the general descriptions for each of the interfaces:

- **IWindowsLiveLogin** Provides functionality for interacting with the Windows Live ID service.
- **IMembershipService** Provide user profile information and authorization methods. In your companion application, it is an abstraction of the ASP.NET Membership Service.
- **IFormsAuthentication** Provides for ASP.NET Forms Authentication abstraction.
- **IReferenceRepository** Provides reference resources, such as lists of states and other model-specific information.
- **IActivitiesRepository** An interface for retrieving and updating activity information.

You'll add two constructors to this class: one for general run-time use, which uses the *ServiceFactory* class to get references to the needed interfaces, and one to enable tests to inject specific instances of the interfaces to use.

```
public AccountController() :
    this(
        new ServiceFactory().GetMembershipService(),
        new WindowsLiveLogin(true),
        new FormsAuthenticationService(),
        new ServiceFactory().GetReferenceRepositoryInstance(),
        new ServiceFactory().GetActivitiesRepositoryInstance())
{
}
```

```
public AccountController(
    IMembershipService membershipService,
    IWindowsLiveLogin windowsLogin,
    IFormsAuthentication formsAuthentication,
    IReferenceRepository referenceRepository,
    IActivitiesRepository activitiesRepository)
{
    this.membershipService = membershipService;
    this.windowsLogin = windowsLogin;
    this.formsAuthentication = formsAuthentication;
    this.referenceRepository = referenceRepository;
    this.activitiesRepository = activitiesRepository;
}
```

Authenticating the User

The first real functionality you'll implement in this controller is that of signing in and out of the application. Most of the methods that you'll implement later require authentication, so this is a good place to start.

The companion application uses a few technologies together at the same time to give the user a smooth authentication experience: Windows Live ID, ASP.NET Forms Authentication, and ASP.NET Membership Services. These three technologies are used in the LiveID action you'll implement next.

Start by creating the following method, in the *AccountController* class:

```
public ActionResult LiveId()
{
    return Redirect("~/");
}
```

This method will be the primary action invoked when interacting with the Windows Live ID services. Right now, if it is invoked, it will just redirect the user to the root of the application.



Note The call to *Redirect* returns *RedirectResult*, and although this example uses a string to define the target of the redirection, various overloads can be used for different situations.

A few different types of actions can be taken when Windows Live ID returns a user to your application. The user can be signing into Windows Live ID, signing out, or clearing the Windows Live ID cookies. Windows Live ID uses a query string parameter called *action* on the URL when it returns a user, so you'll use a switch to branch the logic depending on the value of the parameter.

Add the following to the *LiveId* method above the return statement:

```
string action = Request.QueryString["action"];
switch (action)
{
    case "logout":
        this.formsAuthentication.SignOut();
        return Redirect("~/");

    case "clearcookie":
        this.formsAuthentication.SignOut();
        string type;
        byte[] content;
        this.windowsLogin.GetClearCookieResponse(out type, out content);
        return new FileStreamResult(new MemoryStream(content), type);
}
```

See Also Full documentation of the Windows Live ID system can be found on the <http://dev.live.com/> Web site.

The code you just added handles the two sign-out actions for Windows Live ID. In both cases, you use the *IFormsAuthentication* interface to remove the ASP.NET Forms Authentication cookie so that any future http requests (until the user signs in again) will not be considered authenticated. In the second case, you went one step further to clear the Windows Live ID cookies (the ones that remember your login name, but not your password).

Handling the sign-in scenario requires a bit more code because you have to check whether the authenticating user is in your Membership Database and, if not, create a profile for the user. However, before that, you must pass the data that Windows Live ID sent you to your Windows Live ID interface so that it can validate the information and give you a *WindowsLiveLogin.User* object:

```
default:
    // login
    NameValueCollection tokenContext;
    if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
    {
        tokenContext = Request.Form;
    }
    else
    {
        tokenContext = new NameValueCollection(Request.QueryString);
        tokenContext["stoken"] =
            System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
    }

    var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
```

At this point in the case for logging in, either *liveIdUser* will be a reference to an authenticated *WindowsLiveLogin.User* object or it will be null. With this in mind, you can add your next section of the code, which takes action when the *liveIdUser* value is not null:

```
if (liveIdUser != null)
{
    var returnUrl = liveIdUser.Context;
    var userId = new Guid(liveIdUser.Id).ToString();
    if (!this.membershipService.ValidateUser(userId, userId))
    {
        this.formsAuthentication.SignIn(userId, false);
        this.membershipService.CreateUser(userId, userId, string.Empty);
        var profile = this.membershipService.CreateProfile(userId);
        profile.FullName = "New User";
        profile.State = string.Empty;
        profile.City = string.Empty;
        profile.PreferredActivityTypeId = 0;
        this.membershipService.UpdateProfile(profile);

        if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
        return RedirectToAction("Index", new { returnUrl = returnUrl });
    }
    else
    {
        this.formsAuthentication.SignIn(userId, false);
        if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
        return Redirect(returnUrl);
    }
}
break;
```

The call to the *ValidateUser* method on the *IMembershipService* reference allows the application to check whether the user has been to this site before and whether there will be a profile for the user. Because the user is authenticated with Windows Live ID, you are using the user's ID value (which is a GUID) as both the user name and password to the ASP.NET Membership Service.

If the user does not have a user record with the application, you create one by calling the *CreateUser* method and then also create a user settings profile via *CreateProfile*. The profile is filled with some defaults and saved back to its store, and the user is redirected to the primary input page so that he can update the information.



Note *Controller.RedirectToAction* determines which URL to create based on the combination of input parameters. In this case, you want to redirect the user to the *Index* action of this controller, as well as pass the current return URL value.

The other action that takes place in this code is that the user is signed in to ASP.NET Forms authentication so that a cookie will be created, providing identity information on future requests that require authentication.

The settings profile is managed by ASP.NET Membership Services as well and is declared in the web.config file of the application:

```
<system.web>
...
<profile enabled="true">
  <properties>
    <add name="FullName" type="string" />
    <add name="State" type="string" />
    <add name="City" type="string" />
    <add name="PreferredActivityTypeId" type="int" />
  </properties>

  <providers>
    <clear />
    <add name="AspNetSqlProfileProvider"
type="System.Web.Profile.SqlProfileProvider,
      System.Web, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a"
      connectionStringName="ApplicationServices"
      applicationName="/" />
  </providers>
</profile>
...
</system.web>
```

At this point, the *LiveID* method is complete and should look like the following code. The application can now take authentication information from Windows Live ID, prepare an ASP.NET MembershipService profile, and create an ASP.NET Forms Authentication ticket.

```
public ActionResult LiveId()
{
  string action = Request.QueryString["action"];
  switch (action)
  {
    case "logout":
      this.formsAuthentication.SignOut();
      return Redirect("~/");

    case "clearcookie":
      this.formsAuthentication.SignOut();
      string type;
      byte[] content;
      this.windowsLogin.GetClearCookieResponse(out type, out content);
      return new FileStreamResult(new MemoryStream(content), type);

    default:
      // login
      NameValueCollection tokenContext;
      if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
      {
        tokenContext = Request.Form;
      }
      else
      {

```

```

        tokenContext = new NameValueCollection(Request.QueryString);
        tokenContext["stoken"] =
            System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
    }

    var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);

    if (liveIdUser != null)
    {
        var returnUrl = liveIdUser.Context;
        var userId = new Guid(liveIdUser.Id).ToString();
        if (!this.membershipService.ValidateUser(userId, userId))
        {
            this.formsAuthentication.SignIn(userId, false);
            this.membershipService.CreateUser(userId, string.Empty);
            var profile = this.membershipService.CreateProfile(userId);
            profile.FullName = "New User";
            profile.State = string.Empty;
            profile.City = string.Empty;
            profile.PreferredActivityTypeId = 0;
            this.membershipService.UpdateProfile(profile);

            if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
            return RedirectToAction("Index", new { returnUrl = returnUrl });
        }
        else
        {
            this.formsAuthentication.SignIn(userId, false);
            if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
            return Redirect(returnUrl);
        }
    }
    break;
}
return Redirect("~/");
}

```

Of course, the user has to be able to get to the Windows Live ID login page in the first place before logging in. Currently in the Plan My Night application, there is a Windows Live ID login button. However, there are cases where the application will want the user to be redirected to the login page from code. To cover this scenario, you need to add a small method called *Login* to your controller:

```

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
    return Redirect(redirect);
}

```

This method simply retrieves the login URL for Windows Live and redirects the user to that location. This also satisfies a configuration value in your web.config file for ASP.NET

Forms Authentication in that any request requiring authentication will be redirected to this method:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" name="XAUTH" timeout="2880" path="~/\" />
</authentication>
```

Retrieving the Profile for the Current User

Now with the authentication methods defined, which satisfies your first goal for this controller—signing users in and out in the application—you can move on to retrieving data for the current user.

The *Index* method, which is the default method for the controller based on the URL mapping configuration in *Global.asax*, will be where you retrieve the current user's data and return a view displaying that data. The *Index* method that was initially created when the *AccountController* class was created should be replaced with the following:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Index(string returnUrl)
{
    var profile = this.membershipService.GetCurrentProfile();
    var model = new ProfileViewModel
    {
        Profile = profile,
        ReturnUrl = returnUrl ?? this.GetReturnUrl()
    };

    this.InjectStatesAndActivityTypes(model);

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

Visual Studio 2003 Attributes such as *[Authorize()]* might not have been in common use in Visual Studio 2003; however, ASP.NET MVC makes use of them often. Attributes allow for metadata to be defined about the target they decorate. This allows for the information to be examined at run time (via reflection) and for action to be taken if deemed necessary.

The *Authorize* attribute is very handy because it declares that this method can be invoked only for http requests that are already authenticated. If a request is not authenticated, it will be redirected to the ASP.NET Forms Authentication configured login target, which you just finished setting up. The *AcceptVerbs* attribute also restricts how this method can be invoked, by specifying which Http verbs can be used. In this case, you are restricting this method to HTTP GET verb requests. You've added a string parameter, *returnUrl*, to the method signature so that when the user is finished viewing or updating her information, she can be returned to what she was looking at previously.



Note This highlights a part of the ASP.NET MVC framework called *Model Binding*, details of which are beyond the scope of this book. However, you should know that it attempts to find a source for *returnUrl* (a form field, routing table data, or query string parameter with the same name) and binds it to this value when invoking the method. If the Model Binder cannot find a suitable source, the value will be null. This behavior can cause problems for value types that cannot be null, because it will throw an *InvalidOperationException*.

The main portion of this method is straightforward: it takes the return of the *GetCurrentProfile* method on the ASP.NET Membership Service interface and sets up a view model object for the view to use. The call to *GetReturnUrl* is an example of an extension method defined in the PlanMyNight.Infrastructure project. It's not a member of the Controller class, but in the development environment it makes for much more readable code. (See Figure 2-5.)

```

File Edit View Refactor Project Build Debug Team Data Tools Test Window Help
Debug Any CPU GetReturnUrl
New Work Item
MvcExtensions.cs SiteMaster Source Control Explorer
Microsoft.Samples.PlanMyNight.Infrastructure.Mvc.MvcExtensions - % GetAbsoluteUrl(ControllerBase controller, string path)
return LinkBuilder.BuildUrlFromExpression<T>(
    controller.ControllerContext.RequestContext,
    controller.Url.RouteCollection,
    action);
}
public static string GetAbsoluteUrl(this ControllerBase controller, string path)
{
    return String.Concat(controller.ControllerContext.HttpContext.Request.Url.Scheme,
        "://", controller.ControllerContext.HttpContext.Request.ServerVariables["HTTP_HOST"], path);
}
public static bool IsAjaxCall(this ControllerBase controller)
{
    return !string.IsNullOrEmpty(controller.Request.ContentType) &&
        controller.Request.ContentType.Contains("application/json");
}
public static string GetReturnUrl(this ControllerBase controller)
{
    if (controller.Request.ServerVariables != null &&
        !string.IsNullOrEmpty(controller.Request.ServerVariables["HTTP_REFERER"]))
        return controller.Request.ServerVariables["HTTP_REFERER"];
    return "";
}
}
100 % Ln 21 Col 17 Ch 17 INS
Ready

```

FIGURE 2-5 Example of extension methods in *MvcExtensions.cs*

Visual Studio 2003 In .NET Framework 1.1, which Visual Studio 2003 used, extension methods did not exist. Rather than calling *this.GetReturnUrl()* and also having the method appear in IntelliSense for this object, you would have to type **MvcExtensions.GetReturnUrl(this)**, passing in the controller as a parameter. Extension methods certainly make the code more readable and do not require the developer to know the static class the extension method exists under. For IntelliSense to work, the name space does need to be listed in the *using* clauses.

InjectStatesAndActivityTypes is a method you need to implement. It gathers data from the reference repository for names of states and the activity repository. It makes two collections

of *SelectListItem* (an HTML class for MVC): one for the list of states, and the other for the list of different activity types available in the application. It also sets the respective value.

```
private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null && o.Id ==
                profile.PreferredActivityTypeId)
        }).ToList();

    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null && o.Abbreviation ==
                profile.State)
        }).ToList();

    states.Insert(0, new SelectListItem {
        Text = "Any state",
        Value = string.Empty
    });

    model.PreferredActivityTypes = types;
    model.States = states;
}
```

Visual Studio 2003 In Visual Studio 2003, the *InjectStatesAndActivities* method takes longer to implement because a developer cannot use the LINQ extensions (the call to *Select*) and Lambda expressions, which are a form of anonymous delegate that the *Select* method applies to each member of the collection being enumerated. Instead, the developer would have to write out his own loop and enumerate each item manually.

Updating the Profile Data

Having completed the infrastructure needed to retrieve data for the current profile, you can move on to updating the data in the model from a form submission by the user. After this, you can create your view pages and see how all this ties together. The *Update* method is simple; however, it does introduce some new features not seen yet:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
public ActionResult Update(UserProfile profile)
{
```

```
var returnUrl = Request.Form["returnUrl"];
if (!ModelState.IsValid)
{
    // validation error
    return this.IsAjaxCall() ? new JsonResult { JsonRequestBehavior =
        JsonRequestBehavior.AllowGet, Data = ModelState }
        : this.Index(returnUrl);
}

this.membershipService.UpdateProfile(profile);
if (this.IsAjaxCall())
{
    return new JsonResult { JsonRequestBehavior = JsonRequestBehavior.AllowGet,
        Data = new { Update = true, Profile = profile, ReturnUrl = returnUrl } };
}
else
{
    return RedirectToAction("UpdateSuccess", "Account", new { returnUrl =
        returnUrl });
}
}
```

The *ValidateAntiForgeryToken* attribute ensures that the form has not been tampered with. To use this feature, you need to add an *AntiForgeryToken* to your view's input form. The check on the *ModelState* to see whether it is valid is your first look at input validation. This is a look at the server-side validation, and ASP.NET MVC offers an easy-to-use feature to make sure that incoming data meets some rules. The *UserProfile* object that is created for input to this method, via MVC Model Binding, has had one of its properties decorated with a *System.ComponentModel.DataAnnotations.Required* attribute. During Model Binding, the MVC framework evaluates *DataAnnotation* attributes and marks the *ModelState* as valid only when all of the rules pass.

In the case where the *ModelState* is not valid, the user is redirected to the *Index* method where the *ModelState* will be used in the display of the input form. Or, if the request was an AJAX call, a *JsonResult* is returned with the *ModelState* data attached to it.

Visual Studio 2003 Because in ASP.NET MVC requests are routed through controllers rather than pages, the same URL can handle a number of requests and respond with the appropriate view. In Visual Studio 2003, a developer would have to create two different URLs and call a method in a third class to perform the functionality.

When the *ModelState* is valid, the profile is updated in the membership service and a JSON result is returned for AJAX requests with the success data, or in the case of "normal"

requests, the user is redirected to the *UpdateSuccess* action on the Account controller. The *UpdateSuccess* method is the final method you need to implement to finish off this controller:

```
public ActionResult UpdateSuccess(string returnUrl)
{
    var model = new ProfileViewModel
    {
        Profile = this.membershipService.GetCurrentProfile(),
        ReturnUrl = returnUrl
    };
    return View(model);
}
```

The method is used to return a success view to the browser, display some of the updated data, and provide a link to return the user to where she was when she started the profile update process.

Now that you've reached the end of the Account controller implementation, you should have a class that resembles the following listing:

```
using System;
using System.Collections.Specialized;
using System.IO;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using WindowsLiveId;

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    [HandleErrorWithContentType()]
    [OutputCache(NoStore = true, Duration = 0, VaryByParam = "*")]
    public class AccountController : Controller
    {
        private readonly IWindowsLiveLogin windowsLogin;
        private readonly IMembershipService membershipService;
        private readonly IFormsAuthentication formsAuthentication;
        private readonly IReferenceRepository referenceRepository;
        private readonly IActivitiesRepository activitiesRepository;

        public AccountController() :
            this(
                new ServiceFactory().GetMembershipService(),
                new WindowsLiveLogin(true),
                new FormsAuthenticationService(),
                new ServiceFactory().GetReferenceRepositoryInstance(),
                new ServiceFactory().GetActivitiesRepositoryInstance())
        {
        }
    }
}
```

```
{
}

public AccountController(IMembershipService membershipService,
                        IWindowsLiveLogin windowsLogin,
                        IFormsAuthentication formsAuthentication,
                        IReferenceRepository referenceRepository,
                        IActivitiesRepository activitiesRepository)
{
    this.membershipService = membershipService;
    this.windowsLogin = windowsLogin;
    this.formsAuthentication = formsAuthentication;
    this.referenceRepository = referenceRepository;
    this.activitiesRepository = activitiesRepository;
}

public ActionResult LiveId()
{
    string action = Request.QueryString["action"];
    switch (action)
    {
        case "logout":
            this.formsAuthentication.SignOut();
            return Redirect("~/");
        case "clearcookie":
            this.formsAuthentication.SignOut();
            string type;
            byte[] content;
            this.windowsLogin.GetClearCookieResponse(out type, out content);
            return new FileStreamResult(new MemoryStream(content), type);
        default:
            // login
            NameValueCollection tokenContext;
            if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
            {
                tokenContext = Request.Form;
            }
            else
            {
                tokenContext = new NameValueCollection(Request.QueryString);
                tokenContext["token"] =
                    System.Web.HttpUtility.UrlEncode(tokenContext["token"]);
            }

            var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
            if (liveIdUser != null)
            {
                var returnUrl = liveIdUser.Context;
                var userId = new Guid(liveIdUser.Id).ToString();
                if (!this.membershipService.ValidateUser(userId, userId))
                {
                    this.formsAuthentication.SignIn(userId, false);
                    this.membershipService.CreateUser(
                        userId, userId, string.Empty);
                    var profile =
```

```
        this.membershipService.CreateProfile(userId);
        profile.FullName = "New User";
        profile.State = string.Empty;
        profile.City = string.Empty;
        profile.PreferredActivityTypeId = 0;
        this.membershipService.UpdateProfile(profile);
        if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
        return RedirectToAction("Index", new { returnUrl =
            returnUrl });
    }
    else
    {
        this.formsAuthentication.SignIn(userId, false);
        if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
        return Redirect(returnUrl);
    }
}
break;
}
return Redirect("~/");
}

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
    return Redirect(redirect);
}

[Authorize()]
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Index(string returnUrl)
{
    var profile = this.membershipService.GetCurrentProfile();
    var model = new ProfileViewModel
    {
        Profile = profile,
        ReturnUrl = returnUrl ?? this.GetReturnUrl()
    };

    this.InjectStatesAndActivityTypes(model);
    return View("Index", model);
}

[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
public ActionResult Update(UserProfile profile)
{
    var returnUrl = Request.Form["returnUrl"];
    if (!ModelState.IsValid)
    {
        // validation error
        return this.IsAjaxCall() ?
```

```

        new JsonResult { JsonRequestBehavior =
            JsonRequestBehavior.AllowGet, Data = ModelState }
            : this.Index(returnUrl);
    }
    this.membershipService.UpdateProfile(profile);
    if (this.IsAjaxCall())
    {
        return new JsonResult {
            JsonRequestBehavior = JsonRequestBehavior.AllowGet,
            Data = new {
                Update = true,
                Profile = profile,
                returnUrl = returnUrl } };
    }
    else
    {
        return RedirectToAction("UpdateSuccess",
            "Account", new { returnUrl = returnUrl });
    }
}
public ActionResult UpdateSuccess(string returnUrl)
{
    var model = new ProfileViewModel
    {
        Profile = this.membershipService.GetCurrentProfile(),
        returnUrl = returnUrl
    };
    return View(model);
}

private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes()
        .Select(o => new SelectListItem { Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null &&
                o.Id == profile.PreferredActivityTypeId) })
        .ToList();
    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null &&
                o.Abbreviation == profile.State) })
        .ToList();
    states.Insert(0,
        new SelectListItem { Text = "Any state",
            Value = string.Empty });
    model.PreferredActivityTypes = types;
    model.States = states;
}
}
}
}

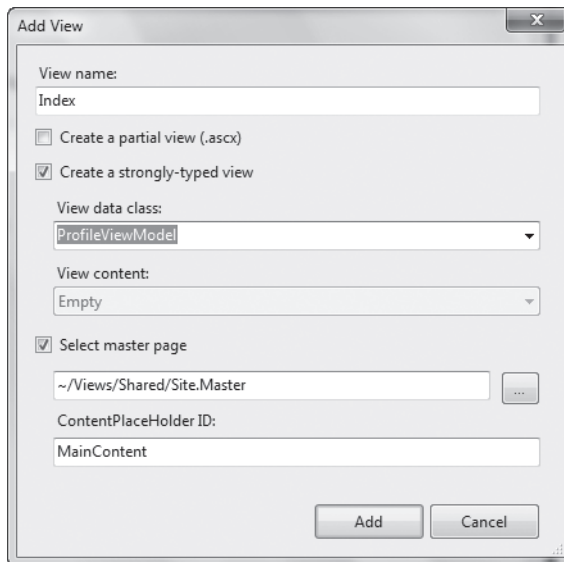
```


Creating the Account View

In the previous section, you created a controller with functionality that allows a user to update her information and view it. In this section, you're going to walk through the Visual Studio 2010 features that enable you to create the views that display this functionality to the user.

To create the Index view for the Account controller:

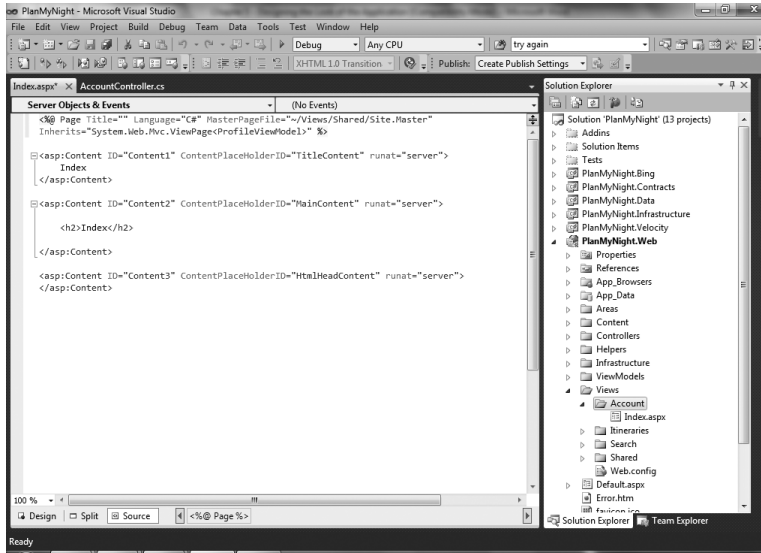
1. Navigate to the Views folder in the PlanMyNight.Web project.
2. Click the right mouse button on the Views folder, expand the Add submenu, and select New Folder.
3. Name the new folder **Account**.
4. Click the right mouse button on the new Account folder, expand the Add submenu, and select View.
5. Fill out the Add View dialog box as shown here:



The screenshot shows the 'Add View' dialog box with the following configuration:

- View name: Index
- Create a partial view (.ascx)
- Create a strongly-typed view
- View data class: ProfileViewModel
- View content: Empty
- Select master page: ~/Views/Shared/Site.Master
- ContentPlaceHolder ID: MainContent

- Click Add. You should see an HTML page with some `<asp:Content>` controls in the markup:



You might notice that it doesn't look much different from what you are used to seeing in Visual Studio 2003. By default, ASP.NET MVC 2 uses the ASP.NET Web Forms view engine, so there will be some commonality between MVC and Web Forms pages. The primary differences at this point are that the *page* class derives from *System.Web.Mvc.ViewPage<ProfileViewModel>* and there is no code-behind file. MVC does not use code-behind files, like ASP.NET Web Forms does, to enforce a strict separation of concerns. MVC pages are generally edited in markup view; the designer view is primarily for ASP.NET Web Forms applications.

For this page skeleton to become the main view for the Account controller, you should change the title content to be more in line with the other views:

```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Plan My Night - Profile
</asp:Content>
```

Next you need to add the client scripts you are going to use in the content placeholder for the *HtmlHeadContent*:

```
<asp:Content ID="Content3" ContentPlaceHolderID="HtmlHeadContent" runat="server">
<% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery-1.3.2.min.js"),
    "http://ajax.microsoft.com/ajax/jquery/jquery-1.3.2.min.js"); %>
<% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery.validate.js"),
    "http://ajax.microsoft.com/ajax/jquery.validate/1.5.5/jquery.validate.min.js"); %>
```

```

<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/MicrosoftMvcJQueryValidation.js"), "pmn"); %>
<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.common.js"), "pmn"); %>
<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.profile.js"), "pmn"); %>
<%= Ajax.RenderClientScripts() %>
</asp:Content>

```

This script makes use of extension methods for the *System.Web.Mvc.AjaxHelper*, which are found in the *PlanMyNight.Infrastructure* project, under the MVC folder.

With the head content setup, you can look at the main content of the view:

```

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
<div class="panel" id="profileForm">
    <div class="innerPanel">
        <h2><span>My Profile</span></h2>
        <% Html.EnableClientValidation(); %>
        <% using (Html.BeginForm("Update", "Account")) %>
        <% { %>
        <%=Html.AntiForgeryToken()%>
        <div class="items">
            <fieldset>
                <p>
                    <label for="FullName">Name:</label>
                    <%=Html.EditorFor(m => m.Profile.FullName)%>
                    <%=Html.ValidationMessage("Profile.FullName",
                        new { @class = "field-validation-error-wrapper" })%>
                </p>
                <p>
                    <label for="State">State:</label>
                    <%=Html.DropDownListFor(m => m.Profile.State, Model.States)%>
                </p>
                <p>
                    <label for="City">City:</label>
                    <%=Html.EditorFor(m => m.Profile.City, Model.Profile.City)%>
                </p>
                <p>
                    <label for="PreferredActivityTypeId">Preferred activity:</label>
                    <%=Html.DropDownListFor(m =>
                        m.Profile.PreferredActivityTypeId,
                        Model.PreferredActivityTypes)%>
                </p>
            </fieldset>
            <div class="submit">
                <%=Html.Hidden("returnUrl", Model.ReturnUrl)%>
                <%=Html.SubmitButton("submit", "Update")%>
            </div>
        </div>
        <div class="toolbox"></div>
        <% } %>
    </div>
</div>
</asp:Content>

```

Aside from some inline code, this looks to be fairly normal HTML markup. We're going to focus our attention on the inline code pieces to demonstrate the power they bring (as well as the simplicity).

Visual Studio 2003 In Visual Studio 2003, it was more commonplace to use server-side controls to display data and other display-time logic. However, because ASP.NET MVC view pages do not have a code-behind file, server-side logic executed in the view at render time must be done in the same file with the markup. ASP.NET Web Forms controls can still be used. Our example makes use of the `<asp:Content>` control. However, the functionality of ASP.NET Web Forms controls is generally limited because there is no code-behind file.

MVC makes a lot of use of what is known as HTML helpers. The methods contained under *System.Web.Mvc.HtmlHelper* emit small, standards-compliant HTML tags for various uses. This requires the MVC developer to type more markup than a Web Forms developer in some cases, but the developer has more direct control over the output. The strongly typed version of this extension class (*HtmlHelper<TModel>*) can be referenced in the view markup via the *ViewPage<TModel>.Html* property.

These are the *HTML* methods used in this form, which are only a fraction of what is available by default:

- *Html.EnableClientValidation* enables data validation to be performed on the client side based on the strongly typed ModelState dictionary.
- *Html.BeginForm* places a `<form>` tag in the markup and closes the form at the end of the *using* section. It takes various parameters for options, but the most common parameter is the name of the action and the controller to invoke that action on. This allows the MVC framework to generate the specific URL to target the form to at run time, rather than having to input a string URL into the markup.
- *Html.AntiForgeryToken* places a hidden field in the form with a check value that is also stored in a cookie in the visitor's browser and validated when the target of the form has the *ValidateAntiForgeryToken* attribute. Remember that you added this attribute to the *Update* method in the controller.
- *Html.EditorFor* is an overloaded method that inserts a text box into the markup. This is the strongly typed version of the *Html.Editor* method.
- *Html.DropDownListFor* is an overloaded method that places a drop-down list into the markup. This is the strongly typed version of the *Html.DropDownList* method.
- *Html.ValidationMessage* is a helper that will display a validation error message when a given key is present in the ModelState dictionary.

- *Html.Hidden* places a hidden field in the form, with the name and value that is passed in.
- *Html.SubmitButton* creates a Submit button for the form.



Note With the Index view markup complete, you only need to add the view for the *UpdateSuccess* action before you can see your results.

To create the UpdateSuccess view:

1. Expand the PlanMyNight.Web project in Solution Explorer, and then expand the Views folder.
2. Click the right mouse button on the Account folder.
3. Open the Add submenu, and click View.
4. Fill out the Add View dialog box so that it looks like this:

The screenshot shows the 'Add View' dialog box with the following configuration:

- View name: UpdateSuccess
- Create a partial view (.ascx)
- Create a strongly-typed view
- View data class: ProfileViewModel
- View content: Empty
- Select master page
- ~/Views/Shared/Site.Master
- ContentPlaceHolder ID: MainContent

After the view page is created, fill in the title content so that it looks like this:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">Plan My Night - Profile Updated</asp:Content>
```

And the placeholder for *MainContent* should look like this:

```
<asp:Content ContentPlaceholderID="MainContent" runat="server">
<div class="panel" id="profileForm">
  <div class="innerPanel">
    <h2><span>My Profile</span></h2>
    <div class="items">
      <p>Your profile has been successfully updated.</p>
      <h3>>> <a href="<%=Html.AttributeEncode(Model.ReturnUrl ??
        Url.Content("~/"))%>">Continue</a></h3>
    </div>
    <div class="toolbox"></div>
  </div>
</div>
</asp:Content>
```

To see the views created, you must perform an edit to the Site.Master file (located in the Views/Shared folder from the Web project's root). Line 33 of the file is commented out, and the comment tags should be removed so that it matches the following example:

```
<%=Html.ActionLink<AccountController>(c => c.Index(null), "My Profile")%>
```

With this last view created, you can now compile and launch the application. Click the Sign In button, as seen in the top right corner of Figure 2-6, and sign in to Windows Live ID.

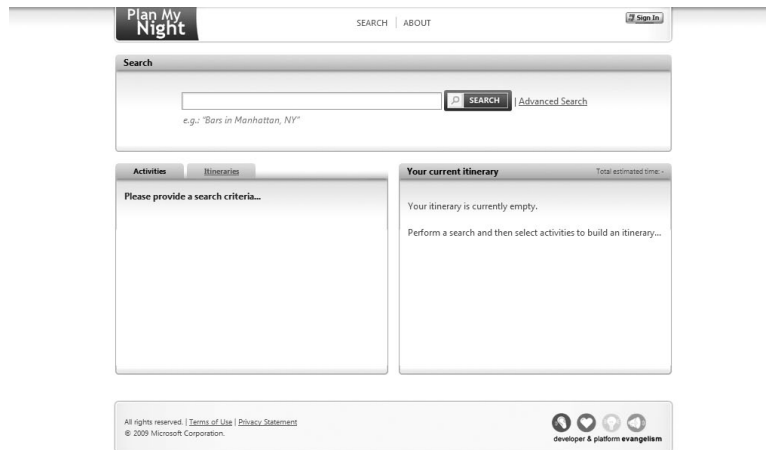


FIGURE 2-6 Plan My Night default screen

After you've signed in, you should be redirected to the Index view of the Account controller you created, shown in Figure 2-7.

Plan My Night SEARCH | MY PROFILE | MY ITINERARIES | ABOUT Hello New User! Sign Out

My Profile

Name:

State:

City:

Preferred activity:

[Update](#)

All rights reserved | [Terms of Use](#) | [Privacy Statement](#)
© 2009 Microsoft Corporation

developer & platform evangelism

FIGURE 2-7 Profile settings screen returned from the *Index* method of the Account controller

If instead you are returned to the search page, just click the My Profile link, located in the links at the center and top of the interface. To see the new data-validation features at work, try to save the form without filling in the Full Name field. You should get a result that looks like Figure 2-8.

My Profile

Name: ⚠ Full name is required.

State:

City:

Preferred activity:

[Update](#)

FIGURE 2-8 Example of failed validation during Model Binding checks

Because you enabled client-side validation, there was no post back. To see the server-side validation work, you would have to edit the `Index.aspx` file in the `Account` folder and comment out the call to `Html.EnableClientValidation`. The tight integration and support of AJAX and other JavaScript in MVC applications allows for server-side operations such as validation to be moved to the client side much more easily than they were previously.

Visual Studio 2003 In ASP.NET MVC applications, the value of the ID attribute for a particular HTML element is not transformed, like it is in ASP.NET Web Forms 1.0. In Visual Studio 2003, a developer would have to make sure to set the *UniqueID* of a control/element into a JavaScript variable so that it could be accessed by external JavaScript. This was done to make sure the ID was unique. However, it was always an extra layer of complexity added to the interaction between ASP.NET 1.0 Web Forms controls and JavaScript. In MVC, this transformation does not happen, but it is up to the developers to ensure uniqueness of the ID. It should also be noted that ASP.NET 4.0 Web Forms now supports disabling the ID transformation on a per-control basis, if the developer so wishes.

With the completed `Account` controller and related views, you have filled in the missing “core” functionality of `Plan My Night`, while taking a brief tour of some new features in Visual Studio 2010 and MVC 2.0 applications. But MVC is not the only choice for Web developers. ASP.NET Web Forms has been the primary application type for ASP.NET since it was released, and it continues to be improved upon in Visual Studio 2010. In the next section, we’ll explore creating an ASP.NET Web Form with the Visual Designer to be used in the MVC application.

Using the Designer View to Create a Web Form

Applications will encounter an unexpected condition at some point in their lifetime of use. The companion application is no different, and when it does encounter an unexpected condition, it returns an error screen like that shown in Figure 2-9.

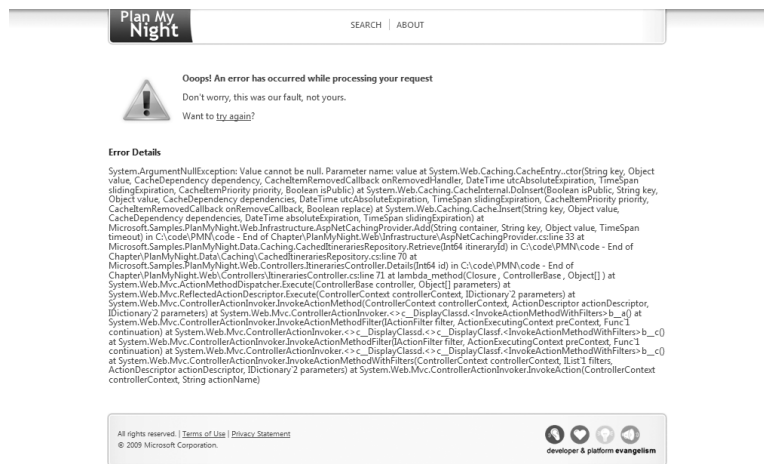
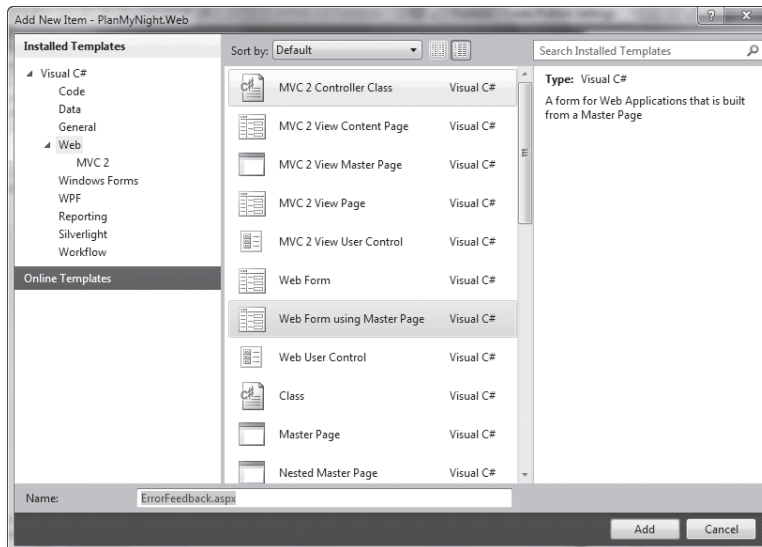


FIGURE 2-9 Example of an error screen in the `Plan My Night` application

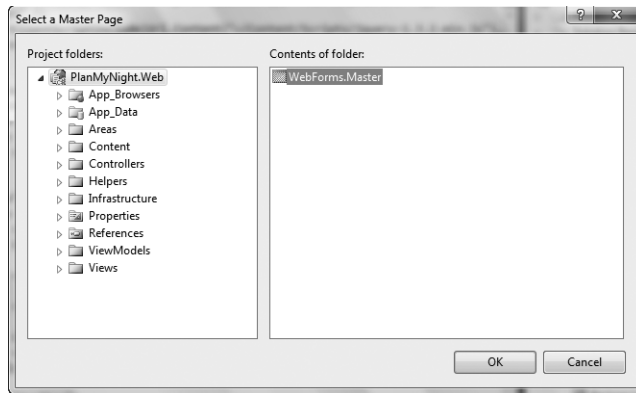
Currently, a user who sees this screen really has only the option of trying his action again or using the navigation links along the top area of the application. (Of course, that might also cause another error.) Adding an option for the user to provide feedback allows the developers to gain information about the situation that might not be apparent by using the standard exception message and stack trace. To show a different way to create a user interface component for Plan My Night, the error feedback page is going to be created as an ASP.NET Web Form using primarily the Designer view in Visual Studio. Before you can begin designing the form, you need to create a base form file to work from.

To create a new Web form:

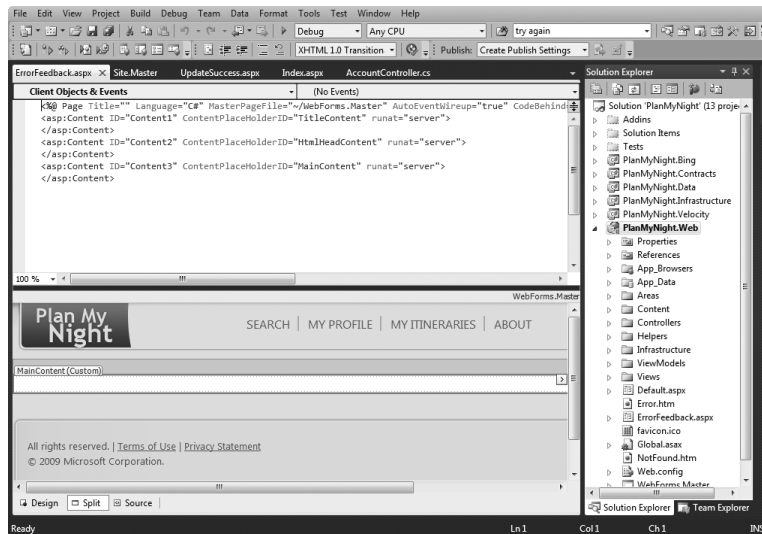
1. Open the context menu on the PlanMyNight.Web project (by clicking the right mouse button), open the Add submenu, and select New Item.
2. In the Add New Item dialog box, select Web Form Using Master Page and call the item **ErrorFeedback.aspx** in the Name field.



3. The dialog screen to associate a master page with this Web form will appear. On the Project Folders side, ensure that the main PlanMyNight.Web folder is selected and then select the WebForms.Master item on the right.



- The resulting page can be shown in the source mode (or Design view) instead of Split view. Switch the view to Split (located at the bottom of the window, just like in previous Visual Studio versions). When you are done, the screen should look similar to this:



Note Split view is recommended so that you can see the source the designer is generating and to add extra markup as needed.

It's a good idea to pin the control toolbox open on the screen because you'll be dragging controls and elements to the content area during this section. The toolbox, if not present already, can be found under the View menu.

Start by dragging a div element (under the HTML group) from the toolbox into the MainContent section of the designer. A div tab will appear, indicating that the new element you added is the currently selected element. Open the context menu for the div, and choose Properties (which can also be opened by pressing the F4 key). With the Properties window open, edit the (*Id*) property to have a value of *profileForm*. (Casing is important.) Also, change the *Class* property to have a value of *panel*. After editing the values, the size of your content area will have changed, because CSS is applied in the Design view.

Visual Studio 2003 A much-needed update to the Web Forms designer surface from Visual Studio 2003 is the application of CSS. This allows the developer to see in real-time how the style changes are applied, without having to run the application. When viewed in Visual Studio 2003, the designer for the search.aspx page will appear similar to Figure 2-10.

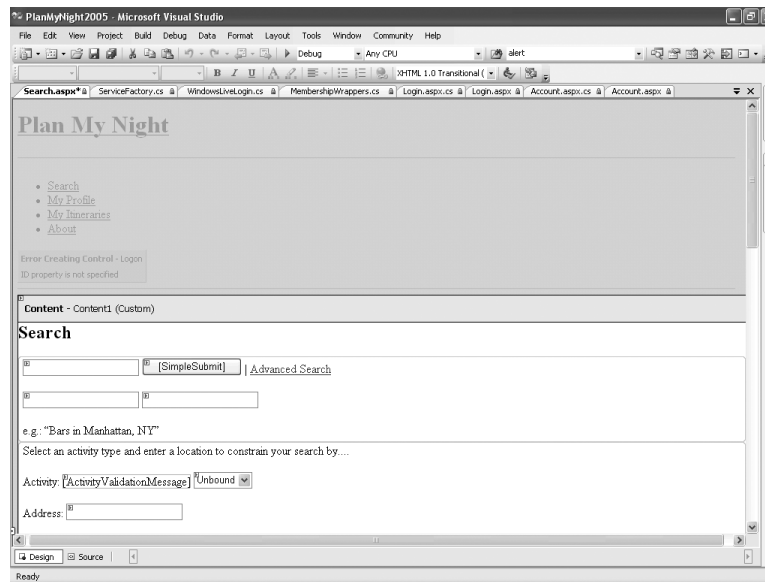


FIGURE 2-10 Designer view of an ASP.NET Web page in Visual Studio 2003

Drag another div inside the first one, and set its *class* property to *innerPanel*. In the markup panel, add the following markup to the *innerPanel*:

```
<h2><span>Error Feedback</span></h2>
```

After the close of the `<h2>` tag, add a new line and open the context menu. Choose Insert Snippet, and follow the click path of ASP.NET > form. This will create a server-side form tag for you to insert Web controls into. Inside the form tag, place a div tag with the class attribute set to *items* and then a fieldset tag inside the div tag.

Next drag a TextBox control (found under Standard) from the toolbox and drop it inside the fieldset tag. Set the ID of the text box to **FullName**. Add a `<label>` tag before this control in the markup view, set its *for* property to the ID of the textbox and set its value to **Full Name:** (making sure to include the colon). To set the value of a `<label>` tag, place the text between the `<label>` and `</label>` tags. Surround these two elements with a `<p>`, and you should have something like Figure 2-11 in the Design view.

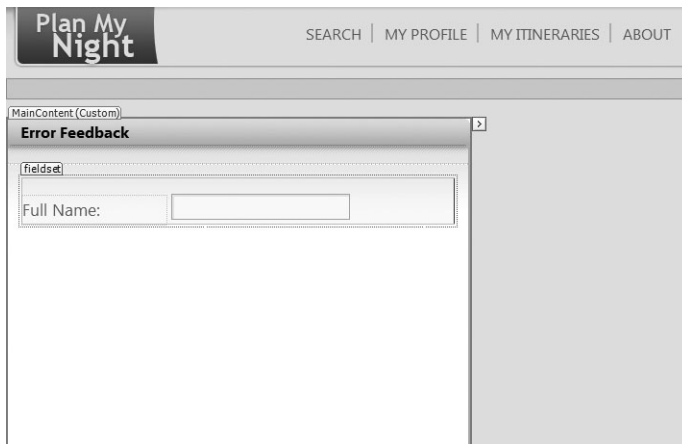


FIGURE 2-11 Current state of ErrorFeedback.aspx in the Design view

Add another text box, and label it in a similar manner as the first, but set the ID of the text box to **EmailAddress** and the label value to **Email Address:** (making sure to include the colon). Repeat the process a third time, setting the TextBox ID and label value to **Comments**. There should now be three labels and three single-line TextBox controls in the Design view. The Comments control needs multiline input, so open its property page and set *TextMode* to *Multiline*, *Rows* to 5, and *Columns* to 40. This should create a much wider text box in which the user can enter comments.

Use the Insert Snippet feature again, after the Comments text box, and insert a “div with class” tag (HTML>divc). Set the class of the div tag to *submit*, and drag a Button control from the toolbox into this div. Set the Button’s *Text* property to *Send Feedback*.

The designer should show something similar to what you see in Figure 2-12, and at this point you have a page that will submit a form.

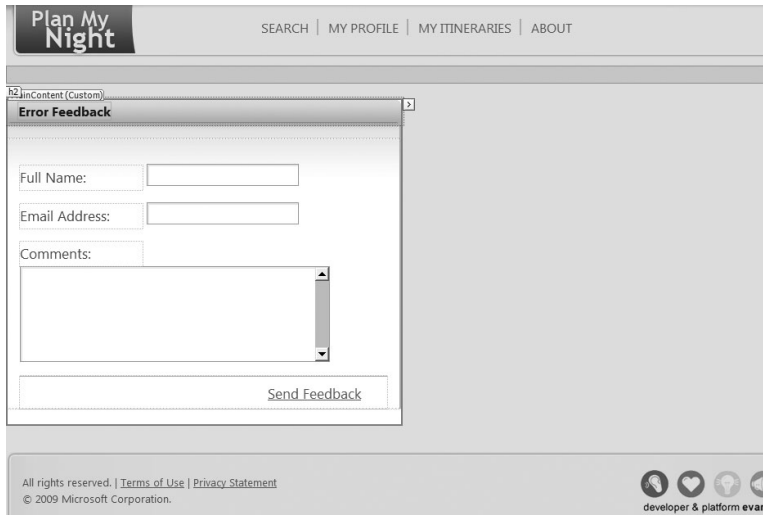


FIGURE 2-12 The ErrorFeedback.aspx form with a complete field set

However, it does not perform any validation on the data being submitted. To do this, you’ll take advantage of some of the validation controls present in ASP.NET. You’ll make the Full Name and Comments boxes required fields and perform a regex validation of the e-mail address to ensure that it matches the right pattern.

Under the Validation group of the toolbox are some premade validation controls you’ll use. Drag a *RequiredFieldValidator* object from the toolbox, and drop it to the right of the Full Name text box. Open the properties for the validation control, and set the *ControlToValidate* property to *FullName*. (It’s a drop-down list of controls on the page.) Also, set the *CssClass* to *field-validation-error*. This changes the display of the error to a red triangle used elsewhere

in the application. Finally, change the Error Message property to **Name is Required**. (See Figure 2-13.)

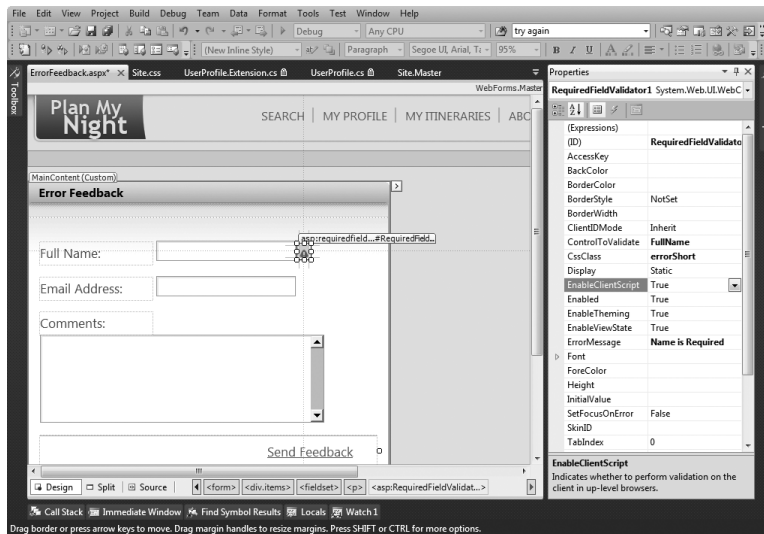


FIGURE 2-13 Validation control example

Repeat these steps for the Comments box, but substitute the *ErrorMessage* and *ControlToValidate* property values as appropriate.

For the Email Address field, you want to make sure the user types in a valid e-mail address, so for this field drag a *RegularExpressionValidator* control from the toolbox and drop it next to the Email Address text box. The property values are similar for this control in that you set the *ControlToValidate* property to *EmailAddress* and the *CssClass* property to *field-validation-error*. However, with this control you define the regular expression to be applied to the input data. This is done with the *ValidationExpression* property, and it should be set like this:

```
[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}
```

The error message for this validator should say something like “Must enter a valid e-mail address.”

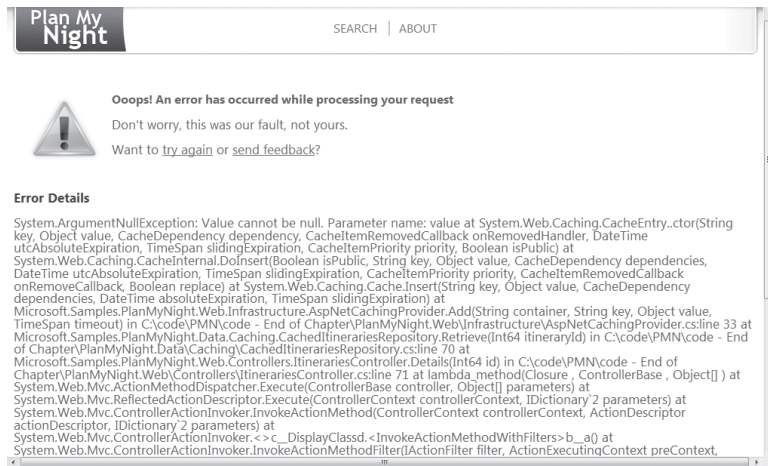
The form is complete. To see it in the application, you need to add the option of providing feedback to a user when the user encounters an error. In Solution Explorer, navigate the *PlanMyNight.Web* project tree to the Views folder and then to the Shared subfolder. Open the *Error.aspx* file in the markup viewer, and go to line 35. This is the line of the error screen where you ask the user if she wants to try her action again and where you'll put the option for sending the feedback. After the question text in the same paragraph, add the following markup:

```
or <a href="/ErrorFeedback.aspx">send feedback</a>?
```

This will add an option to go to the form you just created whenever there is a general error in the MVC application. To see your form, you'll have to cause an error in your application.

To cause an error in the Plan My Night application:

1. Start the application.
2. After the default search page is up, type the following into the browser address bar:
http://www.planmynight.net:48580/Itineraries/Details/38923828.
3. Because it is highly unlikely such an itinerary ID exists in the database, an error screen will be shown.



4. With the error screen visible, click the link to go to the feedback form. Try to submit the form with invalid data.

Error Feedback

Full Name: ⚠ Name is Required

Email Address:

Comments:

[Send Feedback](#)

ASP.NET uses client-side script (when the browser supports it) to perform the validation, so no postbacks occur until the data passes. On the server side, when the server does receive a postback, a developer can check the validation state with the *Page.IsValid* property in the code-behind. However, because you used client-side validation (which is on by default), this will always be *true*. The only code in the code-behind that needs to be added is to redirect the user on a postback (and check the *Page.IsValid* property, in case client validation missed something):

```
protected void Page_Load(object sender, EventArgs e)
{
    if (this.IsPostBack && this.IsValid)
    {
        this.Response.Redirect("/", true);
    }
}
```

This really isn't very useful to the user, but our goal in this section was to work with the designer to create an ASP.NET Web Form. This added a new interface to the PlanMyNight. Web project, but what if you wanted to add new functionality to the application in a more modular sense, such as some degree of functionality that can be added or removed without having to compile the main application project. This is where an extensibility framework like the Managed Extensibility Framework (MEF) can show the benefits it brings.

Extending the Application with MEF

A new technology available in Visual Studio 2010 as part of the .NET Framework 4 is the Managed Extensibility Framework (MEF). The Managed Extensibility Framework provides developers with a simple (yet powerful) mechanism to allow their applications to be extended by third parties after the application has been shipped. Even within the same application, MEF allows developers to create applications that completely isolate components, allowing them to be managed or changed independently. It uses a resolution container to map components that provide a particular function (exporters) and components that require that functionality (importers), without the two concrete components having to know about each other directly. Resolutions are done on a contract basis only, which easily allows components to be interchanged or introduced to an application with very little overhead.

See Also MEF's community Web site, containing in-depth details about the architecture, can be found at <http://mef.codeplex.com>.

The companion Plan My Night application has been designed with extendibility in mind, and it has three “add-in” module projects in the solution, under the Addins solution folder. (See Figure 2-14.)

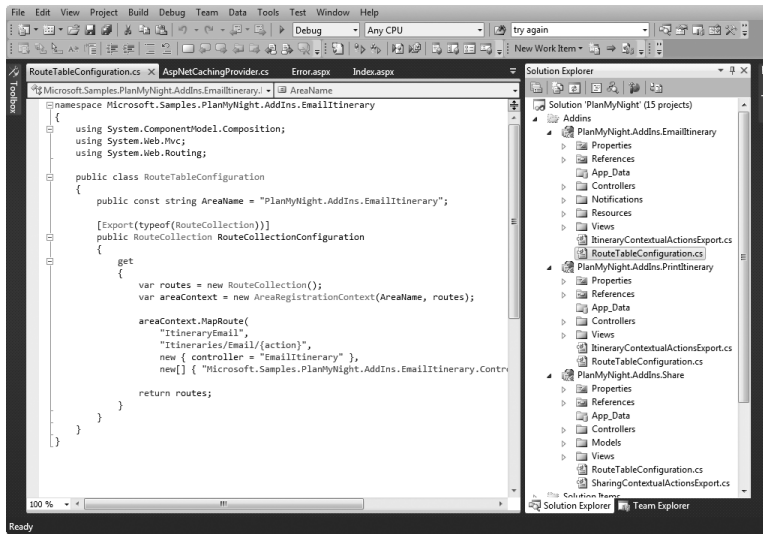


FIGURE 2-14 The Plan My Night application add-ins

`PlanMyNight.AddIns.EmailItinerary` adds the ability to e-mail itinerary lists to anyone the user sees fit to receive them. `PlanMyNight.AddIns.PrintItinerary` provides a printer-friendly view of the itinerary. Lastly, `PlanMyNight.AddIns.Share` adds in social-media sharing functions (so that the user can post a link to an itinerary) as well as URL-shortening operations. None of these projects reference the main `PlanMyNight.Web` application or are referenced by it. They do have references to the `PlanMyNight.Contracts` and `PlanMyNight.Infrastructure` projects, so they can export (and import in some cases) the correct contracts via MEF as well as use any of the custom extensions in the infrastructure project.



Note Before doing the next step, if the Web application is not already running, launch the `PlanMyNight.Web` project so that the UI is visible to you.

To add the modules to your running application, run the `DeployAllAddins.bat` file, found in the same folder as the `PlanMyNight.sln` file. This will create new folders under the `Areas` section of the `PlanMyNight.Web` project. These new folders, one for each plug-in, will contain the files needed to add their functionality to the main Web application. The plug-ins appear in the application as extra options under the current itinerary section of the search results page and on the itinerary details page. After the batch file is finished running, go to

the interface for Plan My Night, search for an activity, and add it to the current itinerary. You should notice some extra options under the itinerary panel other than just New and Save. (See Figure 2-15.)

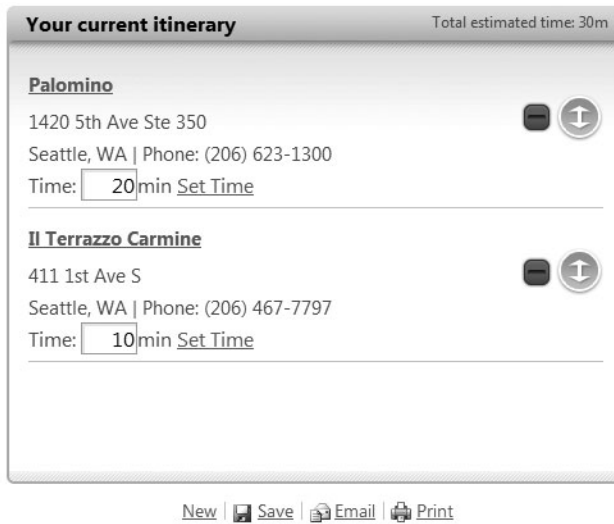


FIGURE 2-15 Location of the e-mail add-in in the UI

The social sharing options will show in the interface only after the itinerary is saved and marked public. (See Figure 2-16.)



FIGURE 2-16 Location of the social-sharing add-in in the UI

Visual Studio 2003 Visual Studio 2003 does not have anything that compares to MEF. To support plug-ins, a developer would have to either write the plug-in framework from scratch or purchase a commercial package. Either of the two options led to proprietary solutions an external developer would have to understand in order to create a component for them. Adding MEF to the .NET Framework helps to cut down the entry barriers to producing extendible applications and the plug-in modules for them.

Print Itinerary Add-in Explained

To demonstrate how these plug-ins wire into the application, let's have a look at the `PrintItinerary.Addin` project. When you expand the project you should see something like the structure shown in Figure 2-17.

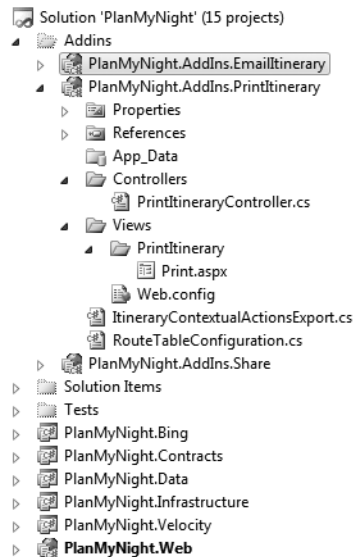


FIGURE 2-17 Structure of the `PrintItinerary` project

Some of this structure is similar to the `PlanMyNight.Web` project (Controllers and Views). That's because this add-in will be placed in an MVC application as an area. If you look more closely at the `PrintItineraryController.cs` file in the Controller folder, you can see it is similar in structure to the controller you created earlier in this chapter (and similar to any of the other controllers in the Web application). However, some key differences set it apart from the controllers that are compiled in the primary `PlanMyNight.Web` application.

Focusing on the class definition, you'll notice some extra attributes:

```
[Export("PrintItinerary", typeof(IController))]  
[PartCreationPolicy(CreationPolicy.NonShared)]
```

These two attributes describe this type to the MEF resolution container. The first attribute, *Export*, marks this class as providing an *IController* under the contract name of *PrintItinerary*. The second attribute declares that this object supports only nonshared creation and cannot be created as a shared/singleton object. Defining these two attributes are all you need to do to have the type used by MEF. In fact, *PartCreationPolicy* is an optional attribute, but it should be defined if the type cannot handle all the creation policy types.

Further into the *PrintItineraryController.cs* file, the constructor is decorated with an *ImportingConstructor* attribute:

```
[ImportingConstructor]  
public PrintItineraryController(IServiceFactory serviceFactory) :  
this(  
    serviceFactory.GetItineraryContainerInstance(),  
    serviceFactory.GetItinerariesRepositoryInstance(),  
    serviceFactory.GetActivitiesRepositoryInstance())  
{  
}  
}
```

The *ImportingConstructor* attribute informs MEF to provide the parameters when creating this object. In this particular case, MEF provides an instance of *IServiceFactory* for this object to use. Where the instance comes from is of no concern to the *this* class and really assists with creating modular applications. For our purposes, the *IServiceFactory* contracted is being exported by the *ServiceFactory.cs* file in the *PlanMyNight.Web* project.

The *RouteTableConfiguration.cs* file registers the URL route information that should be directed to the *PrintItineraryController*. This route, and the routes of the other add-ins, are registered in the application during the *Application_Start* method in the *Global.asax.cs* file of *PlanMyNight.Web*:

```
// MEF Controller factory  
var controllerFactory = new MefControllerFactory(container);  
ControllerBuilder.Current.SetControllerFactory(controllerFactory);  
  
// Register routes from Addins  
foreach (RouteCollection routes in container.GetExportedValues<RouteCollection>())  
{  
    foreach (var route in routes)  
    {  
        RouteTable.Routes.Add(route);  
    }  
}
```

The *controllerFactory*, which was initialized with a MEF container containing path information to the Areas subfolder (so that it enumerated all the plug-ins), is assigned to be the controller factory for the lifetime of the application. This allows controllers imported via MEF to be usable anywhere in the application. The routes these plug-ins respond to are then retrieved from the MEF container and registered into the MVC routing table.

The *ItineraryContextualActionsExport.cs* file exports information to create the link to this plug-in, as well as metadata for displaying it. This information is used in the *ViewModelExtensions.cs* file, in the *PlanMyNight.Web* project, when building a view model for display to the user:

```
// get addin links and toolboxes
var addinBoxes = new List<RouteValueDictionary>();
var addinLinks = new List<ExtensionLink>();

addinBoxes.AddRange(AddinExtensions.GetActionsFor("ItineraryToolbox", model.Id == 0 ? null :
new { id = model.Id }));

addinLinks.AddRange(AddinExtensions.GetLinksFor("ItineraryLinks", model.Id == 0 ? null : new
{ id = model.Id }));
```

The call to *AddinExtensions.GetLinksFor* enumerates over exports in the MEF Export provider and returns a collection of them to be added to the local *addinLinks* collection. These are then used in the view to display more options when they are present.

Summary

In this chapter, we explored a few of the many new features and technologies found in Visual Studio 2010 that were used to create the companion Plan My Night application. We walked through creating a controller and its associated view and how the ASP.NET MVC framework offers Web developers a powerful option for creating Web applications. We also explored how using the Managed Extensibility Framework in application design can allow plug-in modules to be developed external to the application and loaded at run time. In the next chapter, we'll explore how debugging applications has been improved in Visual Studio 2010.

Chapter 3

From 2003 to 2010: Debugging an Application

After reading this chapter on debugging, you will be able to

- Use the new debugger features of Microsoft Visual Studio 2010
- Create unit tests and execute them in Visual Studio 2010
- Compare what was available to you or see what was different for you as a developer in Visual Studio 2003

As we were writing this book, we realized how much the debugging tools and developer aids have evolved over the last three versions of Visual Studio. Focusing on debugging an application and writing unit tests just increases the opportunities we have to work with Visual Studio 2010.

Visual Studio 2010 Debugging Features

In this chapter, you'll go through the different debugging features using a modified Plan My Night application. If you installed the companion content at the default location, you'll find the modified Plan My Night application at the following location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 3\Code. Double-click the PlanMyNight.sln file.

First, before diving into the debugging session itself, you'll need to set up a few things:

1. In Solution Explorer, ensure that PlanMyNight.Web is the startup project. If the project name is not in bold, right-click on PlanMyNight.Web and select Set As StartUp Project.
2. To get ready for the next steps, in the PlanMyNight.Web solution open the Global.asax.cs file by clicking the triangle beside the Global.asax folder and then double-clicking the Global.asax.cs file as shown in Figure 3-1.

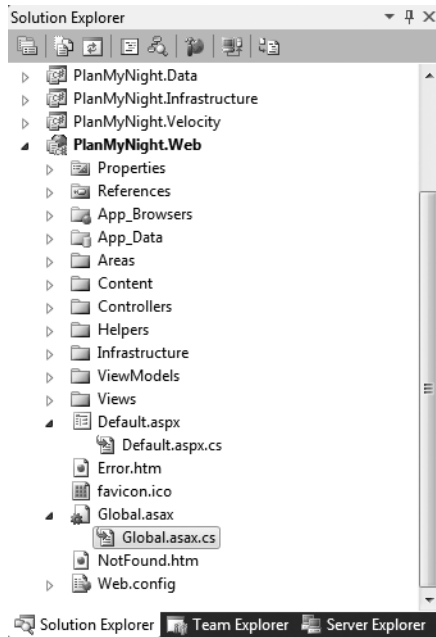


FIGURE 3-1 Solution Explorer before opening the file Global.asax.cs

Managing Your Debugging Session

Using the Plan My Night application, you'll examine how a developer can manage and share breakpoints. And with the use of new breakpoint enhancements, you'll learn how to inspect the different data elements in the application in a much faster and more efficient way. You'll also look at new minidumps and the addition of a new intermediate language (IL) interpreter that allows you to evaluate managed code properties and functions during minidump debugging.

New Breakpoint Enhancements

At this point, you have the Global.ascx.cs file opened in your editor. The following steps walk you through some ways to manage and share breakpoints:

1. Navigate to the *Application_BeginRequest(object sender, EventArgs e)* method, and set a breakpoint on the line that reads *var url = HttpContext.Current.Request.Url;* by clicking in the left margin or pressing F9. Look at Figure 3-2 to see this in action.

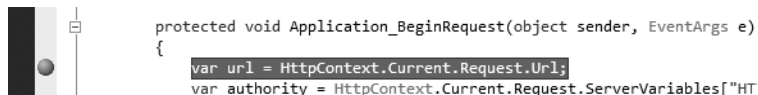


FIGURE 3-2 Creating a breakpoint

2. Press F5 to start the application in debug mode. You should see the developer Web server starting in the system tray and a new browser window opening. The application should immediately stop at the breakpoint you just created. The Breakpoints window might not be visible even after starting the application in debug mode. If that is the case, you can make it visible by going to the Debug menu and selecting Windows and then Breakpoints, or you can use the following keyboard shortcut: Ctrl+D+B.

Visual Studio 2003 Here is an area where there's a night-and-day difference between Visual Studio 2003 and Visual Studio 2010. Thankfully, the ease of debugging in Visual Studio 2010 is not comparable to the pain developers experienced in Visual Studio 2003.

Debugging Web applications in Visual Studio 2003 required you to complete a series of careful tasks to be successful. For instance, you had to enable debugging in a few different places in the Configuration Manager; you had to configure IIS carefully to avoid getting the dreaded Temporary ASP.NET Files Access Denied security exception. Basically, it was not developer friendly; it required lots of configuration and still there was no guarantee of success. Many articles were written to help solve the intricacies of debugging an ASP.NET Web application with Internet Information Services (IIS) 5 and 6 and Visual Studio 2003. A good reference that you might have consulted a few times is this one: [http://msdn.microsoft.com/en-us/library/aa290100\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa290100(VS.71).aspx). Many of those blogs or articles were just there to allow professional developers to debug on their workstations. Doing some research in online forums revealed how much folks like you were struggling for years through dozens of tough Web application debugging situations. For instance, debugging an ASP.NET application along with a few Web services was quite a challenge to set up.

The creation of the personal Web server (also known as "Cassini") improved things tremendously, but a few enhancements in Visual Studio 2010 make it easier to modify the necessary configuration files, modify project settings, and deploy to IIS. These improvements help developers write good code and spend less time trying to debug it. You'll find that it definitely feels different to debug in Visual Studio 2010—you'll find it refreshing and easier.

You should now see the Breakpoints window as shown in Figure 3-3.

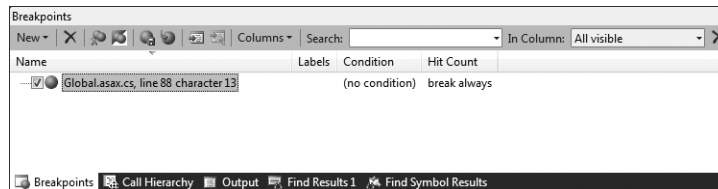


FIGURE 3-3 Breakpoints window

3. In the same method, add three more breakpoints so that the editor and the Breakpoints window look like those shown in Figure 3-4.

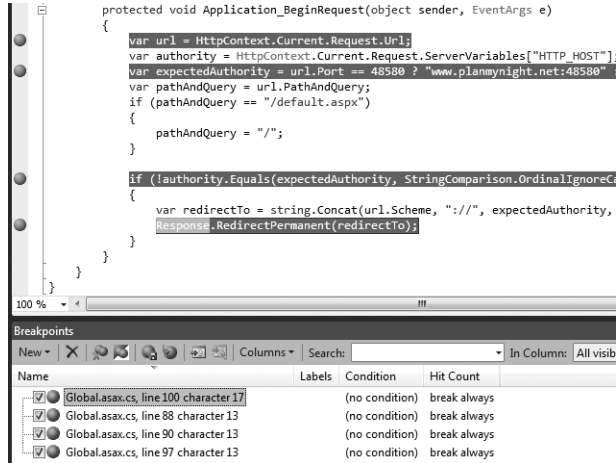


FIGURE 3-4 Code editor and Breakpoints window with three new breakpoints

Visual Studio 2003 As a reader and a professional developer who used Visual Studio 2003 often, you probably noticed a series of new buttons as well as new fields in the Breakpoints window in this exercise. As a reminder, take a look at Figure 3-5 for a quick comparison of what it looks like in Visual Studio 2003.

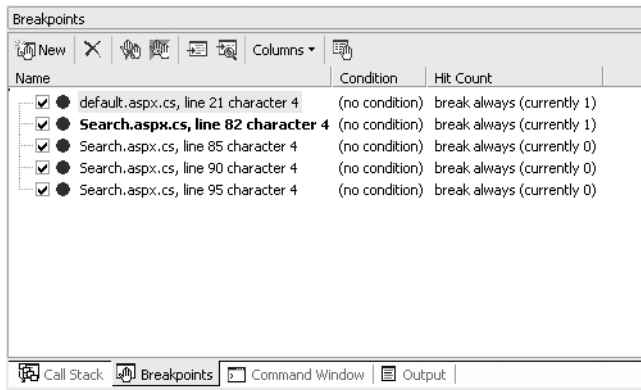


FIGURE 3-5 Visual Studio 2003 Breakpoints window

4. Notice that the Labels column is now available to help you index and search breakpoints. It is a really nice and useful feature that Visual Studio 2010 brings to the table. To use this feature, you simply right-click on a breakpoint in the Breakpoints window

and select Edit Labels or use the keyboard shortcut Alt+F9, L. Take a look at Figure 3-6 as a reference.

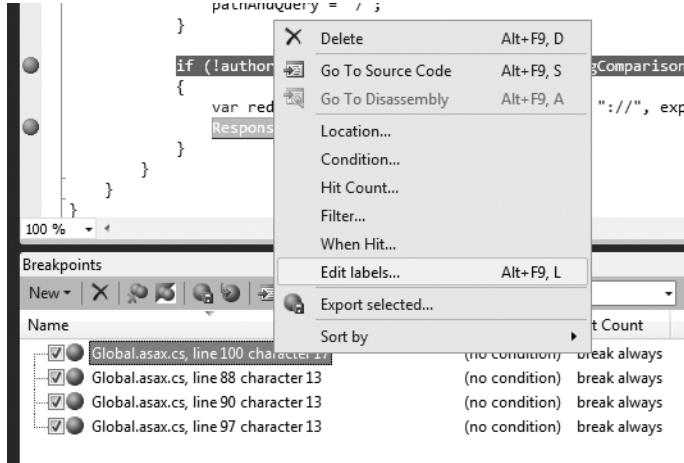


FIGURE 3-6 Edit Labels option

5. In the Edit Breakpoint Labels window, add labels for the selected breakpoint (which is the first one in the Breakpoints window). Type **ContextRequestUrl** in the Type A New Label text box, and click Add. Repeat this operation on the next breakpoint, and type a label name of **Url**. When you are done, click OK. You should see a window that looks like Figure 3-7 while you are entering them, and to the right you should see the Breakpoints window after you are done with those two operations.

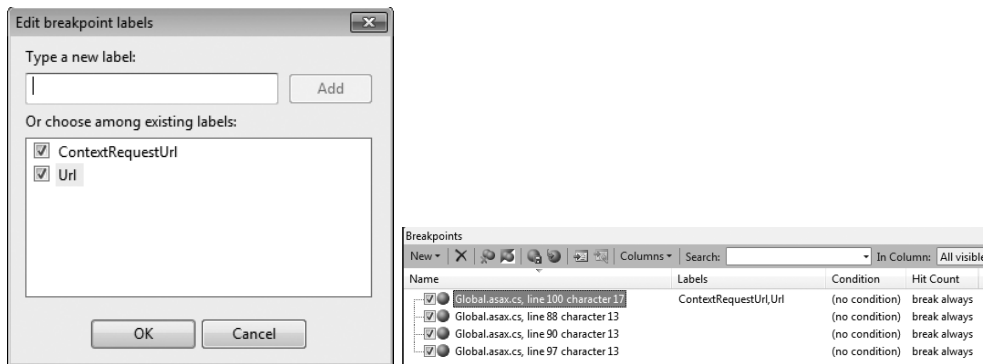


FIGURE 3-7 Adding labels that show up in the Breakpoints window



Note You can also right-click on the breakpoint in the left margin and select Edit Labels to accomplish the same tasks just outlined.



Note You'll see that when adding labels to a new breakpoint you can choose any of the existing labels you have already entered. You'll find these in the Or Choose Among Existing Labels area, which is shown in the Edit Breakpoint Labels dialog box on the left in Figure 3-7.

- Using any of the ways you just learned, add labels for each of the breakpoints and make sure your Breakpoints window looks like Figure 3-8 after you're done.

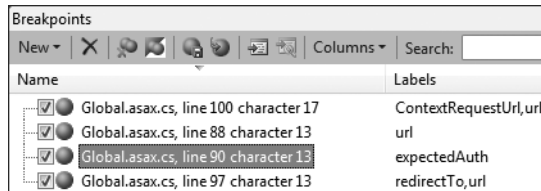


FIGURE 3-8 Breakpoints window with all labels entered




When you have a lot of code and you are in the midst of a debugging session, it would be great to be able to filter the displayed list of breakpoints. That's exactly what the new Search feature in Visual Studio 2010 allows you to do.


- To see the Search feature in action, just type **url** in the search text box and you'll see the list of breakpoint is filtered down to breakpoints containing *url* in one of their labels.

In a team environment where you have many developers and testers working together, often two people at some point in time are working on the same bugs. In Visual Studio 2003, the two people needed to sit near each other, send one another screen shots, or send one another line numbers of where to put breakpoints to refine where they should look while debugging a particular bug.



Important One of the great new additions to breakpoint management in Visual Studio 2010 is that you can now export breakpoints to a file and then send them to a colleague, who can then import them into his own environment. Another scenario that this feature is useful for is to share breakpoints between machines. We'll see how to do that next.

- In the Breakpoints window, click the Export button  to export your breakpoints to a file, and then save the file on your desktop. Name the file **breakexports.xml**.
- Delete all the breakpoints either by clicking the Delete All Breakpoints Matching The Current Search Criteria button  or by selecting all the breakpoints and clicking the Delete The Selected Breakpoints button . The only purpose of deleting them is to simulate two developers sharing them or one developer sharing breakpoints between two machines.

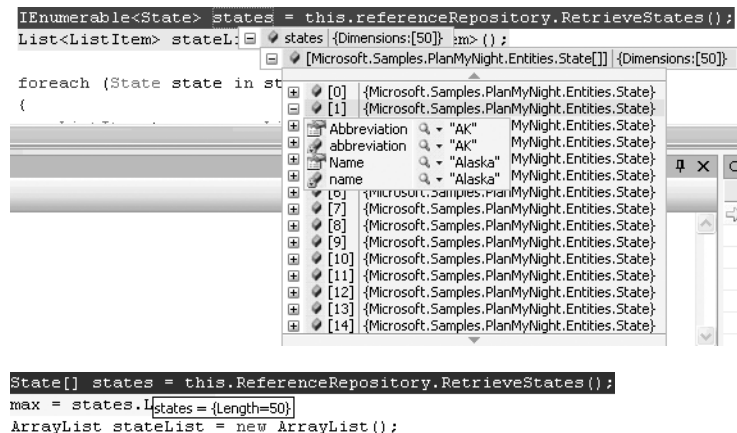
10. You'll now import your breakpoints by clicking the Import button  and loading them from your desktop. Notice that all your breakpoints with all of their properties are back and loaded in your environment. For the purposes of this chapter, delete all the breakpoints.

Visual Studio 2003 Putting breakpoints in client-side code (JavaScript) and stopping and tracing was not really friendly. As a developer, you had to place a debugger (or stop in VBScript) statement in your client-side code and then trace into the code. But there was no IntelliSense support for client-side code. In Visual Studio 2010, you get great support for JavaScript as well as for the latest jQuery iteration. It was already good in Visual Studio 2008, but the integration in Visual Studio 2010 is faster and you don't have to do anything to get it.

Inspecting the Data

When you are debugging your applications, you know how much time one can spend stepping into the code and inspecting the content of variables, arguments, and so forth. Maybe you can remember when you were learning to write code, a while ago, when debuggers weren't a reality or when they were really rudimentary. Do you remember (maybe not—you might not be as old as we are) how many *printf* or *WriteLn* statements you had to write to inspect the content of different data elements.

Visual Studio 2010 From the days of Visual Studio 2003, there already was a big improvement from the days of writing to the console with all kinds of statements because we had a real debugger with new functionalities. New data visualizers allowed you to see XML as a well-formed XML snippet and not as a long string. Furthermore, with those data visualizers, you could view arrays in a more useful way, with the list of elements and their indices, and you accomplished that by simply hovering your mouse over the object. Take a look at Figure 3-9 for an example.



```

IEnumerable<State> states = this.referenceRepository.RetrieveStates();
List<ListItem> stateList;
states {Dimensions:[50]} :m> ();
[Microsoft.Samples.PlanMyNight.Entities.State[]] {Dimensions:[50]}
foreach (State state in st
(
  [0] {Microsoft.Samples.PlanMyNight.Entities.State}
  [1] {Microsoft.Samples.PlanMyNight.Entities.State}
  Abbreviation "AK" MyNight.Entities.State}
  abbreviation "AK" MyNight.Entities.State}
  Name "Alaska" MyNight.Entities.State}
  name "Alaska" MyNight.Entities.State}
  [7] {Microsoft.Samples.PlanMyNight.Entities.State}
  [8] {Microsoft.Samples.PlanMyNight.Entities.State}
  [9] {Microsoft.Samples.PlanMyNight.Entities.State}
  [10] {Microsoft.Samples.PlanMyNight.Entities.State}
  [11] {Microsoft.Samples.PlanMyNight.Entities.State}
  [12] {Microsoft.Samples.PlanMyNight.Entities.State}
  [13] {Microsoft.Samples.PlanMyNight.Entities.State}
  [14] {Microsoft.Samples.PlanMyNight.Entities.State}

State[] states = this.ReferenceRepository.RetrieveStates();
max = states.Length;
ArrayList stateList = new ArrayList();

```

FIGURE 3-9 Collection view versus an array view in the debugger in Visual Studio 2010 and in Visual Studio 2003

Visual Studio 2003 In Visual Studio 2003, there was no other way to do data visualizations apart from Watch, Locals, and Quick Watch or some of the more rudimentary ways described earlier. You couldn't hover over a field in an array to get its content. If you had XML or formatted text in a string, you couldn't read it in any meaningful way other than opening Notepad or another file in Visual Studio and pasting the content of a string to see the formatted content of a variable.

Although those DataTip data visualization techniques are still available in Visual Studio 2010, a few great enhancements have been added that make DataTips even more useful. The DataTip enhancements have been added in conjunction with another new feature of Visual Studio 2010, multimonitor support. Floating DataTips can be valuable to you as a developer. Having the ability to put them on a second monitor can make your life a lot easier while debugging because it keeps the data that always needs to be in context right there on the second monitor. The following steps demonstrate how to use these features:

1. In the Global.ascx.cs file, insert breakpoints on lines 89 and 91, lines starting with the source code *var authority* and *var pathAndQuery*, respectively.
2. You are now going to experiment with the new DataTip features. Start the debugger by pressing F5. When the debugger hits the first breakpoint, move your mouse over the word *url* and click on the pushpin as seen in Figure 3-10.

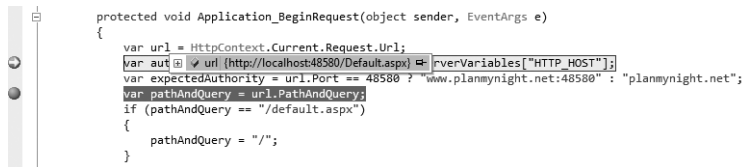


FIGURE 3-10 The new DataTip pushpin feature

3. To the right of the line of code, you should see the pinned DataTip (as seen in Figure 3-11 on the left). If you hover your mouse over the DataTip, you'll get the DataTip management bar (as seen in Figure 3-11 on the right).



FIGURE 3-11 On the left is the pinned DataTip, and on the right is the DataTip management bar



Note You should also see in the breakpoint gutter a blue pushpin indicating that the DataTip is pinned. The pushpin should look like this: . Because you have a breakpoint on that line, the pushpin is actually underneath it. To see the pushpin, just toggle the breakpoint by clicking on it in the gutter. Toggle once to disable the breakpoint and another time to get it back.



Note If you click on the double arrow pointing down in the DataTip management bar, you can insert a comment for this DataTip, as shown in Figure 3-12. You can also remove the DataTip altogether by clicking the X button in the DataTip management bar.

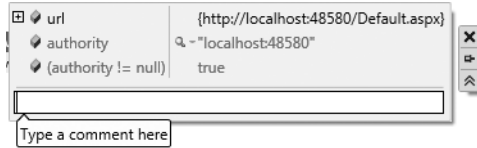


FIGURE 3-12 Inserting a comment for a DataTip

- One nice feature of the new DataTip is that you can insert any expression to be evaluated right there in your debugging session. For instance, right-click on the DataTip name, in this case on *url*, select Add Expression, type **authority**, and then add another one like this: **(authority != null)**. You'll see that the expressions are evaluated immediately and will continue to be evaluated for the rest of the debugging session every time your debugger stops on those breakpoints. At this point in the debugging session, the expression should evaluate to *null* and *false*, respectively.
- Press F10 to execute the line where the debugger stopped, and look at the *url* DataTip as well as both expressions. They should contain values based on the current context. Take a look at Figure 3-13 to see this in action.

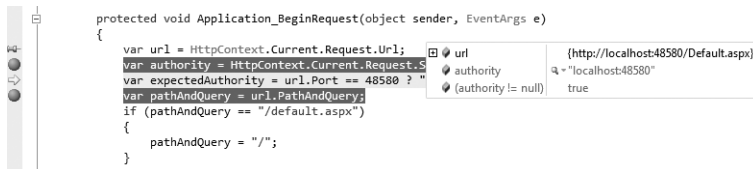


FIGURE 3-13 The *url* pinned DataTip with the two evaluated expressions

- Although it is nice to be able to have a mini-watch window where it matters—right there where the code is executing—you can also see that it is superimposed on the source code being debugged. Keep in mind that you can move the DataTip window anywhere you want in the code editor by simply dragging it. Take a look at Figure 3-14 for an example.



FIGURE 3-14 Move the pinned DataTip away from the source code

7. Because it is pinned, the DataTip window stays where you pinned it, so it will not be in view if you trace into another file. But in some cases, you need the DataTip window to be visible at all times. For instance, keeping it visible is interesting for global variables that are always in context or for multimonitor scenarios. To move a DataTip, you have to first unpin it by clicking the pushpin in the DataTip management bar. You'll see that it turns yellow. That indicates you can now move it wherever you want—for instance, over Solution Explorer, to a second monitor, over your desktop, or to any other window. Take a look at Figure 3-15 for an example.



FIGURE 3-15 Unpinned DataTip over Solution Explorer and the Windows desktop



Note If the DataTip is not pinned, the debugger stops in another file and method, and the DataTip contains items that are out of context, the DataTip windows will look like Figure 3-16. You can retry to have the debugger evaluate the value of an element by

clicking on this button: . However, if that element has no meaning in this context, it's possible that nothing happens.

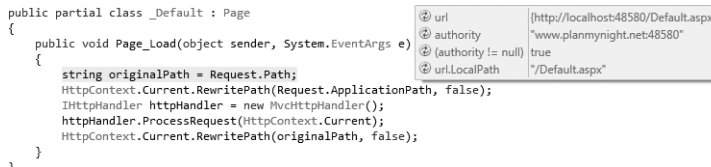


FIGURE 3-16 DataTip window with out-of-context items



Note You'll get an error message if you try to pin outside the editor, as seen in Figure 3-17.

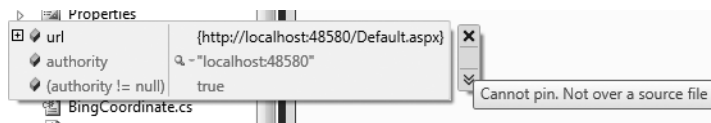


FIGURE 3-17 Error message that appears when trying to pin a DataTip outside the code editor



Note Your port number might be different than in the screen shots just shown. This is normal—it is a random port used by the personal Web server included with Visual Studio.



Note You can also pin any child of a pinned item. For instance, if you look at url and expand its content by pressing the plus sign (+), you'll see that you can also pin a child element, as seen in Figure 3-18.

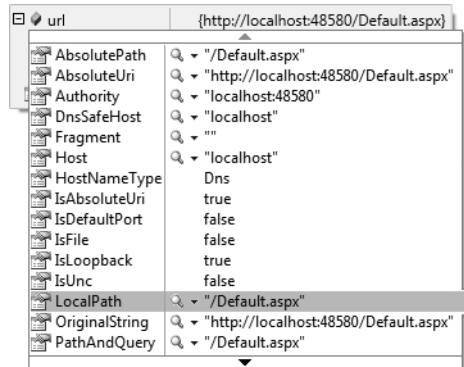


FIGURE 3-18 Pinned child element within the url DataTip


- Before stopping the debugger, go back to the Global.ascx.cs if you are not already there and re-pin the DataTip window. Then stop the debugging session by clicking the Stop Debugging button in the debug toolbar () or by pressing Shift+F5. Now if you hover your mouse over the blue pushpin in the breakpoint gutter, you'll see the values from the last debug session which is a nice enhancement over the watch window. Take a look at Figure 3-19 for what you should see.



FIGURE 3-19 Values from the last debug session for a pinned DataTip



Note As with the breakpoints, you can export or import the DataTips by going to the Debug menu and selecting Export DataTips or Import DataTips, respectively.

Using the Minidump Debugger

Many times in real-world situations, you'll have access to a minidump from your product support team. Apart from their bug descriptions and repro steps, it might be the only thing you have to help debug a customer application. Visual Studio 2010 adds a few enhancements to the minidump debugging experience.

Visual Studio 2003 In Visual Studio 2003, you could debug managed application or minidump files, but you had to use an extension if your code was written in managed code. You had to use a tool called SOS and load it in the debugger using the Immediate window. You had to attach the debugger both in native and managed mode, and you couldn't expect to have information in the call stack or Locals window. You had to use commands for SOS in the Immediate window to help you go through minidump files. With application written in native code, you used normal debugging windows and tools. To read more about this or just to refresh your knowledge of the topic, you can read the *Bug Slayer* column in MSDN magazine here: <http://msdn.microsoft.com/en-us/magazine/cc164138.aspx>.

Let's see the new enhancements to the minidump debugger. First you need to create a crash from which you'll be able to generate a minidump file:

1. In Solution Explorer in the PlanMyNight.Web project, rename the file Default.aspx to **DefaultA.aspx**. Note the *A* appended to the word "Default."
2. Make sure you have no breakpoints left in your project. To do that, look in the Breakpoints window and delete any breakpoints left there using any of the ways you learned earlier in the chapter.
3. Press F5 to start debugging the application. Depending on your machine speed, soon after the build process is complete you should see an unhandled exception of type *HttpException*. Although the bug is simple in this case, let's go through the steps of creating the minidump file and debugging it. Take a look at Figure 3-20 to see what you should see at this point.

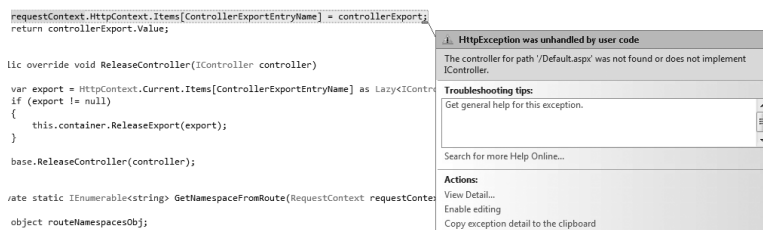


FIGURE 3-20 The unhandled exception you should expect

4. It is time to create the minidump file for this exception. Go to the Debug menu, and select Save Dump As as seen in Figure 3-21. You should see the name of the process from which the exception was thrown. In this case, the process from which the exception was thrown was Cassini or the Personal Web Server in Visual Studio. Keep the file name proposed (WebDev.WebServer40.dmp), and save the file on your desktop. Note that it might take some time to create the file because the minidump file size will be close to 300 MB.

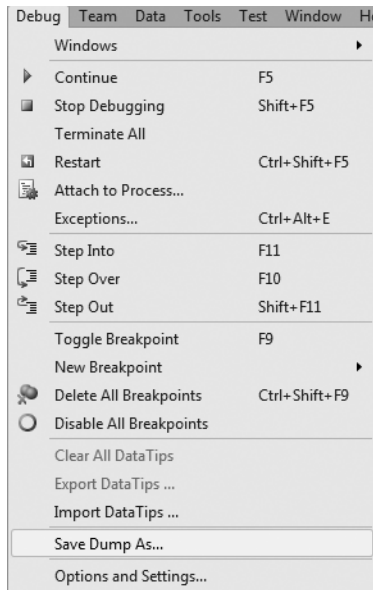


FIGURE 3-21 Saving the minidump file

5. Stop Debugging by pressing Shift+F5 or the Stop Debugging button.
6. Next, go to the File menu and close your solution.
7. In the File menu, click Open and point to the desktop to load your minidump file named WebDev.WebServer40.dmp. Doing so opens the Minidump File Summary page, which gives you some summary information about the bug you are trying to fix. (Figure 3-22 shows what you should see.) Before you start to debug, you'll get basic information from that page such as the following: process name, process architecture, operating system version, CLR version, modules loaded, as well as some actions you can take from that point. From this place, you can set the paths to the symbol files. Conveniently, the Modules list contains the version and path on disk of your module, so finding the symbols and source code is easy. The CLR version is 4.0; therefore, you can debug here in Visual Studio 2010.

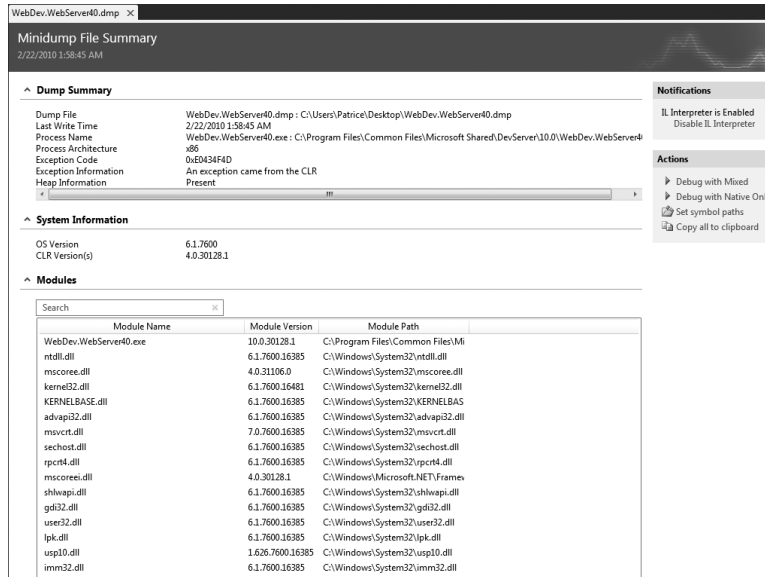


FIGURE 3-22 Minidump summary page

- To start debugging, locate the Actions list on the right side of the Minidump File Summary page and click Debug With Mixed.
- You should see almost immediately a first-chance exception like the one shown in Figure 3-23. In this case, it tells you what the bug is; however, this won't always be the case. Continue by clicking the Break button.

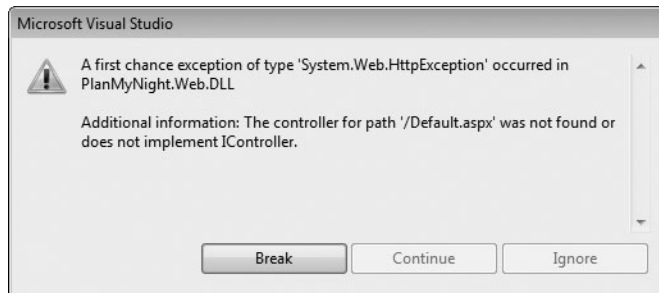


FIGURE 3-23 First-chance exception

- You should see a green line indicating which instruction caused the exception. If you look at the source code, you'll see in your Autos window that the *controllerExport* variable is *null*, and just before that we specified that if the variable was *null* we wanted to have an *HttpException* thrown if the file to load was not found. In this case, the file to look for is *Default.aspx*, as you can see in the Locals window in the *controllerName* variable. You can glance at many other variables, objects, and so forth in the Locals and Autos windows containing the current context. Here, you have only one call that

belongs to your code, so the call stack indicates that the code before and after is external to your process. If you had a deeper chain of calls in your code, you could step back and forth in the code and look at the variables. Figure 3-24 shows a summary view of all that.

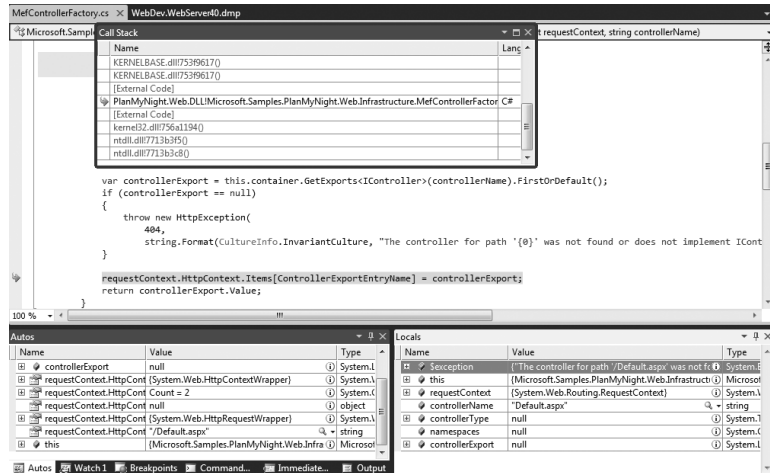


FIGURE 3-24 Autos, Locals, and Call Stack windows, and the next instruction to execute

11. OK, you found the bug, so stop the debugging by pressing Shift+F5 or clicking the Stop Debugging button. Then fix the bug by reloading the PlanMyNight solution and renaming the file back to **default.aspx**. Then rebuild the solution by going to the Build menu and selecting Rebuild Solution. Then press F5, and the application should be working again.

Web.Config Transformations

This next new feature, while small, is one that will delight many developers because it saves them time while debugging. The feature is the Web.Config transformations that allow you to have transform files that show the differences between the debug and release environments. As an example, connection strings are often different from one environment to the other; therefore, by creating transform files with the different connection strings—because ASP.NET provides tools to change (transform) web.config files—you'll always end up with the right connection strings for the right environment. To learn more about how to do this, take a look at the following article on MSDN: <http://go.microsoft.com/fwlink/?LinkId=125889>.

Creating Unit Tests

Most of the unit test framework and tools are unchanged in Visual Studio 2010 Professional. It is in other versions of Visual Studio 2010 that the change in test management and test tools is really apparent. Features such as UI Unit Tests, IntelliTrace, and Microsoft Test Manager 2010 are available in other product versions like Visual Studio 2010 Premium and

Visual Studio 2010 Ultimate. To see which features are covered in the Application Lifecycle Management and for more specifics, refer to the following article on MSDN: [http://msdn.microsoft.com/en-us/library/ee789810\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/ee789810(VS.100).aspx).

Visual Studio 2003 With Visual Studio 2003, your options to create unit tests and execute them were limited to third-party tools and frameworks like NUnit and other commercial products created by Microsoft partners.

In this part of the chapter, we'll simply show you how to add a unit test for a class you'll find in the Plan My Night application. We won't spend time defining what a unit test is or what it should contain; rather, we'll show you within Visual Studio 2010 how to add tests and execute them.

You'll add unit tests to the Plan My Night application for the Print Itinerary Add-in. To create unit tests, open the solution from the companion content folder. If you do not remember how to do this, you can look at the first page of this chapter for instructions. After you have the solution open, just follow these steps:

1. In Solution Explorer, expand the project PlanMyNight.Web and then expand the Helpers folder. Then double-click on the file ViewHelper.cs to open it in the code editor. Take a look at Figure 3-25 to make sure you are at the right place.

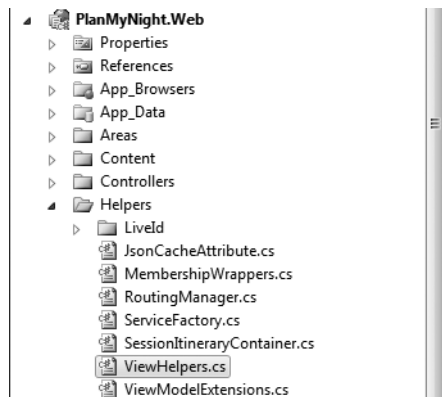


FIGURE 3-25 The PlanMyNight.Web project and ViewHelper.cs file in Solution Explorer

2. In the code editor, you can add unit tests in two different ways. You can right-click on a class name or on a method name and select Create Unit Tests. You can also go to the Test menu and select New Test. We'll explore the first way of creating unit tests. This

way Visual Studio automatically generates some source code for you. Right-click the *GetFriendlyTime* method, and select Create Unit Tests. Figure 3-26 shows what it looks like.

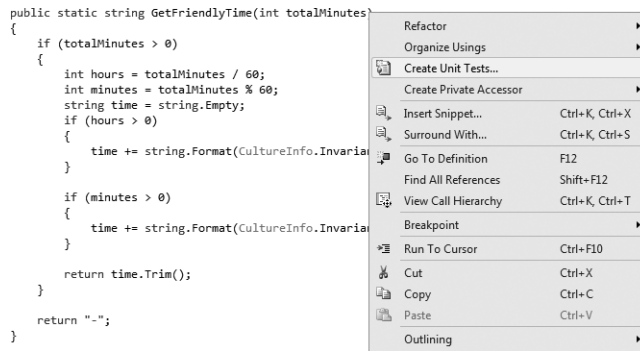


FIGURE 3-26 Contextual menu to create unit tests from by right-clicking on a class name

3. After selecting Create Unit Tests, you'll be presented with a dialog that, by default, shows the method you selected from that class. To select where you want to create the unit tests, click on the drop-down combo box at the bottom of this dialog and select *PlanMyNight.Web.Tests*. If you didn't have an existing location, you would have simply selected Create A New Visual C# Test Project from the list. Figure 3-27 shows what you should be seeing.

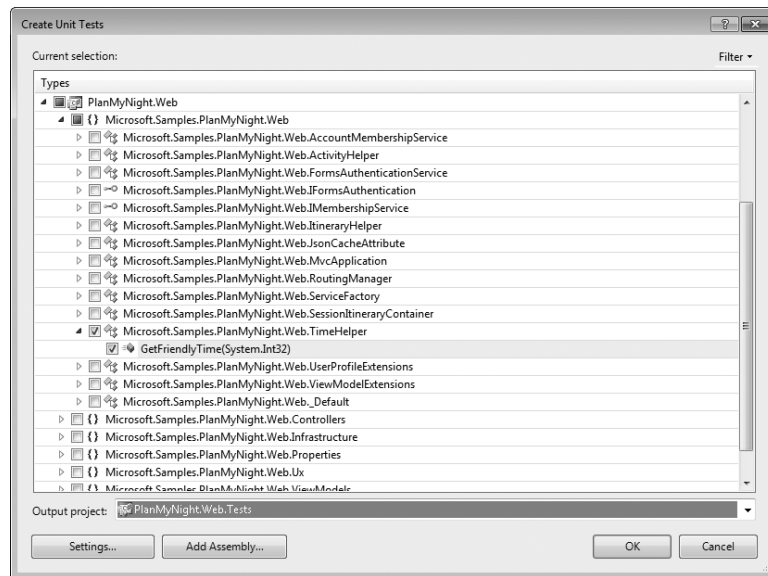


FIGURE 3-27 Selecting the method you want to create a unit test against

4. After you click OK, the dialog switches to a test-case generation mode and displays a progress bar. After this is complete, a new file is created named *TimeHelperTest.cs* that has autogenerated code stubs for you to modify.
5. Remove the method and its attributes because you'll create three new test cases for that method. Remove the following code:

```

/// <summary>
///A test for GetFriendlyTime
///</summary>
// TODO: Ensure that the UrlToTest attribute specifies a URL to an ASP.NET page
// (for example, // http://.../Default.aspx). This is necessary for the unit test to be
// executed on the web server, // whether you are testing a page, web service, or a WCF
// service.
[TestMethod()]
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("C:\\Users\\Patrice\\Documents\\Chapter 3\\code\\
PlanMyNight.Web", "/")]
[UrlToTest("http://localhost:48580/")]
public void GetFriendlyTimeTest()
{
    int totalMinutes = 0; // TODO: Initialize to an appropriate value
    string expected = string.Empty; // TODO: Initialize to an appropriate value
    string actual;
    actual = TimeHelper.GetFriendlyTime(totalMinutes);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}

```

6. Add the three simple test cases validating three key scenarios used by PlanMyNight. To do that, insert the following source code right below the method attributes that were left behind when you deleted the block of code in step 5:

```

[TestMethod]
public void ZeroReturnsSlash()
{
    Assert.AreEqual("-", TimeHelper.GetFriendlyTime(0));
}

[TestMethod]
public void LessThan60MinutesReturnsValueInMinutes()
{
    Assert.AreEqual("10m", TimeHelper.GetFriendlyTime(10));
}

[TestMethod()]
public void MoreThan60MinutesReturnsValueInHoursAndMinutes()
{
    Assert.AreEqual("2h 3m", TimeHelper.GetFriendlyTime(123));
}

```


7. In the PlanMyNight.Web.Tests project, create a solution folder called **Helpers**. Then move your TimeHelperTests.cs file to that folder so that your project looks like Figure 3-28 where you are done.

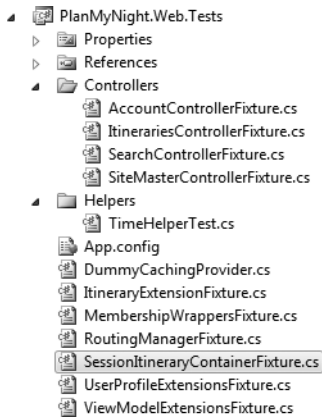


FIGURE 3-28 TimeHelperTest.cs in its Helpers folder

8. It is time to execute your newly created tests. To execute only your newly created tests, go into the code editor and place your cursor on the class named *public class TimeHelperTest*. Then you can either go to the Test menu, select Run, and finally select Test In Current Context or accomplish the same thing using the keyboard shortcut CTRL+R, T. Look at Figure 3-29 for a reference.

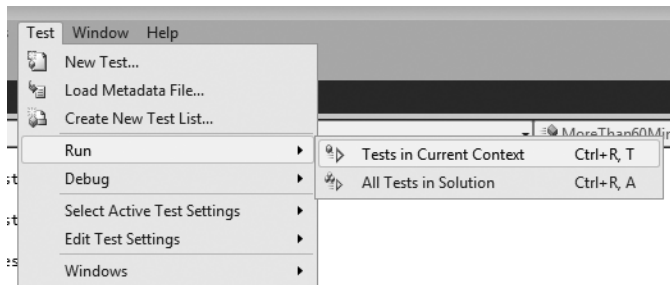


FIGURE 3-29 Test execution menu

9. Performing this action executes only your three tests. You should see the Test Results window (shown in Figure 3-30) appear at the bottom of your editor with the test results.

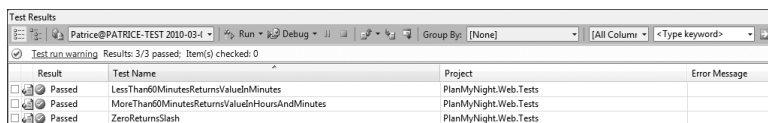


FIGURE 3-30 Test Results window for your newly created tests



More Info Depending on what you select, you might have a different behavior when you choose the Tests In Current Context option. For instance, if you select a test method like *ZeroReturnsSlash*, you'll execute only this test case. However, if you click outside the test class, you could end up executing every test case, which is the equivalent of choosing All Tests In Solution.

New Threads Window

The emergence of computers with multiple cores and the fact that language features give developers many tools to take advantage of those cores creates a new problem: the difficulty of debugging concurrency in applications. The new Threads window enables you, the developer, to pause threads and search the calling stack to see artifacts similar to those you see when using the famous SysInternals Process Monitor (<http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>). You can display the Threads window by going to Debug and selecting Windows And Threads while debugging an application. Take a look at Figure 3-31 to see the Threads window as it appears while debugging Plan My Night.

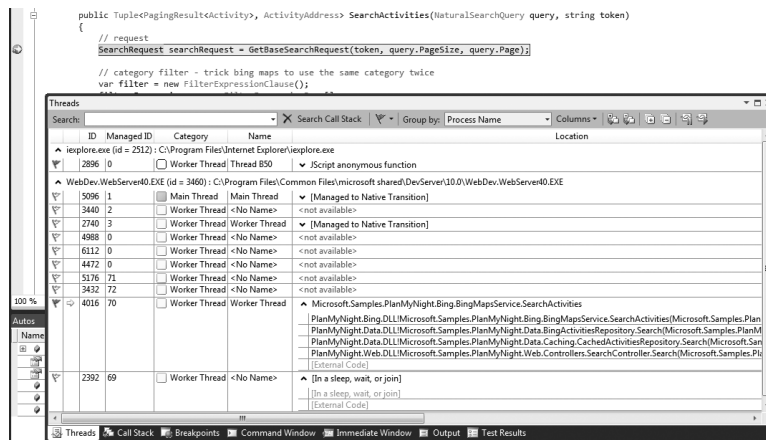


FIGURE 3-31 Displaying the Threads window while debugging Plan My Night

The Threads window allows you to freeze threads and then thaw them whenever you are ready to let them continue. It can be really useful when you are trying to isolate particular effects. You can debug both managed code and unmanaged code. If your application uses thread, you'll definitely love this new feature of the debugger in Visual Studio 2010.

Visual Studio 2003 In Visual Studio 2003, the Thread window was rudimentary. It enabled you to suspend or switch to a thread. It was tough to know more about those threads because you didn't have any more information from within Visual Studio 2003. There was no filtering, call-stack searching and expansion, and grouping. The columns were in a fixed order. Back then, the notion of multiple cores existed but wasn't as much in use as it is today. Furthermore, developing for multicore CPUs wasn't facilitated by the .NET Framework in any way like it is today with .NET 4.0 and libraries like PLINQ.

Summary

In this chapter, you learned how to manage your debugging sessions by using new break-point enhancements and employing new data-inspection and data-visualization techniques. You also learned how to use the new minidump debugger and tools to help you solve real customer problems from the field. The chapter also showed you how to raise the quality of your code by writing unit tests and how Visual Studio 2010 Professional can help you do this. Multicore machines are now the norm, and so are multithreaded applications. Therefore, the fact that Visual Studio 2010 Professional has specific debugger tools for finding issues in multithreaded applications is great news.

Finally, throughout this chapter you also saw how Visual Studio 2010 Professional has raised the bar in terms of debugging applications and has given professional developers the tools to debug today's feature-rich experiences. You saw that it is a clear improvement over what was available in Visual Studio 2003. The exercises in this chapter scratched the surface of how you'll save time and money by moving to this new debugging environment and showed that Visual Studio 2010 is more than a small iteration in the evolution of Visual Studio. It represents a huge leap in productivity for developers. The gap between Visual Studio 2003 and Visual Studio 2010 in terms of debugging is less severe than in earlier versions. The quantity of information provided by the debugger, the way to visualize the information, and the ease of use and configuration of the projects are the biggest changes that help you be more productive. You'll also see in the next chapter how easy it is to deploy Web applications, their databases, IIS settings, and all of their configurations.

The various versions of Visual Studio 2010 give you a great list of improvements related to the debugger and testing. My personal favorites are IntelliTrace—[http://msdn.microsoft.com/en-us/library/dd264915\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(VS.100).aspx)—which is available only in Visual Studio 2010 Ultimate and Microsoft Test Manager. IntelliTrace enables test teams to have much better experiences using Visual Studio 2010 and Visual Studio 2010 Team Foundation Server—[http://msdn.microsoft.com/en-us/library/bb385901\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb385901(VS.100).aspx).

Chapter 4

From 2003 to 2010: Deploying an Application

After reading this chapter on deployment techniques, you will be able to

- Deploy a Web application and an SQL database using Web Deployment Packages
- Deploy a Web application using One-Click Publish

Deploying a Web application is never easy—it should be, but it never is. Whether you are trying to deploy to your monthly paid host company or to a datacenter, you usually have to push the files to some location using FTP, using another custom upload tool, or packaging them in a .zip file. Then someone—either you or a system engineer—has to perform other configuration steps so that your users can hit your Web application successfully. This chapter will bring you the latest and greatest (and easiest) ways to deploy your Web application, and it will show you how to do so while having more control and going through fewer manual steps. It will also offer you comparisons with all three previous versions of Microsoft Visual Studio.

Visual Studio 2010 Web Deployment Packages

In this chapter, you'll deploy your application using Web Deployment Packages. You'll also see some of the pain points of doing this that were present in previous versions of Visual Studio. We'll take the examples mentioned in this chapter's introduction and go through the different steps one had to go through to deploy the application. Then we'll compare how it is done in each version of Visual Studio from 2003 all the way up to 2010. If you installed the companion content at the default location, you'll find the modified Plan My Night application at the following location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 4\Code. Double-click the PlanMyNight.sln file.

Let's start with the example of deploying to a monthly paid shared hosting company. Here are the big steps one would have to take to deploy a Web application in such a scenario:

1. Get the files that are needed for your application to your Web hosting company using FTP or their custom control panel upload tool. If you want only the files needed to execute, you need to sort them out and know exactly which ones you need and transfer only those files.
2. After the files are copied, you have to go to the control panel and make sure all the files are in the right place and then go through the configuration of your application

for various Internet Information Services (IIS) settings. Depending on the host company, you might have to create an application in IIS, select the application pool and its type, and so forth.

3. If your application has a SQL database, you then have to create it, and perhaps populate some domain tables by executing SQL scripts.
4. You might also have to create some security settings for SQL and your application in general.
5. Finally, you'll probably have to modify your `web.config` to match some of the servers and modify some database-related configuration options such as your connection strings.

Similarly, in an enterprise you often have to deploy to datacenters and on servers for which you won't likely have access to physically or even remotely with remote desktop. In most enterprises, you won't even have a chance to talk to the engineers doing the deployments because often it happens during off-peak hours and in other time zones. And the only thing they have is the reliability of your scripts and your deployment documentation. This means your deployment scripts must be bulletproof and your deployment documentation needs to be thorough—because it will be tested by other engineers with little or no knowledge of your project—so that nothing is assumed.

With any deployment technologies, the ideal situation is to come up with the deployment packages as you develop your product. If you wait until after the code is completed, it is extremely hard to do a good job and your work is a lot more error prone. Here is where Visual Studio 2010 and the Web Deployment Tool come in handy.

Visual Studio 2010 and Web Deployment Packages

Using the Plan My Night application and Visual Studio 2010, you'll examine how as a developer you can deploy your Web application and get it to a state where you can deploy it with confidence. Using the tools, you'll be able to test it and refine it using IIS on your machine and then deploy it to your shared hosting service. In an enterprise, you do this and then add the deployment package creation in your MSBuild or TFSBuild processes.

What Was Available Before Visual Studio 2010

In Visual Studio 2003, your options to deploy Web applications without buying a specialized tool were roughly limited to the following three:

1. Using the command **`xcopy`** for deployment, also known simply as *XCOPY deployment*.
2. Using the Copy Project option in Visual Studio 2003.
3. Using the Visual Studio 2003 Web Setup Project option.

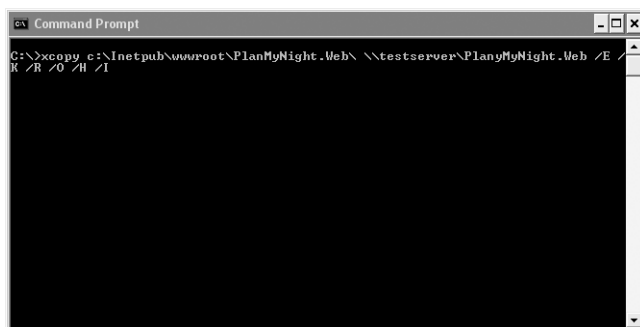
Options 1 and 2 were useful for simple deployment, but when things got more complicated the only options were either option 3 or using a product like InstallShield.

Option 1 was basically a process of manually copying the necessary files using the command **xcopy** and then doing the rest of the configuration either through batch files, your favorite scripting language, or manually.

Option 2 involved doing something similar to option 1, except that you had fewer decisions to make as to which files were going to your server and how they would get there. You could use the FrontPage extensions or a file share, and you could choose to just use the files needed to run the application or all the files in the project or folder in which your application was residing. You still had to configure similarly to option 1.

Option 3 enabled you to create an MSI and configure pretty much everything for your Web application, but it didn't give you lots of control over IIS settings, SQL databases, and other things that most Web applications use. And on top of that, it wasn't as easy and reliable as one would like.

Look at Figures 4-1, 4-2, and 4-3 for the three most common methods to deploy a Web application in Visual Studio 2003. Figure 4-1 shows the first option listed.



```
Command Prompt
C:\>xcopy c:\inetpub\wwwroot\PlanMyNight.Web\ \\testserver\PlanMyNight.Web /E /R /O /H /I
```

FIGURE 4-1 The XCOPY deployment screen

Figure 4-2 illustrates option 2, the Copy Project method.

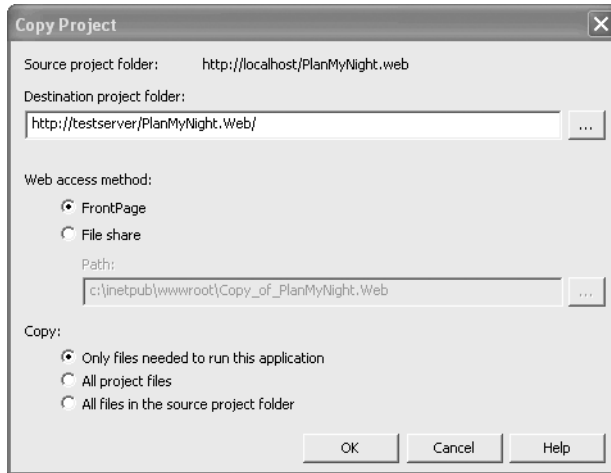


FIGURE 4-2 The Copy Project dialog box

And finally, option 3, the creation of a Web Setup Project, is shown in Figure 4-3.

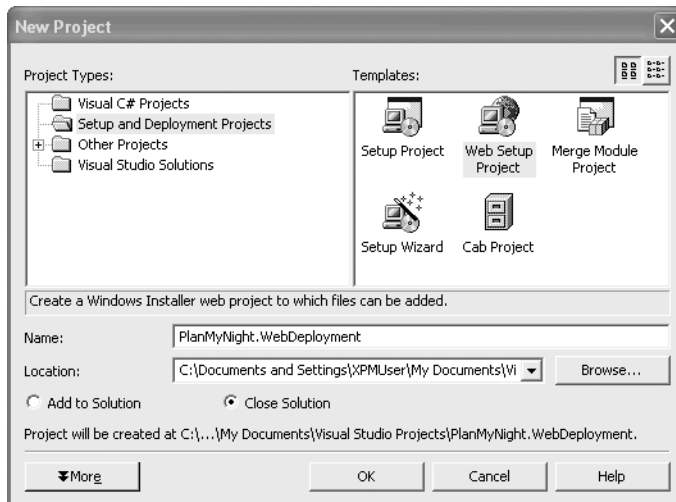


FIGURE 4-3 The New Project dialog box, showing the Web Setup Project option

In Visual Studio 2005 and 2008, new notions regarding Web applications started to emerge. Starting in the Visual Studio 2005 era, a Web application could be created in two different

ways. It could be created as a Web site directly in the file system—on disk and without a project file—or as a Web Application Project, like many other normal project types.



Note The Web Application Project became part of Visual Studio 2005 as an add-on a few months after the product shipped. People requested that Microsoft have both options: a Web site on disk and a Web Application Project similar to the proven project template that had shipped in Visual Studio 2003. Those projects are easier to manage in an enterprise environment and can be easily integrated with MSBuild.

The Web Deployment Project was created as an add-on in Visual Studio 2005 and 2008. The feature set is pretty much the same in both versions except that the Visual Studio 2008 version had many bug fixes and improvements.

The Web Deployment Project doesn't alter the Web site or the Web Application Project. Instead, it takes one as input and creates an entirely different project. In fact, it never touches the original source code, but rather it creates a new project with the necessary files based on the configuration options you selected. The Web Deployment Project is only a project file and nothing else. With a few dialogs, it enables you, the developer, to specify how and where you want to deploy your application.

At build time, it then mainly uses two utilities to turn those options into a Web site matching your selections. Those two command-line tools are called `aspnet_compiler` and `aspnet_merge`. In a nutshell, the first one compiles your projects composing your Web application and the other one merges their output and copies the files either to a folder or to a virtual folder in IIS.

You can have one deployment configuration per regular configuration (debug and release) and any custom configuration you might create. Figure 4-4 shows the beginning of the Web Deployment Project creation.

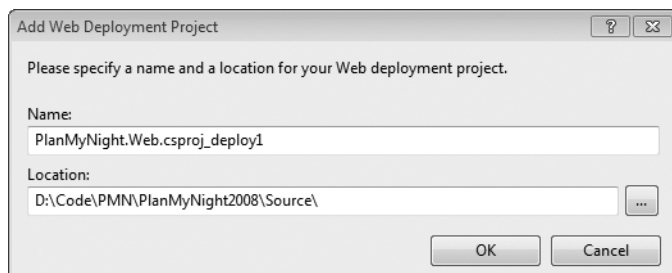


FIGURE 4-4 Add Web Deployment Project dialog

Now let's look in Visual Studio 2008 at the Web Deployment Project properties. Figure 4-5 gives you an overall sense of what it can help you with.

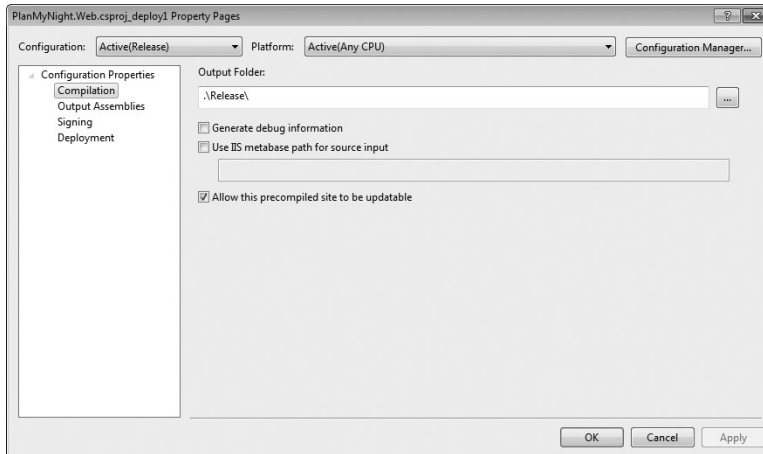


FIGURE 4-5 Web Deployment Project property page

Because it is a normal project file with MSBuild directives, you can modify “pre” and “post” build steps as well as run additional scripts. To do that, you have to edit the project file and modify the appropriate MSBuild targets. Finally, look at Figure 4-6 for a fragment for the PlanMyNight Web Deployment Project project file in Visual Studio 2008.

```

1 |<!--
2 | Microsoft Visual Studio 2008 Web Deployment Project
3 | http://go.microsoft.com/fwlink/?LinkID=104956
4 |
5 | -->
6 | <Project ToolsVersion="3.5" DefaultTargets="Build" xmlns="http://schemas.microsoft.cc
7 | <PropertyGroup>
8 |   <Configuration Condition="'$(Configuration)' == ''">Debug</Configuration>
9 |   <Platform Condition="'$(Platform)' == ''">AnyCPU</Platform>
10 |   <ProductVersion>9.0.21022</ProductVersion>
11 |   <SchemaVersion>2.0</SchemaVersion>
12 |   <ProjectGuid>{00000000-0000-0000-0000-000000000000}</ProjectGuid>
13 |   <SourceWebPhysicalPath>.\PlanMyNight.Web</SourceWebPhysicalPath>
14 |   <SourceWebProject>{74235ED3-1BD7-494C-B8E2-A7C37DD66C12}|PlanMyNight.Web\PlanMyNi
15 |   <SourceWebVirtualPath>PlanMyNight.Web.csproj</SourceWebVirtualPath>
16 |   <TargetFrameworkVersion>v3.5</TargetFrameworkVersion>
17 | </PropertyGroup>
18 | <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
19 |   <DebugSymbols>true</DebugSymbols>
20 |   <OutputPath>.\Debug</OutputPath>
21 |   <EnableUpdateable>true</EnableUpdateable>
22 |   <UseMerge>true</UseMerge>
23 |   <SingleAssemblyName>PlanMyNight.Web.csproj_deploy1</SingleAssemblyName>
24 | </PropertyGroup>
25 | <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
26 |   <DebugSymbols>false</DebugSymbols>
27 |   <OutputPath>.\Release</OutputPath>
28 |   <EnableUpdateable>true</EnableUpdateable>
29 |   <UseMerge>true</UseMerge>
30 |   <SingleAssemblyName>PlanMyNight.Web.csproj_deploy1</SingleAssemblyName>
31 | </PropertyGroup>
32 | </ItemGroup>
33 | </ItemGroup>
34 | <Import Project="$(MSBuildExtensionsPath)\Microsoft\WebDeployment\v9.0\Microsoft.We
35 | <!-- To modify your build process, add your task inside one of the targets below an
36 |     Other similar extension points exist. see Microsoft.WebDeployment.targets.

```

FIGURE 4-6 Web Deployment package project file source

It might not be apparent, but even though the Web Deployment Project was a huge improvement over what was in Visual Studio 2003, it was still not easy enough, not inclusive of all needs, and not powerful enough. It was a good step. Here is where the new Web Deployment Packages, the new Web Application deployment project, and Visual Studio 2010 come to the rescue.

What Are Web Deployment Packages?

A Web deployment package is a compressed (.zip) file that contains all the necessary files and metadata to set up your application in IIS, copy the application files to their destination, configure the different applications in IIS, and set up related resources such as localization resources, certificates, registry settings, installing assemblies in the GAC, and, finally, setting up databases.

Those packages are then installed on the destination server using the msdeploy tool. You can read the latest news about the msdeploy tool here: <http://blogs.iis.net/msdeploy/default.aspx>. A good analogy to this type of solution is what MSIs and the Windows installer are for the client desktop.



Note InstallShield and the Wix Toolset are two other great solutions. They both have pros and cons, and they both can work with other types of applications. The msdeploy tool is simply more specialized and therefore a bit easier to work with. As of the writing of this chapter (May 2010), Wix 3.5 is not out yet and a change in plans potentially is taking the custom action for IIS 7 out of that release. If it does ship, it will be no earlier than July 2010. (To follow what is going on in the Wix world, read Rob Mensching's blog: <http://robmensching.com/blog/>.) Therefore, the Wix toolset is not an option I would recommend just yet, unless you are installing on IIS 6. InstallShield is out already, supports Visual Studio 2010, and can definitely create good packages to deploy Web applications; however, it isn't free.

In Visual Studio 2010, you create your packages by creating settings in the Package/Publish Web tab of the project properties page. Those settings allow you to specify what you put into a deployment package. Let's see this in action.

1. Make sure you have the PlanMyNight solution opened. Then right-click on the PlanMyNight.Web project, and select Package/Publish Settings. Let's look at Figure 4-7 to make sure you are at the right place.

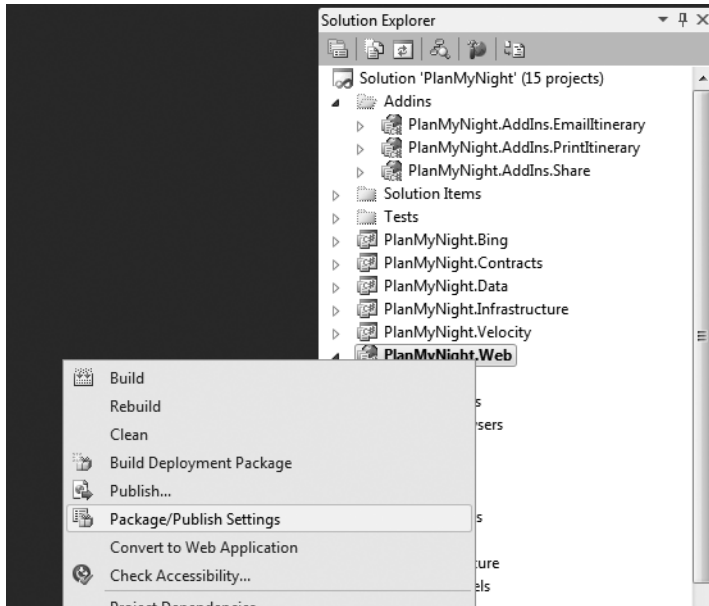


FIGURE 4-7 The Package/Publish Settings option

- Now let's take a look at Figure 4-8 to look at the content of that tab and select the same settings for the PlanMyNight.Web application you have opened.

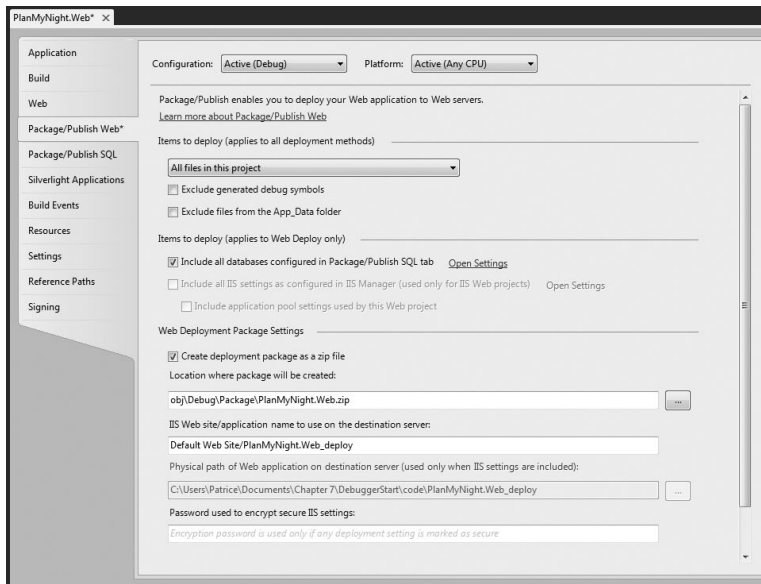


FIGURE 4-8 The Package/Publish Web tab

3. Similarly, in the Package/Publish SQL tab you can find the settings related to creating a package for your database. Click on the Package/Publish SQL tab, or click on the Open Settings link in the Package/Publish Web tab.
4. Click the Import From Web.Config button, and then copy the same string from the source connection string to the destination connection string. Make sure your SQL settings look like Figure 4-9. Save the file.

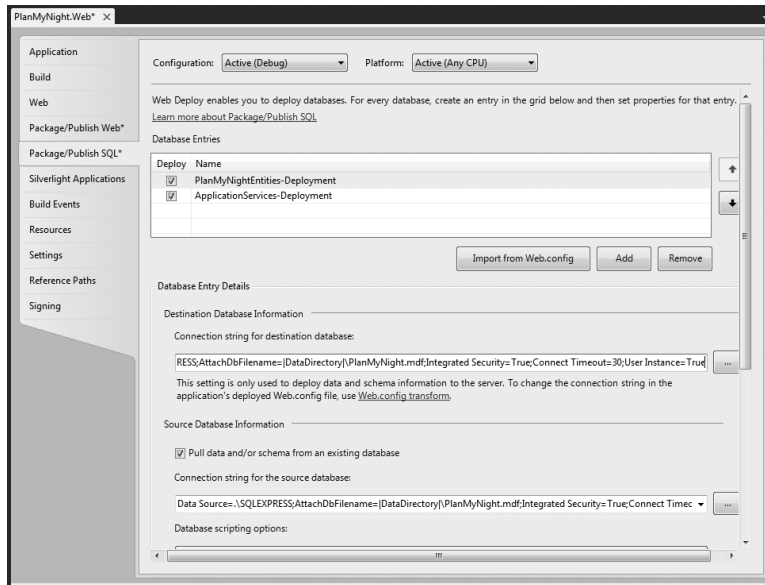


FIGURE 4-9 The Package/Publish SQL tab

5. Now it's time to build the package. Right-click on the project name, and select Build Deployment Package. The build process for the application will start. When it is completed, the output will become the input to the package's creation. If all goes well, the package should be created at the location you specified in the package settings.
6. The package folder should contain the package .zip file, a command file that invokes Web Deploy to make it easier to install the package from the command line, a SetParameters.xml file containing all the parameters passed to Web Deploy, and a SourceManifest.xml containing the parameters Visual Studio 2010 used to create the package.

You have just created the package using Visual Studio 2010, but packages can also be created by using MSBuild at the command line or MSBuild using Windows PowerShell or TFSBuild. Now if you have access to the server, you can take this package and the command file and deploy it. But if the server is a shared hosting company or another datacenter, you have to publish it differently. And here is where One-Click Publish can help.

One-Click Publish

One-Click Publish is a Visual Studio 2010 tool that allows you to deploy a package based on various technologies. Most importantly, it can use Web Deploy to publish the package you created in the previous steps. It can also use FTP and FrontPage Extension. Let's see what the Publish Profile looks like. To do so, right-click on the PlanMyNight.Web Web project and select Publish. You should see the dialog shown in Figure 4-10, which displays the Publish Profile information.

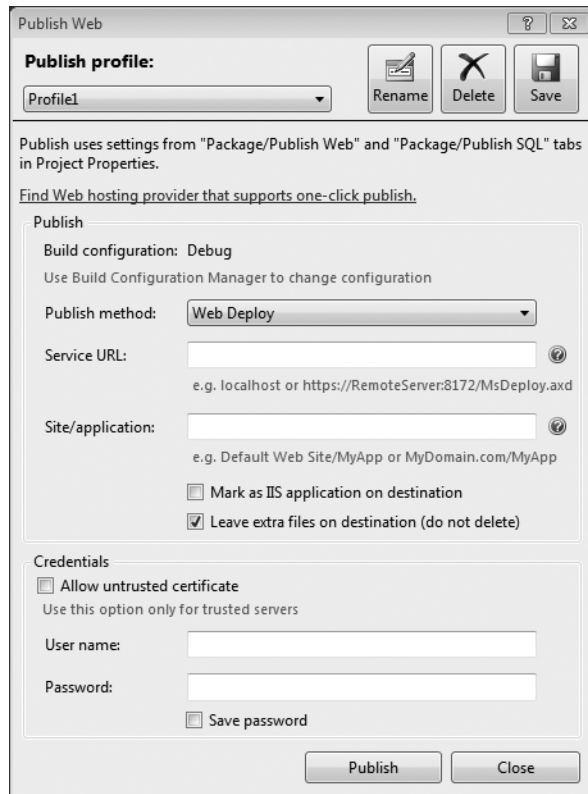


FIGURE 4-10 Publish Profile dialog

If you publish to a shared hosting company and have Web Deploy installed on its servers, the hosting company will give you the service URL to use—something like `https:<sharedhost>:8172/MsDeploy.axd`. Or you put the name of the server if it was on your intranet. Finally, the site/application corresponds to the `IISWebSiteName/IISWebApplication`. The bottom part of the dialog shown in Figure 4-10 is used to enter credentials, if needed, for your Web hosting company or your intranet.



Note To try it, you just have to delete your database, delete your IIS Web application and settings, and then publish using this tool.

You can do a lot more than use what the UI gives you access to. Integrating your deployment in your TFSBuild or MSBuild allows you to change many things at different steps of the process.

The options offered out of the box have been discussed here in this chapter. Of course, Visual Studio 2010 has many more ways to deploy other types of applications.

Summary

In this chapter, we reviewed many ways to deploy Web applications. You saw how to do it in the previous three versions of Visual Studio. You learned that you have a lot more control of the process and that it's a lot easier in Visual Studio 2010. Web Deploy is a new technology that allows Web developers to become really proficient at preparing and deploying complex installations and databases in an easy and extensible manner. As you witnessed in these short examples, you can deploy your software in a confident and timely manner. It is a great improvement over the XCOPY deployment most of us had to work with in the early days of ASP.NET.

Finally, remember to keep abreast of all the changes in the msdeploy technologies by following the teams' blog at <http://blogs.iis.net/msdeploy/default.aspx>. The Wix Toolset is also well integrated into Visual Studio and is therefore another excellent alternative.

Part II

Moving from Microsoft Visual Studio 2005 to Visual Studio 2010

Authors Patrice Pelland, Ken Haines, and Pascal Pare

In this part:

From 2005 to 2010: Business Logic and Data (Pascal)	117
From 2005 to 2010: Designing the Look and Feel (Ken)	153
From 2005 to 2010: Debugging an Application (Patrice)	195

Chapter 5

From 2005 to 2010: Business Logic and Data

After reading this chapter, you will be able to

- Use the Entity Framework (EF) to build a data access layer using an existing database or with the Model-First approach
- Generate entity types from the Entity Data Model (EDM) Designer using the ADO.NET Entity Framework POCO templates
- Get data from Web services
- Learn about data caching using the Microsoft Windows Server AppFabric (formerly known by the codename "Velocity")

Application Architecture

The Plan My Night (PMN) application allows the user to manage his itinerary activities and share them with others. The data is stored in a Microsoft SQL Server database. Activities are gathered from searches to the Bing Maps Web services.

Let's have a look at the high-level block model of the data model for the application, which is shown in Figure 5-1.

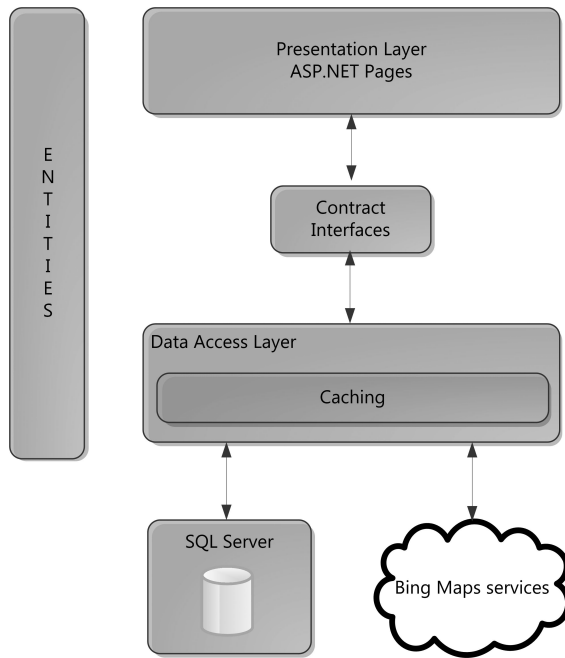


FIGURE 5-1 Plan My Night application architecture diagram

Defining contracts and entity classes that are cleared of any persistence-related code constraints allows us to put them in an assembly that has no persistence-aware code. This approach ensures a clean separation between the Presentation and Data layers.

Let's identify the contract interfaces for the major components of the PMN application:

- *ItinerariesRepository* is the interface to our data store (a Microsoft SQL Server database).
- *IActivitiesRepository* allows us to search for activities (using Bing Maps Web services).
- *ICachingProvider* provides us with our data-caching interface (ASP.NET caching or Windows Server AppFabric caching).



Note This is not an exhaustive list of the contracts implemented in the PMN application.

PMN stores the user's itineraries into an SQL database. Other users will be able to comment and rate each other's itineraries. Figure 5-2 shows the tables used by the PMN application.

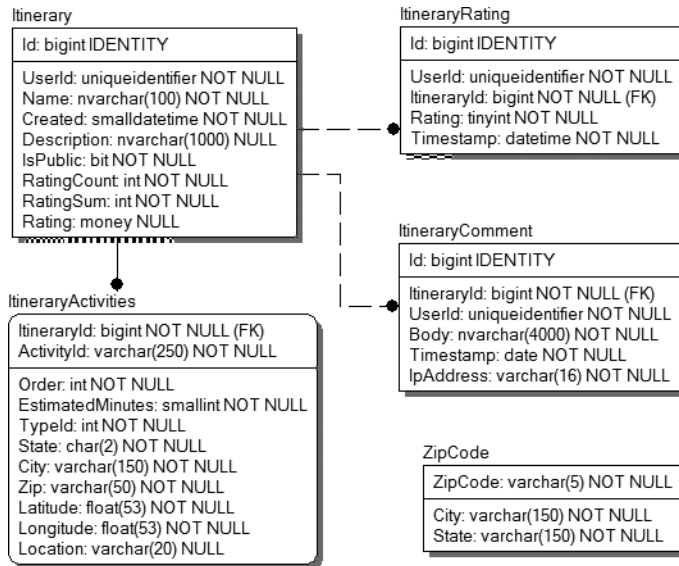


FIGURE 5-2 PlanMyNight database schema



Important The Plan My Night application uses the ASP.NET Membership feature to provide secure credential storage for the users. The user store tables are not shown in Figure 5-2. You can learn more about this feature on MSDN: [ASP.NET 4 - Introduction to Membership \(http://msdn.microsoft.com/en-us/library/yh26yfzy\(VS.100\).aspx\)](http://msdn.microsoft.com/en-us/library/yh26yfzy(VS.100).aspx).



Note The ZipCode table is used as a reference repository to provide a list of available Zip Codes and cities so that you can provide autocomplete functionality when the user is entering a search query in the application.

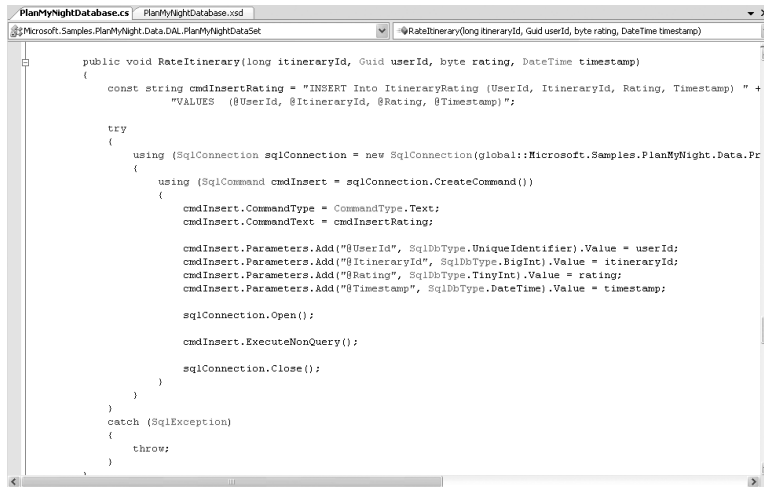
Plan My Night Data in Microsoft Visual Studio 2005

It would be straightforward to create the Plan My Night application in Visual Studio 2005 because it offers all the required tools to help you to code the application. However, some of the technologies used back then required you to write a lot more code to achieve the same goals.

Let's take a look at how you could create the required data layer in Visual Studio 2005. One approach would have been to write the data layer using ADO.NET *DataSet* or *DataReader*

directly. (See Figure 5-3.) This solution offers you great flexibility because you have complete control over access to the database. On the other hand, it also has some drawbacks:

- You need to know the SQL syntax.
- All queries are specialized. A change in requirement or in the tables will force you to update the queries affected by these changes.
- You need to map the properties of your entity classes using the column name, which is a tedious and error-prone process.
- You have to manage the relations between tables yourself.



```

PlanMyNightDatabase.cs | PlanMyNightDatabase.csod
@Microsoft.Samples.PlanMyNight.Data.DAL.PlanMyNightDataSet
@RateItinerary(long itineraryId, Guid userId, byte rating, DateTime timestamp)

public void RateItinerary(long itineraryId, Guid userId, byte rating, DateTime timestamp)
{
    const string cmdInsertRating = "INSERT INTO ItineraryRating (UserId, ItineraryId, Rating, Timestamp) " +
        "VALUES (@UserId, @ItineraryId, @Rating, @Timestamp)";

    try
    {
        using (SqlConnection sqlConnection = new SqlConnection(global::Microsoft.Samples.PlanMyNight.Data.Pr
        {
            using (SqlCommand cmdInsert = sqlConnection.CreateCommand())
            {
                cmdInsert.CommandType = CommandType.Text;
                cmdInsert.CommandText = cmdInsertRating;

                cmdInsert.Parameters.Add("@UserId", SqlDbType.UniqueIdentifier).Value = userId;
                cmdInsert.Parameters.Add("@ItineraryId", SqlDbType.BigInt).Value = itineraryId;
                cmdInsert.Parameters.Add("@Rating", SqlDbType.TinyInt).Value = rating;
                cmdInsert.Parameters.Add("@Timestamp", SqlDbType.DateTime).Value = timestamp;

                sqlConnection.Open();
                cmdInsert.ExecuteNonQuery();
                sqlConnection.Close();
            }
        }
    }
    catch (SqlException)
    {
        throw;
    }
}

```

FIGURE 5-3 ADO.NET Insert query

Another approach would be to use the *DataSet* designer available in Visual Studio 2005. Starting from a database with the PMN tables, you could use the TableAdapter Configuration Wizard to import the database tables as shown in Figure 5-4. The generated code offers you a typed *DataSet*. One of the benefits is type checking at design time, which gives you the advantage of statement completion. There are still some pain points with this approach:

- You still need to know the SQL syntax, although you have access to the query builder directly from the *DataSet* designer.
- You still need to write specialized SQL queries to match each of the requirements of your data contracts.
- You have no control of the generated classes. For example, changing the *DataSet* to add or remove a query for a table will rebuild the generated *TableAdapter* classes and might change the index used for a query. This makes it difficult to write predictable code using these generated items.

- The generated classes associated with the tables are persistence aware, so you will have to create another set of simple entities and copy the data from one to the other. This means more processing and memory usage.

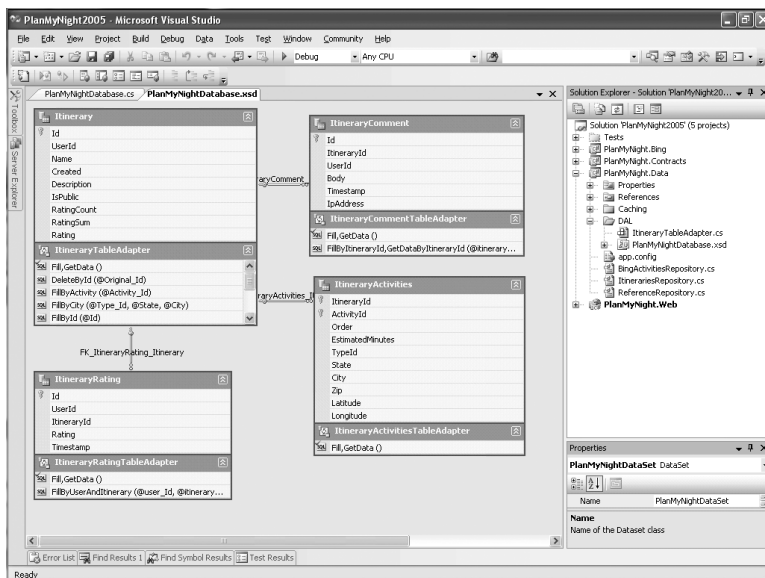


FIGURE 5-4 DataSet designer in Visual Studio 2005

In the next sections of this chapter, you'll explore some of the new features of Visual Studio 2010 that will help you create the PMN data layer with less code, give you more control of the generated code, and allow you to easily maintain and expand it.

Data with the Entity Framework in Visual Studio 2010

The ADO.NET Entity Framework (EF) allows you to easily create the data access layer for an application by abstracting the data from the database and exposing a model closer to the business requirements of the application. The EF has been considerably enhanced in the .NET Framework 4 release.

See Also The MSDN Data Developer Center offers a lot of resources about the [ADO.NET Entity Framework](http://msdn.microsoft.com/en-us/data/aa937723.aspx) (<http://msdn.microsoft.com/en-us/data/aa937723.aspx>) in .NET 4.

You'll use the PlanMyNight project as an example of how to build an application using some of the features of the EF. The next two sections demonstrate two different approaches to generating the data model of PMN. In the first one, you let the EF generate the Entity Data Model (EDM) from an existing database. In the second part, you use a Model First approach, where you first create the entities from the EF designer and generate the Data Definition Language (DDL) scripts to create a database that can store your EDM.

EF: Importing an Existing Database

You'll start with an existing solution that already defines the main projects of the PMN application. If you installed the companion content at the default location, you'll find the solution at this location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\Code\ExistingDatabase. Double-click the PlanMyNight.sln file.

This solution includes all the projects in the following list, as shown in Figure 5-5:

- PlanMyNight.Data: Application data layer
- PlanMyNight.Contracts: Entities and contracts
- PlanMyNight.Bing: Bing Maps services
- PlanMyNight.Web: Presentation layer
- PlanMyNight.AppFabricCaching: AppFabric caching

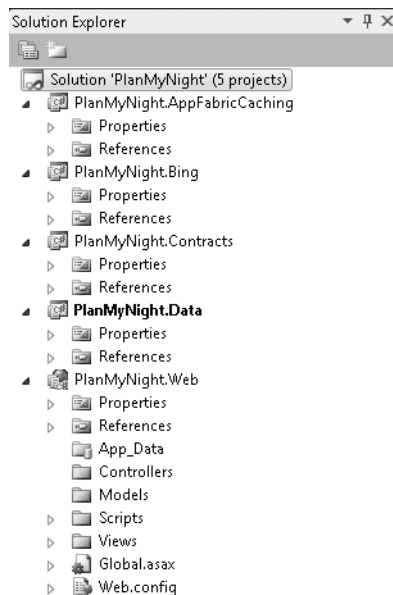


FIGURE 5-5 PlanMyNight solution

The EF allows you to easily import an existing database. Let's walk through this process.

The first step is to add an EDM to the PlanMyNight.Data project. Right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item, and change its name to **PlanMyNight.edmx**, as shown in Figure 5-6.

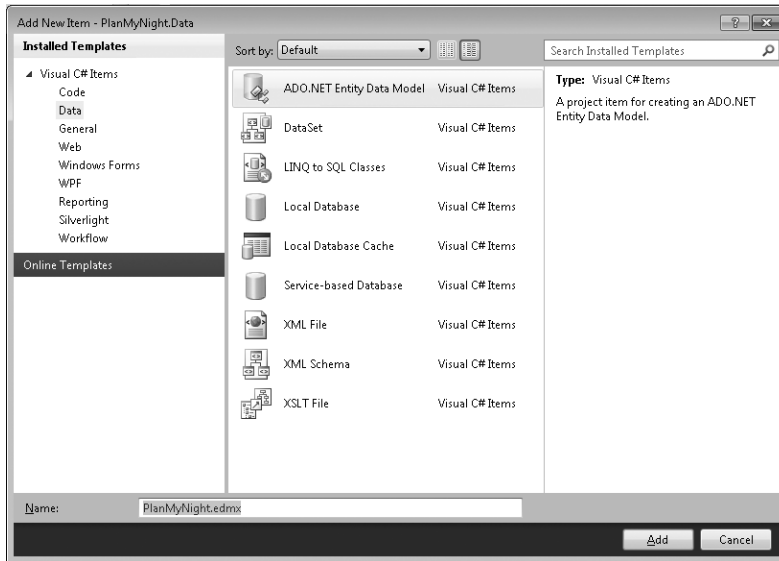


FIGURE 5-6 Add New Item dialog with ADO.NET Entity Data Model selected

The first dialog of the Entity Data Model Wizard allows you to choose the model content. You'll generate the model from an existing database. Select Generate From Database and then click Next.

You need to connect to an existing database file. Click New Connection. Select Microsoft SQL Server Database File from the Choose Data Source dialog, and click Continue. Select the %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\ExistingDatabase\PlanMyNight.Web\App_Data\PlanMyNight.mdf file. (See Figure 5-7.)

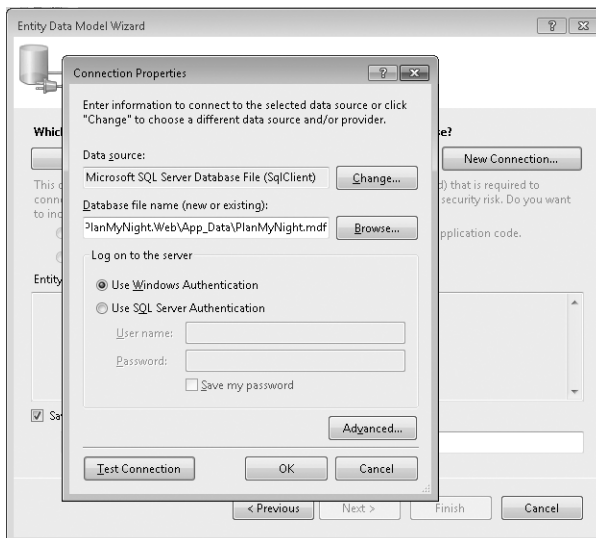


FIGURE 5-7 EDM Wizard database connection

Leave the other fields in the form as is for now and click Next.



Note You'll get a warning stating that the local data file is not in the current project. Click No to close the dialog because you do not want to copy the database file to the current project.

From the Choose Your Database Objects dialog, select the Itinerary, ItineraryActivities, ItineraryComment, ItineraryRating, and ZipCode tables and the UserProfile view. Select the *RetrieveItinerariesWithinArea* stored procedure. Change the Model Namespace value to **Entities** as shown in Figure 5-8.

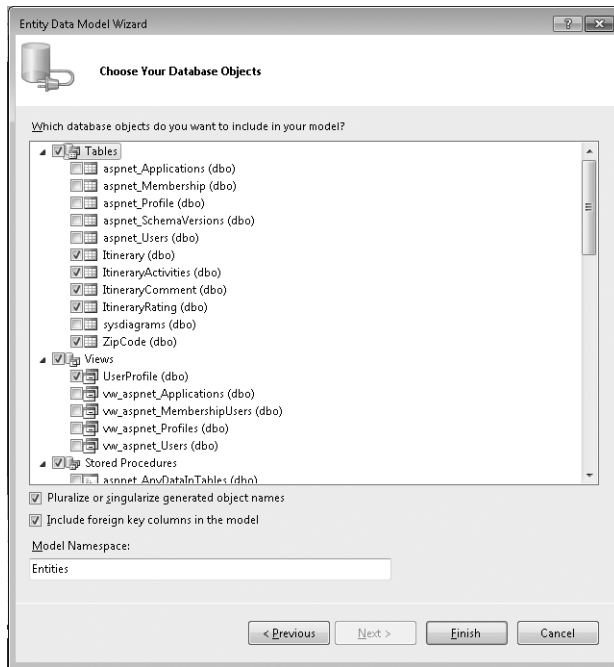


FIGURE 5-8 EDM Wizard, Choose Your Database Objects page

Click Finish to generate your EDM.

Fixing the Generated Data Model

You now have a model representing a set of entities matching your database. The wizard has generated all the navigation properties associated with the foreign keys from the database.

The PMN application requires only the navigation property *ItineraryActivities* from the *Itinerary* table, so you can go ahead and delete all the other navigation properties. You'll also need to rename the *ItineraryActivities* navigation property to **Activities**. Refer to Figure 5-9 for the updated model.

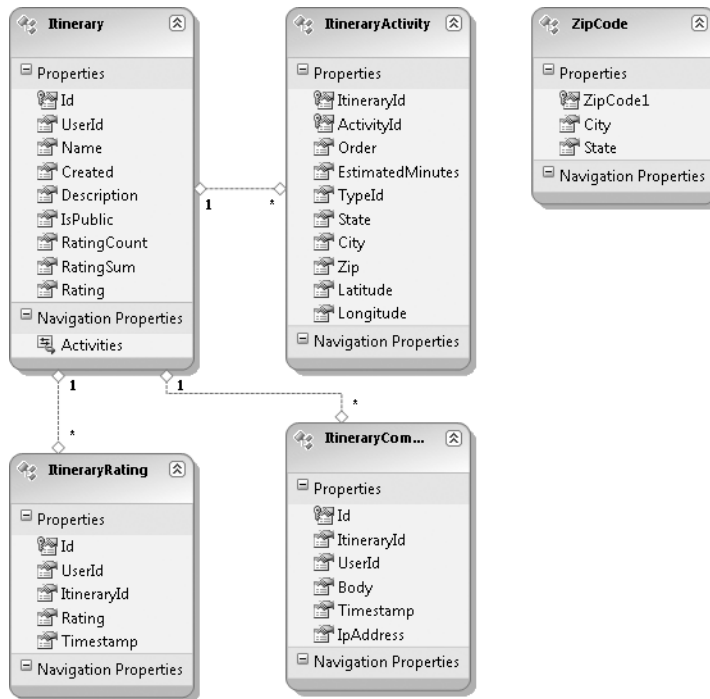


FIGURE 5-9 Model imported from the PlanMyNight database

Notice that one of the properties of the *ZipCode* entity has been generated with the name *ZipCode1* because the table itself is already named *ZipCode* and the name has to be unique. Let's fix the property name by double-clicking it. Change the name to **Code**, as shown in Figure 5-10.

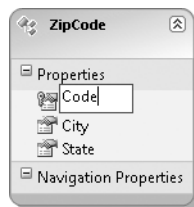


FIGURE 5-10 *ZipCode* entity

Build the solution by pressing Ctrl+Shift+B. When looking at the output window, you'll notice two messages from the generated EDM. You can discard the first one because the Location column is not required in PMN. The second message reads as follows:

The table/view 'dbo.UserProfile' does not have a primary key defined and no valid primary key could be inferred. This table/view has been excluded. To use the entity, you will need to review your schema, add the correct keys, and uncomment it.

When looking at the UserProfile view, you'll notice it does not explicitly define a primary key even though the UserName column is unique.

You need to modify the EDM manually to fix the UserProfile view mapping so that you can access the UserProfile data from the application.

From the project explorer, right-click the PlanMyNight.edmx file and then select Open With. Choose XML (Text) Editor from the Open With dialog as shown in Figure 5-11. Click OK to open the XML file associated with your model.

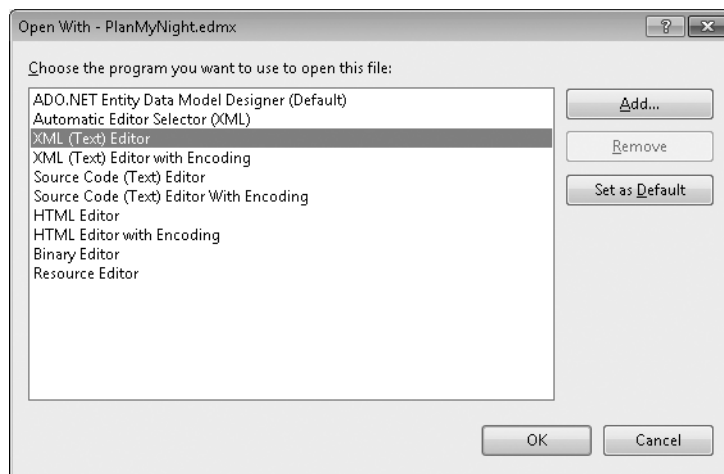


FIGURE 5-11 Open PlanMyNight.edmx in the XML Editor



Note You'll get a warning stating that the PlanMyNight.edmx file is already open. Click Yes to close it.

The generated code was commented out by the code-generation tool because there was no primary key defined. To be able to use the UserProfile view from the designer, you need to uncomment the UserProfile entity type and add the Key tag to it. Search for UserProfile in the file. Uncomment the entity type, add a Key tag and set its name to **UserName**, and make the *UserName* property not nullable. Refer to Listing 5-1 to see the updated entity type.

LISTING 5-1 UserProfile Entity Type XML Definition

```
<EntityType Name="UserProfile">
  <Key>
    <PropertyRef Name="UserName"/>
  </Key>
  <Property Name="UserName" Type="uniqueidentifier" Nullable="false" />
  <Property Name="FullName" Type="varchar" MaxLength="500" />
  <Property Name="City" Type="varchar" MaxLength="500" />
  <Property Name="State" Type="varchar" MaxLength="500" />
  <Property Name="PreferredActivityTypeId" Type="int" />
</EntityType>
```

If you close the XML file and try to open the EDM Designer, you'll get the following error message in the designer: "The Entity Data Model Designer is unable to display the file you requested. You can edit the model using the XML Editor."

There is a warning in the Error List pane that can give you a little more insight into what this error is all about:

Error 11002: Entity type 'UserProfile' has no entity set.

You need to define an entity set for the UserProfile type so that it can map the entity type to the store schema. Open the PlanMyNight.edmx file in the XML editor so that you can define an entity set for UserProfile. At the top of the file, just above the Itinerary entity set, add the XML code shown in Listing 5-2.

LISTING 5-2 UserProfile EntitySet XML Definition

```
<EntitySet Name="UserProfile" EntityType="Entities.Store.UserProfile"
  store:Type="Views" store:Schema="dbo" store:Name="UserProfile">
  <DefiningQuery>
    SELECT
      [UserProfile].[UserName] AS [UserName],
      [UserProfile].[FullName] AS [FullName],
      [UserProfile].[City] AS [City],
      [UserProfile].[State] AS [State],
      [UserProfile].[PreferredActivityTypeId] as [PreferredActivityTypeId]
    FROM [dbo].[UserProfile] AS [UserProfile]
  </DefiningQuery>
</EntitySet>
```

Save the EDM XML file, and reopen the EDM Designer. Figure 5-12 shows the UserProfile view in the Entities.Store section of the Model Browser.



Tip You can open the Model Browser from the View menu by clicking Other Windows and selecting the Entity Data Model Browser item.

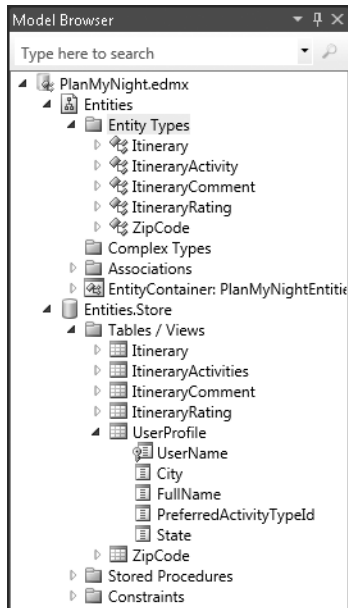


FIGURE 5-12 Model Browser with the UserProfile view

Now that the view is available in the store metadata, you add the UserProfile entity and map it to the UserProfile view. Right-click in the background of the EDM Designer, select Add, and then choose Entity. You'll see the dialog shown in Figure 5-13.

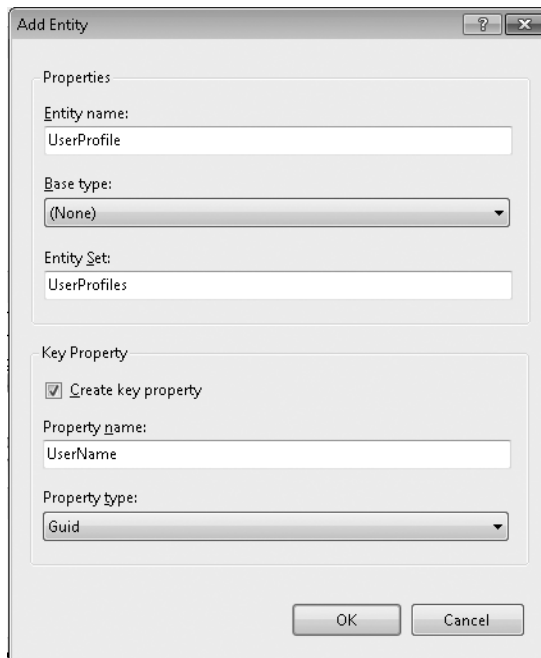


FIGURE 5-13 Add Entity dialog

Complete the dialog as shown in Figure 5-13, and click OK to generate the entity.

You need to add the remaining properties: *City*, *State*, and *PreferredActivityTypeId*. To do so, right-click the *UserProfile* entity, select *Add*, and then select *Scalar Property*. After the property is added, set the *Type*, *Max Length*, and *Unicode* field values. Table 5-1 shows the expected values for each of the fields.

TABLE 5-1 UserProfile Entity Properties

Name	Type	Max Length	Unicode
<i>FullName</i>	<i>String</i>	500	False
<i>City</i>	<i>String</i>	500	False
<i>State</i>	<i>String</i>	500	False
<i>PreferredActivityTypeId</i>	<i>Int32</i>	NA	NA

Now that you have created the *UserProfile* entity, you need to map it to the *UserProfile* view. Right-click the *UserProfile* entity, and select *Table Mapping* as shown in Figure 5-14.

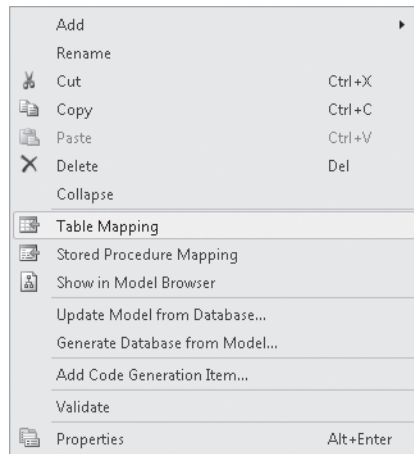


FIGURE 5-14 Table Mapping menu item

Then select the *UserProfile* view from the drop-down box as shown in Figure 5-15. Ensure that all the columns are correctly mapped to the entity properties. The *UserProfile* view of our store is now accessible from the code through the *UserProfile* entity.

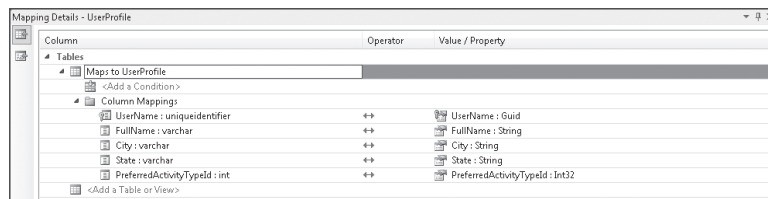


FIGURE 5-15 UserProfile mapping details

Stored Procedure and Function Imports

The Entity Data Model Wizard has created an entry in the storage model for the *RetrievalItinerariesWithinArea* stored procedure you selected in the last step of the wizard. You need to create a corresponding entry to the conceptual model by adding a Function Import entry.

From the Model Browser, open the Stored Procedures folder in the Entities.Store section. Right-click *RetrievalItineraryWithinArea*, and then select Add Function Import. The Add Function Import dialog appears as shown in Figure 5-16. Specify the return type by selecting Entities and then select the Itinerary item from the drop-down box. Click OK.

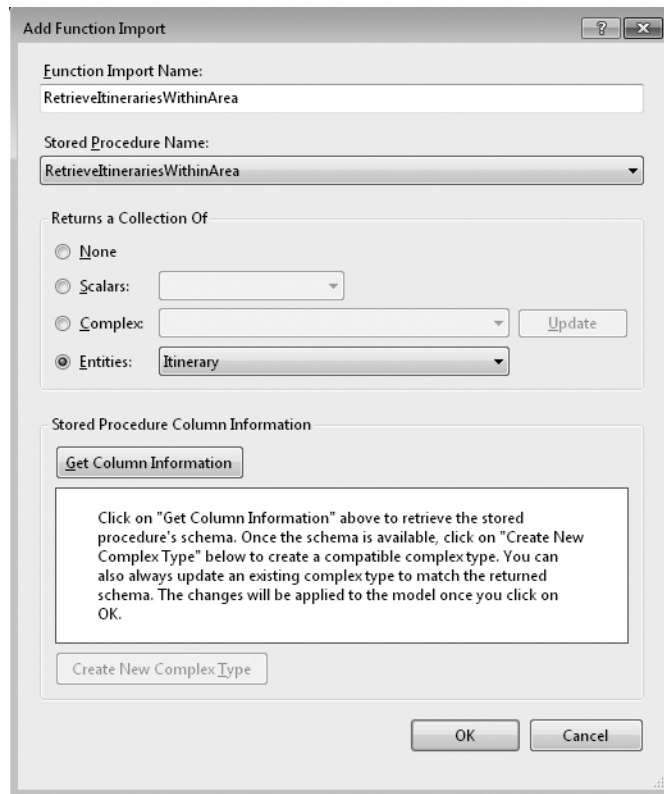


FIGURE 5-16 Add Function Import dialog

The *RetrievelTinerariesWithinArea* function import was added to the Model Browser as shown in Figure 5-17.

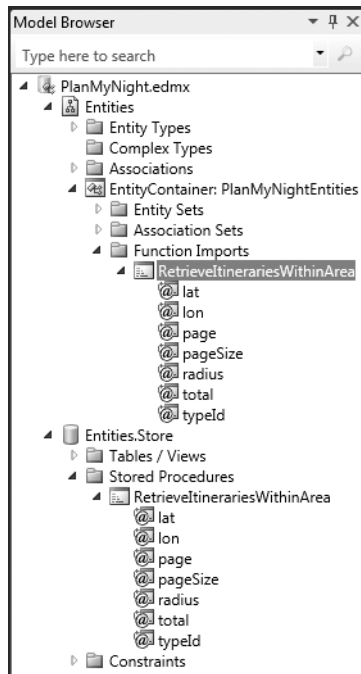


FIGURE 5-17 Function Imports in the Model Browser

You can now validate the EDM by right-clicking on the design surface and selecting Validate. There should be no error or warning.

EF: Model First

In the prior section, you saw how to use the EF designer to generate the model by importing an existing database. The EF designer in Visual Studio 2010 also supports the ability to generate the Data Definition Language (DDL) file that will allow you to create a database based on your entity model. In this section, you'll use a new solution to learn how to generate a database script from a model.

You can start from an empty model by selecting the Empty model option from the Entity Data Model Wizard. (See Figure 5-18.)



Note To get the wizard, right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item.

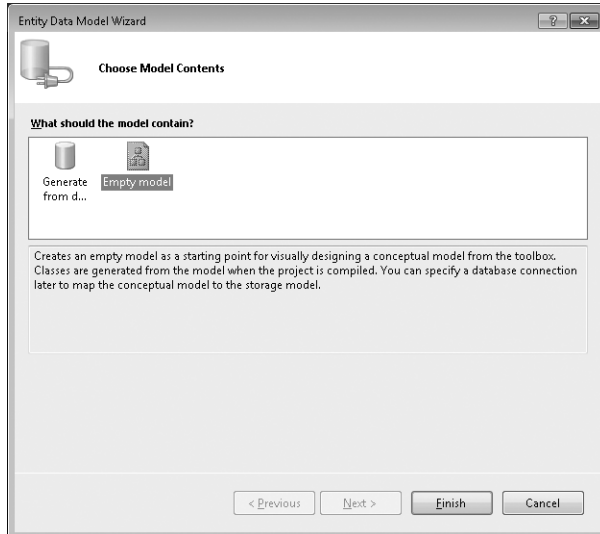


FIGURE 5-18 EDM Wizard: Empty model

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\Code\ModelFirst by double-clicking the PlanMyNight.sln file.

The PlanMyNight.Data project from this solution already contains an EDM file named PlanMyNight.edmx with some entities already created. These entities match the data schema you saw in Figure 5-2.

The Entity Model designer lets you easily add an entity to your data model. Let's add the missing ZipCode entity to the model. From the toolbox, drag an Entity item into the designer, as shown in Figure 5-19. Rename the entity as **ZipCode**. Rename the *Id* property as **Code**, and change its type to *String*.

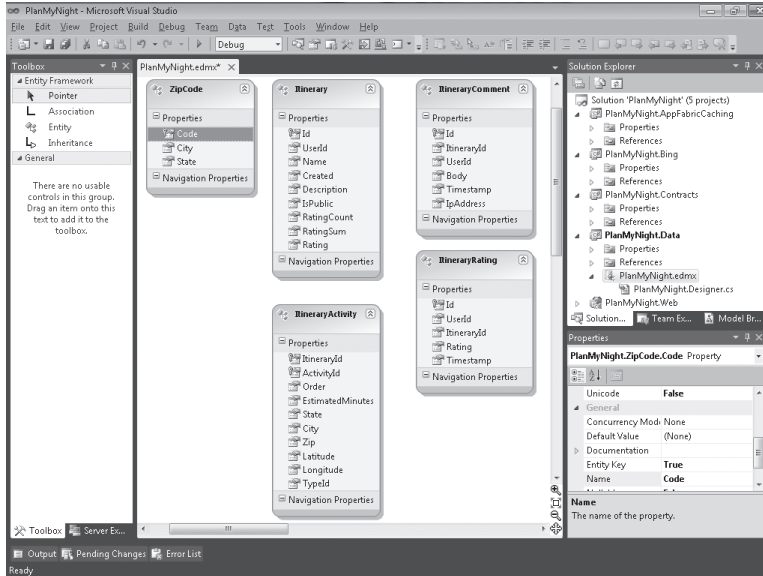


FIGURE 5-19 Entity Model designer

You need to add the *City* and *State* properties to the entity. Right-click the ZipCode entity, select Add, and then choose Scalar Property. Ensure that each property has the values shown in Table 5-2.

TABLE 5-2 ZipCode Entity Properties

Name	Type	Fixed Length	Max Length	Unicode
<i>Code</i>	<i>String</i>	False	5	False
<i>City</i>	<i>String</i>	False	150	False
<i>State</i>	<i>String</i>	False	150	False

Add the relations between the ItineraryComment and Itinerary entities. Right-click the designer background, select Add, and then choose Association. (See Figure 5-20.)

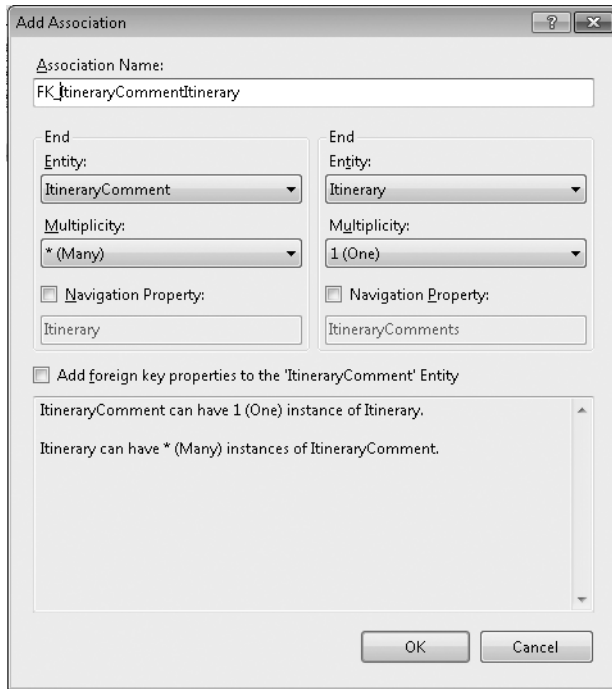


FIGURE 5-20 Add Association dialog for *FK_ItineraryCommentItinerary*

Set the association name to **FK_ItineraryCommentItinerary** and then select the entity and the multiplicity for each end, as shown in Figure 5-20. After the association is created, double-click the association line to set the Referential Constraint as shown in Figure 5-21.

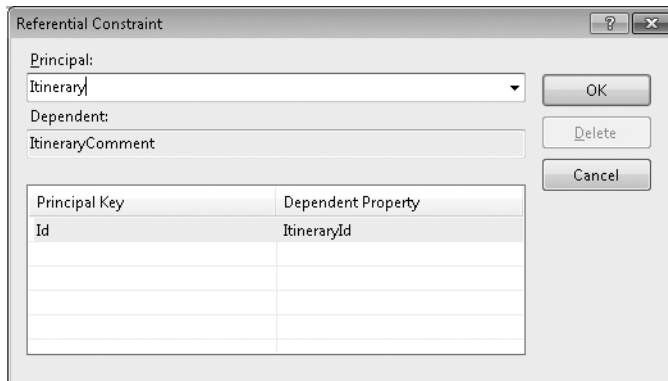


FIGURE 5-21 Association Referential Constraint dialog

Add the association between the ItineraryRating and Itinerary entities. Right-click the designer background, select Add, and then choose Association. Set the association name to **FK_ItineraryItineraryRating** and then select the entity and the multiplicity for each end as in the previous step, except set the first end to **ItineraryRating**. Double-click on the association line, and set the Referential Constraint as shown in Figure 5-21. Note that the Dependent field will read ItineraryRating instead of ItineraryComment.

Create a new association between the ItineraryActivity and Itinerary entities. For the FK_ItineraryItineraryActivity association, you also want to create a navigation property and name it **Activities**, as shown in Figure 5-22. After the association is created, set the Referential Constraint for this association by double-clicking on the association line.

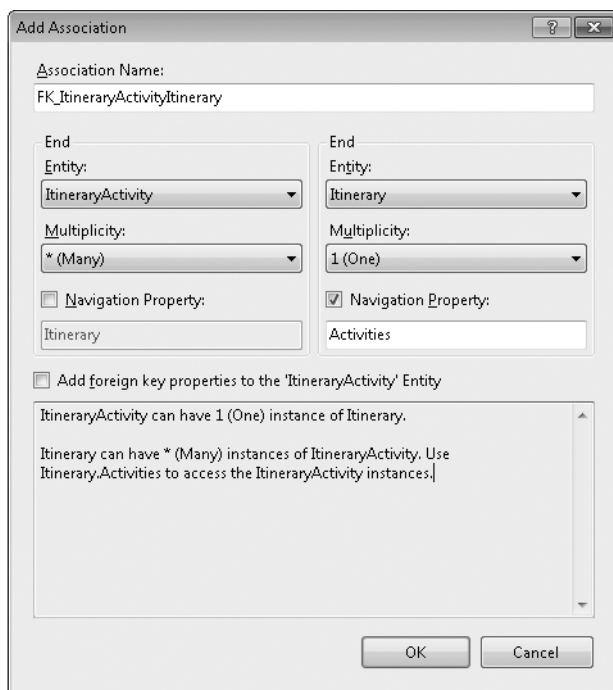


FIGURE 5-22 Add Association dialog for *FK_ItineraryActivityItinerary*

Generating the Database Script from the Model

Your data model is now completed, but there is no mapping or store associated with it. The EF designer offers the possibility of generating a database script from our model.

Right-click on the designer surface, and choose Generate Database From Model as shown in Figure 5-23.

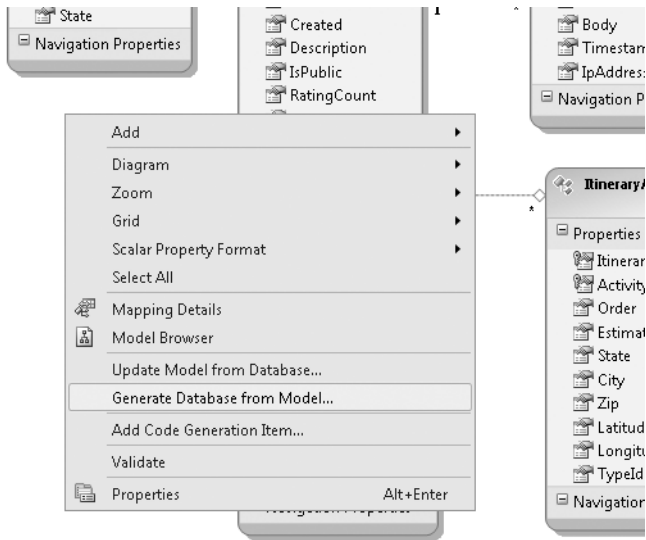


FIGURE 5-23 Generate Database From Model menu item

The Generate Database Wizard requires a data connection. The wizard uses the connection information to translate the model types to the database type and to generate a DDL script targeting this database.

Select New Connection, select Microsoft SQL Server Database File from the Choose Data Source dialog, and click Continue. Select the database file located at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\Code\ModelFirst\Data\PlanMyNight.mdf. (See Figure 5-24.)



FIGURE 5-24 Generate a script database connection

After your connection is configured, click Next to get to the final page of the wizard, as shown in Figure 5-25. When you click Finish, the generated T-SQL PlanMyNight.edmx.sql file is added to your project. The DDL script will generate the primary and foreign key constraints for your model.

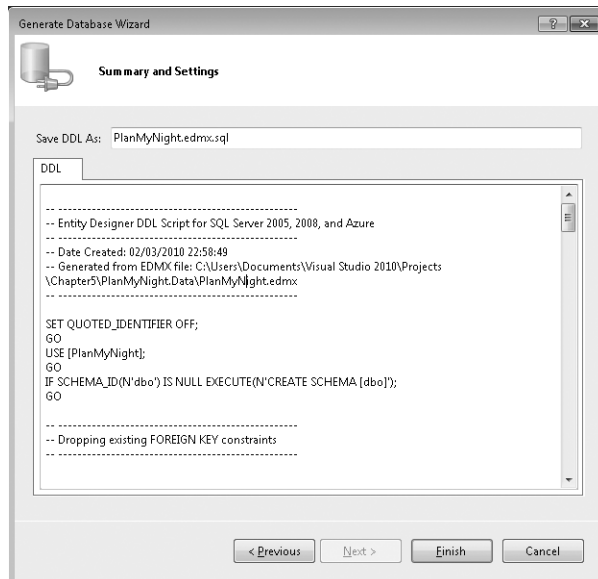


FIGURE 5-25 Generated T-SQL file

The EDM is also updated to ensure your newly created store is mapped to the entities. You can now use the generated DDL script to add the tables to the database. Also, you now have a data layer that exposes strongly typed entities that you can use in your application.



Important Generating the complete PMN database would require adding the remaining tables, stored procedures, and triggers used by the application. Instead of performing all these operations, we will go back to the solution we had at the end of the “EF: Importing an Existing Database” section.

POCO Templates

The EDM Designer uses T4 templates to generate the code for the entities. So far, we have let the designer create the entities using the default templates. You can take a look at the code generated by opening the `PlanMyNight.Designer.cs` file associated with `PlanMyNight.edmx`. The generated entities are based on the `EntityType` type and decorated with attributes to allow the EF to manage them at run time.



Note T4 stands for *Text Template Transformation Toolkit*. T4 support in Visual Studio 2010 allows you to easily create your own templates and generate any type of text file (Web, resource, or source). To learn more about the code generation in Visual Studio 2010, visit [Code Generation and Text Templates](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx) ([http://msdn.microsoft.com/en-us/library/bb126445\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx)).

The EF also supports POCO entity types. POCO classes are simple objects with no attributes or base class related to the framework. (Listing 5-3, in the next section, shows the POCO class for the `ZipCode` entity.) The EF uses the names of the types and the properties of these objects to map them to the model at run time.



Note POCO stands for *Plain-Old CLR Objects*.

ADO.NET POCO Entity Generator

Let's re-open the `%userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\Code\ExistingDatabase\PlanMyNight.sln` file.

Open the `PlanMyNight.edmx` file, right-click on the design surface, and choose `Add Code Generation Item`. This opens a dialog like the one shown in Figure 5-26, where you can select the template you want to use. Select the `ADO.NET POCO Entity Generator` template, and name it **PlanMyNight.tt**. Then click the `Add` button.



Note You might get a security warning about running this text template. Click OK to close the dialog because the source for this template is trusted.

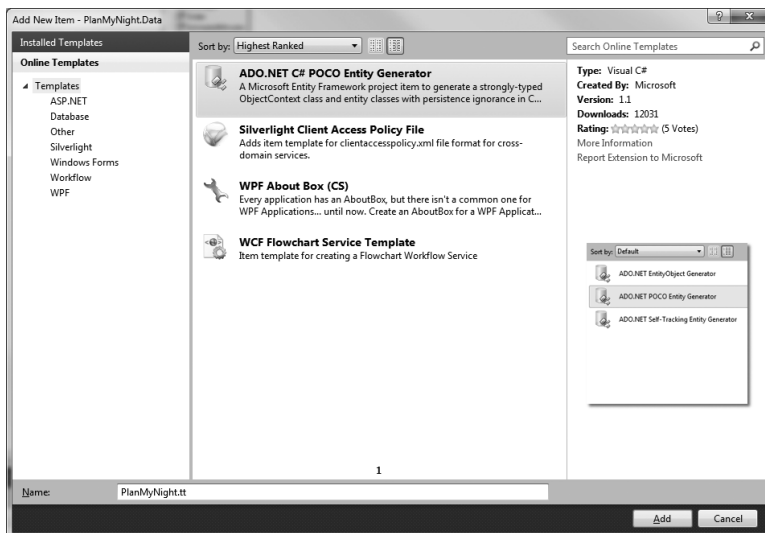


FIGURE 5-26 Add New Item dialog

Two files, `PlanMyNight.tt` and `PlanMyNight.Context.tt`, have been added to your project, as shown in Figure 5-27. These files replace the default code-generation template, and the code is no longer generated in the `PlanMyNight.Designer.cs` file.

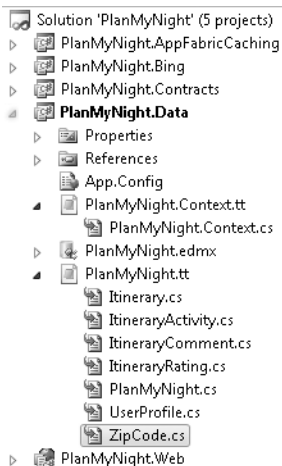


FIGURE 5-27 Added templates

The `PlanMyNight.tt` template produces a class file for each entity in the model. Listing 5-3 shows the POCO version of the `ZipCode` class.

LISTING 5-3 POCO Version of the *ZipCode* Class

```
namespace Microsoft.Samples.PlanMyNight.Data
{
    public partial class ZipCode
    {
        #region Primitive Properties
        public virtual string Code
        {
            get;
            set;
        }
        public virtual string City
        {
            get;
            set;
        }
        public virtual string State
        {
            get;
            set;
        }
    }
    #endregion
}
```



Tip C# 3.0 introduced a new feature called *automatic properties*. The backing field is created at compile time if the compiler finds empty *get* or *set* blocks.

The other file, *PlanMyNight.Context.cs*, generates the *ObjectContext* object for the *PlanMyNight.edmx* model. This is the object you'll use to interact with the database.



Tip The POCO templates will automatically update the generated classes to reflect the changes to your model when you save the *.edmx* file.

Moving the Entity Classes to the Contracts Project

We have designed the PMN application architecture to ensure that the presentation layer was persistence ignorant by moving the contracts and entity classes to an assembly that has no reference to the storage.

Visual Studio 2005 Even though it was possible to extend the XSD processing with code-generator tools, it was not easy and you had to maintain these tools. The EF uses T4 templates to generate both the database schema and the code. These templates can easily be customized to your needs.

The ADO.NET POCO templates split the generation of the entity classes into a separate template, allowing you to easily move these entities to a different project.

You are going to move the `PlanMyNight.tt` file to the `PlanMyNight.Contracts` project. Right-click the `PlanMyNight.tt` file, and select `Cut`. Right-click the `Entities` folder in the `PlanMyNight.Contracts` project, and select `Paste`. The result is shown in Figure 5-28.

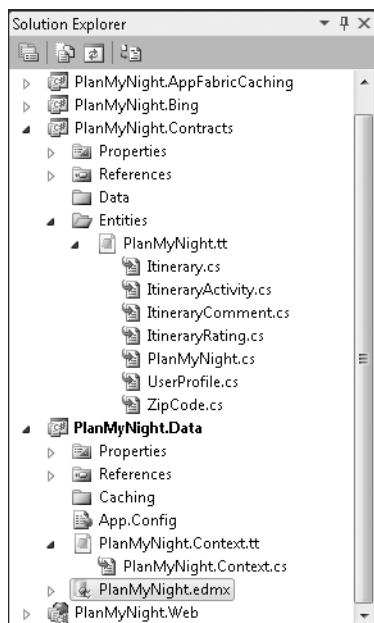


FIGURE 5-28 POCO template moved to the Contracts project

The `PlanMyNight.tt` template relies on the metadata from the EDM model to generate the entity type's code. You need to fix the relative path used by the template to access the EDMX file.

Open the `PlanMyNight.tt` template and locate the following line:

```
string inputFile = @"PlanMyNight.edmx";
```

Fix the file location so that it points to the `PlanMyNight.edmx` file in the `PlanMyNight.Data` project:

```
string inputFile = @"..\..\PlanMyNight.Data\PlanMyNight.edmx";
```

The entity classes are regenerated when you save the template.

You also need to update the `PlanMyNight.Context.tt` template in the `PlanMyNight.Contracts` project because the entity classes are now in the *Microsoft.Samples.PlanMyNight.Entities*

namespace instead of the *Microsoft.Samples.PlanMyNight.Data* namespace. Open the *PlanMyNight.Context.tt* file, and update the *using* section to include the new namespace:

```
using System;
using System.Data.Objects;
using System.Data.EntityClient;
using Microsoft.Samples.PlanMyNight.Entities;
```

Build the solution by pressing Ctrl+Shift+B. The project should now compile successfully.

Putting It All Together

Now that you have created the generic code layer to interact with your SQL database, you are ready to start implementing the functionalities specific to the PMN application. In the upcoming sections, you'll walk through this process, briefly look at getting the data from the Bing Maps services, and get a quick introduction to the Microsoft Windows Server AppFabric Caching feature used in PMN.

There is a lot of plumbing pieces of code required to get this all together. To simplify the process, you'll use an updated solution where the contracts, entities, and most of the connecting pieces to the Bing Maps services have been coded. The solution will also include the *PlanMyNight.Data.Test* project to help you validate the code from the *PlanMyNight.Data* project.



Note Testing in Visual Studio 2010 will be covered in Chapter 7.

Getting Data from the Database

At the beginning of this chapter, we decided to group the operations on the *Itinerary* entity into the *ItinerariesRepository* repository interface. Some of these operations are

- Searching for *Itinerary* by *Activity*
- Searching for *Itinerary* by *ZipCode*
- Searching for *Itinerary* by *Radius*
- Adding a new *Itinerary*

Let's take a look at the corresponding methods in the *ItinerariesRepository* interface:

- *SearchByActivity* allows searching for itineraries by activity and returning a page of data.
- *SearchByZipCode* allows searching for itineraries by Zip Code and returning a page of data.

- *SearchByRadius* allows searching for itineraries from a specific location and returning a page of data.
- *Add* allows you to add an itinerary to the database.

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\Code\Final by double-clicking the PlanMyNight.sln file.

Select the PlanMyNight.Data project, and open the ItinerariesRepository.cs file. This is the *ItinerariesRepository* interface implementation. Using the PlanMyNightEntities Object Context you generated earlier, you can write LINQ queries against your model, and the EF will translate these queries to native T-SQL that will be executed against the database.



Note LINQ stands for *Language Integrated Query* and was introduced in the .NET Framework 3.5. It adds native data querying capability to the .NET Framework so that you don't have to worry about learning or maintaining custom SQL queries. LINQ allows you to use strongly typed objects, and the Visual Studio IntelliSense lets you select the properties or methods that are in the current context as shown in Figure 5-29. To learn more about LINQ, visit the [.NET Framework Developer Center](http://msdn.microsoft.com/en-us/netframework/aa904594.aspx) (<http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>).

```
public PagingResult<Itinerary> SearchByActivity(string activityId, int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.

```

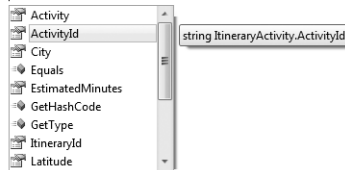


FIGURE 5-29 IntelliSense support for LINQ queries

Navigate to the *SearchByActivity* function definition. This method must return a set of itineraries where the *IsPublic* flag is set to true and where one of their activities has the same *activityId* that was passed in the argument to the function. You also need to order the result itinerary list by the rating field.

Visual Studio 2005 Implementing each method to retrieve the itinerary in Visual Studio 2005 would have required writing tailored SQL. With the EF and LINQ, any query becomes trivial and changes can be easily implemented at the code level!

Using standard LINQ operators, you can implement *SearchByActivity* as shown in Listing 5-4. Add the highlighted code to the *SearchByActivity* method body.

LISTING 5-4 *SearchByActivity* Implementation

```

public PagingResult<Itinerary> SearchByActivity(string activityId, int pageSize, int
pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.ActivityId == activityId)
                        && itinerary.IsPublic
                    orderby itinerary.Rating
                    select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}

```



Note The resulting paging is implemented in the *PageResults* method:

```

private static PagingResult<Itinerary> PageResults(IQueryable<Itinerary> query, int
page, int pageSize)
{
    int rowCount = query.Count();
    if (pageSize > 0)
    {
        query = query.Skip((page - 1) * pageSize)
                    .Take(pageSize);
    }
    var result = new PagingResult<Itinerary>(query.ToArray())
    {
        PageSize = pageSize,
        CurrentPage = page,
        TotalItems = rowCount
    };
    return result;
}

```

IQueryable<Itinerary> is passed to this function so that it can add the paging to the base query composition. Passing *IQueryable* instead of *IEnumerable* ensures that the T-SQL created for the query against the repository will be generated only when *query.ToArray* is called.

The *SearchByZipCode* function method is similar to the *SearchByActivity* method, but it also adds a filter on the Zip Code of the activity. Here again, LINQ support makes it easy to implement, as shown in Listing 5-5. Add the highlighted code to the *SearchByZipCode* method body.

LISTING 5-5 *SearchByZipCode* Implementation

```

public PagingResult<Itinerary> SearchByZipCode(int activityTypeId, string zip, int
pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.TypeId == activityTypeId &&
t.Zip == zip)
                    && itinerary.IsPublic
                    orderby itinerary.Rating
                    select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}

```

The *SearchByRadius* function calls the *RetrieveItinerariesWithinArea* import function that was mapped to a stored procedure. It then loads the activities for each itinerary found. You can copy the highlighted code in Listing 5-6 to the *SearchByRadius* method body in the *ItinerariesRepository.cs* file.

LISTING 5-6 *SearchByRadius* Implementation

```

public PagingResult<Itinerary> SearchByRadius(int activityTypeId,
double longitude, double latitude, double radius,
int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        // Stored Procedure with output parameter
        var totalOutput = new ObjectParameter("total", typeof(int));
        var items = ctx.RetrieveItinerariesWithinArea(activityTypeId,
latitude, longitude, radius, pageSize, pageNumber, totalOutput).ToArray();

        foreach (var item in items)
        {
            item.Activities.ToList().AddRange(this.Retrieve(item.Id).Activities);
        }

        int total = totalOutput.Value == DBNull.Value ? 0 : (int)totalOutput.Value;

        return new PagingResult<Itinerary>(items)
        {
            TotalItems = total,

```

```
        PageSize = pageSize,  
        CurrentPage = pageNumber  
    };  
}  
}
```

The *Add* method allows you to add *Itinerary* to the data store. Implementing this functionality becomes trivial because your contract and context object use the same entity object. Copy and paste the highlighted code in Listing 5-7 to the *Add* method body.

LISTING 5-7 *Add* Implementation

```
public void Add(Itinerary itinerary)  
{  
    using (var ctx = new PlanMyNightEntities())  
    {  
        ctx.Itineraries.AddObject(itinerary);  
        ctx.SaveChanges();  
    }  
}
```

There you have it! You have completed the *ItinerariesRepository* implementation using the context object generated using the EF designer. Run all the tests in the solution by pressing Ctrl+R, A. The tests related to the *ItinerariesRepository* implementation should all succeed.

Getting Data from the Bing Maps Web Services

PMN relies on the Bing Maps services to allow the user to search for activities to add to her itineraries. To get a Bing Maps Key to use in the PMN application, you need to create a Bing Maps Developer Account. You can create a free developer account on the Bing Maps Account Center.

See Also *Microsoft Bing Maps Web services is a set of programmable Simple Object Access Protocol (SOAP) services that allow you to match addresses to the map, search for points of interest, integrate maps and imagery, return driving directions, and incorporate other location intelligence into your Web application. You can learn more about these services by visiting the site for the [Bing Maps Web Services SDK](http://msdn.microsoft.com/en-us/library/cc980922.aspx) (<http://msdn.microsoft.com/en-us/library/cc980922.aspx>).*

Visual Studio 2005 In Visual Studio 2005, if you had to add a reference to a Web service you would have selected the Add Web Service Reference from the contextual menu to bring up the Add Web Reference dialog and then added a reference to a Web service to your project. (See Figure 5-30.)

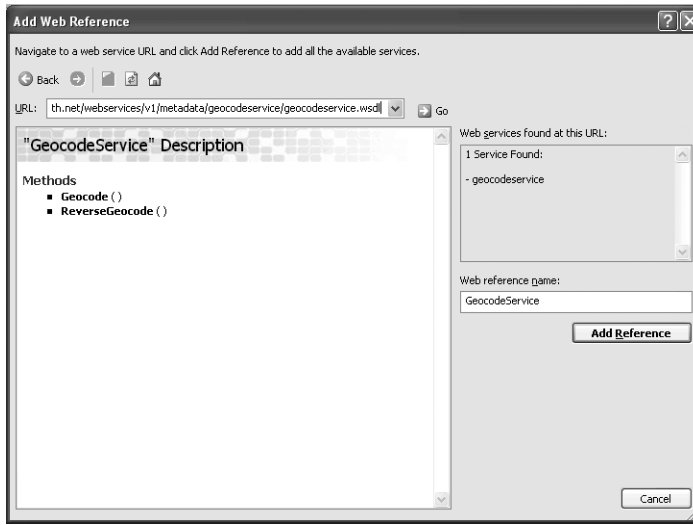


FIGURE 5-30 Visual Studio 2005 Add Web Reference dialog

Introduced in the .NET Framework 3.0, the Windows Communication Foundation (WCF) services brought the ASMX Web services and other communication technologies into a unified programming model.

Visual Studio 2010 provides tools for working with WCF services. You can bring up the new Add Service Reference dialog by right-clicking on a project node and selecting Add Service Reference as shown in Figure 5-31. In this dialog, you first need to specify the service metadata address in the Address field and then click Go to view the available service endpoints. You can then specify a namespace for the generated code in the Namespace text box and then click OK to add the proxy to your project.

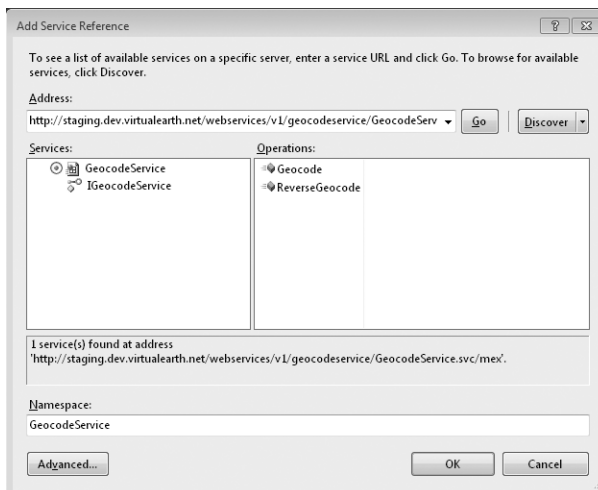


FIGURE 5-31 Add Service Reference dialog



Tip Click the Discover button to look for WCF services in the current solution.

See Also Click the Advanced button to access the Service Reference Settings dialog. This dialog lets you tweak the configuration of the WCF service proxy. You can add the .NET Framework 2.0 style reference by clicking the Add Web Service button. To learn more about these settings, visit [MSDN - Configure Service Reference Dialog Box](http://msdn.microsoft.com/en-us/library/bb514724(VS.100).aspx) ([http://msdn.microsoft.com/en-us/library/bb514724\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb514724(VS.100).aspx)).

The generated WCF proxy can be used in the same way you used the ASMX-style proxy, as shown in Listing 5-8.

LISTING 5-8 Using a Web Service Proxy

```
public BingCoordinate GeocodeAddress(ActivityAddress address, string token)
{
    ...
    Microsoft.Samples.PlanMyNight.Bing.VEGeocodingService.GeocodeResponse geocodeResponse
    = null;
    // Make the geocode request
    using (var geocodeService = new
    Microsoft.Samples.PlanMyNight.Bing.VEGeocodingService.GeocodeServiceClient())
    {
        try
        {
            geocodeResponse = geocodeService.Geocode(geocodeRequest);
            geocodeService.Close();
        }
        catch
        {
            geocodeService.Abort();
        }
    }

    if (geocodeResponse != null && geocodeResponse.Results != null && geocodeResponse.
    Results.Length > 0)
    {
        var location = geocodeResponse.Results[0].Locations[0];
        return new BingCoordinate { Latitude = (float)location.Latitude, Longitude =
        (float)location.Longitude };
    }

    return default(BingCoordinate);
}
```

Parallel Programming

With the advances in multicore computing, it is becoming more and more important for developers to be able to write parallel applications. Visual Studio 2010 and the .NET Framework 4.0 provide new ways to express concurrency in applications. The Task Parallel Library (TPL) is now part of the Base Class Library (BCL) for the .NET Framework. This means that every .NET application can now access the TPL without adding any assembly reference.

PMN stores only the Bing Activity ID for each *ItineraryActivity* to the database. When it's time to retrieve the entire Bing Activity object, a function that iterates through each of the *ItineraryActivity* instances for the current *Itinerary* is used to populate the Bing Activity entity from the Bing Maps Web services.

One way of performing this operation is to sequentially call the service for each activity in the *Itinerary*, as shown in Listing 5-9. This function waits for each call to *RetrieveActivity* to complete before making another call, which has the effect of making its execution time linear.

LISTING 5-9 Activity Sequential Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    foreach (var item in itinerary.Activities.Where(i =>i.Activity == null))
    {
        item.Activity = this.RetrieveActivity(item.ActivityId);
    }
}
```

In the past, if you wanted to parallelize this task, you had to use threads and then hand off work to them. With the TPL, all you have to do now is use a *Parallel.ForEach* that will take care of the threading for you, as seen in Listing 5-10.

LISTING 5-10 Activity Parallel Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    Parallel.ForEach(itinerary.Activities.Where(i =>i.Activity == null),
        item =>
        {
            item.Activity = this.RetrieveActivity(item.ActivityId);
        });
}
```

See Also *The .NET Framework 4.0 now includes the Parallel LINQ libraries (in System.Core.dll). PLINQ introduces the .AsParallel extension to perform parallel operations in LINQ queries. You can also easily enforce the treatment of a data source as if it was ordered by using the .AsOrdered extensions. Some new thread-safe collections have also been added in the System.Collections.Concurrent namespace. You can learn more about these new features from [Parallel Computing on MSDN](http://msdn.microsoft.com/en-us/concurrency/default.aspx) (<http://msdn.microsoft.com/en-us/concurrency/default.aspx>).*

AppFabric Caching

PMN is a data-driven application that gets its data from the application database and the Bing Maps Web services. One of the challenges you might face when building a Web application is managing the needs of a large number of users, including performance and response time. The operations that use the data store and the services used to search for activities can increase the usage of server resources dramatically for items that are shared across many users. For example, many users have access to the public itineraries, so displaying these will generate numerous calls to the database for the same items. Implementing caching at the Web tier will help reduce usage of the resources at the data store and help mitigate latency for recurring searches to the Bing Maps Web services. Figure 5-32 shows the architecture for an application implementing a caching solution at the front-end server.

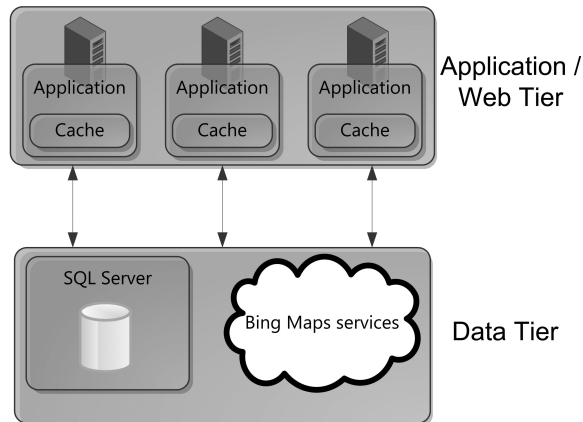


FIGURE 5-32 Typical Web application architecture

Using this approach reduces the pressure on the data layer, but the caching is still coupled to a specific server serving the request. Each Web tier server will have its own cache, but you can still end up with an uneven distribution of the processing to these servers.

Windows Server AppFabric caching offers a distributed, in-memory cache platform. The AppFabric client library allows the application to access the cache as a unified view event if

the cache is distributed across multiple computers, as shown in Figure 5-33. The API provides simple *get* and *set* methods to retrieve and store any serializable common language runtime (CLR) objects easily. The AppFabric cache allows you to add a cache computer on demand, thus making it possible to scale in a manner that is transparent to the client. Another benefit is that the cache can also share copies of the data across the cluster, thereby protecting data against failure.

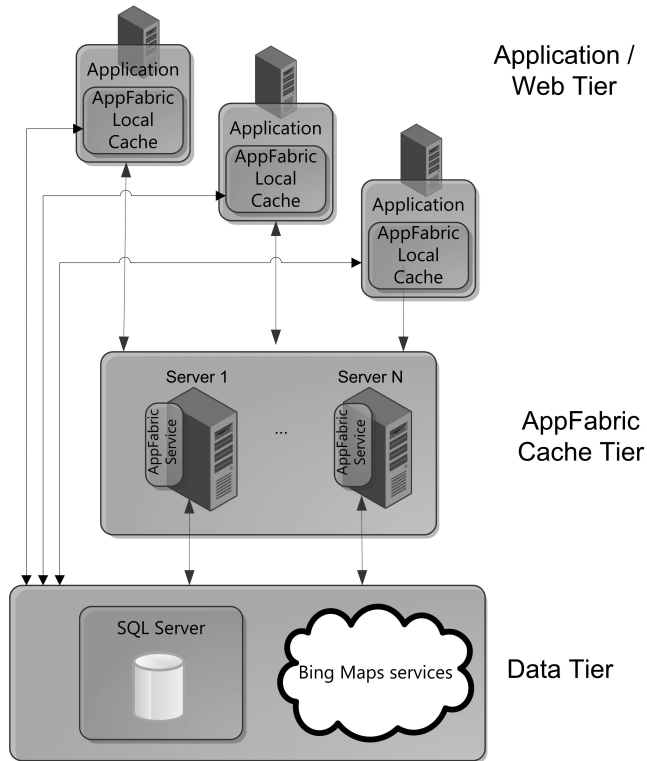


FIGURE 5-33 Web application using Windows Server AppFabric caching

See Also Windows Server AppFabric caching is available as a set of extensions to the .NET Framework 4.0. For more information about how to get, install, and configure Windows Server AppFabric, please visit [Windows Server AppFabric](http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx) (<http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx>).

See Also PMN can be configured to use either ASP.NET caching or Windows Server AppFabric caching. A complete walkthrough describing how to add Windows Server AppFabric caching to PMN is available here: [PMN: Adding Caching using Velocity](http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/) (<http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/>).

Summary

In this chapter, you used a few of the new Visual Studio 2010 features to structure the data layer of the Plan My Night application using the Entity Framework version 4.0 to access a database. You also were introduced to automated entity generation using the ADO.NET Entity Framework POCO templates and to the Windows Server AppFabric caching extensions.

In the next chapter, you will explore how the ASP.NET MVC framework and the Managed Extensibility Framework can help you build great Web applications.

Chapter 6

From 2005 to 2010: Designing the Look and Feel

After reading this chapter, you will be able to

- Create an ASP.NET MVC controller that interacts with the data model
- Create an ASP.NET MVC view that displays data from the controller and validates user input
- Extend the application with an external plug-in using the Managed Extensibility Framework

Web application development in Microsoft Visual Studio has certainly made significant improvements over the years since ASP.NET 1.0 was released. Visual Studio 2005 and .NET Framework 2.0 included things such as more efficient view state, partial classes, and generic types (plus many others) to help developers create efficient applications that were easy to manage.

The spirit of improvement to assist developers in creating world-class applications is very much alive in Visual Studio 2010. In this chapter, we'll explore some of the new features as we add functionality to the Plan My Night companion application.



Note The companion application is an ASP.NET MVC 2 project, but a Web developer has a choice in Visual Studio 2010 to use this new form of ASP.NET application or the more traditional ASP.NET (referred to in the community as *Web Forms for distinction*). ASP.NET 4.0 has many improvements to help developers and is still a very viable approach to creating Web applications.

We'll be using a modified version of the companion application's solution to work our way through this chapter. If you installed the companion content in the default location, the correct solution can be found at Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 6\ in a folder called UserInterface-Start.

Introducing the PlanMyNight.Web Project

The user interface portion of Plan My Night in Visual Studio 2010 was developed as an ASP.NET MVC application, the layout of which differs from what a developer might be accustomed to when developing an ASP.NET Web Forms application in Visual Studio 2005. Some

items in the project (as seen in Figure 6-1) will look familiar (such as Global.asax), but others are completely new, and some of the structure is required by the ASP.NET MVC framework.

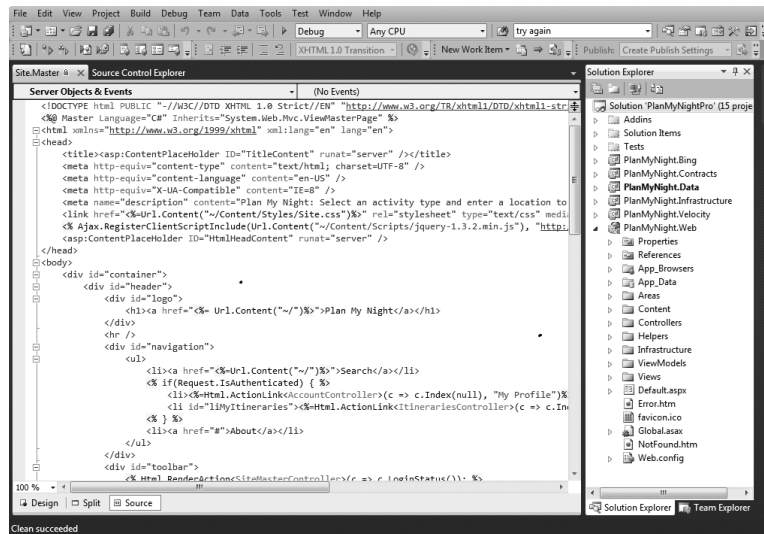


FIGURE 6-1 PlanMyNight.Web project view

Here are the items required by ASP.NET MVC:

- **Areas** This folder is used by the ASP.NET MVC framework to organize large Web applications into smaller components, without using separate solutions or projects. This feature is not used in the Plan My Night application but is called out because this folder is created by the MVC project template.
- **Controllers** During request processing, the ASP.NET MVC framework looks for controllers in this folder to handle the request.
- **Views** The Views folder is actually a structure of folders. The layer immediately inside the Views folder is named for each of the classes found in the Controllers folder, plus a Shared folder. The Shared subfolder is for common views, partial views, master pages, and anything else that will be available to all controllers.

See Also More information about ASP.NET MVC components, as well as how its request processing differs from ASP.NET Web Forms, can be found at <http://asp.net/mvc>.

In most cases, the web.config file is the last file in a project's root folder. However, it has received a much-needed update in Visual Studio 2010: Web.config Transformation. This feature allows for a base web.config file to be created but then to have build-specific web.config

files override the settings of the base at build, deployment, and run times. These files appear under the base web.config file, as seen in Figure 6-2.

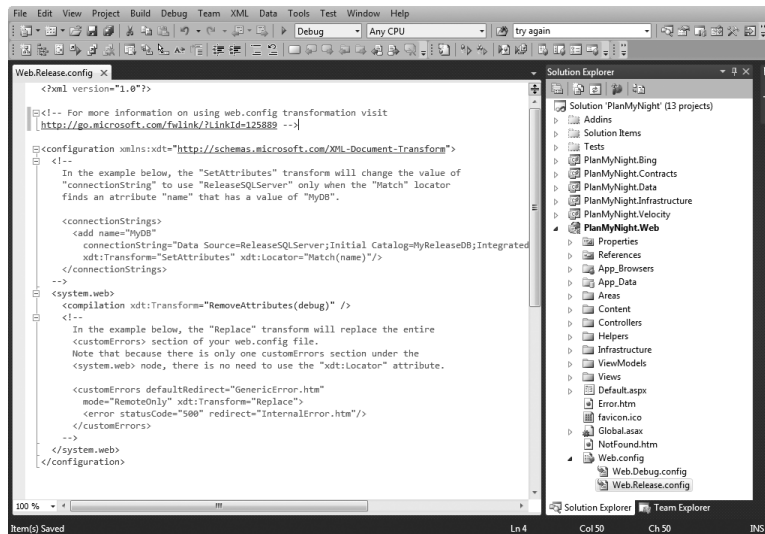


FIGURE 6-2 A web.config file with build-specific files expanded

Visual Studio 2005 When working on a project in Visual Studio 2005, do you recall needing to remember not to overwrite the web.config file with your debug settings? Or needing to remember to update web.config when it was published for a retail build with the correct settings? This is no longer an issue in Visual Studio 2010. The settings in the web.Release.config file will be used during release builds to override the values in web.config, and the same goes for web.Debug.config in debug builds.

Other sections of the project include the following:

- **Content** A collection of folders containing images, scripts, and style files
- **Helpers** Includes miscellaneous classes, containing a number of extension methods, that add functionality to types used in the project
- **Infrastructure** Contains items related to dealing with the lower level infrastructure of ASP.NET MVC (for example, caching and controller factories)
- **ViewModels** Contains data entities filled out by controller classes and used by views to display data

Running the Project

If you compile and run the project, you should see a screen similar to Figure 6-3.

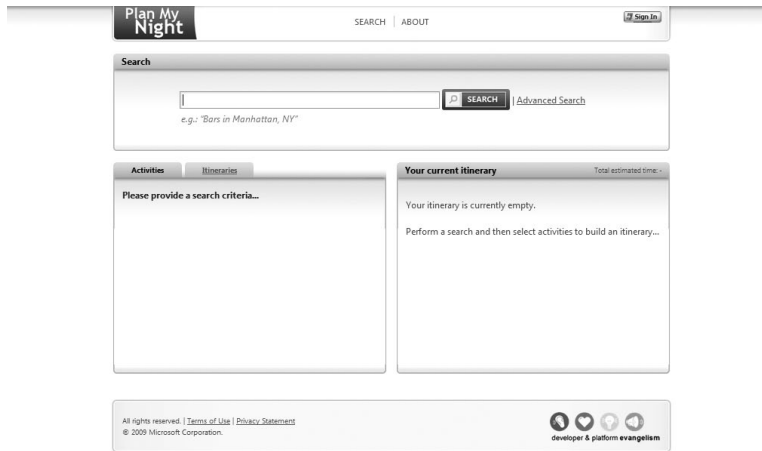


FIGURE 6-3 Default page of the Plan My Night application

The searching functionality and the ability to organize an initial list of itinerary items all work, but if you attempt to save the itinerary you are working on, or if you log in with Windows Live ID, the application will return a 404 Not Found error screen (as shown in Figure 6-4).

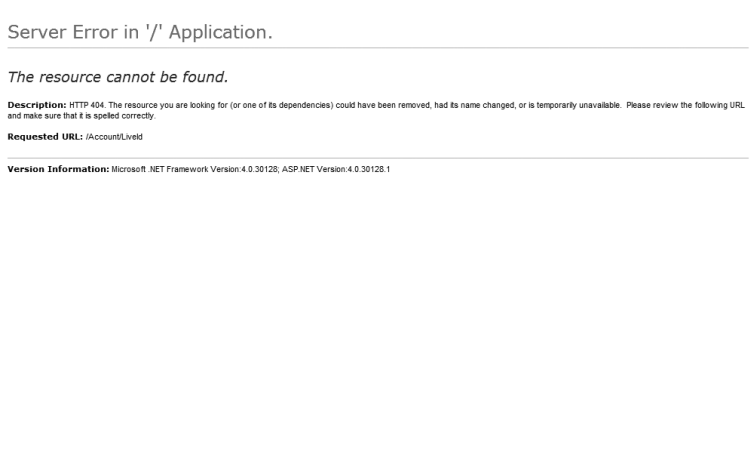


FIGURE 6-4 Error screen returned when logging in to the Plan My Night application

You get this error message because currently the project does not include an account controller to handle these requests.

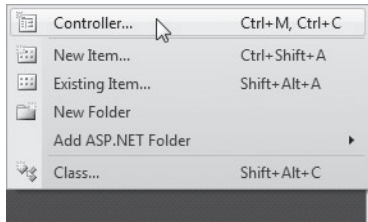
Creating the Account Controller

The *AccountController* class provides some critical functionality to the companion Plan My Night application:

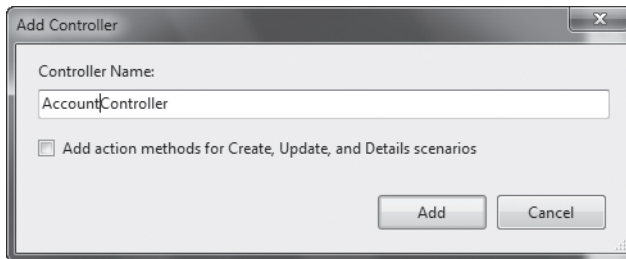
- It handles signing users in and out of the application (via Windows Live ID).
- It provides actions for displaying and updating user profile information.

To create a new ASP.NET MVC controller:

1. Use Solution Explorer to navigate to the Controllers folder in the PlanMyNight.Web project, and click the right mouse button.
2. Open the Add submenu, and select the Controller item.



3. Fill in the name of the controller as **AccountController**.



Note Leave the Add Action Methods For Create, Update, And Delete Scenarios check box blank. Selecting the box inserts some “starter” action methods, but because you will not be using the default methods, there is no reason to create them.

After you click the Add button in the Add Controller dialog box, you should have a basic *AccountController* class open, with a single *Index* method in its body:

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

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{

public class AccountController : Controller
    {
        //
        // GET: /Account/

        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Visual Studio 2005 A difference to be noted from developing ASP.NET Web Forms applications in Visual Studio 2005 is that ASP.NET MVC applications do not have a companion code-behind file for each of their .aspx files. Controllers like the one you are currently creating perform the logic required to process input and prepare output. This approach allows for a clear separation of display and business logic, and it's a key aspect of ASP.NET MVC.

Implementing the Functionality

To communicate with any of the data layers and services (the Model), you'll need to add some instance fields and initialize them. Before that, you need to add some namespaces to your *using* block:

```
using System.IO;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using System.Collections.Specialized;
using WindowsLiveId;
```

Now, let's add the instance fields. These fields are interfaces to the various sections of your Model:

```
public class AccountController : Controller
{
    private readonly I WindowsLiveLogin windowsLogin;
    private readonly I MembershipService membershipService;
    private readonly I FormsAuthentication formsAuthentication;
    private readonly I ReferenceRepository referenceRepository;
    private readonly I ActivitiesRepository activitiesRepository;
    .
    .
    .
}
```



Note Using interfaces to interact with all external dependencies allows for better portability of the code to various platforms. Also, during testing, dependencies can be mimicked much easier when using interfaces, making for more efficient isolation of a specific component.

As mentioned, these fields represent parts of the Model this controller will interact with to meet its functional needs. Here are the general descriptions for each of the interfaces:

- **IWindowsLiveLogin** Provides functionality for interacting with the Windows Live ID service.
- **IMembershipService** Provides user profile information and authorization methods. In your companion application, it is an abstraction of the ASP.NET Membership Service.
- **IFormsAuthentication** Provides for ASP.NET Forms Authentication abstraction.
- **IReferenceRepository** Provides reference resources, such as lists of states and other model-specific information.
- **IActivitiesRepository** An interface for retrieving and updating activity information.

You'll add two constructors to this class: one for general run-time use, which uses the *ServiceFactory* class to get references to the needed interfaces, and one to enable tests to inject specific instances of the interfaces to use.

```
public AccountController() :
    this(
        new ServiceFactory().GetMembershipService(),
        new WindowsLiveLogin(true),
        new FormsAuthenticationService(),
        new ServiceFactory().GetReferenceRepositoryInstance(),
        new ServiceFactory().GetActivitiesRepositoryInstance())
{
}

public AccountController(
    IMembershipService membershipService,
```

```
IWindowsLiveLogin windowsLogin,  
IFormsAuthentication formsAuthentication,  
IReferenceRepository referenceRepository,  
IActivitiesRepository activitiesRepository)  
{  
    this.membershipService = membershipService;  
    this.windowsLogin = windowsLogin;  
    this.formsAuthentication = formsAuthentication;  
    this.referenceRepository = referenceRepository;  
    this.activitiesRepository = activitiesRepository;  
}
```

Authenticating the User

The first real functionality you'll implement in this controller is that of signing in and out of the application. Most of the methods you'll implement later require authentication, so this is a good place to start.

The companion application uses a few technologies together at the same time to give the user a smooth authentication experience: Windows Live ID, ASP.NET Forms Authentication, and ASP.NET Membership Services. These three technologies are used in the LiveID action you'll implement next.

Start by creating the following method in the *AccountController* class:

```
public ActionResult LiveId()  
{  
    return Redirect("~/");  
}
```

This method will be the primary action invoked when interacting with the Windows Live ID services. Right now, if it is invoked, it will just redirect the user to the root of the application.



Note The call to *Redirect* returns *RedirectResult*, and although this example uses a string to define the target of the redirection, various overloads can be used for different situations.

A few different types of actions can be taken when Windows Live ID returns a user to your application. The user can be signing in to Windows Live ID, signing out, or clearing the Windows Live ID cookies. Windows Live ID uses a query string parameter called *action* on the URL when it returns a user, so you'll use a switch to branch the logic depending on the value of the parameter.

Add the following to the *LiveId* method above the return statement:

```
string action = Request.QueryString["action"];
switch (action)
{
    case "logout":
        this.formsAuthentication.SignOut();
        return Redirect("~/");

    case "clearcookie":
        this.formsAuthentication.SignOut();
        string type;
        byte[] content;
        this.windowsLogin.GetClearCookieResponse(out type, out content);
        return new FileStreamResult(new MemoryStream(content), type);
}
```

See also *Full documentation of the Windows Live ID system can be found on the <http://dev.live.com/> Web site.*

The code you just added handles the two sign-out actions for Windows Live ID. In both cases, you use the *IFormsAuthentication* interface to remove the ASP.NET Forms Authentication cookie so that any future http requests (until the user signs in again) will not be considered authenticated. In the second case, you went one step further to clear the Windows Live ID cookies (the ones that remember your login name but not your password).

Handling the sign-in scenario requires a bit more code because you have to check whether the authenticating user is in your Membership Database and, if not, create a profile for the user. However, before that, you must pass the data that Windows Live ID sent you to your Windows Live ID interface so that it can validate the information and give you a *WindowsLiveLogin.User* object:

```
default:
    // login
    NameValueCollection tokenContext;
    if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
    {
        tokenContext = Request.Form;
    }
    else
    {
        tokenContext = new NameValueCollection(Request.QueryString);
        tokenContext["stoken"] =
            System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
    }

    var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
```

At this point in the case for logging in, either *liveIdUser* will be a reference to an authenticated *WindowsLiveLogin.User* object or it will be null. With this in mind, you can add your next section of the code, which takes action when the *liveIdUser* value is not null:

```

if (liveIdUser != null)
{
    var returnUrl = liveIdUser.Context;
    var userId = new Guid(liveIdUser.Id).ToString();
    if (!this.membershipService.ValidateUser(userId, userId))
    {
        this.formsAuthentication.SignIn(userId, false);
        this.membershipService.CreateUser(userId, userId, string.Empty);
        var profile = this.membershipService.CreateProfile(userId);
        profile.FullName = "New User";
        profile.State = string.Empty;
        profile.City = string.Empty;
        profile.PreferredActivityTypeId = 0;
        this.membershipService.UpdateProfile(profile);

        if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
        return RedirectToAction("Index", new { returnUrl = returnUrl });
    }
    else
    {
        this.formsAuthentication.SignIn(userId, false);
        if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
        return Redirect(returnUrl);
    }
}
break;

```

The call to the *ValidateUser* method on the *IMembershipService* reference allows the application to check whether the user has been to this site before and whether there will be a profile for the user. Because the user is authenticated with Windows Live ID, you are using the user's ID value (which is a GUID) as both the user name and password to the ASP.NET Membership Service.

If the user does not have a user record with the application, you create one by calling the *CreateUser* method and then also create a user settings profile via *CreateProfile*. The profile is filled with some defaults and saved back to its store, and the user is redirected to the primary input page so that he can update the information.



Note *Controller.RedirectToAction* determines which URL to create based on the combination of input parameters. In this case, you want to redirect the user to the *Index* action of this controller, as well as pass the current return URL value.

The other action that takes place in this code is that the user is signed in to ASP.NET Forms authentication so that a cookie will be created, providing identity information on future requests that require authentication.

The settings profile is managed by ASP.NET Membership Services as well and is declared in the web.config file of the application:

```
<system.web>
...
<profile enabled="true">
  <properties>
    <add name="FullName" type="string" />
    <add name="State" type="string" />
    <add name="City" type="string" />
    <add name="PreferredActivityTypeId" type="int" />
  </properties>

  <providers>
    <clear />
    <add name="AspNetSqlProfileProvider"
type="System.Web.Profile.SqlProfileProvider,
      System.Web, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a"
      connectionStringName="ApplicationServices"
      applicationName="/" />
  </providers>
</profile>
...
</system.web>
```

At this point, the *LiveID* method is complete and should look like the following code. The application can now take authentication information from Windows Live ID, prepare an ASP.NET MembershipService profile, and create an ASP.NET Forms Authentication ticket.

```
public ActionResult LiveId()
{
  string action = Request.QueryString["action"];
  switch (action)
  {
    case "logout":
      this.formsAuthentication.SignOut();
      return Redirect("~/");

    case "clearcookie":
      this.formsAuthentication.SignOut();
      string type;
      byte[] content;
      this.windowsLogin.GetClearCookieResponse(out type, out content);
      return new FileStreamResult(new MemoryStream(content), type);

    default:
      // login
      NameValueCollection tokenContext;
      if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
      {
        tokenContext = Request.Form;
      }
      else
```

```

    {
        tokenContext = new NameValueCollection(Request.QueryString);
        tokenContext["stoken"] =
            System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
    }

    var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);

    if (liveIdUser != null)
    {
        var returnUrl = liveIdUser.Context;
        var userId = new Guid(liveIdUser.Id).ToString();
        if (!this.membershipService.ValidateUser(userId, userId))
        {
            this.formsAuthentication.SignIn(userId, false);
            this.membershipService.CreateUser(userId, userId, string.Empty);
            var profile = this.membershipService.CreateProfile(userId);
            profile.FullName = "New User";
            profile.State = string.Empty;
            profile.City = string.Empty;
            profile.PreferredActivityTypeId = 0;
            this.membershipService.UpdateProfile(profile);

            if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
            return RedirectToAction("Index", new { returnUrl = returnUrl });
        }
        else
        {
            this.formsAuthentication.SignIn(userId, false);
            if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
            return Redirect(returnUrl);
        }
    }
    break;
}
return Redirect("~/");
}

```

Of course, the user has to be able to get to the Windows Live ID login page in the first place before logging in. Currently in the Plan My Night application, there is a Windows Live ID login button. However, there are cases where the application will want the user to be redirected to the login page from code. To cover this scenario, you need to add a small method called *Login* to your controller:

```

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
    return Redirect(redirect);
}

```

This method simply retrieves the login URL for Windows Live and redirects the user to that location. This also satisfies a configuration value in your web.config file for ASP.NET Forms Authentication in that any request requiring authentication will be redirected to this method:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" name="XAUTH" timeout="2880" path="~/\" />
</authentication>
```

Retrieving the Profile for the Current User

Now with the authentication methods defined, which satisfies your first goal for this controller—signing users in and out in the application—you can move on to retrieving data for the current user.

The *Index* method, which is the default method for the controller based on the URL mapping configuration in *Global.asax*, will be where you retrieve the current user's data and return a view displaying that data. The *Index* method that was initially created when the *AccountController* class was created should be replaced with the following:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Index(string returnUrl)
{
    var profile = this.membershipService.GetCurrentProfile();
    var model = new ProfileViewModel
    {
        Profile = profile,
        ReturnUrl = returnUrl ?? this.GetReturnUrl()
    };

    this.InjectStatesAndActivityTypes(model);

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

Visual Studio 2005 Attributes, such as *[Authorize()]*, might not have been in common use in Visual Studio 2005; however, ASP.NET MVC makes use of them often. Attributes allow for metadata to be defined about the target they decorate. This allows for the information to be examined at run time (via reflection) and for action to be taken if deemed necessary.

The *Authorize* attribute is very handy because it declares that this method can be invoked only for http requests that are already authenticated. If a request is not authenticated, it will be redirected to the ASP.NET Forms Authentication configured login target, which you just finished setting up. The *AcceptVerbs* attribute also restricts how this method can be invoked, by specifying which Http verbs can be used. In this case, you are restricting this method to HTTP GET verb requests. You've added a string parameter, *returnUrl*, to the method signature so that when the user is finished viewing or updating her information, she can be returned to what she was looking at previously.



Note This highlights a part of the ASP.NET MVC framework called *Model Binding*, details of which are beyond the scope of this book. However, you should know that it attempts to find a source for *returnUrl* (a form field, routing table data, or query string parameter with the same name) and binds it to this value when invoking the method. If the Model Binder cannot find a suitable source, the value will be null. This behavior can cause problems for value types that cannot be null, because it will throw an *InvalidOperationException*.

The main portion of this method is straightforward: it takes the return of the *GetCurrentProfile* method on the ASP.NET Membership Service interface and sets up a view model object for the view to use. The call to *GetReturnUrl* is an example of an extension method defined in the *PlanMyNight.Infrastructure* project. It's not a member of the *Controller* class, but in the development environment it makes for much more readable code. (See Figure 6-5.)

```

File Edit View Refactor Project Build Debug Team Data Tools Test Window Help
Debug Any CPU GetReturnUrl
New Work Item
MvcExtensions.cs SiteMaster Source Control Explorer
Microsoft.Samples.PlanMyNight.Infrastructure.Mvc.MvcExtensions - % GetAbsoluteUrl(ControllerBase controller, string path)
return LinkBuilder.BuildUrlFromExpressionT<(
    controller.ControllerContext.RequestContext,
    controller.Url.RouteCollection,
    action);
}
public static string GetAbsoluteUrl(this ControllerBase controller, string path)
{
    return String.Concat(controller.ControllerContext.HttpContext.Request.Url.Scheme,
        "://", controller.ControllerContext.HttpContext.Request.ServerVariables["HTTP_HOST"], path);
}
public static bool IsAjaxCall(this ControllerBase controller)
{
    return !string.IsNullOrEmpty(controller.Request.ContentType) &&
        controller.Request.ContentType.Contains("application/json");
}
public static string GetReturnUrl(this ControllerBase controller)
{
    if (controller.Request.ServerVariables != null &&
        !string.IsNullOrEmpty(controller.Request.ServerVariables["HTTP_REFERER"]))
        return controller.Request.ServerVariables["HTTP_REFERER"];
    return "~/";
}
}
}
100 % Ln 21 Col 17 Ch 17 INS
Ready

```

FIGURE 6-5 Example of extension methods in *MvcExtensions.cs*

Visual Studio 2005 In .NET Framework 2.0, which Visual Studio 2005 used, extension methods did not exist. Rather than calling *this.GetReturnUrl()* and also having the method appear in IntelliSense for this object, you would have to type **MvcExtensions.GetReturnUrl(this)**, passing in the controller as a parameter. Extension methods certainly make the code more readable and do not require the developer to know the static class the extension method exists under. For IntelliSense to work, the namespace needs to be listed in the *using* clauses.

InjectStatesAndActivityTypes is a method you need to implement. It gathers data from the reference repository for names of states and the activity repository. It makes two collections of *SelectListItem* (an HTML class for MVC): one for the list of states, and the other for the list of different activity types available in the application. It also sets the respective value.

```
private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null && o.Id ==
                profile.PreferredActivityTypeId)
        }).ToList();

    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null && o.Abbreviation ==
                profile.State)
        }).ToList();

    states.Insert(0, new SelectListItem {
        Text = "Any state",
        Value = string.Empty
    });

    model.PreferredActivityTypes = types;
    model.States = states;
}
```

Visual Studio 2005 In Visual Studio 2005, the *InjectStatesAndActivities* method takes longer to implement because a developer cannot use the LINQ extensions (the call to *Select*) and Lambda expressions, which are a form of anonymous delegate that the *Select* method applies to each member of the collection being enumerated. Instead, the developer would have to write out his own loop and enumerate each item manually.

Updating the Profile Data

Having completed the infrastructure needed to retrieve data for the current profile, you can move on to updating the data in the model from a form submission by the user. After this,

you can create your view pages and see how all this ties together. The *Update* method is simple; however, it does introduce some new features not seen yet:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
public ActionResult Update(UserProfile profile)
{
    var returnUrl = Request.Form["returnUrl"];
    if (!ModelState.IsValid)
    {
        // validation error
        return this.IsAjaxCall() ? new JsonResult { JsonRequestBehavior =
            JsonRequestBehavior.AllowGet, Data = ModelState }
            : this.Index(returnUrl);
    }

    this.membershipService.UpdateProfile(profile);
    if (this.IsAjaxCall())
    {
        return new JsonResult { JsonRequestBehavior = JsonRequestBehavior.AllowGet,
            Data = new { Update = true, Profile = profile, ReturnUrl = returnUrl } };
    }
    else
    {
        return RedirectToAction("UpdateSuccess", "Account", new { returnUrl =
            returnUrl });
    }
}
```

The *ValidateAntiForgeryToken* attribute ensures that the form has not been tampered with. To use this feature, you need to add an *AntiForgeryToken* to your view's input form. The check on the *ModelState* to see whether it is valid is your first look at input validation. This is a look at the server-side validation, and ASP.NET MVC offers an easy-to-use feature to make sure that incoming data meets some rules. The *UserProfile* object that is created for input to this method, via MVC Model Binding, has had one of its properties decorated with a *System.ComponentModel.DataAnnotations.Required* attribute. During Model Binding, the MVC framework evaluates *DataAnnotation* attributes and marks the *ModelState* as valid only when all of the rules pass.

In the case where the *ModelState* is not valid, the user is redirected to the *Index* method where the *ModelState* will be used in the display of the input form. Or, if the request was an AJAX call, a *JsonResult* is returned with the *ModelState* data attached to it.

Visual Studio 2005 Because in ASP.NET MVC requests are routed through controllers rather than pages, the same URL can handle a number of requests and respond with the appropriate view. In Visual Studio 2005, a developer would have to create two different URLs and call a method in a third class to perform the functionality.

When the *ModelState* is valid, the profile is updated in the membership service and a JSON result is returned for AJAX requests with the success data, or in the case of “normal” requests, the user is redirected to the *UpdateSuccess* action on the Account controller. The *UpdateSuccess* method is the final method you need to implement to finish off this controller:

```
public ActionResult UpdateSuccess(string returnUrl)
{
    var model = new ProfileViewModel
    {
        Profile = this.membershipService.GetCurrentProfile(),
        ReturnUrl = returnUrl
    };
    return View(model);
}
```

The method is used to return a success view to the browser, display some of the updated data, and provide a link to return the user to where she was when she started the profile update process.

Now that you’ve reached the end of the Account controller implementation, you should have a class that resembles the following listing:

```
using System;
using System.Collections.Specialized;
using System.IO;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using WindowsLiveId;

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    [HandleErrorWithContentType()]
    [OutputCache(NoStore = true, Duration = 0, VaryByParam = "")]
    public class AccountController : Controller
    {
        private readonly IWindowsLiveLogin windowsLogin;
        private readonly IMembershipService membershipService;
        private readonly IFormsAuthentication formsAuthentication;
        private readonly IReferenceRepository referenceRepository;
        private readonly IActivitiesRepository activitiesRepository;

        public AccountController() :
            this(
```

```
        new ServiceFactory().GetMembershipService(),
        new WindowsLiveLogin(true),
        new FormsAuthenticationService(),
        new ServiceFactory().GetReferenceRepositoryInstance(),
        new ServiceFactory().GetActivitiesRepositoryInstance())
    {
    }

    public AccountController(IMembershipService membershipService,
        IWindowsLiveLogin windowsLogin,
        IFormsAuthentication formsAuthentication,
        IReferenceRepository referenceRepository,
        IActivitiesRepository activitiesRepository)
    {
        this.membershipService = membershipService;
        this.windowsLogin = windowsLogin;
        this.formsAuthentication = formsAuthentication;
        this.referenceRepository = referenceRepository;
        this.activitiesRepository = activitiesRepository;
    }

    public ActionResult LiveId()
    {
        string action = Request.QueryString["action"];
        switch (action)
        {
            case "logout":
                this.formsAuthentication.SignOut();
                return Redirect("~/");
            case "clearcookie":
                this.formsAuthentication.SignOut();
                string type;
                byte[] content;
                this.windowsLogin.GetClearCookieResponse(out type, out content);
                return new FileStreamResult(new MemoryStream(content), type);
            default:
                // login
                NameValueCollection tokenContext;
                if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
                {
                    tokenContext = Request.Form;
                }
                else
                {
                    tokenContext = new NameValueCollection(Request.QueryString);
                    tokenContext["stoken"] =
                        System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
                }

                var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
                if (liveIdUser != null)
                {
                    var returnUrl = liveIdUser.Context;
                    var userId = new Guid(liveIdUser.Id).ToString();
                }
            }
        }
    }
}
```



```
        if (!this.membershipService.ValidateUser(userId, userId))
        {
            this.formsAuthentication.SignIn(userId, false);
            this.membershipService.CreateUser(
                userId, userId, string.Empty);
            var profile =
                this.membershipService.CreateProfile(userId);
            profile.FullName = "New User";
            profile.State = string.Empty;
            profile.City = string.Empty;
            profile.PreferredActivityTypeId = 0;
            this.membershipService.UpdateProfile(profile);
            if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
            return RedirectToAction("Index", new { returnUrl =
                returnUrl });
        }
        else
        {
            this.formsAuthentication.SignIn(userId, false);
            if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
            return Redirect(returnUrl);
        }
    }
    break;
}
return Redirect("~/");
}

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
    return Redirect(redirect);
}

[Authorize()]
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Index(string returnUrl)
{
    var profile = this.membershipService.GetCurrentProfile();
    var model = new ProfileViewMode1
    {
        Profile = profile,
        ReturnUrl = returnUrl ?? this.GetReturnUrl()
    };

    this.InjectStatesAndActivityTypes(model);
    return View("Index", model);
}

[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
```

```

public ActionResult Update(UserProfile profile)
{
    var returnUrl = Request.Form["returnUrl"];
    if (!ModelState.IsValid)
    {
        // validation error
        return this.IsAjaxCall() ?
            new JsonResult { JsonRequestBehavior =
                JsonRequestBehavior.AllowGet, Data = ModelState }
                : this.Index(returnUrl);
    }
    this.membershipService.UpdateProfile(profile);
    if (this.IsAjaxCall())
    {
        return new JsonResult {
            JsonRequestBehavior = JsonRequestBehavior.AllowGet,
            Data = new {
                Update = true,
                Profile = profile,
                ReturnUrl = returnUrl } };
    }
    else
    {
        return RedirectToAction("UpdateSuccess",
            "Account", new { returnUrl = returnUrl });
    }
}

public ActionResult UpdateSuccess(string returnUrl)
{
    var model = new ProfileViewModel
    {
        Profile = this.membershipService.GetCurrentProfile(),
        ReturnUrl = returnUrl
    };
    return View(model);
}

private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes()
        .Select(o => new SelectListItem { Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null &&
                o.Id == profile.PreferredActivityTypeId) })
        .ToList();
    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null &&
                o.Abbreviation == profile.State) })
        .ToList();
}

```

```
        states.Insert(0,
            new SelectListItem { Text = "Any state",
                                Value = string.Empty });
        model.PreferredActivityTypes = types;
        model.States = states;
    }
}
}
```

Creating the Account View

In the previous section, you created a controller with functionality that allows a user to update her information and view it. In this section, you're going to walk through the Visual Studio 2010 features that enable you to create the views that display this functionality to the user.

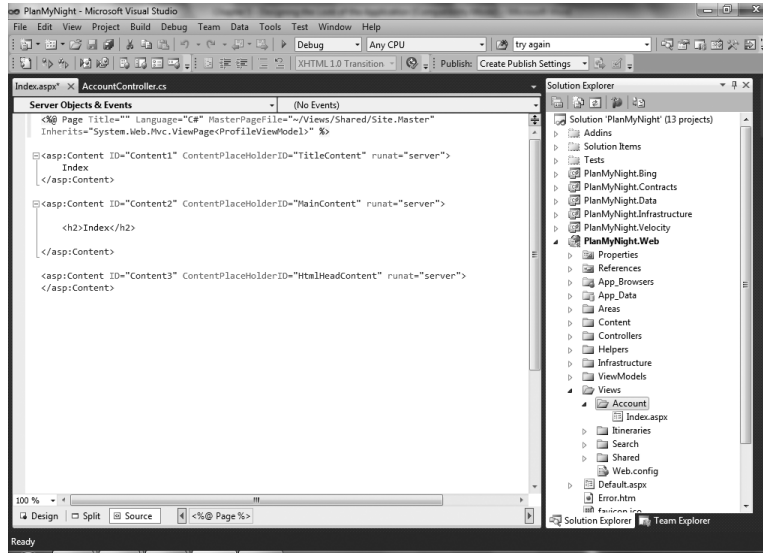
To create the Index view for the Account controller:

1. Navigate to the Views folder in the PlanMyNight.Web project.
2. Click the right mouse button on the Views folder, expand the Add submenu, and select New Folder.
3. Name the new folder **Account**.
4. Click the right mouse button on the new Account folder, expand the Add submenu, and select View.
5. Fill out the Add View dialog box as shown here:

The screenshot shows the 'Add View' dialog box with the following configuration:

- View name: Index
- Create a partial view (.ascx)
- Create a strongly-typed view
- View data class: ProfileViewModel
- View content: Empty
- Select master page
- ~/Views/Shared/Site.Master
- ContentPlaceHolder ID: MainContent

- Click OK. You should see an HTML page with some `<asp:Content>` controls in the markup:



You might notice that it doesn't look much different from what you are used to seeing in Visual Studio 2005. By default, ASP.NET MVC 2 uses the ASP.NET Web Forms view engine, so there will be some commonality between MVC and Web Forms pages. The primary differences at this point are that the *page* class derives from *System.Web.Mvc.ViewPage<ProfileViewModel>* and there is no code-behind file. MVC does not use code-behind files, like ASP.NET Web Forms does, to enforce a strict separation of concerns. MVC pages are generally edited in markup view; the designer view is primarily for ASP.NET Web Forms applications.

For this page skeleton to become the main view for the Account controller, you should change the title content to be more in line with the other views:

```

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
Plan My Night - Profile
</asp:Content>

```

Next you need to add the client scripts you are going to use in the content placeholder for the *HtmlHeadContent*:

```

<asp:Content ID="Content3" ContentPlaceHolderID="HtmlHeadContent" runat="server">
<% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery-1.3.2.min.js"),
    "http://ajax.microsoft.com/ajax/jquery/jquery-1.3.2.min.js"); %>
<% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery.validate.js"),
    "http://ajax.microsoft.com/ajax/jquery.validate/1.5.5/jquery.validate.min.js"); %>
<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/MicrosoftMvcjQueryValidation.js"), "pmm"); %>

```

```

<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.common.js"), "pmn"); %>
<% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.profile.js"), "pmn"); %>
<%= Ajax.RenderClientScripts() %>
</asp:Content>

```

This script makes use of extension methods for the *System.Web.Mvc.AjaxHelper*, which are found in the *PlanMyNight.Infrastructure* project, under the MVC folder.

With the head content set up, you can look at the main content of the view:

```

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
<div class="panel" id="profileForm">
    <div class="innerPanel">
        <h2><span>My Profile</span></h2>
        <% Html.EnableClientValidation(); %>
        <% using (Html.BeginForm("Update", "Account")) %>
        <% { %>
        <%=Html.AntiForgeryToken()%>
        <div class="items">
            <fieldset>
                <p>
                    <label for="FullName">Name:</label>
                    <%=Html.EditorFor(m => m.Profile.FullName)%>
                    <%=Html.ValidationMessage("Profile.FullName",
                        new { @class = "field-validation-error-wrapper" })%>
                </p>
                <p>
                    <label for="State">State:</label>
                    <%=Html.DropDownListFor(m => m.Profile.State, Model.States)%>
                </p>
                <p>
                    <label for="City">City:</label>
                    <%=Html.EditorFor(m => m.Profile.City, Model.Profile.City)%>
                </p>
                <p>
                    <label for="PreferredActivityTypeId">Preferred activity:</label>
                    <%=Html.DropDownListFor(m =>
                        m.Profile.PreferredActivityTypeId,
                        Model.PreferredActivityTypes)%>
                </p>
            </fieldset>
            <div class="submit">
                <%=Html.Hidden("returnUrl", Model.ReturnUrl)%>
                <%=Html.SubmitButton("submit", "Update")%>
            </div>
        </div>
        <div class="toolbox"></div>
        <% } %>
    </div>
</div>
</asp:Content>

```

Aside from some inline code, this looks to be fairly normal HTML markup. We're going to focus our attention on the inline code pieces to demonstrate the power they bring (as well as the simplicity).

Visual Studio 2005 In Visual Studio 2005, it was more commonplace to use server-side controls to display data, and other display-time logic. However, because ASP.NET MVC view pages do not have a code-behind file, server-side logic executed in the view at render time must be done in the same file with the markup. ASP.NET Web Forms controls can still be used. Our example makes use of the `<asp:Content>` control. However, the functionality of ASP.NET Web Forms controls is generally limited because there is no code-behind file.

MVC makes a lot of use of what is known as HTML helpers. The methods contained under *System.Web.Mvc.HtmlHelper* emit small, standards-compliant HTML tags for various uses. This requires the MVC developer to type more markup than a Web Forms developer in some cases, but the developer has more direct control over the output. The strongly typed version of this extension class (*HtmlHelper<TModel>*) can be referenced in the view markup via the *ViewPage<TModel>.Html* property.

These are the *HTML* methods used in this form, which are only a fraction of what is available by default:

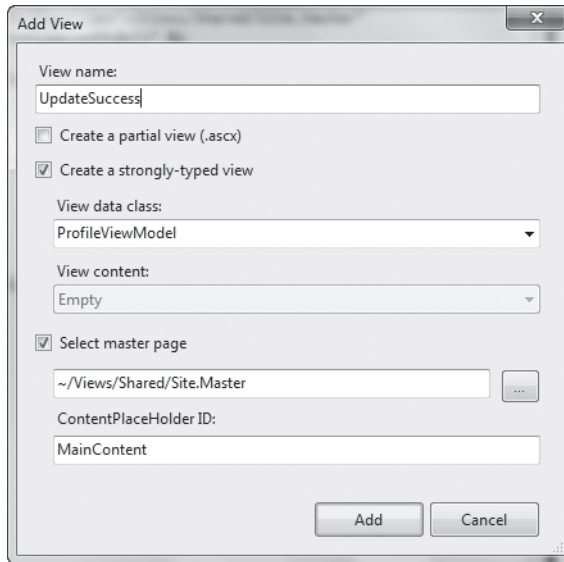
- *Html.EnableClientValidation* enables data validation to be performed on the client side based on the strongly typed ModelState dictionary.
- *Html.BeginForm* places a `<form>` tag in the markup and closes the form at the end of the *using* section. It takes various parameters for options, but the most common parameter is the name of the action and the controller to invoke that action on. This allows the MVC framework to generate the specific URL to target the form to at run time, rather than having to input a string URL into the markup.
- *Html.AntiForgeryToken* places a hidden field in the form with a check value that is also stored in a cookie in the visitor's browser and validated when the target of the form has the *ValidateAntiForgeryToken* attribute. Remember that you added this attribute to the *Update* method in the controller.
- *Html.EditorFor* is an overloaded method that inserts a text box into the markup. This is the strongly typed version of the *Html.Editor* method.
- *Html.DropDownListFor* is an overloaded method that places a drop-down list into the markup. This is the strongly typed version of the *Html.DropDownList* method.
- *Html.ValidationMessage* is a helper that will display a validation error message when a given key is present in the ModelState dictionary.
- *Html.Hidden* places a hidden field in the form, with the name and value that is passed in.
- *Html.SubmitButton* creates a Submit button for the form.



Note With the Index view markup complete, you only need to add the view for the *UpdateSuccess* action before you can see your results.

To create the UpdateSuccess view:

1. Expand the PlanMyNight.Web project in Solution Explorer, and then expand the Views folder.
2. Click the right mouse button on the Account folder.
3. Open the Add submenu, and click View.
4. Fill out the Add View dialog box so that it looks like this:



After the view page is created, fill in the title content so that it looks like this:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">Plan My Night - Profile Updated</asp:Content>
```

And the placeholder for *MainContent* should look like this:

```
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
<div class="panel" id="profileForm">
  <div class="innerPanel">
    <h2><span>My Profile</span></h2>
    <div class="items">
      <p>Your profile has been successfully updated.</p>
      <h3>> <a href="<%=Html.AttributeEncode(Model.ReturnUrl ??
        Url.Content("~/"))%>">Continue</a></h3>
    </div>
    <div class="toolbox"></div>
  </div>
</div>
</asp:Content>
```

To see the views created, you must perform an edit to the Site.Master file (located in the Views/Shared folder from the Web project's root). Line 33 of the file is commented out, and the comment tags should be removed so that it matches the following example:

```
<%=Html.ActionLink<AccountController>(c =>c.Index(null), "My Profile")%>
```

With this last view created, you can now compile and launch the application. Click the Sign In button, as seen in the top right corner of Figure 6-6, and sign in to Windows Live ID.

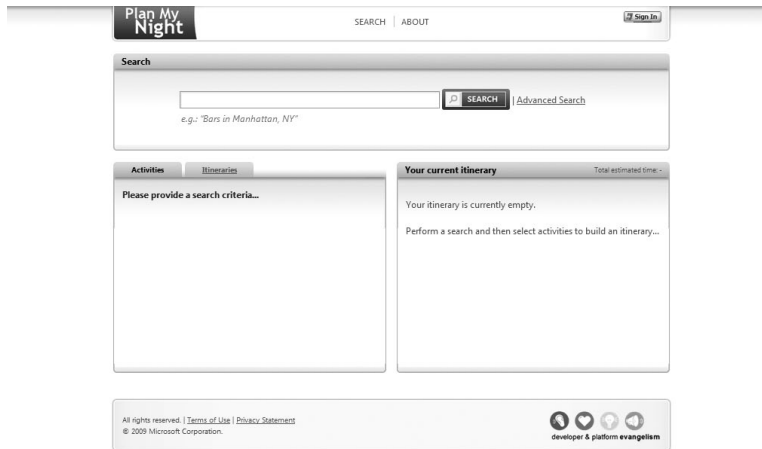


FIGURE 6-6 Plan My Night default screen

After you've signed in, you should be redirected to the Index view of the Account controller you created, shown in Figure 6-7.

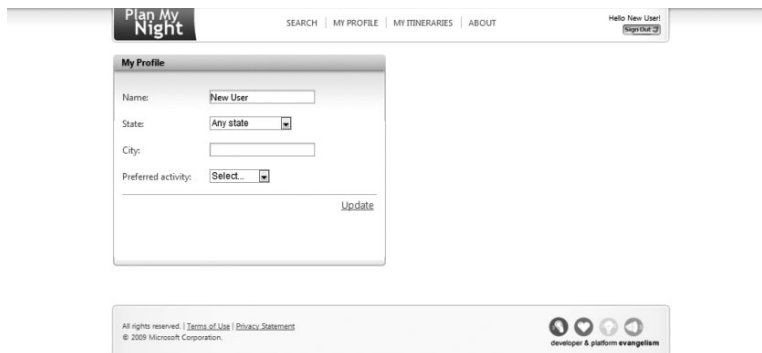
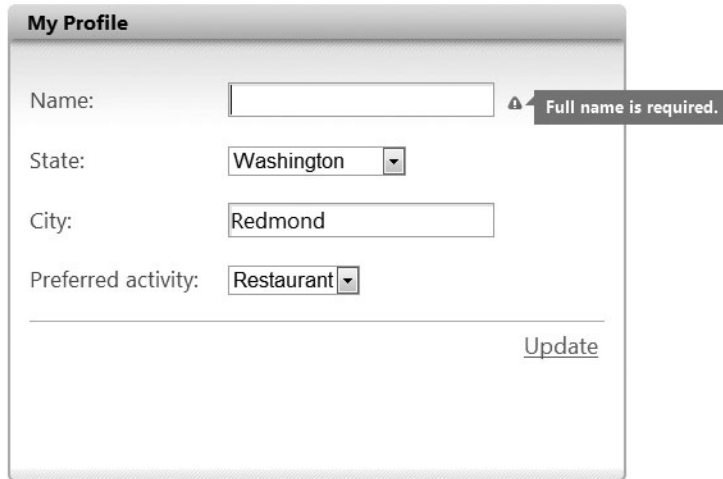


FIGURE 6-7 Profile settings screen returned from the *Index* method of the Account controller

If instead you are returned to the search page, just click the My Profile link, located in the links at the center and top of the interface. To see the new data-validation features at work, try to save the form without filling in the Full Name field. You should get a result that looks like Figure 6-8.



The screenshot shows a web form titled "My Profile". It contains four input fields: "Name:" (empty), "State:" (dropdown menu with "Washington" selected), "City:" (text box with "Redmond" entered), and "Preferred activity:" (dropdown menu with "Restaurant" selected). A red error message box with a warning icon is positioned over the "Name:" field, displaying the text "Full name is required.". At the bottom right of the form, there is an "Update" button.

FIGURE 6-8 Example of failed validation during Model Binding checks

Because you enabled client-side validation, there was no post back. To see the server-side validation work, you would have to edit the `Index.aspx` file in the `Account` folder and comment out the call to `Html.EnableClientValidation`. The tight integration and support of AJAX and other JavaScript in MVC applications allows for server-side operations such as validation to be moved to the client side much more easily than they were previously.

Visual Studio 2005 In ASP.NET MVC applications, the value of the ID attribute for a particular HTML element is not transformed, like it is in ASP.NET Web Forms 2.0. In Visual Studio 2005, a developer would have to make sure to set the *UniqueID* of a control/element into a JavaScript variable so that it could be accessed by external JavaScript. This was done to make sure the ID was unique. However, it was always an extra layer of complexity added to the interaction between ASP.NET 2.0 Web Forms controls and JavaScript. In MVC, this transformation does not happen, but it is up to the developers to ensure uniqueness of the ID. It should also be noted that ASP.NET 4.0 Web Forms now supports disabling the ID transformation on a per-control basis, if the developer so wishes.

With the completed Account controller and related views, you have filled in the missing “core” functionality of Plan My Night, while taking a brief tour of some new features in Visual Studio 2010 and MVC 2.0 applications. But MVC is not the only choice for Web developers. ASP.NET Web Forms has been the primary application type for ASP.NET since it was released, and it continues to be improved upon in Visual Studio 2010. In the next section, we’ll explore creating an ASP.NET Web Form with the Visual Designer to be used in the MVC application.

Using the Designer View to Create a Web Form

Applications will encounter an unexpected condition at some point in their lifetime of use. The companion application is no different, and when it does encounter an unexpected condition, it returns an error screen like that shown in Figure 6-9.

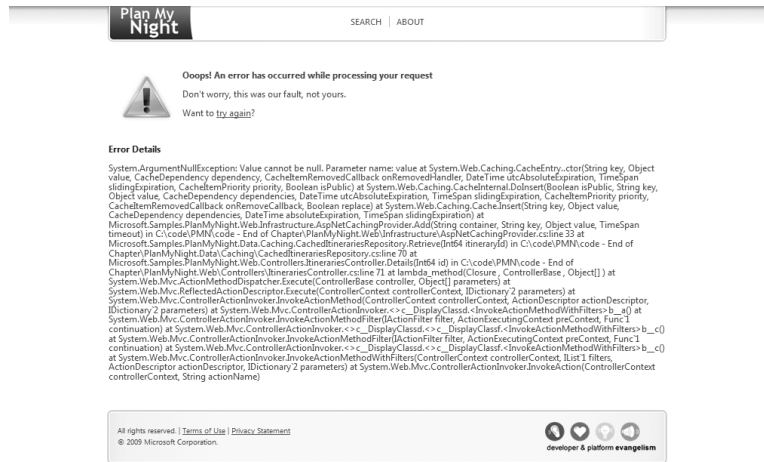
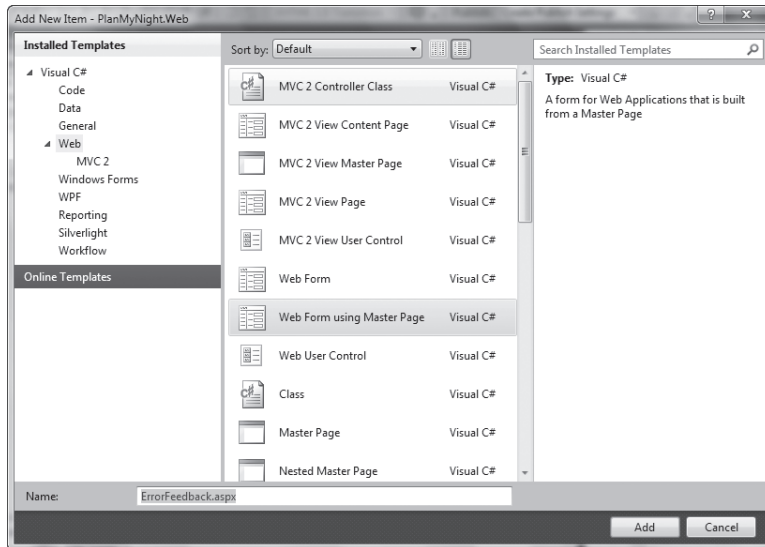


FIGURE 6-9 Example of an error screen in the Plan My Night application

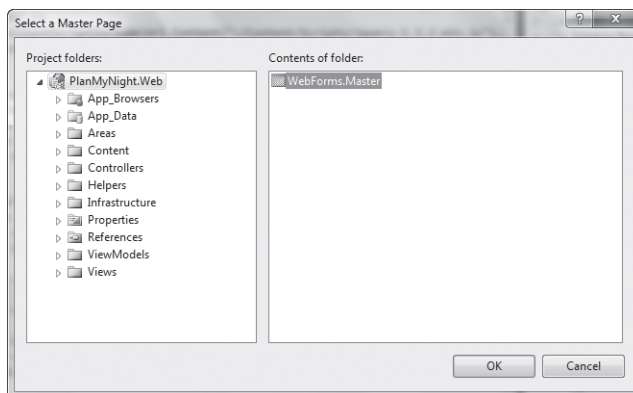
Currently, a user who sees this screen really has only the option of trying his action again or using the navigation links along the top area of the application. (Of course, that might also cause another error.) Adding an option for the user to provide feedback allows the developers to gain information about the situation that might not be apparent by using the standard exception message and stack trace. To show a different way to create a user interface component for Plan My Night, the error feedback page is going to be created as an ASP.NET Web Form using primarily the Designer view in Visual Studio. Before you can begin designing the form, you need to create a base form file to work from.

To create a new Web form:

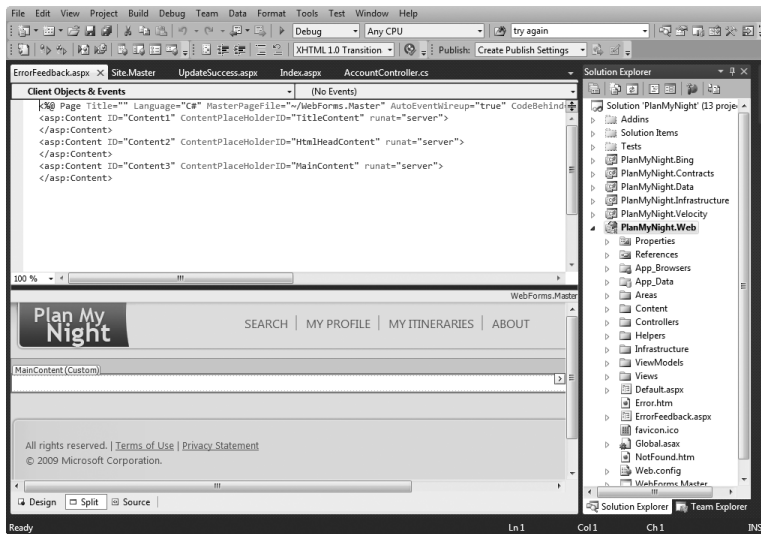
1. Open the context menu on the PlanMyNight.Web project (by clicking the right mouse button), open the Add submenu, and select New Item.
2. In the Add New Item dialog box, select Web Form Using Master Page and call the item **ErrorFeedback.aspx** in the Name field.



3. The dialog screen to associate a master page with this Web form will appear. On the Project Folders side, ensure that the main PlanMyNight.Web folder is selected and then select the WebForms.Master item on the right.



- The resulting page can be shown in the source mode (or Design view) instead of Split view. Switch the view to Split (located at the bottom of the window, just like in previous Visual Studio versions). When you are done, the screen should look similar to this:



Note Split view is recommended so that you can see the source the designer is generating and to add extra markup as needed.

It's a good idea to pin the control toolbox open on the screen because you'll be dragging controls and elements to the content area during this section. The toolbox, if not present already, can be found under the View menu.

Start by dragging a `div` element (under the HTML group) from the toolbox into the `MainContent` section of the designer. A `div` tab will appear, indicating that the new element you added is the currently selected element. Open the context menu for the `div`, and choose Properties (which can also be opened by pressing the F4 key). With the Properties window open, edit the (*Id*) property to have a value of `profileForm`. (Casing is important.) Also, change the *Class* property to have a value of `panel`. After editing the values, the size of your content area will have changed, because CSS is applied in the Design view.

Visual Studio 2005 A much-needed update to the Web Forms designer surface from Visual Studio 2005 is the application of CSS. This allows the developer to see in real-time how the style changes are applied, without having to run the application. When viewed in Visual Studio 2005, the designer for the search.aspx page will appear similar to Figure 6-10.

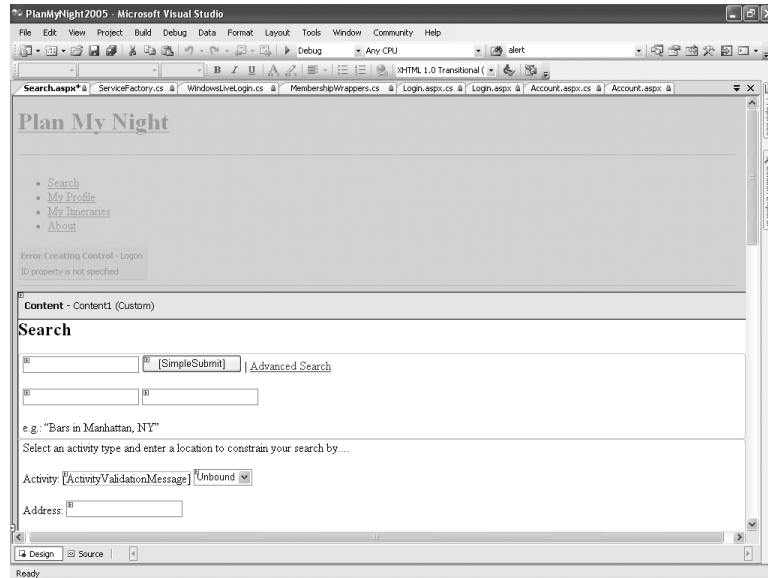


FIGURE 6-10 Designer view of an ASP.NET Web page in Visual Studio 2005

Drag another div inside the first one, and set its *class* property to *innerPanel*. In the markup panel, add the following markup to the *innerPanel*:

```
<h2><span>Error Feedback</span></h2>
```

After the close of the `<h2>` tag, add a new line and open the context menu. Choose Insert Snippet, and follow the click path of ASP.NET > formr. This will create a server-side form tag for you to insert Web controls into. Inside the form tag, place a div tag with the class attribute set to *items* and then a fieldset tag inside the div tag.

Next drag a TextBox control (found under Standard) from the toolbox and drop it inside the fieldset tag. Set the ID of the text box to **FullName**. Add a `<label>` tag before this control in the markup view, set its *for* property to the ID of the text box, and set its value to **Full Name:** (making sure to include the colon). To set the value of a `<label>` tag, place the text between the `<label>` and `</label>` tags. Surround these two elements with a `<p>`, and you should have something like Figure 6-11 in the Design view.

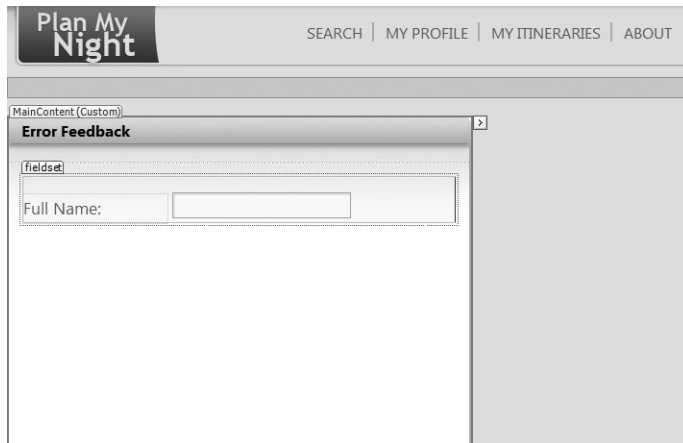


FIGURE 6-11 Current state of ErrorFeedback.aspx in the Design view

Add another text box and label it in a similar manner as the first, but set the ID of the text box to **EmailAddress** and the label value to **Email Address:** (making sure to include the colon). Repeat the process a third time, setting the TextBox ID and label value to **Comments**. There should now be three labels and three single-line TextBox controls in the Design view. The Comments control needs multiline input, so open its property page and set *TextMode* to *Multiline*, *Rows* to *5*, and *Columns* to *40*. This should create a much wider text box in which the user can enter comments.

Use the Insert Snippet feature again, after the Comments text box, and insert a “div with class” tag (HTML>divc). Set the class of the div tag to *submit*, and drag a Button control from the toolbox into this div. Set the Button’s *Text* property to *Send Feedback*.

The designer should show something similar to what you see in Figure 6-12, and at this point you have a page that will submit a form.

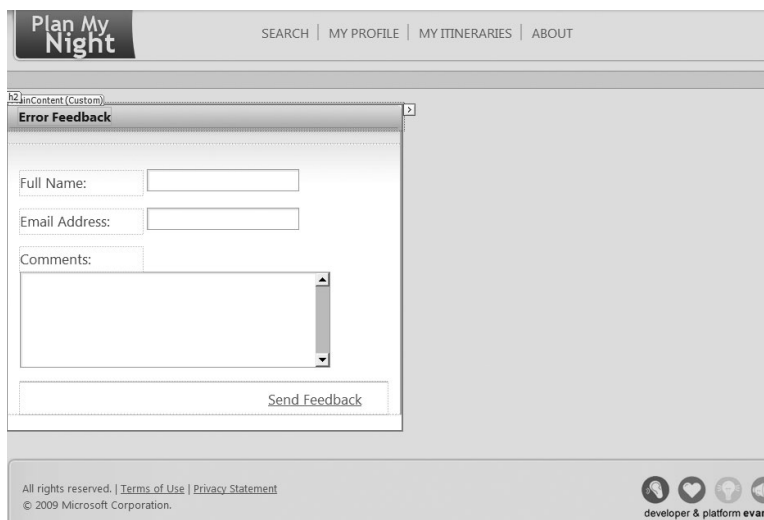


FIGURE 6-12 The ErrorFeedback.aspx form with a complete field set

However, it does not perform any validation on the data being submitted. To do this, you'll take advantage of some of the validation controls present in ASP.NET. You'll make the Full Name and Comments boxes required fields and perform a regex validation of the e-mail address to ensure that it matches the right pattern.

Under the Validation group of the toolbox are some premade validation controls you'll use. Drag a *RequiredFieldValidator* object from the toolbox, and drop it to the right of the Full Name text box. Open the properties for the validation control, and set the *ControlToValidate* property to *FullName*. (It's a drop-down list of controls on the page.) Also, set the *CssClass* to *field-validation-error*. This changes the display of the error to a red triangle used elsewhere in the application. Finally, change the Error Message property to **Name is Required**. (See Figure 6-13.)

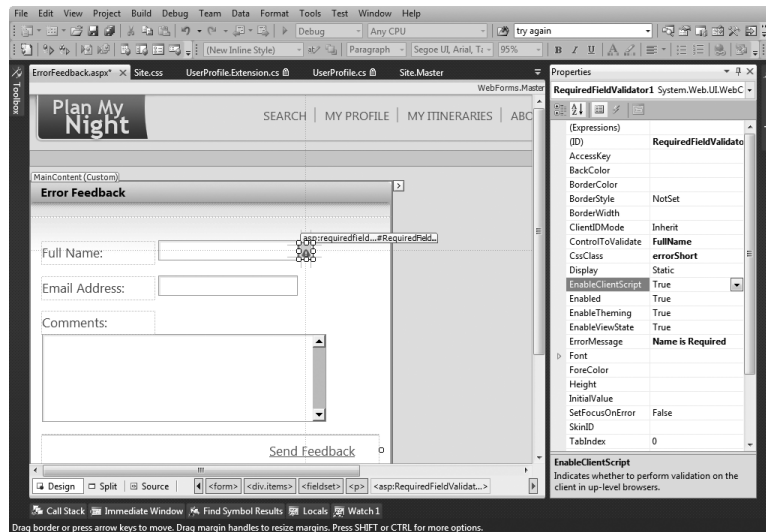


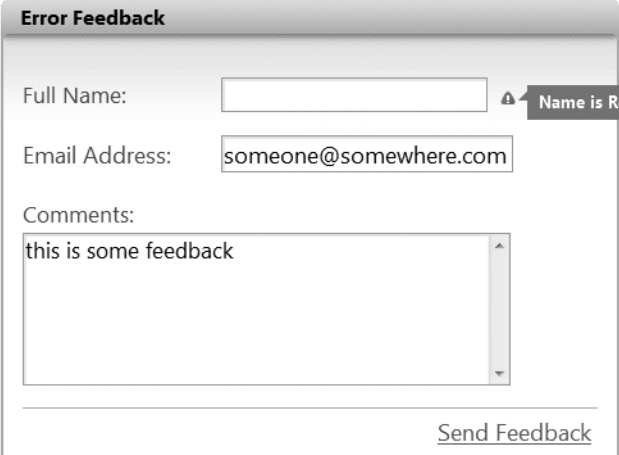
FIGURE 6-13 Validation control example

Repeat these steps for the Comments box, but substitute the *ErrorMessage* and *ControlToValidate* property values as appropriate.

For the Email Address field, you want to make sure the user types in a valid e-mail address, so for this field drag a *RegularExpressionValidator* control from the toolbox and drop it next to the Email Address text box. The property values are similar for this control in that you set the *ControlToValidate* property to *EmailAddress* and the *CssClass* property to *field-validation-error*. However, with this control you define the regular expression to be applied to the input data. This is done with the *ValidationExpression* property, and it should be set like this:

```
[A-Za-z0-9_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}
```


4. With the error screen visible, click the link to go to the feedback form. Try to submit the form with invalid data.



The screenshot shows a web form titled "Error Feedback". It has three input fields: "Full Name:" (empty), "Email Address:" (filled with "someone@somewhere.com"), and "Comments:" (filled with "this is some feedback"). A red error message "Name is Required" with a warning icon is displayed next to the Full Name field. A "Send Feedback" button is at the bottom right.

ASP.NET uses client-side script (when the browser supports it) to perform the validation, so no postbacks occur until the data passes. On the server side, when the server does receive a postback, a developer can check the validation state with the *Page.IsValid* property in the code-behind. However, because you used client-side validation (which is on by default), this will always be *true*. The only code in the code-behind that needs to be added is to redirect the user on a postback (and check the *Page.IsValid* property, in case client validation missed something):

```
protected void Page_Load(object sender, EventArgs e)
{
    if (this.IsPostBack && this.IsValid)
    {
        this.Response.Redirect("/", true);
    }
}
```

This really isn't very useful to the user, but our goal in this section was to work with the designer to create an ASP.NET Web Form. This added a new interface to the PlanMyNight .Web project, but what if you wanted to add new functionality to the application in a more modular sense, such as some degree of functionality that can be added or removed without having to compile the main application project. This is where an extensibility framework like the Managed Extensibility Framework (MEF) can show the benefits it brings.

Extending the Application with MEF

A new technology available in Visual Studio 2010 as part of the .NET Framework 4 is the Managed Extensibility Framework (MEF). The Managed Extensibility Framework provides developers with a simple (yet powerful) mechanism to allow their applications to be extended by third parties after the application has been shipped. Even within the same application, MEF allows developers to create applications that completely isolate components, allowing them to be managed or changed independently. It uses a resolution container to map components that provide a particular function (exporters) and components that require that functionality (importers), without the two concrete components having to know about each other directly. Resolutions are done on a contract basis only, which easily allows components to be interchanged or introduced to an application with very little overhead.

See Also MEF's community Web site, containing in-depth details about the architecture, can be found at <http://mef.codeplex.com>.

The companion Plan My Night application has been designed with extensibility in mind, and it has three "add-in" module projects in the solution, under the Addins solution folder. (See Figure 6-14.)

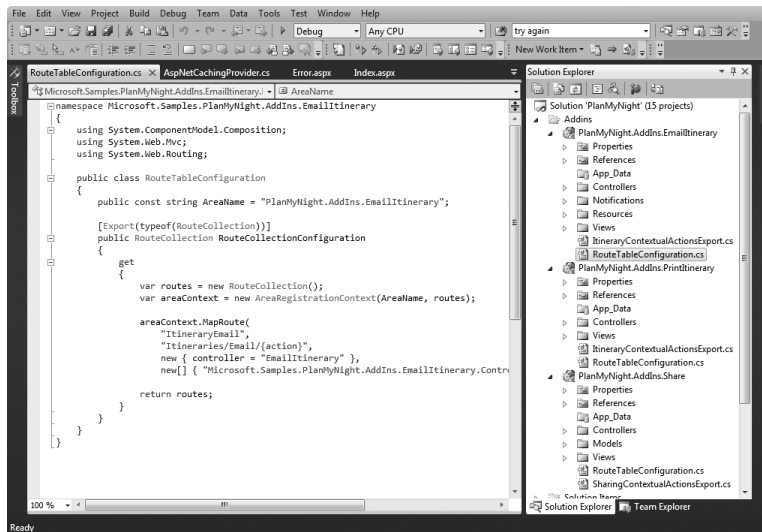


FIGURE 6-14 The Plan My Night application add-ins

PlanMyNight.Addins.EmailItinerary adds the ability to e-mail itinerary lists to anyone the user sees fit to receive them. PlanMyNight.Addins.PrintItinerary provides a printer-friendly view of the itinerary. Lastly, PlanMyNight.Addins.Share adds in social-media sharing functions (so that the user can post a link to an itinerary) as well as URL-shortening operations. None of

these projects reference the main PlanMyNight.Web application or are referenced by it. They do have references to the PlanMyNight.Contracts and PlanMyNight.Infrastructure projects, so they can export (and import in some cases) the correct contracts via MEF as well as use any of the custom extensions in the infrastructure project.



Note Before doing the next step, if the Web application is not already running, launch the PlanMyNight.Web project so that the UI is visible to you.

To add the modules to your running application, run the DeployAllAddins.bat file, found in the same folder as the PlanMyNight.sln file. This will create new folders under the Areas section of the PlanMyNight.Web project. These new folders, one for each plug-in, will contain the files needed to add their functionality to the main Web application. The plug-ins appear in the application as extra options under the current itinerary section of the search results page and on the itinerary details page. After the batch file is finished running, go to the interface for PlanMyNight, search for an activity, and add it to the current itinerary. You should notice some extra options under the itinerary panel other than just New and Save. (See Figure 6-15.)

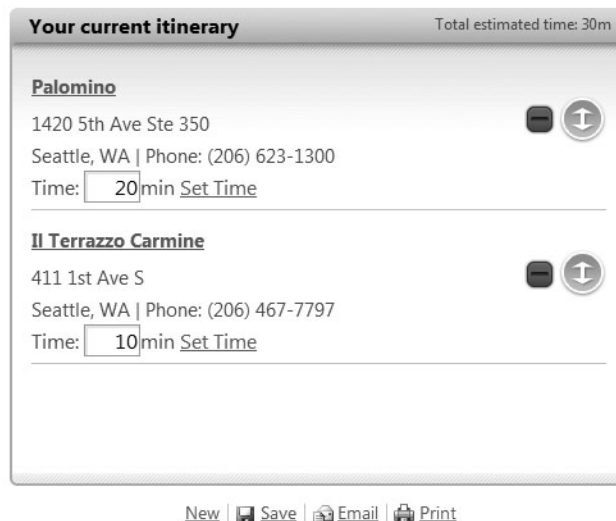


FIGURE 6-15 Location of the e-mail add-in in the UI

The social sharing options will show in the interface only after the itinerary is saved and marked public. (See Figure 6-16.)

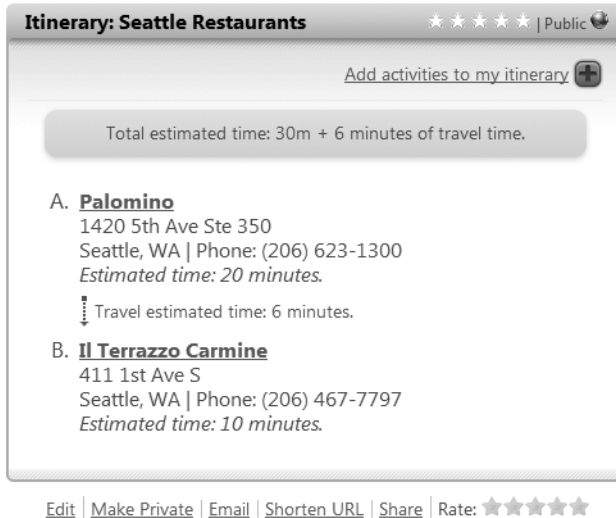


FIGURE 6-16 Location of the social-sharing add-in in the UI

Visual Studio 2005 Visual Studio 2005 does not have anything that compares to MEF. To support plug-ins, a developer would have to either write the plug-in framework from scratch or purchase a commercial package. Either of the two options led to proprietary solutions an external developer would have to understand in order to create a component for them. Adding MEF to the .NET Framework helps to cut down the entry barriers to producing extendible applications and the plug-in modules for them.

Print Itinerary Add-in Explained

To demonstrate how these plug-ins wire into the application, let's have a look at the `PrintItinerary.Addin` project. When you expand the project, you should see something like the structure shown in Figure 6-17.

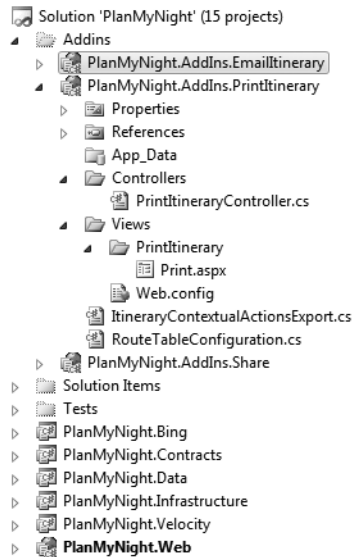


FIGURE 6-17 Structure of the PrintItinerary project

Some of this structure is similar to the PlanMyNight.Web project (Controllers and Views). That's because this add-in will be placed in an MVC application as an area. If you look more closely at the PrintItineraryController.cs file in the Controller folder, you can see it is similar in structure to the controller you created earlier in this chapter (and similar to any of the other controllers in the Web application). However, some key differences set it apart from the controllers that are compiled in the primary PlanMyNight.Web application.

Focusing on the class definition, you'll notice some extra attributes:

```
[Export("PrintItinerary", typeof(ILogger))]
[PartCreationPolicy(CreationPolicy.NonShared)]
```

These two attributes describe this type to the MEF resolution container. The first attribute, *Export*, marks this class as providing an *ILogger* under the contract name of *PrintItinerary*. The second attribute declares that this object supports only nonshared creation and cannot be created as a shared/singleton object. Defining these two attributes are all you need to do to have the type used by MEF. In fact, *PartCreationPolicy* is an optional attribute, but it should be defined if the type cannot handle all the creation policy types.

Further into the `PrintItineraryController.cs` file, the constructor is decorated with an *ImportingConstructor* attribute:

```
[ImportingConstructor]
public PrintItineraryController(IServiceFactory serviceFactory) :
this(
    serviceFactory.GetItineraryContainerInstance(),
    serviceFactory.GetItinerariesRepositoryInstance(),
    serviceFactory.GetActivitiesRepositoryInstance())
{
}
```

The *ImportingConstructor* attribute informs MEF to provide the parameters when creating this object. In this particular case, MEF provides an instance of *IServiceFactory* for this object to use. Where the instance comes from is of no concern to the *this* class and really assists with creating modular applications. For our purposes, the *IServiceFactory* contracted is being exported by the `ServiceFactory.cs` file in the `PlanMyNight.Web` project.

The `RouteTableConfiguration.cs` file registers the URL route information that should be directed to the `PrintItineraryController`. This route, and the routes of the other add-ins, are registered in the application during the *Application_Start* method in the `Global.asax.cs` file of `PlanMyNight.Web`:

```
// MEF Controller factory
var controllerFactory = new MefControllerFactory(container);
ControllerBuilder.Current.SetControllerFactory(controllerFactory);

// Register routes from Addins
foreach (RouteCollection routes in container.GetExportedValues<RouteCollection>())
{
    foreach (var route in routes)
    {
        RouteTable.Routes.Add(route);
    }
}
```

The *controllerFactory*, which was initialized with an MEF container containing path information to the `Areas` subfolder (so that it enumerated all the plug-ins), is assigned to be the controller factory for the lifetime of the application. This allows controllers imported via MEF to be usable anywhere in the application. The routes these plug-ins respond to are then retrieved from the MEF container and registered in the MVC routing table.

The `ItineraryContextualActionsExport.cs` file exports information to create the link to this plug-in, as well as metadata for displaying it. This information is used in the

ViewModelExtensions.cs file, in the PlanMyNight.Web project, when building a view model for display to the user:

```
// get addin links and toolboxes
var addinBoxes = new List<RouteValueDictionary>();
var addinLinks = new List<ExtensionLink>();

addinBoxes.AddRange(AddinExtensions.GetActionsFor("ItineraryToolbox", model.Id == 0 ? null :
new { id = model.Id }));

addinLinks.AddRange(AddinExtensions.GetLinksFor("ItineraryLinks", model.Id == 0 ? null : new
{ id = model.Id }));
```

The call to *AddinExtensions.GetLinksFor* enumerates over exports in the MEF Export provider and returns a collection of them to be added to the local *addinLinks* collection. These are then used in the view to display more options when they are present.

Summary

In this chapter, we explored a few of the many new features and technologies found in Visual Studio 2010 that were used to create the companion Plan My Night application. We walked through creating a controller and its associated view and how the ASP.NET MVC framework offers Web developers a powerful option for creating Web applications. We also explored how using the Managed Extensibility Framework in application design can allow plug-in modules to be developed external to the application and loaded at run time. In the next chapter, we'll explore how debugging applications has been improved in Visual Studio 2010.

Chapter 7

From 2005 to 2010: Debugging an Application

After reading this chapter, you will be able to

- Use the new debugger features of Microsoft Visual Studio 2010
- Create unit tests and execute them in Visual Studio 2010
- Compare what was available to you as a developer in Visual Studio 2005

As we were writing this book, we realized how much the debugging tools and developer aids have evolved over the last three versions of Visual Studio. Focusing on debugging an application and writing unit tests just increases the opportunities we have to work with Visual Studio 2010.

Visual Studio 2010 Debugging Features

In this chapter, you'll go through the different debugging features using a modified Plan My Night application. If you installed the companion content at the default location, you'll find the modified Plan My Night application at the following location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 7\Code. Double-click the PlanMyNight.sln file.

First, before diving into the debugging session itself, you'll need to set up a few things:

1. In Solution Explorer, ensure that PlanMyNight.Web is the startup project. If the project name is not in bold, right-click on PlanMyNight.Web and select Set As StartUp Project.
2. To get ready for the next steps, in the PlanMyNight.Web solution open the Global.asax.cs file by clicking the triangle beside the Global.asax folder and then double-clicking the Global.asax.cs file, as shown in Figure 7-1:

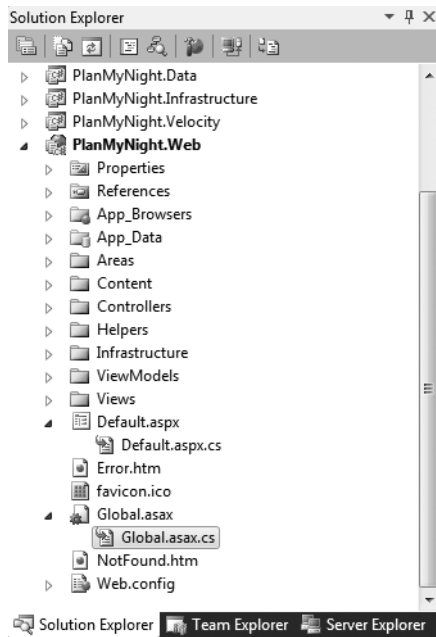


FIGURE 7-1 Solution Explorer before opening the file Global.asax.cs

Managing Your Debugging Session

Using the Plan My Night application, you'll examine how a developer can manage and share breakpoints. And with the use of new breakpoint enhancements, you'll learn how to inspect the different data elements in the application in a much faster and more efficient way. You'll also look at new minidumps and the addition of a new intermediate language (IL) interpreter that allows you to evaluate managed code properties and functions during minidump debugging.

New Breakpoint Enhancements

At this point, you have the Global.ascx.cs file opened in your editor. The following steps walk you through some ways to manage and share breakpoints:

1. Navigate to the *Application_BeginRequest(object sender, EventArgs e)* method, and set a breakpoint on the line that reads *var url = HttpContext.Current.Request.Url;* by clicking in the left margin or pressing F9. Look at Figure 7-2 to see this in action:

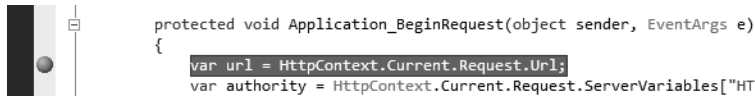


FIGURE 7-2 Creating a breakpoint

2. Press F5 to start the application in debug mode. You should see the developer Web server starting in the system tray and a new browser window opening. The application should immediately stop at the breakpoint you just created. The Breakpoints window might not be visible even after starting the application in debug mode. If that is the case, you can make it visible by going to the Debug menu and selecting Windows and then Breakpoints, or you can use the keyboard shortcut: Ctrl+D+B.

You should now see the Breakpoints window as shown in Figure 7-3:

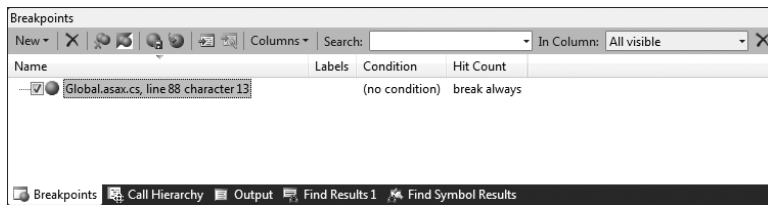


FIGURE 7-3 Breakpoints window

3. In the same method, add three more breakpoints so that the editor and the Breakpoints window look like those shown in Figure 7-4:

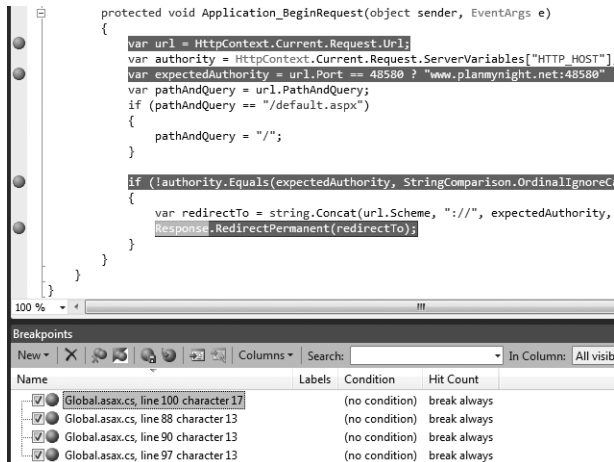


FIGURE 7-4 Code editor and Breakpoints window with three new breakpoints

Visual Studio 2005 As a reader and a professional developer who used Visual Studio 2005 often, you probably noticed a series of new buttons as well as new fields in the Breakpoints window in this exercise. As a reminder, take a look at Figure 7-5 for a quick comparison of what it looks like in Visual Studio 2005.

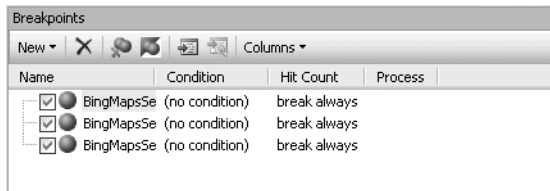


FIGURE 7-5 Visual Studio 2005 Breakpoints window

4. Notice that the Labels column is now available to help you index and to search breakpoints. It is a really nice and useful feature that Visual Studio 2010 brings to the table. To use this feature, you simply right-click on a breakpoint in the Breakpoints window and select Edit Labels or use the keyboard shortcut Alt+F9, L, as shown in Figure 7-6:

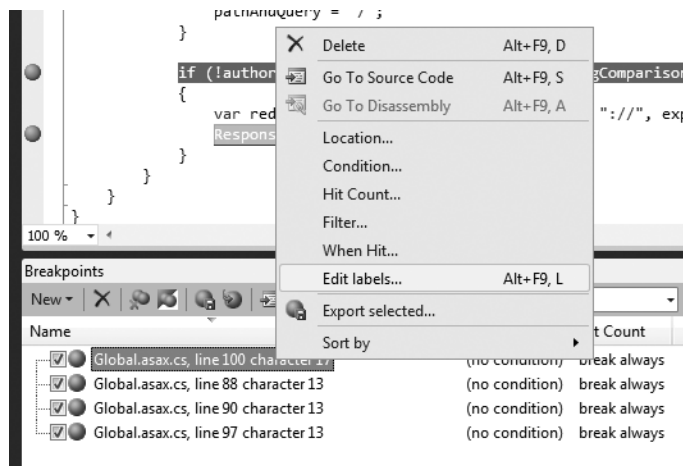


FIGURE 7-6 Edit Labels option

5. In the Edit Breakpoint Labels window, add labels for the selected breakpoint (which is the first one in the Breakpoints window). Type **ContextRequestUrl** in the Type A New Label text box, and click Add. Repeat this operation on the next breakpoint, and type a label name of **url**. When you are done, click OK. You should see a window that

looks like Figure 7-7 while you are entering them, and to the right you should see the Breakpoints window after you are done with those two operations:

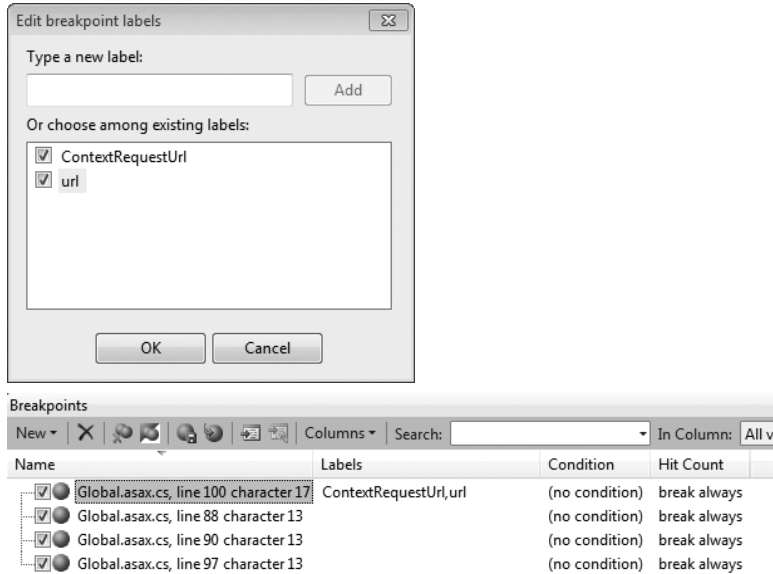


FIGURE 7-7 Adding labels that show up in the Breakpoints window



Note You can also right-click on the breakpoint in the left margin and select Edit Labels to accomplish the same tasks I just outlined.



Note You'll see that when adding labels to a new breakpoint you can choose any of the existing labels you have already entered. You'll find these in the Or Choose Among Existing Labels area, which is shown in the Edit Breakpoint Labels dialog box on the left in the preceding figure.

- Using any of the ways you just learned, add labels for each of the breakpoints, and make sure your Breakpoints window looks like Figure 7-8 after you're done.

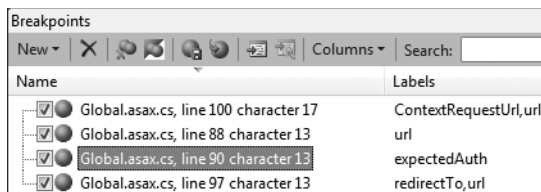


FIGURE 7-8 Breakpoints window with all labels entered





When you have a lot of code and are in the midst of a debugging session, it would be great to be able to filter the displayed list of breakpoints. That's exactly what the new Search feature in Visual Studio 2010 allows you to do.

7. To see the Search feature in action, just type **url** in the search text box and you'll see the list of breakpoints is filtered down to breakpoints containing *url* in one of their labels.

In a team environment where you have many developers and testers working together, often two people at some point in time are working on the same bugs. In Visual Studio 2005, the two people needed to sit near each other, send one another screen shots, or send one another the line numbers of where to put breakpoints to refine where they should look while debugging a particular bug.



Important One of the great new additions to breakpoint management in Visual Studio 2010 is that you can now export breakpoints to a file and then send them to a colleague, who can then import them into his own environment. Another scenario that this feature is useful for is to share breakpoints between machines. We'll see how to do that next.

8. In the Breakpoints window, click the Export button  to export your breakpoints to a file, and then save the file on your desktop. Name the file **breakexports.xml**.
9. Delete all the breakpoints either by clicking the Delete All Breakpoints Matching The Current Search Criteria button  or by selecting all the breakpoints and clicking the Delete The Selected Breakpoints button . The only purpose of deleting them is to simulate two developers sharing them or one developer sharing breakpoints between two machines.
10. You'll now import your breakpoints by clicking the Import button  and loading them from your desktop. Notice that all your breakpoints with all of their properties are back and loaded in your environment. For the purposes of this chapter, delete all the breakpoints.

Inspecting the Data

When you are debugging your applications, you know how much time one can spend stepping into the code and inspecting the content of variables, arguments, and so forth. Maybe you can remember when you were learning to write code, a while ago, when debuggers weren't a reality or when they were really rudimentary. Do you remember (maybe not—you might not be as old as we are) how many *printf* or *WriteLn* statements you had to write to inspect the content of different data elements?

Visual Studio 2005 In Visual Studio 2005, things were already a big improvement from the days of writing all kinds of statements to the console, because Visual Studio had a real debugger with new functionalities. New data visualizers allowed you to see XML as a well-formed XML snippet and not as a long string. Furthermore, with those data visualizers, you could view arrays in a more useful way, with the list of elements and their indices, and you accomplished that by simply hovering your mouse over the object. Take a look at Figure 7-9 for an example:

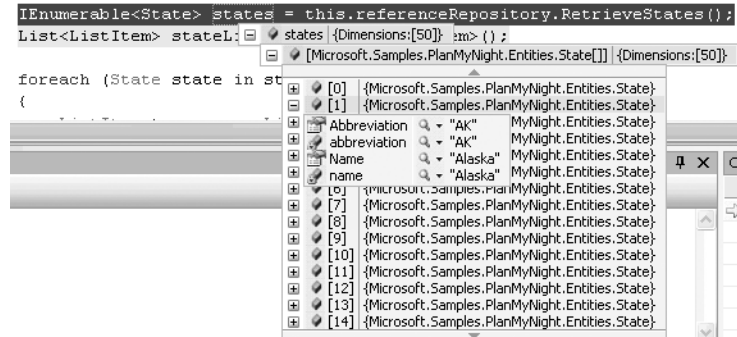


FIGURE 7-9 Collection view versus an array view in the debugger in Visual Studio 2010 and in Visual Studio 2005

Although those DataTip data visualization techniques are still available in Visual Studio 2010, a few great enhancements have been added that make DataTips even more useful. The DataTip enhancements have been added in conjunction with another new feature of Visual Studio 2010, multimonitor support. Floating DataTips can be valuable to you as a developer. Having the ability to put DataTips on a second monitor can make your life a lot easier while debugging, because it keeps the data that always needs to be in context right there on the second monitor. The following steps demonstrate how to use these features:

1. In the Global.ascx.cs file, insert breakpoints on lines 89 and 91, lines starting with the source code `var authority` and `var pathAndQuery`, respectively.
2. You are now going to experiment with the new DataTip features. Start the debugger by pressing F5. When the debugger hits the first breakpoint, move your mouse over the word `url` and click on the pushpin, as seen in Figure 7-10:

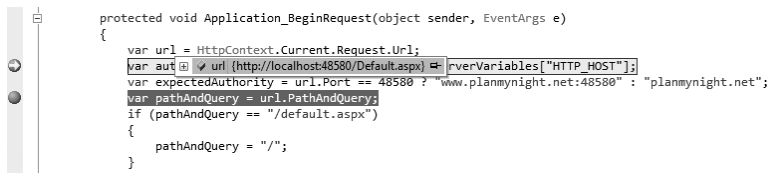



FIGURE 7-10 The new DataTip pushpin feature

- To the right of the line of code, you should see the pinned DataTip (as seen in the following figure on the left). If you hover your mouse over the DataTip, you'll get the DataTip management bar (as seen in Figure 7-11 on the right):



FIGURE 7-11 On the left is the pinned DataTip, and on the right is the DataTip management bar



Note You should also see in the breakpoint gutter a blue pushpin indicating that the DataTip is pinned. The pushpin should look like this: . Because you have a breakpoint on that line, the pushpin is actually underneath it. To see the pushpin, just toggle the breakpoint by clicking on it in the gutter. Toggle once to disable the breakpoint and another time to get it back.



Note If you click the double arrow pointing down in the DataTip management bar, you can insert a comment for this DataTip, as shown in Figure 7-12. You can also remove the DataTip altogether by clicking the X button in the DataTip management bar.

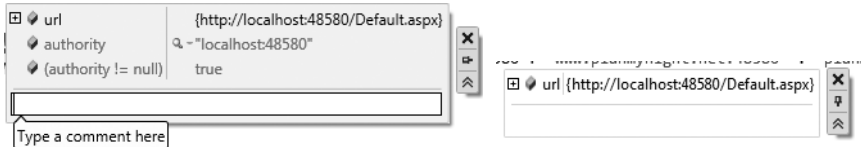


FIGURE 7-12 Inserting a comment for a DataTip

- One nice feature of the new DataTip is that you can insert any expression to be evaluated right there in your debugging session. For instance, right-click on the DataTip name, in this case *url*, select **Add Expression**, type **authority**, and then add another one like this: **(authority != null)**. You'll see that the expressions are evaluated immediately and will continue to be evaluated for the rest of the debugging session every time your debugger stops on those breakpoints. At this point in the debugging session, the expression should evaluate to *null* and *false*, respectively.
- Press F10 to execute the line where the debugger stopped, and look at the url DataTip as well as both expressions. They should contain values based on the current context, as shown in Figure 7-13:

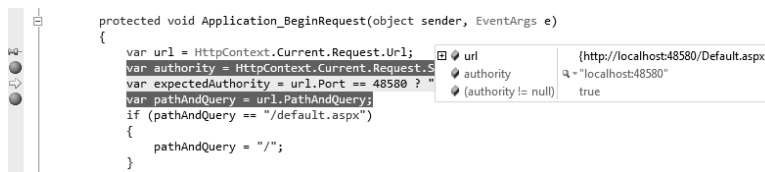


FIGURE 7-13 The url pinned DataTip with the two evaluated expressions

6. Although it is nice to be able to have a mini-watch window where it matters—right there where the code is executing—you can also see that it is superimposed on the source code being debugged. Keep in mind that you can move the DataTip window anywhere you want in the code editor by simply dragging it, as illustrated in Figure 7-14:

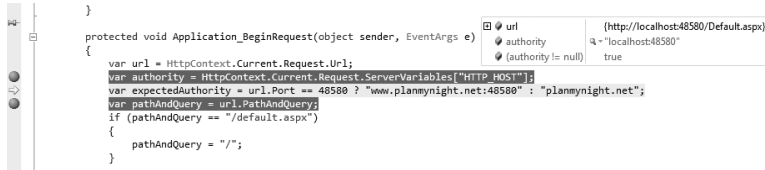


FIGURE 7-14 Move the pinned DataTip away from the source code

7. Because it is pinned, the DataTip window stays where you pinned it, so it will not be in view if you trace into another file. But in some cases, you need the DataTip window to be visible at all times. For instance, keeping it visible is interesting for global variables that are always in context or for multimonitor scenarios. To move a DataTip, you have to first unpin it by clicking the pushpin in the DataTip management bar. You'll see that it turns yellow. That indicates you can now move it wherever you want—for instance, over Solution Explorer, to a second monitor, over your desktop, or to any other window. Take a look at Figure 7-15 for an example:

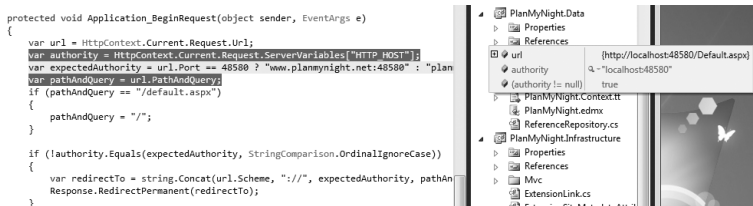



FIGURE 7-15 Unpinned DataTip over Solution Explorer and the Windows desktop



Note If the DataTip is not pinned, the debugger stops in another file and method, and the DataTip contains items that are out of context, the DataTip windows will look like Figure 7-16. You can retry to have the debugger evaluate the value of an element by clicking on this button: . However, if that element has no meaning in this context, it's possible that nothing happens.

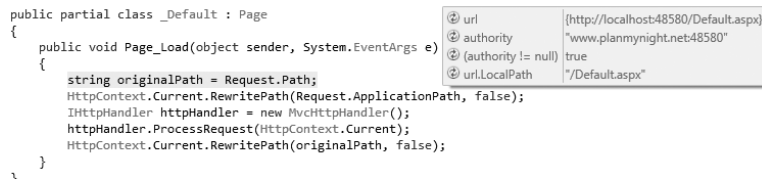


FIGURE 7-16 DataTip window with out-of-context items



Note You'll get an error message if you try to pin outside the editor, as seen in Figure 7-17:

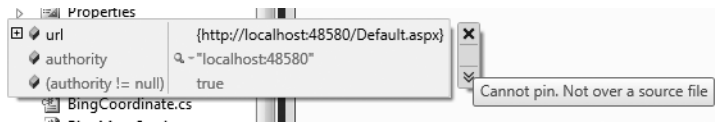


FIGURE 7-17 Error message that appears when trying to pin a DataTip outside the code editor



Note Your port number might be different than in the screen shots just shown. This is normal—it is a random port used by the personal Web server included with Visual Studio.



Note You can also pin any child of a pinned item. For instance, if you look at url and expand its content by pressing the plus sign (+), you'll see that you can also pin a child element, as seen in Figure 7-18:

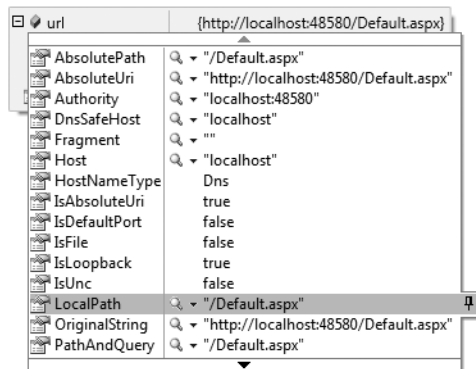



FIGURE 7-18 Pinned child element within the url DataTip

- Before stopping the debugger, go back to the `Global.ascs.cs` if you are not already there and re-pin the DataTip window. Then stop the debugging session by clicking the Stop Debugging button in the debug toolbar () or by pressing Shift+F5. Now if you hover your mouse over the blue pushpin in the breakpoint gutter, you'll see the values from the last debug session, which is a nice enhancement to the watch window. Take a look at Figure 7-19 for what you should see:

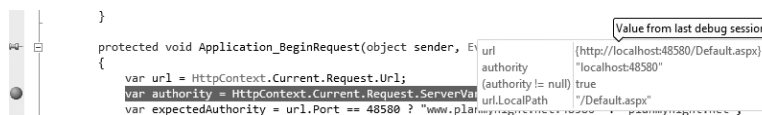


FIGURE 7-19 Values from the last debug session for a pinned DataTip



Note As with the breakpoints, you can export or import the DataTips by going to the Debug menu and selecting Export DataTips or Import DataTips, respectively.

Using the Minidump Debugger

Many times in real-world situations, you'll have access to a minidump from your product support team. Apart from their bug descriptions and repro steps, it might be the only thing you have to help debug a customer application. Visual Studio 2010 adds a few enhancements to the minidump debugging experience.

Visual Studio 2005 In Visual Studio 2005, you could debug managed application or minidump files, but you had to use an extension if your code was written in managed code. You had to use a tool called SOS and load it in the debugger using the Immediate window. You had to attach the debugger both in native and managed mode, and you couldn't expect to have information in the call stack or Locals window. You had to use commands for SOS in the Immediate window to help you go through minidump files. With application written in native code, you used normal debugging windows and tools. To read more about this or just to refresh your knowledge of the topic, you can read the *Bug Slayer* column in MSDN magazine here: <http://msdn.microsoft.com/en-us/magazine/cc164138.aspx>.

Let's see the new enhancements to the minidump debugger. First you need to create a crash from which you'll be able to generate a minidump file:

1. In Solution Explorer in the PlanMyNight.Web project, rename the file Default.aspx to **DefaultA.aspx**. Note the *A* appended to the word "Default."
2. Make sure you have no breakpoints left in your project. To do that, look in the Breakpoints window and delete any breakpoints left there using any of the ways you learned earlier in the chapter.
3. Press F5 to start debugging the application. Depending on your machine speed, soon after the build process is complete you should see an unhandled exception of type *HttpException*. Although the bug is simple in this case, let's go through the steps of creating the minidump file and debugging it. Figure 7-20 shows what you should see at this point:

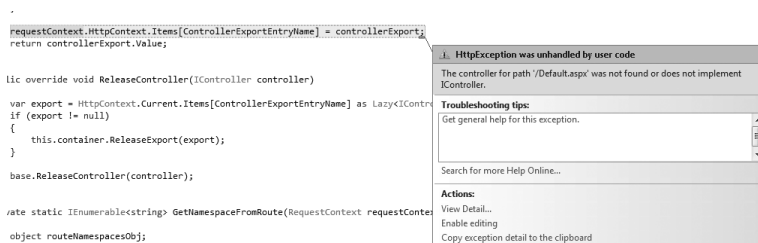


FIGURE 7-20 The unhandled exception you should expect

4. It is time to create the minidump file for this exception. Go to the Debug menu, and select Save Dump As, as seen in Figure 7-21. You should see the name of the process from which the exception was thrown. In this case, the process from which the exception was thrown was Cassini or the Personal Web Server in Visual Studio. Keep the file name proposed (WebDev.WebServer40.dmp), and save the file on your desktop. Note that it might take some time to create the file because the minidump file size will be close to 300 MB.

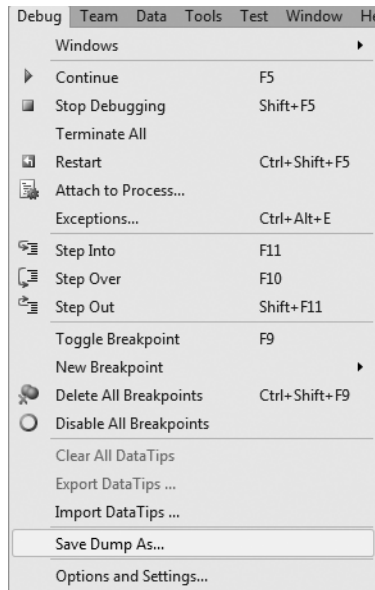


FIGURE 7-21 Saving the minidump file

5. Stop Debugging by pressing Shift+F5 or the Stop Debugging button.
6. Next, go to the File menu and close your solution.
7. In the File menu, click Open and point to the desktop to load your minidump file named WebDev.WebServer40.dmp. Doing so opens the Minidump File Summary page, which gives you some summary information about the bug you are trying to fix. (Figure 7-22 shows what you should see.) Before you start to debug, you'll get basic information from that page such as the following: process name, process architecture, operating system version, CLR version, modules loaded, as well as some actions you can take from that point. From this place, you can set the paths to the symbol files. Conveniently, the Modules list contains the version and path on disk of your module, so finding the symbols and source code is easy. The CLR version is 4.0; therefore, you can debug here in Visual Studio 2010.

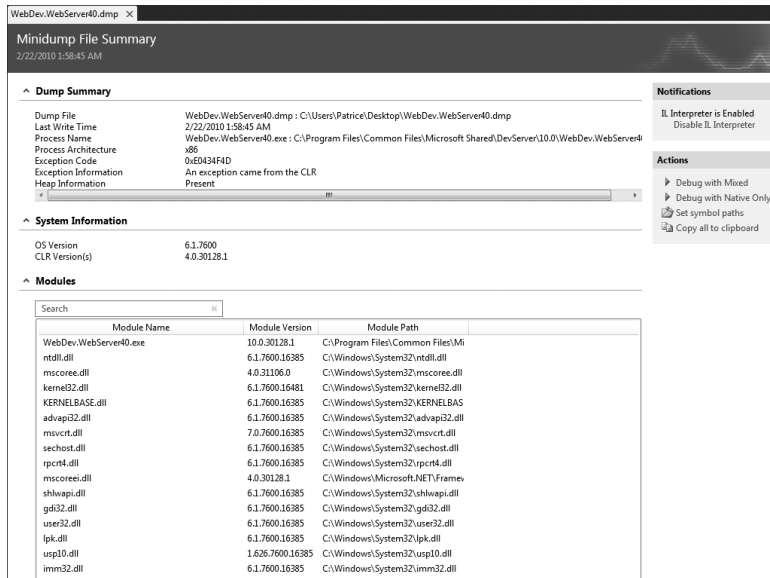


FIGURE 7-22 Minidump summary page

- To start debugging, locate the Actions list on the right side of the Minidump File Summary page and click Debug With Mixed.
- You should see almost immediately a first-chance exception like the one shown in Figure 7-23. In this case, it tells you what the bug is; however, this won't always be the case. Continue by clicking the Break button.

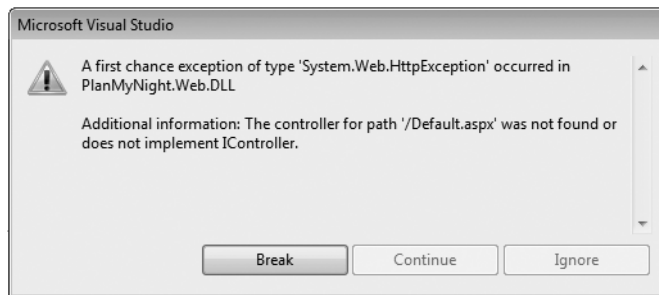


FIGURE 7-23 First-chance exception

- You should see a green line indicating which instruction caused the exception. If you look at the source code, you'll see in your Autos window that the *controllerExport* variable is *null*, and that just before that we specified that if the variable was *null* we wanted to have an *HttpException* thrown if the file to load was not found. In this case, the file to look for is *Default.aspx*, as you can see in the Locals window in the *controllerName* variable. You can glance at many other variables, objects, and so forth in the Locals and Autos windows containing the current context. Here, you have only

one call that belongs to your code, so the call stack indicates that the code before and after is external to your process. If you had a deeper chain of calls in your code, you could step back and forth in the code and look at the variables. Figure 7-24 shows a summary view of all that:

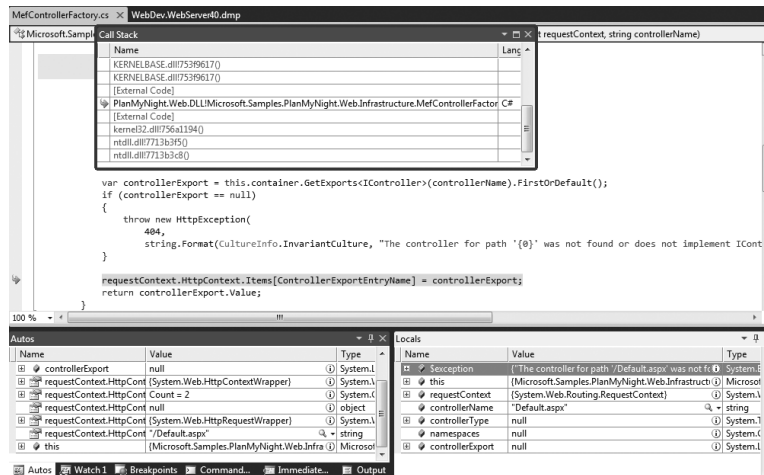


FIGURE 7-24 Autos, Locals, and Call Stack windows, and the next instruction to execute

- OK, you found the bug, so stop the debugging by pressing Shift+F5 or clicking the Stop Debugging button. Then fix the bug by reloading the PlanMyNight solution and renaming the file back to **default.aspx**. Then rebuild the solution by going to the Build menu and selecting Rebuild Solution. Then press F5, and the application should be working again.

Web.Config Transformations

This next new feature, while small, is one that will delight many developers because it saves them time while debugging. The feature is the Web.Config transformations that allow you to have transform files that show the differences between the debug and release environments. As an example, connection strings are often different from one environment to the other; therefore, by creating transform files with the different connection strings—because ASP.NET provides tools to change (transform) web.config files—you'll always end up with the right connection strings for the right environment. To learn more about how to do this, take a look at the following article on MSDN: <http://go.microsoft.com/fwlink/?LinkId=125889>.

Creating Unit Tests

Most of the unit test framework and tools are unchanged in Visual Studio 2010 Professional. It is in other versions of Visual Studio 2010 that the change in test management and test tools is really apparent. Features such as UI Unit Tests, IntelliTrace, and Microsoft Test Manager 2010 are available in other product versions, like Visual Studio 2010 Premium and

Visual Studio 2010 Ultimate. To see which features are covered in the Application Lifecycle Management and for more specifics, refer to the following article on MSDN: [http://msdn.microsoft.com/en-us/library/ee789810\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/ee789810(VS.100).aspx).

Visual Studio 2005 With Visual Studio 2005, you had to own either Visual Studio 2005 Team System or Visual Studio 2005 Team Test to have the ability to create and execute tests out of the box within Visual Studio 2005. Another option back then was to go with a third-party option like NUnit.

In this part of the chapter, we'll show you how to add a unit test for a class you'll find in the Plan My Night application. We won't spend time defining what a unit test is or what it should contain; rather, we'll show you within Visual Studio 2010 how to add tests and execute them.

You'll add unit tests to the Plan My Night application for the Print Itinerary Add-in. To create unit tests, open the solution from the companion content folder. If you do not remember how to do this, you can look at the first page of this chapter for instructions. After you have the solution open, just follow these steps:

1. In Solution Explorer, expand the project PlanMyNight.Web and then expand the Helpers folder. Then double-click on the file ViewHelper.cs to open it in the code editor. Take a look at Figure 7-25 to make sure you are at the right place:

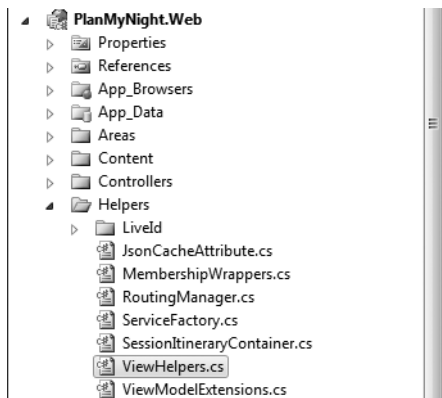


FIGURE 7-25 The PlanMyNight.Web project and ViewHelper.cs file in Solution Explorer

2. In the code editor, you can add unit tests in two different ways. You can right-click on a class name or on a method name and select Create Unit Tests. You can also go to the Test menu and select New Test. We'll explore the first way of creating unit tests. This way Visual Studio automatically generates some source code for you. Right-click on the *GetFriendlyTime* method, and select Create Unit Tests. Figure 7-26 shows what it looks like:

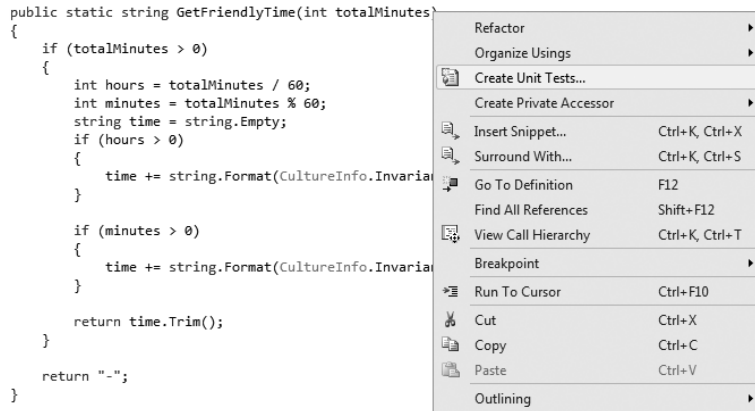


FIGURE 7-26 Contextual menu to create unit tests from by right-clicking on a class name

- After selecting Create Unit Tests, you'll be presented with a dialog that, by default, shows the method you selected from that class. To select where you want to create the unit tests, click on the drop-down combo box at the bottom of this dialog and select `PlanMyNight.Web.Tests`. If you didn't have an existing location, you would have simply selected Create A New Visual C# Test Project from the list. Figure 7-27 shows what you should be seeing:

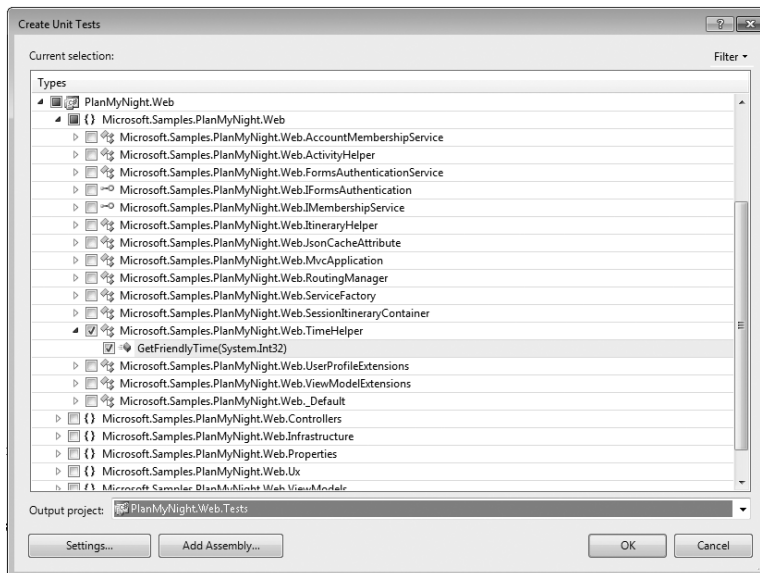


FIGURE 7-27 Selecting the method you want to create a unit test against

- After you click OK, the dialog switches to a test-case generation mode and displays a progress bar. After this is complete, a new file is created named `TimeHelperTest.cs` that has autogenerated code stubs for you to modify.

5. Remove the method and its attributes because you'll create three new test cases for that method. Remove the following code:

```
/// <summary>
///A test for GetFriendlyTime
///</summary>
// TODO: Ensure that the UrlToTest attribute specifies a URL to an ASP.NET page (for
// example, http://.../Default.aspx). This is necessary for the unit test to be
// executed on the web server,
// whether you are testing a page, web service, or a WCF service.
[TestMethod()]
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("C:\\Users\\Patrice\\Documents\\Chapter 7\\code\\
PlanMyNight.Web", "/")]
[UrlToTest("http://localhost:48580/")]
public void GetFriendlyTimeTest()
{
    int totalMinutes = 0; // TODO: Initialize to an appropriate value
    string expected = string.Empty; // TODO: Initialize to an appropriate value
    string actual;
    actual = TimeHelper.GetFriendlyTime(totalMinutes);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}
}
```

6. Add the three simple test cases validating three key scenarios used by Plan My Night. To do that, insert the following source code right below the method attributes that were left behind when you deleted the block of code in step 5:

```
[TestMethod]
public void ZeroReturnsSlash()
{
    Assert.AreEqual("-", TimeHelper.GetFriendlyTime(0));
}

[TestMethod]
public void LessThan60MinutesReturnsValueInMinutes()
{
    Assert.AreEqual("10m", TimeHelper.GetFriendlyTime(10));
}

[TestMethod()]
public void MoreThan60MinutesReturnsValueInHoursAndMinutes()
{
    Assert.AreEqual("2h 3m", TimeHelper.GetFriendlyTime(123));
}
```

7. In the PlanMyNight.Web.Tests project, create a solution folder called **Helpers**. Then move your TimeHelperTests.cs file to that folder so that your project looks like Figure 7-28 when you are done:

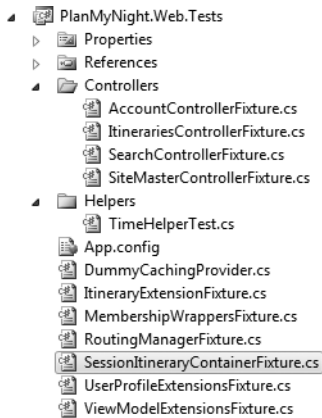


FIGURE 7-28 TimeHelperTest.cs in its Helpers folder

8. It is time to execute your newly created tests. To execute only your newly created tests, go into the code editor and place your cursor on the class named *public class TimeHelperTest*. Then you can either go to the Test menu, select Run, and finally select Test In Current Context or accomplish the same thing using the keyboard shortcut CTRL+R, T. Look at Figure 7-29 for a reference:

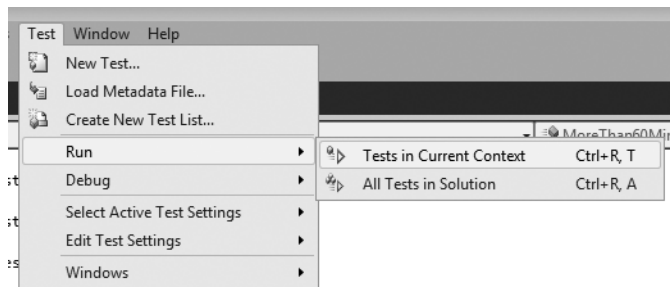


FIGURE 7-29 Test execution menu

9. Performing this action executes only your three tests. You should see the Test Results window (shown in Figure 7-30) appear at the bottom of your editor with the test results.

The screenshot shows the Test Results window with the following data:

Result	Test Name	Project	Error Message
Passed	LessThan60MinutesReturnsValueInMinutes	PlanMyNight.Web.Tests	
Passed	MoreThan60MinutesReturnsValueInHoursAndMinutes	PlanMyNight.Web.Tests	
Passed	ZeroReturnsSlash	PlanMyNight.Web.Tests	

FIGURE 7-30 Test Results window for your newly created tests



More Info Depending on what you select, you might have a different behavior when you choose the Tests In Current Context option. For instance, if you select a test method like *ZeroReturnsSlash*, you'll execute only this test case. However, if you click outside the test class, you could end up executing every test case, which is the equivalent of choosing All Tests In Solution.

New Threads Window

The emergence of computers with multiple cores and the fact that language features give developers many tools to take advantage of those cores creates a new problem: the difficulty of debugging concurrency in applications. The new Threads window enables you, the developer, to pause threads and search the calling stack to see artifacts similar to those you see when using the famous SysInternals Process Monitor (<http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>). You can display the Threads window by going to Debug and selecting Windows And Threads while debugging an application. Take a look at Figure 7-31 to see the Threads window as it appears while debugging Plan My Night.

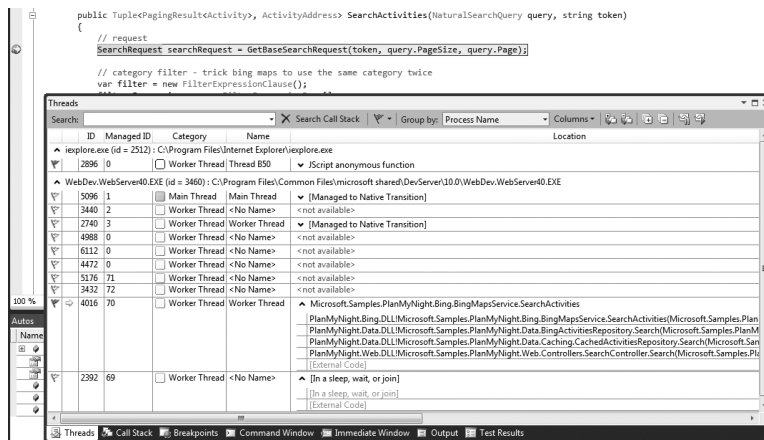


FIGURE 7-31 Displaying the Threads window while debugging Plan My Night

The Threads window allows you to freeze threads and then thaw them whenever you are ready to let them continue. It can be really useful when you are trying to isolate particular effects. You can debug both managed code and unmanaged code. If your application uses threads, you'll definitely love this new feature of the debugger in Visual Studio 2010.

Visual Studio 2005 In Visual Studio 2005, you had to use many tools not integrated into Visual Studio or third-party tools. And in most cases, you still had to rely on your instinct and experience to find concurrency bugs.

Summary

In this chapter, you learned how to manage your debugging sessions by using new break-point enhancements and employing new data-inspection and data-visualization techniques. You also learned how to use the new minidump debugger and tools to help you solve real customer problems from the field. The chapter also showed you how to raise the quality of your code by writing unit tests and how Visual Studio 2010 Professional can help you do this. Multicore machines are now the norm, and so are multithreaded applications. Therefore, the fact that Visual Studio 2010 Professional has specific debugger tools for finding issues in multithreaded applications is great news.

Finally, throughout this chapter you also saw how Visual Studio 2010 Professional has raised the bar in terms of debugging applications and has given professional developers the tools to debug today's feature-rich experiences. You saw that it is a clear improvement over what was available in Visual Studio 2005. The exercises in the chapter scratched the surface of how you'll save time and money by moving to this new debugging environment and showed that Visual Studio 2010 is more than a small iteration in the evolution of Visual Studio. It represents a huge leap in productivity for developers.

The various versions of Visual Studio 2010 give you a great list of improvements related to the debugger and testing. My personal favorites are IntelliTrace—[http://msdn.microsoft.com/en-us/library/dd264915\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(VS.100).aspx)—which is available only in Visual Studio 2010 Ultimate and Microsoft Test Manager. IntelliTrace enables test teams to have much better experiences using Visual Studio 2010 and Visual Studio 2010 Team Foundation Server—[http://msdn.microsoft.com/en-us/library/bb385901\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb385901(VS.100).aspx).

Part III

Moving from Microsoft Visual Studio 2008 to Visual Studio 2010

Authors Patrice Pelland, Ken Haines, and Pascal Pare

In this part:

From 2008 to 2010: Business Logic and Data (Pascal)	217
From 2008 to 2010: Designing the Look and Feel (Ken)	251
From 2008 to 2010: Debugging an Application (Patrice)	293

Chapter 8

From 2008 to 2010: Business Logic and Data

After reading this chapter, you will be able to

- Use the Entity Framework (EF) to build a data access layer using an existing database or with the Model First approach
- Generate entity types from the Entity Data Model (EDM) Designer using the ADO.NET Entity Framework POCO templates
- Learn about data caching using the Microsoft Windows Server AppFabric (formerly known by the codename “Velocity”)

Application Architecture

The Plan My Night (PMN) application allows the user to manage his itinerary activities and share them with others. The data is stored in a Microsoft SQL Server database. Activities are gathered from searches to the Bing Maps Web services.

Let’s have a look at the high-level block model of the data model for the application, which is shown in Figure 8-1.

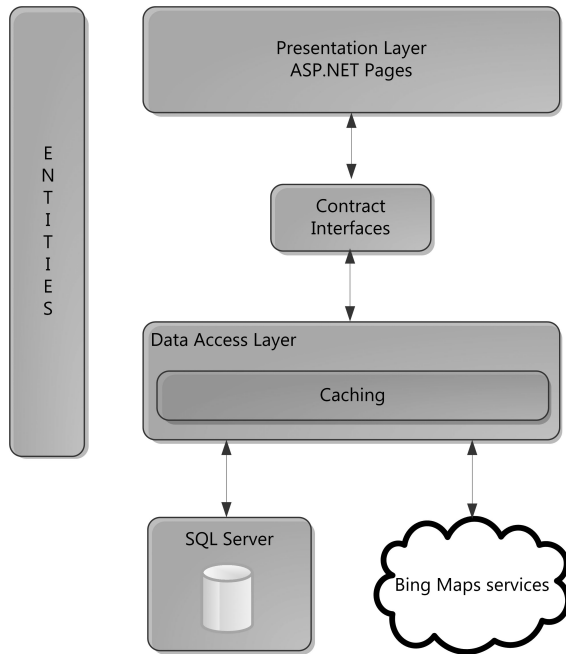


FIGURE 8-1 Plan My Night application architecture diagram

Defining contracts and entity classes that are cleared of any persistence-related code constraints allows us to put them in an assembly that has no persistence-aware code. This approach ensures a clean separation between the Presentation and Data layers.

Let's identify the contract interfaces for the major components of the PMN application:

- *ItinerariesRepository* is the interface to our data store (a Microsoft SQL Server database).
- *IActivitiesRepository* allows us to search for activities (using Bing Maps Web services).
- *ICachingProvider* provides us with our data-caching interface (ASP.NET caching or Windows Server AppFabric caching).



Note This is not an exhaustive list of the contracts implemented in the PMN application.

PMN stores the user's itineraries into an SQL database. Other users will be able to comment and rate each other's itineraries. Figure 8-2 shows the tables used by the PMN application.

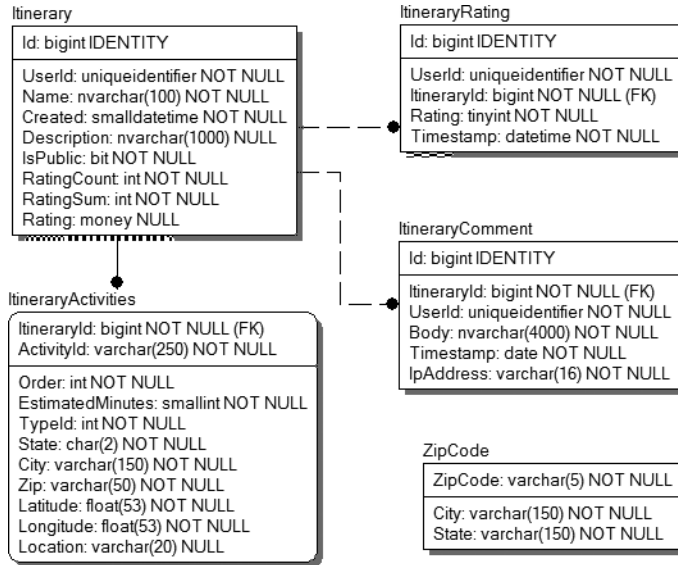


FIGURE 8-2 PlanMyNight database schema



Important The Plan My Night application uses the ASP.NET Membership feature to provide secure credential storage for the users. The user store tables are not shown in Figure 8-2. You can learn more about this feature on MSDN: *ASP.NET 4 - Introduction to Membership* ([http://msdn.microsoft.com/en-us/library/yh26yfzy\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/yh26yfzy(VS.100).aspx)).



Note The ZipCode table is used as a reference repository to provide a list of available Zip Codes and cities so that you can provide autocomplete functionality when the user is entering a search query in the application.

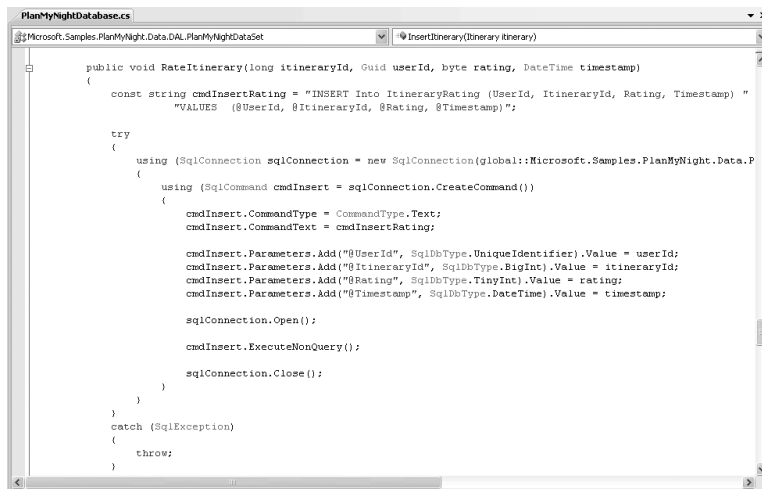
Plan My Night Data in Microsoft Visual Studio 2008

It would be straightforward to create the Plan My Night application in Visual Studio 2008 because it offers all the required tools to help you to code the application. However, some of the technologies used back then required you to write a lot more code to achieve the same goals.

Let's take a look at how you could create the required data layer in Visual Studio 2008. One approach would have been to write the data layer using ADO.NET *DataSet* or *DataReader*

directly. (See Figure 8-3.) This solution offers you great flexibility because you have complete control over access to the database. On the other hand, it also has some drawbacks:

- You need to know the SQL syntax.
- All queries are specialized. A change in requirement or in the tables will force you to update the queries affected by these changes.
- You need to map the properties of your entity classes using the column name, which is a tedious and error-prone process.
- You have to manage the relations between tables yourself.



```
PlanMyNightDatabase.cs
Microsoft.Samples.PlanMyNight.Data.DAL.PlanMyNightDataSet
InsertItinerary(Itinerary itinerary)

public void RateItinerary(long itineraryId, Guid userId, byte rating, DateTime timestamp)
{
    const string cmdInsertRating = "INSERT Into ItineraryRating (UserId, ItineraryId, Rating, Timestamp) "
    "VALUES (@UserId, @ItineraryId, @Rating, @Timestamp)";

    try
    {
        using (SqlConnection sqlConnection = new SqlConnection(global::Microsoft.Samples.PlanMyNight.Data.P
        {
            using (SqlCommand cmdInsert = sqlConnection.CreateCommand())
            {
                cmdInsert.CommandType = CommandType.Text;
                cmdInsert.CommandText = cmdInsertRating;

                cmdInsert.Parameters.Add("@UserId", SqlDbType.UniqueIdentifier).Value = userId;
                cmdInsert.Parameters.Add("@ItineraryId", SqlDbType.BigInt).Value = itineraryId;
                cmdInsert.Parameters.Add("@Rating", SqlDbType.TinyInt).Value = rating;
                cmdInsert.Parameters.Add("@Timestamp", SqlDbType.DateTime).Value = timestamp;

                sqlConnection.Open();

                cmdInsert.ExecuteNonQuery();

                sqlConnection.Close();
            }
        }
    }
    catch (SqlException)
    {
        throw;
    }
}
```

FIGURE 8-3 ADO.NET Insert query

Another approach would be to use the *DataSet* designer available in Visual Studio 2008. Starting from a database with the PMN tables, you could use the TableAdapter Configuration Wizard to import the database tables as shown in Figure 8-4. The generated code offers you a typed *DataSet*. One of the benefits is type checking at design time, which gives you the advantage of statement completion. There are still some pain points with this approach:

- You still need to know the SQL syntax although you have access to the query builder directly from the *DataSet* designer.
- You still need to write specialized SQL queries to match each of the requirements of your data contracts.
- You have no control of the generated classes. For example, changing the *DataSet* to add or remove a query for a table will rebuild the generated *TableAdapter* classes and might change the index used for a query. This makes it difficult to write predictable code using these generated items.

- The generated classes associated with the tables are persistence aware, so you will have to create another set of simple entities and copy the data from one to the other. This means more processing and memory usage.

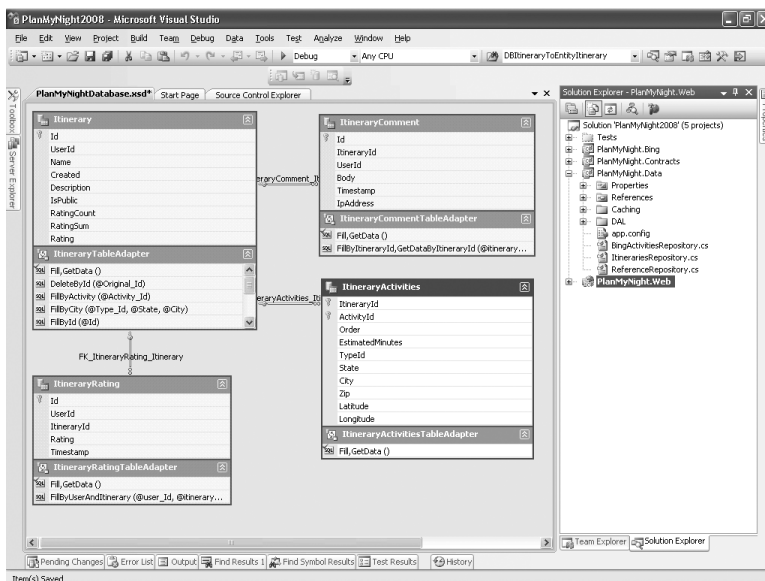


FIGURE 8-4 DataSet designer in Visual Studio 2008

Another technology available in Visual Studio 2008 was LINQ to SQL (L2S). With the Object Relational Designer for L2S, it was easy to add the required database tables. This approach gives you access to strongly typed objects and to LINQ to create the queries required to access your data, so you do not have to explicitly know the SQL syntax. This approach also has its limits:

- LINQ to SQL works only with SQL Server databases.
- You have limited control over the created entities, and you cannot easily update them if your database schema changes.
- The generated entities are persistence aware.



Note As of .NET 4.0, Microsoft recommends the Entity Framework as the data access solution for LINQ to relational scenarios.

In the next sections of this chapter, you'll explore some of the new features of Visual Studio 2010 that will help you create the PMN data layer with less code, give you more control of the generated code, and allow you to easily maintain and expand it.

Data with the Entity Framework in Visual Studio 2010

The ADO.NET Entity Framework (EF) allows you to easily create the data access layer for an application by abstracting the data from the database and exposing a model closer to the business requirements of the application. The EF has been considerably enhanced in the .NET Framework 4 release.

You'll use the PlanMyNight project as an example of how to build an application using some of the features of the EF. The next two sections demonstrate two different approaches to generating the data model of PMN. In the first one, you let the EF generate the Entity Data Model (EDM) from an existing database. In the second part, you use a Model First approach, where you first create the entities from the EF designer and generate the Data Definition Language (DDL) scripts to create a database that can store your EDM.

Visual Studio 2008 The first version of the Entity Framework was released with Visual Studio 2008 Service Pack 1. The second version of the EF included in the .NET Framework 4.0 offers many new features to help you build your data applications. Some of these new enhancements include the following:

- T4 code-generation templates that you can customize to your needs
- The possibility to define your own POCOs (Plain Old CLR Objects) to ensure that your entities are decoupled from the persistence technology
- Model-First development, where you create a model for your entities and let Visual Studio 2010 create your database
- Code-only support so that you can use the Entity Framework using POCO entities and without an EDMX file
- Lazy loading for related entities so that they are loaded only from the database when required
- Self-tracking entities that have the ability to record their own changes on the client and send these changes so that they can be applied to the database store

In the next sections, you'll explore some of these new features.

See Also The MSDN Data Developer Center also offers a lot of resources about the ADO.NET Entity Framework (<http://msdn.microsoft.com/en-us/data/aa937723.aspx>) in .NET 4.

EF: Importing an Existing Database

You'll start with an existing solution that already defines the main projects of the PMN application. If you installed the companion content at the default location, you'll find the

solution at this location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ExistingDatabase. Double-click the PlanMyNight.sln file.

This solution includes all the projects in the following list, as shown in Figure 8-5:

- PlanMyNight.Data: Application data layer
- PlanMyNight.Contracts: Entities and contracts
- PlanMyNight.Bing: Bing Maps services
- PlanMyNight.Web: Presentation layer
- PlanMyNight.AppFabricCaching: AppFabric caching

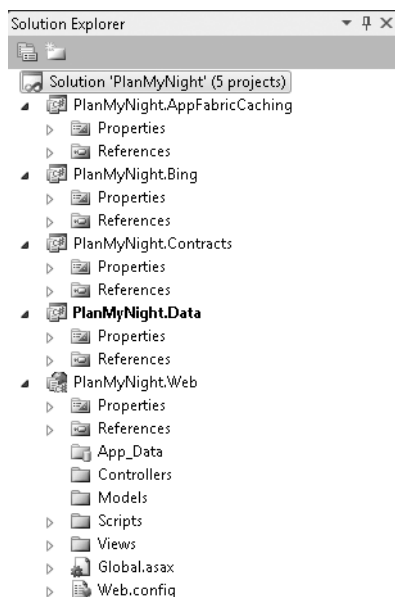


FIGURE 8-5 PlanMyNight solution

The EF allows you to easily import an existing database. Let's walk through this process.

The first step is to add an EDM to the PlanMyNight.Data project. Right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item, and change its name to **PlanMyNight.edmx**, as shown in Figure 8-6.

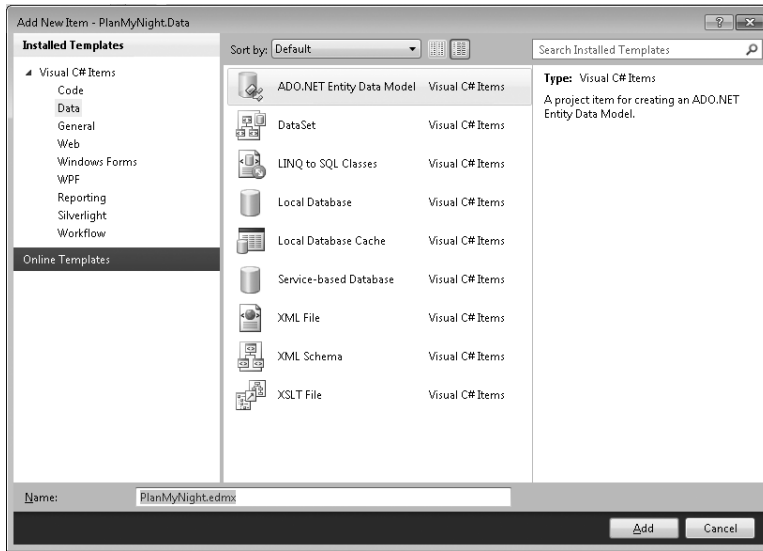


FIGURE 8-6 Add New Item dialog with ADO.NET Entity Data Model selected

The first dialog of the Entity Data Model Wizard allows you to choose the model content. You'll generate the model from an existing database. Select **Generate From Database** and then click **Next**.

You need to connect to an existing database file. Click **New Connection**. Select **Microsoft SQL Server Database File** from the **Choose Data Source** dialog, and click **Continue**. Select the `%userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\ExistingDatabase\PlanMyNight.Web\App_Data\PlanMyNight.mdf` file. (See Figure 8-7.)

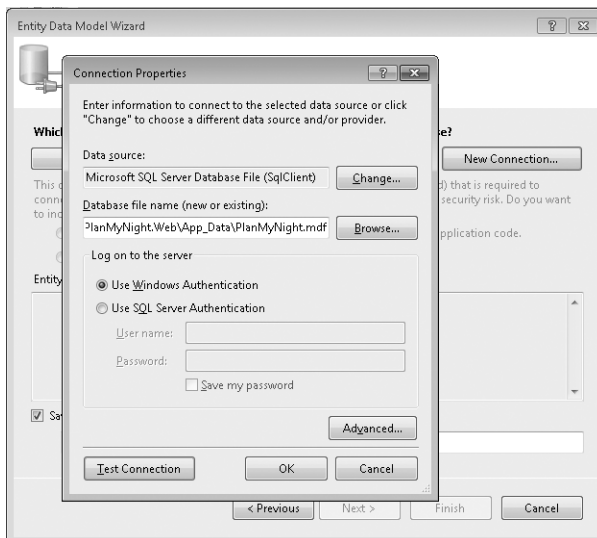


FIGURE 8-7 EDM Wizard database connection

Leave the other fields in the form as is for now and click Next.



Note You'll get a warning stating that the local data file is not in the current project. Click No to close the dialog because you do not want to copy the database file to the current project.

From the Choose Your Database Objects dialog, select the *Itinerary*, *ItineraryActivities*, *ItineraryComment*, *ItineraryRating*, and *ZipCode* tables and the *UserProfile* view. Select the *RetrievalItinerariesWithinArea* stored procedure. Change the Model Namespace value to **Entities** as shown in Figure 8-8.

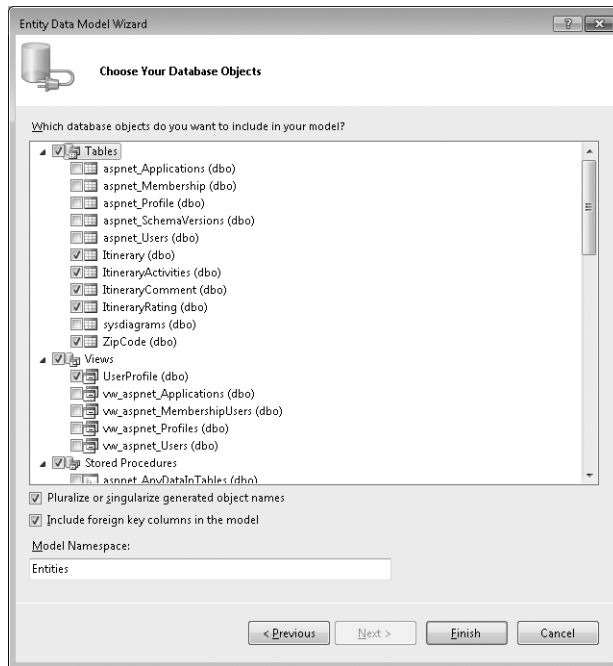


FIGURE 8-8 EDM Wizard: Choose Your Database Objects page

Click Finish to generate your EDM.

Visual Studio 2008 In the first version of the EF, the names associated with *EntityType*, *EntitySet*, and *NavigationProperty* were often wrong when you created a model from the database because it was using the database table name to generate them. You probably do not want to create an instance of the *ItineraryActivities* entity. Instead, you probably want the name to be singularized to *ItineraryActivity*. The Pluralize Or Singularize Generated Object Names check box shown in Figure 8-8 allows you to control whether pluralization or singularization should be attempted.

Fixing the Generated Data Model

You now have a model representing a set of entities matching your database. The wizard has generated all the navigation properties associated with the foreign keys from the database.

The PMN application requires only the navigation property *ItineraryActivities* from the *Itinerary* table, so you can go ahead and delete all the other navigation properties. You'll also need to rename the *ItineraryActivities* navigation property to **Activities**. Refer to Figure 8-9 for the updated model.

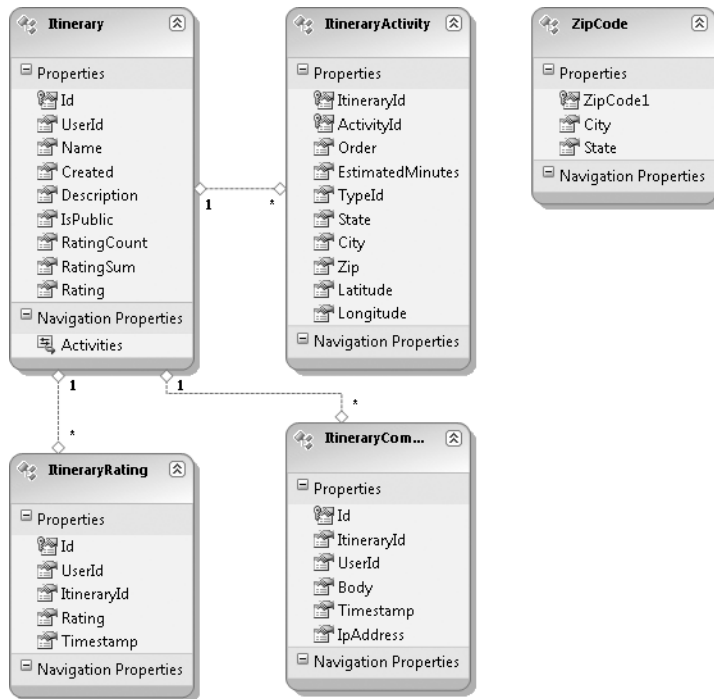


FIGURE 8-9 Model imported from the PlanMyNight database

Notice that one of the properties of the *ZipCode* entity has been generated with the name *ZipCode1* because the table itself is already named *ZipCode* and the name has to be unique. Let's fix the property name by double-clicking it. Change the name to **Code**, as shown in Figure 8-10.

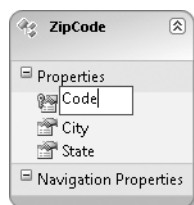


FIGURE 8-10 ZipCode entity

Build the solution by pressing Ctrl+Shift+B. When looking at the output window, you'll notice two messages from the generated EDM. You can discard the first one because the Location column is not required in PMN. The second message reads as follows:

The table/view 'dbo.UserProfile' does not have a primary key defined and no valid primary key could be inferred. This table/view has been excluded. To use the entity, you will need to review your schema, add the correct keys, and uncomment it.

When looking at the UserProfile view, you'll notice it does not explicitly define a primary key even though the UserName column is unique.

You need to modify the EDM manually to fix the UserProfile view mapping so that you can access the UserProfile data from the application.

From the project explorer, right-click the PlanMyNight.edmx file and then select Open With Choose XML (Text) Editor from the Open With dialog as shown in Figure 8-11. Click OK to open the XML file associated with your model.

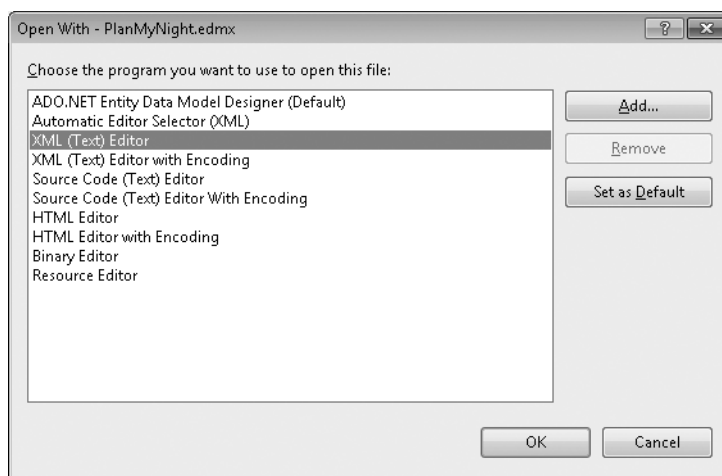


FIGURE 8-11 Open PlanMyNight.edmx in the XML Editor



Note You'll get a warning stating that the PlanMyNight.edmx file is already open. Click Yes to close it.

The generated code was commented out by the code-generation tool because there was no primary key defined. To be able to use the UserProfile view from the designer, you need to uncomment the UserProfile entity type and add the Key tag to it. Search for UserProfile in the file. Uncomment the entity type, add a Key tag and set its name to **UserName** and make the *UserName* property not nullable. Refer to Listing 8-1 to see the updated entity type.

LISTING 8-1 UserProfile Entity Type XML Definition

```

<EntityType Name="UserProfile">
  <Key>
    <PropertyRef Name="UserName"/>
  </Key>
  <Property Name="UserName" Type="uniqueidentifier" Nullable="false" />
  <Property Name="FullName" Type="varchar" MaxLength="500" />
  <Property Name="City" Type="varchar" MaxLength="500" />
  <Property Name="State" Type="varchar" MaxLength="500" />
  <Property Name="PreferredActivityTypeId" Type="int" />
</EntityType>

```

If you close the XML file and try to open the EDM Designer, you'll get the following error message in the designer: "The Entity Data Model Designer is unable to display the file you requested. You can edit the model using the XML Editor."

There is a warning in the Error List pane that can give you a little more insight into what this error is all about:

Error 11002: Entity type 'UserProfile' has no entity set.

You need to define an entity set for the UserProfile type so that it can map the entity type to the store schema. Open the PlanMyNight.edmx file in the XML editor so that you can define an entity set for UserProfile. At the top of the file, just above the Itinerary entity set, add the XML code shown in Listing 8-2.

LISTING 8-2 UserProfile EntitySet XML Definition

```

<EntitySet Name="UserProfile" EntityType="Entities.Store.UserProfile"
  store:Type="Views" store:Schema="dbo" store:Name="UserProfile">
  <DefiningQuery>
    SELECT
      [UserProfile].[UserName] AS [UserName],
      [UserProfile].[FullName] AS [FullName],
      [UserProfile].[City] AS [City],
      [UserProfile].[State] AS [State],
      [UserProfile].[PreferredActivityTypeId] as [PreferredActivityTypeId]
    FROM [dbo].[UserProfile] AS [UserProfile]
  </DefiningQuery>
</EntitySet>

```

Save the EDM XML file, and reopen the EDM Designer. Figure 8-12 shows the UserProfile view in the Entities.Store section of the Model Browser.



Tip You can open the Model Browser from the View menu by clicking Other Windows and selecting the Entity Data Model Browser item.

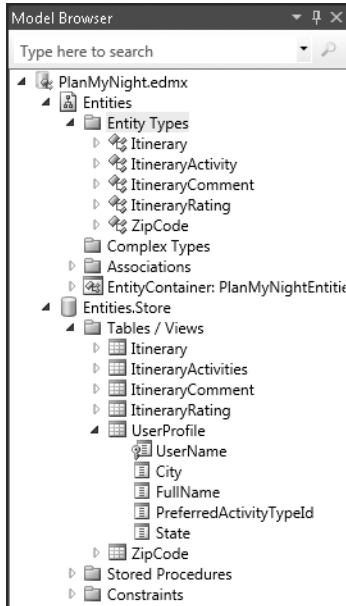


FIGURE 8-12 Model Browser with the UserProfile view

Now that the view is available in the store metadata, you add the UserProfile entity and map it to the UserProfile view. Right-click in the background of the EDM Designer, select Add, and then choose Entity. You'll see the dialog shown in Figure 8-13.

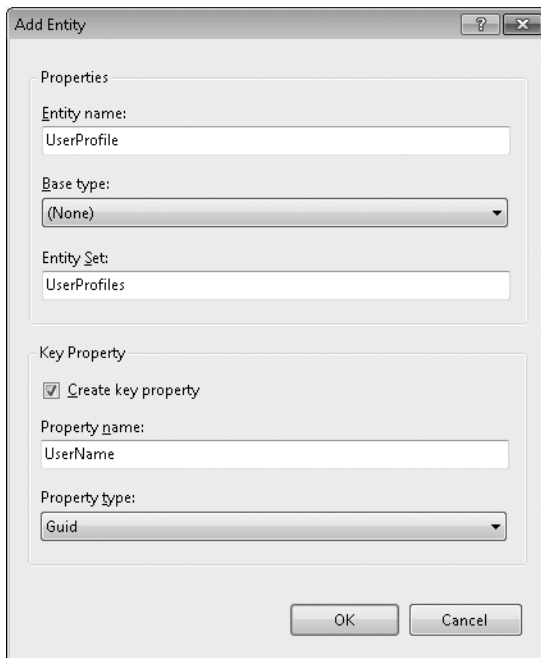


FIGURE 8-13 Add Entity dialog

Complete the dialog as shown in Figure 8-13, and click OK to generate the entity.

You need to add the remaining properties: *City*, *State*, and *PreferredActivityTypeId*. To do so, right-click the *UserProfile* entity, select *Add*, and then select *Scalar Property*. After the property is added, set the *Type*, *Max Length*, and *Unicode* field values. Table 8-1 shows the expected values for each of the fields.

TABLE 8-1 UserProfile Entity Properties

Name	Type	Max Length	Unicode
<i>FullName</i>	<i>String</i>	500	False
<i>City</i>	<i>String</i>	500	False
<i>State</i>	<i>String</i>	500	False
<i>PreferredActivityTypeId</i>	<i>Int32</i>	NA	NA

Now that you have created the *UserProfile* entity, you need to map it to the *UserProfile* view. Right-click the *UserProfile* entity, and select *Table Mapping* as shown in Figure 8-14.

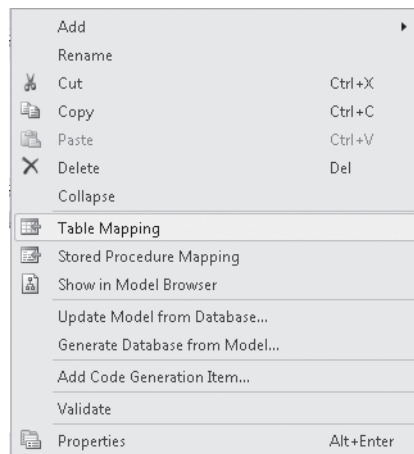


FIGURE 8-14 Table Mapping menu item

Then select the *UserProfile* view from the drop-down box as shown in Figure 8-15. Ensure that all the columns are correctly mapped to the entity properties. The *UserProfile* view of our store is now accessible from the code through the *UserProfile* entity.

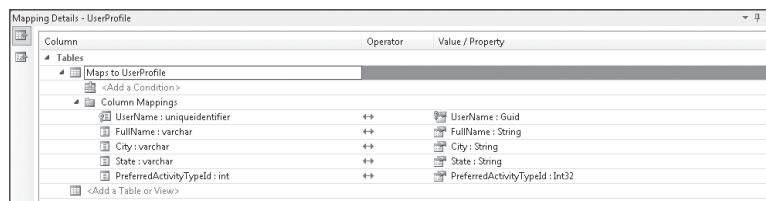


FIGURE 8-15 UserProfile mapping details

Stored Procedure and Function Imports

The Entity Data Model Wizard has created an entry in the storage model for the *RetrievalItinerariesWithinArea* stored procedure you selected in the last step of the wizard. You need to create a corresponding entry to the conceptual model by adding a Function Import entry.

From the Model Browser, open the Stored Procedures folder in the Entities.Store section. Right-click *RetrievalItineraryWithinArea*, and then select Add Function Import. The Add Function Import dialog appears as shown in Figure 8-16. Specify the return type by selecting Entities and then select the Itinerary item from the drop-down box. Click OK.

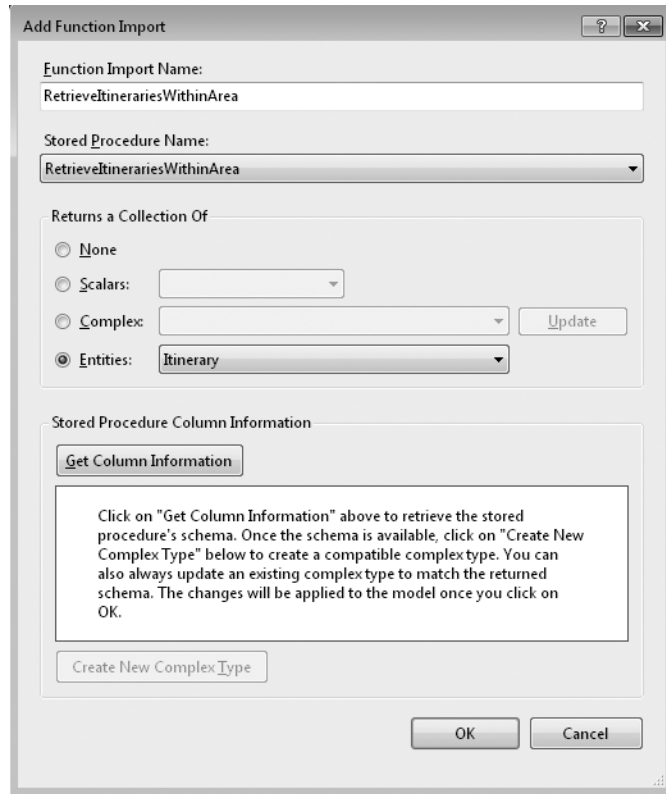


FIGURE 8-16 Add Function Import dialog

The *RetrievalItinerariesWithinArea* function import was added to the Model Browser as shown in Figure 8-17.

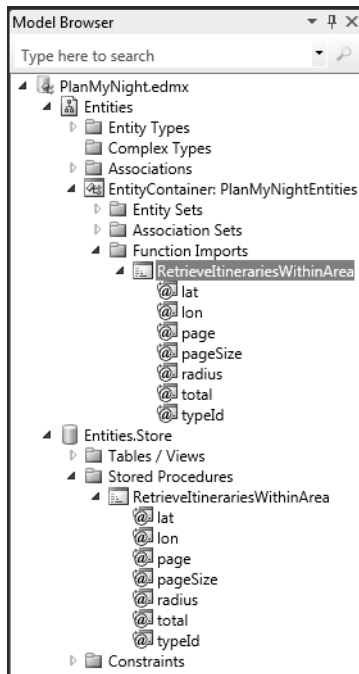


FIGURE 8-17 Function Imports in the Model Browser

You can now validate the EDM by right-clicking on the design surface and selecting *Validate*. There should be no error or warning.

EF: Model First

In the prior section, you saw how to use the EF designer to generate the model by importing an existing database. The EF designer in Visual Studio 2010 also supports the ability to generate the Data Definition Language (DDL) file that will allow you to create a database based on your entity model. In this section, you'll use a new solution to learn how to generate a database script from a model.

You can start from an empty model by selecting the Empty model option from the Entity Data Model Wizard. (See Figure 8-18.)



Note To get the wizard, right-click the PlanMyNight.Data project, select Add, and then choose New Item. Select the ADO.NET Entity Data Model item.

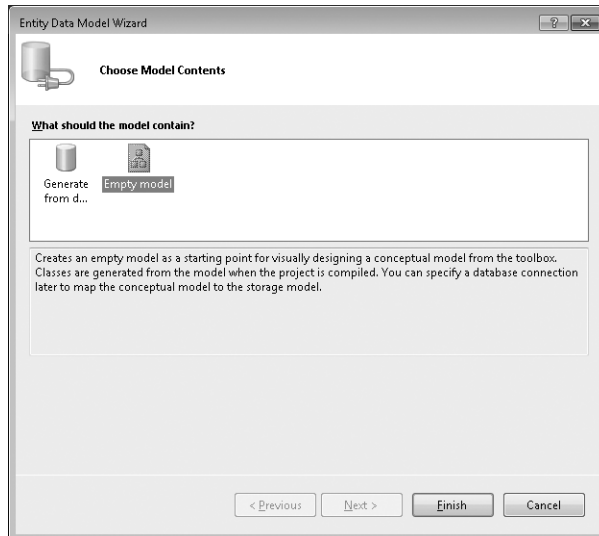


FIGURE 8-18 EDM Wizard: Empty model

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ModelFirst by double-clicking the PlanMyNight.sln file.

The PlanMyNight.Data project from this solution already contains an EDM file named PlanMyNight.edmx with some entities already created. These entities match the data schema you saw in Figure 8-2.

The Entity Model designer lets you easily add an entity to your data model. Let's add the missing ZipCode entity to the model. From the toolbox, drag an Entity item into the designer, as shown in Figure 8-19. Rename the entity as **ZipCode**. Rename the *Id* property as **Code**, and change its type to *String*.

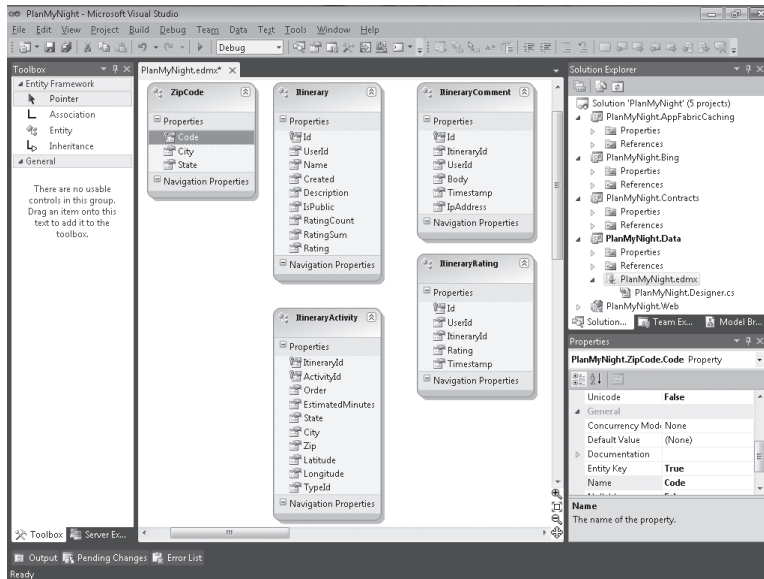


FIGURE 8-19 Entity Model designer

You need to add the *City* and *State* properties to the entity. Right-click the *ZipCode* entity, select Add, and then choose Scalar Property. Ensure that each property has the values shown in Table 8-2.

TABLE 8-2 ZipCode Entity Properties

Name	Type	Fixed Length	Max Length	Unicode
<i>Code</i>	<i>String</i>	False	5	False
<i>City</i>	<i>String</i>	False	150	False
<i>State</i>	<i>String</i>	False	150	False

Add the relations between the *ItineraryComment* and *Itinerary* entities. Right-click the designer background, select Add, and then choose Association. (See Figure 8-20.)

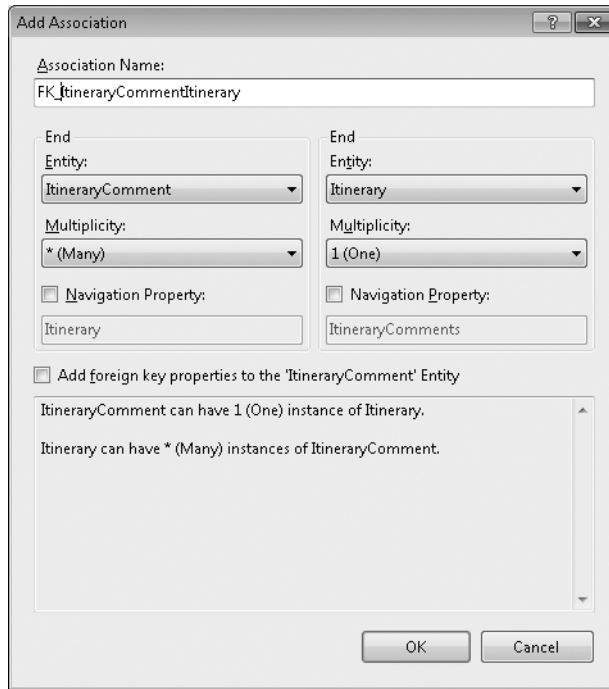


FIGURE 8-20 Add Association dialog for *FK_ItineraryCommentItinerary*

Visual Studio 2008 Foreign key associations are now included in the .NET 4.0 version of the Entity Framework. This allows you to have Foreign properties on your entities. Foreign Key Associations is now the default type of association, but the Independent Associations supported in .NET 3.5 are still available.

Set the association name to **FK_ItineraryCommentItinerary**, and then select the entity and the multiplicity for each end, as shown in Figure 8-20. After the association is created, double-click the association line to set the Referential Constraint as shown in Figure 8-21.

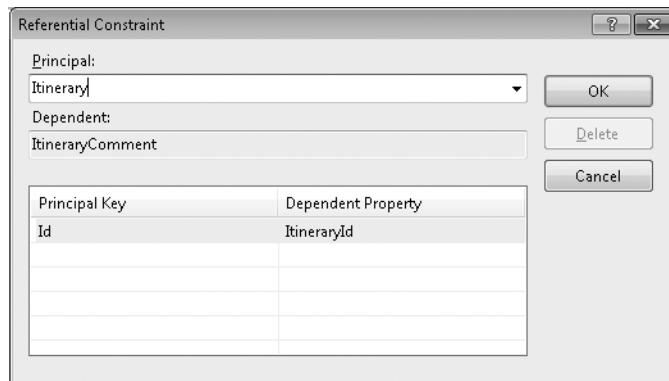


FIGURE 8-21 Association Referential Constraint dialog

Add the association between the *ItineraryRating* and *Itinerary* entities. Right-click the designer background, select **Add**, and then choose **Association**. Set the association name to **FK_ItineraryItineraryRating** and then select the entity and the multiplicity for each end as in the previous step, except set the first end to **ItineraryRating**. Double-click on the association line, and set the Referential Constraint as shown in Figure 8-21. Note that the Dependent field will read *ItineraryRating* instead of *ItineraryComment*. Create a new association between the *ItineraryActivity* and *Itinerary* entities. For the *FK_ItineraryItineraryActivity* association, you want to also create a navigation property and name it **Activities**, as shown in Figure 8-22. After the association is created, set the Referential Constraint for this association by double-clicking on the association line.

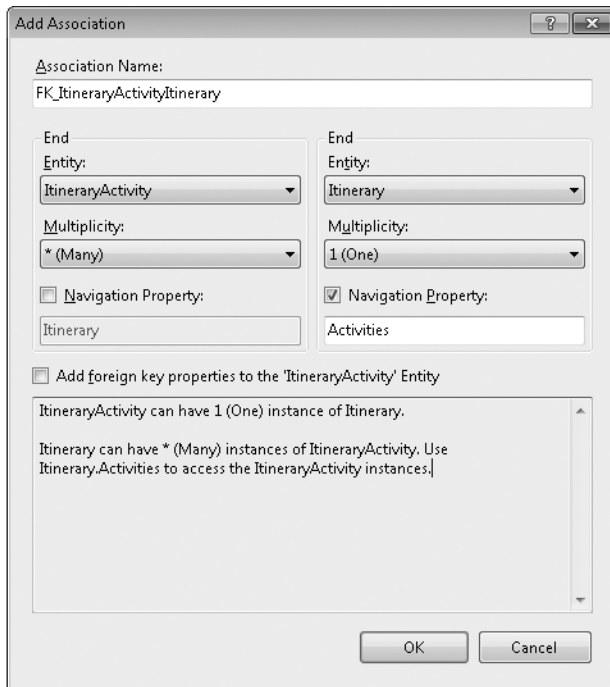


FIGURE 8-22 Add Association dialog for *FK_ItineraryActivityItinerary*

Generating the Database Script from the Model

Your data model is now completed but there is no mapping or store associated with it. The EF designer offers the possibility of generating a database script from our model.

Right-click on the designer surface, and choose Generate Database From Model as shown in Figure 8-23.

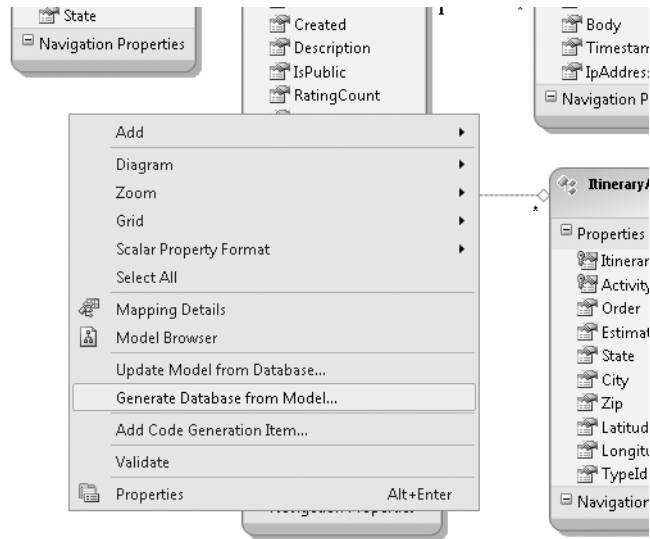


FIGURE 8-23 Generate Database From Model menu item

The Generate Database Wizard requires a data connection. The wizard uses the connection information to translate the model types to the database type and to generate a DDL script targeting this database.

Select New Connection, select Microsoft SQL Server Database File from the Choose Data Source dialog, and click Continue. Select the database file located at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ModelFirst\Data\PlanMyNight.mdf. (See Figure 8-24.)



FIGURE 8-24 Generate a script database connection

After your connection is configured, click Next to get to the final page of the wizard, as shown in Figure 8-25. When you click Finish, the generated T-SQL PlanMyNight.edmx.sql file is added to your project. The DDL script will generate the primary and foreign key constraints for your model.

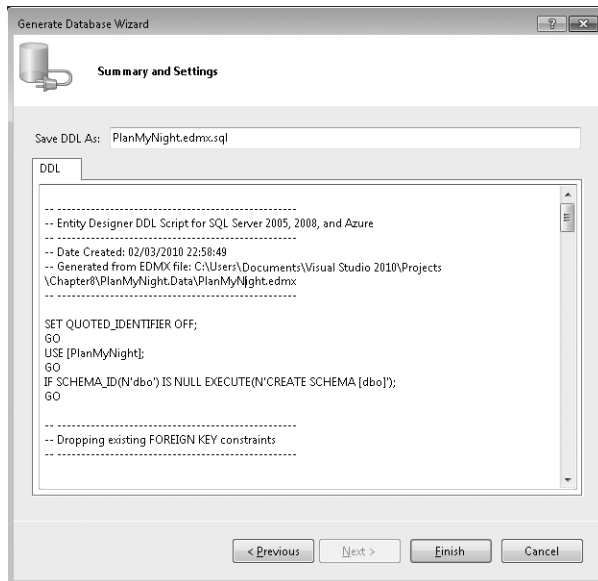


FIGURE 8-25 Generated T-SQL file

The EDM is also updated to ensure your newly created store is mapped to the entities. You can now use the generated DDL script to add the tables to the database. Also, you now have a data layer that exposes strongly typed entities that you can use in your application.



Important Generating the complete PMN database would require adding the remaining tables, stored procedures, and triggers used by the application. Instead of performing all these operations, we will go back to the solution we had at the end of the “EF: Importing an Existing Database” section.

POCO Templates

The EDM Designer uses T4 templates to generate the code for the entities. So far, we have let the designer create the entities using the default templates. You can take a look at the code generated by opening the `PlanMyNight.Designer.cs` file associated with `PlanMyNight.edmx`. The generated entities are based on the `EntityObject` type and decorated with attributes to allow the EF to manage them at run time.



Note T4 stands for *Text Template Transformation Toolkit*. T4 support in Visual Studio 2010 allows you to easily create your own templates and generate any type of text file (Web, resource, or source). To learn more about the code generation in Visual Studio 2010, visit [Code Generation and Text Templates](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx) ([http://msdn.microsoft.com/en-us/library/bb126445\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx)).

The EF also supports POCO entity types. POCO classes are simple objects with no attributes or base class related to the framework. (Listing 8-3, in the next section, shows the POCO class for the `ZipCode` entity.) The EF uses the names of the types and the properties of these objects to map them to the model at run time.



Note POCO stands for *Plain-Old CLR Objects*.

ADO.NET POCO Entity Generator

Let's re-open the `%userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ExistingDatabase\PlanMyNight.sln` file.

Open the `PlanMyNight.edmx` file, right-click on the design surface, and choose `Add Code Generation Item`. This opens a dialog like the one shown in Figure 8-26, where you can select the template you want to use. Select the `ADO.NET POCO Entity Generator` template, and name it **PlanMyNight.tt**. Then click the `Add` button.



Note You might get a security warning about running this text template. Click OK to close the dialog because the source for this template is trusted.

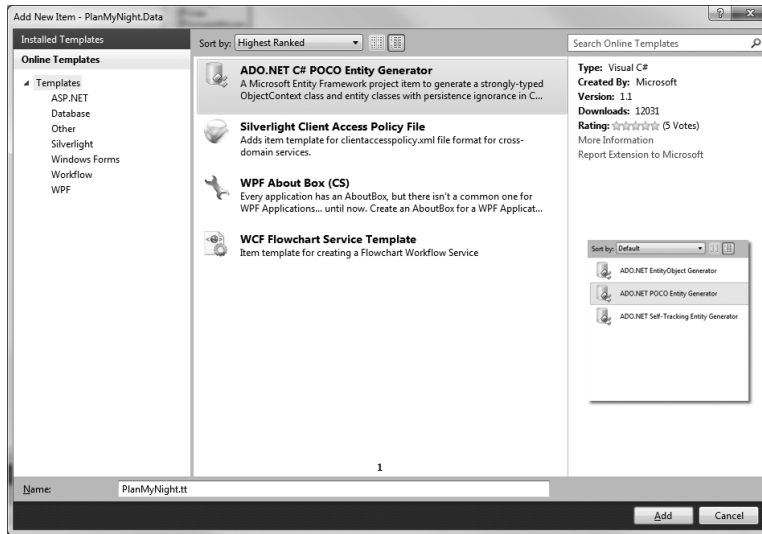


FIGURE 8-26 Add New Item dialog

Two files, `PlanMyNight.tt` and `PlanMyNight.Context.tt`, have been added to your project, as shown in Figure 8-27. These files replace the default code-generation template, and the code is no longer generated in the `PlanMyNight.Designer.cs` file.

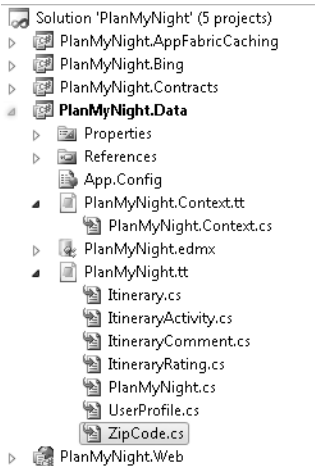


FIGURE 8-27 Added templates

The PlanMyNight.tt template produces a class file for each entity in the model. Listing 8-3 shows the POCO version of the *ZipCode* class.

LISTING 8-3 POCO Version of the *ZipCode* Class

```
namespace Microsoft.Samples.PlanMyNight.Data
{
    public partial class ZipCode
    {
        #region Primitive Properties
        public virtual string Code
        {
            get;
            set;
        }
        public virtual string City
        {
            get;
            set;
        }
        public virtual string State
        {
            get;
            set;
        }
        #endregion
    }
}
```

The other file, *PlanMyNight.Context.cs*, generates the *ObjectContext* object for the *PlanMyNight.edmx* model. This is the object you'll use to interact with the database.



Tip The POCO templates will automatically update the generated classes to reflect the changes to your model when you save the .edmx file.

Moving the Entity Classes to the Contracts Project

We have designed the PMN application architecture to ensure that the presentation layer was persistence ignorant by moving the contracts and entity classes to an assembly that has no reference to the storage.

Visual Studio 2008 Even though it was possible to extend the XSD processing with code-generator tools, it was not easy and you had to maintain these tools. The EF uses T4 templates to generate both the database schema and the code. These templates can easily be customized to your needs.

The ADO.NET POCO templates split the generation of the entity classes into a separate template, allowing you to easily move these entities to a different project.

You are going to move the PlanMyNight.tt file to the PlanMyNight.Contracts project. Right-click the PlanMyNight.tt file, and select Cut. Right-click the Entities folder in the PlanMyNight.Contracts project, and select Paste. The result is shown in Figure 8-28.

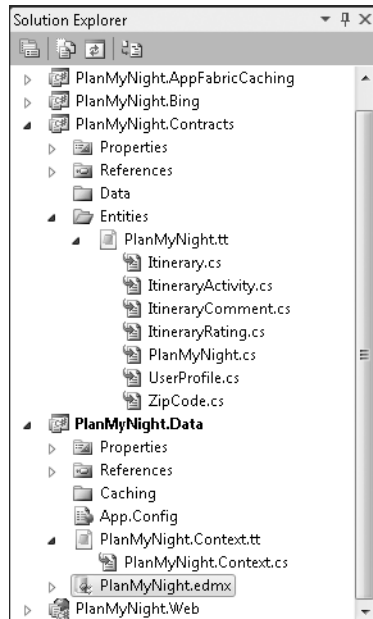


FIGURE 8-28 POCO template moved to the Contracts project

The PlanMyNight.tt template relies on the metadata from the EDM model to generate the entity type's code. You need to fix the relative path used by the template to access the EDMX file.

Open the PlanMyNight.tt template and locate the following line:

```
string inputFile = @"PlanMyNight.edmx";
```

Fix the file location so that it points to the PlanMyNight.edmx file in the PlanMyNight.Data project:

```
string inputFile = @"..\..\PlanMyNight.Data\PlanMyNight.edmx";
```

The entity classes are regenerated when you save the template.

You also need to update the `PlanMyNight.Context.tt` template in the `PlanMyNight.Contracts` project because the entity classes are now in the `Microsoft.Samples.PlanMyNight.Entities` namespace instead of the `Microsoft.Samples.PlanMyNight.Data` namespace. Open the `PlanMyNight.Context.tt` file, and update the `using` section to include the new namespace:

```
using System;
using System.Data.Objects;
using System.Data.EntityClient;
using Microsoft.Samples.PlanMyNight.Entities;
```

Build the solution by pressing `Ctrl+Shift+B`. The project should now compile successfully.

Putting It All Together

Now that you have created the generic code layer to interact with your SQL database, you are ready to start implementing the functionalities specific to the PMN application. In the upcoming sections, you'll walk through this process, briefly look at getting the data from the Bing Maps services, and get a quick introduction to the Microsoft Windows Server AppFabric Caching feature used in PMN.

There is a lot of plumbing pieces of code required to get this all together. To simplify the process, you'll use an updated solution where the contracts, entities, and most of the connecting pieces to the Bing Maps services have been coded. The solution will also include the `PlanMyNight.Data.Test` project to help you validate the code from the `PlanMyNight.Data` project.



Note Testing in Visual Studio 2010 will be covered in Chapter 10.

Getting Data from the Database

At the beginning of this chapter, we decided to group the operations on the `Itinerary` entity in the `ItinerariesRepository` repository interface. Some of these operations are

- Searching for Itinerary by Activity
- Searching for Itinerary by ZipCode
- Searching for Itinerary by Radius
- Adding a new Itinerary

Let's take a look at the corresponding methods in the *ItinerariesRepository* interface:

- *SearchByActivity* allows searching for itineraries by activity and returning a page of data.
- *SearchByZipCode* allows searching for itineraries by Zip Code and returning a page of data.
- *SearchByRadius* allows searching for itineraries from a specific location and returning a page of data.
- *Add* allows you to add an itinerary to the database.

Open the PMN solution at %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\Final by double-clicking the PlanMyNight.sln file.

Select the PlanMyNight.Data project, and open the ItinerariesRepository.cs file. This is the *ItinerariesRepository* interface implementation. Using the PlanMyNightEntities Object Context you generated earlier, you can write LINQ queries against your model, and the EF will translate these queries to native T-SQL that will be executed against the database.

Navigate to the *SearchByActivity* function definition. This method must return a set of itineraries where the *IsPublic* flag is set to true and where one of their activities has the same *activityId* that was passed in the argument to the function. You also need to order the result itinerary list by the rating field.

Using standard LINQ operators, you can implement *SearchByActivity* as shown in Listing 8-4. Add the highlighted code to the *SearchByActivity* method body.

LISTING 8-4 *SearchByActivity* Implementation

```
public PagingResult<Itinerary> SearchByActivity(string activityId, int pageSize, int
pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
                    where itinerary.Activities.Any(t => t.ActivityId == activityId)
                      && itinerary.IsPublic
                    orderby itinerary.Rating
                    select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}
```



Note The resulting paging is implemented in the *PageResults* method:

```
private static PagingResult<Itinerary> PageResults(IQueryable<Itinerary> query, int
page, int pageSize)
{
    int rowCount = query.Count();
    if (pageSize > 0)
    {
        query = query.Skip((page - 1) * pageSize)
            .Take(pageSize);
    }
    var result = new PagingResult<Itinerary>(query.ToArray())
    {
        PageSize = pageSize,
        CurrentPage = page,
        TotalItems = rowCount
    };
    return result;
}
```

IQueryable<Itinerary> is passed to this function so that it can add the paging to the base query composition. Passing *IQueryable* instead of *IEnumerable* ensures that the T-SQL created for the query against the repository will be generated only when *query.ToArray* is called.

The *SearchByZipCode* function method is similar to the *SearchByActivity* method, but it also adds a filter on the Zip Code of the activity. Here again, LINQ support makes it easy to implement, as shown in Listing 8-5. Add the highlighted code to the *SearchByZipCode* method body.

LISTING 8-5 *SearchByZipCode* Implementation

```
public PagingResult<Itinerary> SearchByZipCode(int activityTypeId, string zip, int
pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        var query = from itinerary in ctx.Itineraries.Include("Activities")
            where itinerary.Activities.Any(t => t.TypeId == activityTypeId &&
t.Zip == zip)
                && itinerary.IsPublic
            orderby itinerary.Rating
            select itinerary;

        return PageResults(query, pageNumber, pageSize);
    }
}
```

The *SearchByRadius* function calls the *RetrieveItinerariesWithinArea* import function that was mapped to a stored procedure. It then loads the activities for each itinerary found. You can copy the highlighted code in Listing 8-6 to the *SearchByRadius* method body in the *ItinerariesRepository.cs* file.

LISTING 8-6 *SearchByRadius* Implementation

```
public PagingResult<Itinerary> SearchByRadius(int activityTypeId,
    double longitude, double latitude, double radius,
    int pageSize, int pageNumber)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.ContextOptions.ProxyCreationEnabled = false;

        // Stored Procedure with output parameter
        var totalOutput = new ObjectParameter("total", typeof(int));
        var items = ctx.RetrieveItinerariesWithinArea(activityTypeId, latitude, longitude,
            radius, pageSize, pageNumber, totalOutput).ToArray();

        foreach (var item in items)
        {
            item.Activities.ToList().AddRange(this.Retrieve(item.Id).Activities);
        }

        int total = totalOutput.Value == DBNull.Value ? 0 : (int)totalOutput.Value;

        return new PagingResult<Itinerary>(items)
        {
            TotalItems = total,
            PageSize = pageSize,
            CurrentPage = pageNumber
        };
    }
}
```

The *Add* method allows you to add Itinerary to the data store. Implementing this functionality becomes trivial because your contract and context object use the same entity object. Copy and paste the highlighted code in Listing 8-7 to the *Add* method body.

LISTING 8-7 *Add* Implementation

```
public void Add(Itinerary itinerary)
{
    using (var ctx = new PlanMyNightEntities())
    {
        ctx.Itineraries.AddObject(itinerary);
        ctx.SaveChanges();
    }
}
```

There you have it! You have completed the *ItinerariesRepository* implementation using the context object generated using the EF designer. Run all the tests in the solution by pressing Ctrl+R, A. The tests related to the *ItinerariesRepository* implementation should all succeed.

Parallel Programming

With the advances in multicore computing, it is becoming more and more important for developers to be able to write parallel applications. Visual Studio 2010 and the .NET Framework 4.0 provide new ways to express concurrency in applications. The Task Parallel Library (TPL) is now part of the Base Class Library (BCL) for the .NET Framework. This means that every .NET application can now access the TPL without adding any assembly reference.

PMN stores only the Bing Activity ID for each *ItineraryActivity* to the database. When it's time to retrieve the entire Bing Activity object, a function that iterates through each of the *ItineraryActivity* instances for the current *Itinerary* is used to populate the Bing Activity entity from the Bing Maps Web services.

One way of performing this operation is to sequentially call the service for each activity in the *Itinerary* as shown in Listing 8-8. This function waits for each call to *RetrieveActivity* to complete before making another call, which has the effect of making its execution time linear.

LISTING 8-8 Activity Sequential Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    foreach (var item in itinerary.Activities.Where(i => i.Activity == null))
    {
        item.Activity = this.RetrieveActivity(item.ActivityId);
    }
}
```

In the past, if you wanted to parallelize this task, you had to use threads and then hand off work to them. With the TPL, all you have to do now is use a *Parallel.ForEach* that will take care of the threading for you as seen in Listing 8-9.

LISTING 8-9 Activity Parallel Retrieval

```
public void PopulateItineraryActivities(Itinerary itinerary)
{
    Parallel.ForEach(itinerary.Activities.Where(i => i.Activity == null),
        item =>
        {
            item.Activity = this.RetrieveActivity(item.ActivityId);
        });
}
```

See Also *The .NET Framework 4.0 now includes the Parallel LINQ libraries (in System.Core.dll). PLINQ introduces the .AsParallel extension to perform parallel operations in LINQ queries. You can also easily enforce the treatment of a data source as if it was ordered by using the .AsOrdered extensions. Some new thread-safe collections have also been added in the System.Collections.Concurrent namespace. You can learn more about these new features from [Parallel Computing on MSDN](http://msdn.microsoft.com/en-us/concurrency/default.aspx) (<http://msdn.microsoft.com/en-us/concurrency/default.aspx>).*

AppFabric Caching

PMN is a data-driven application that gets its data from the application database and the Bing Maps Web services. One of the challenges you might face when building a Web application is managing the needs of a large number of users, including performance and response time. The operations that use the data store and the services used to search for activities can increase the usage of server resources dramatically for items that are shared across many users. For example, many users have access to the public itineraries, so displaying these will generate numerous calls to the database for the same items. Implementing caching at the Web tier will help reduce usage of the resources at the data store and help mitigate latency for recurring searches to the Bing Maps Web services. Figure 8-29 shows the architecture for an application implementing a caching solution at the front-end server.

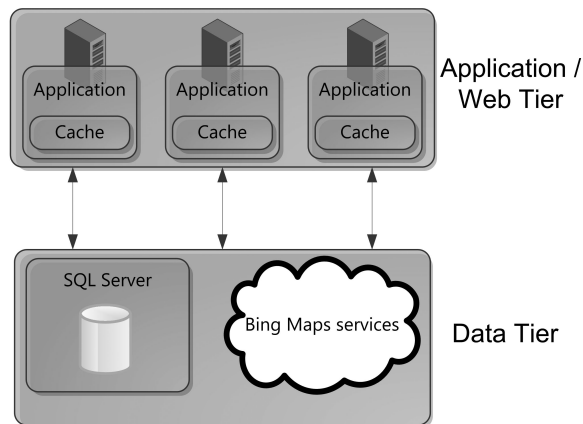


FIGURE 8-29 Typical Web application architecture

Using this approach reduces the pressure on the data layer, but the caching is still coupled to a specific server serving the request. Each Web tier server will have its own cache, but you can still end up with an uneven distribution of the processing to these servers.

Windows Server AppFabric caching offers a distributed, in-memory cache platform. The AppFabric client library allows the application to access the cache as a unified view event if

the cache is distributed across multiple computers, as shown in Figure 8-30. The API provides simple *get* and *set* methods to retrieve and store any serializable common language runtime (CLR) objects easily. The AppFabric cache allows you to add a cache computer on demand, thus making it possible to scale in a manner that is transparent to the client. Another benefit is that the cache can also share copies of the data across the cluster, thereby protecting data against failure.

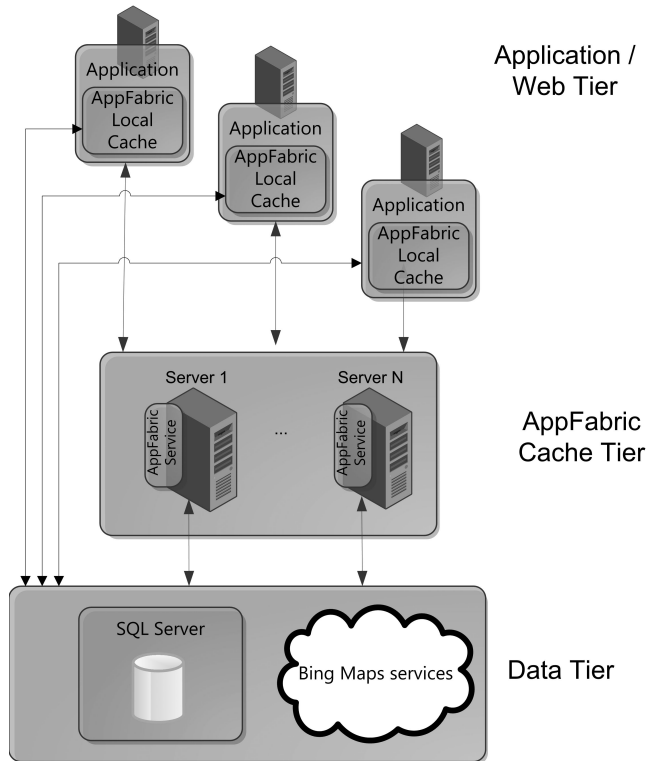


FIGURE 8-30 Web application using Windows Server AppFabric caching

See Also Windows Server AppFabric caching is available as a set of extensions to the .NET Framework 4.0. For more information about how to get, install, and configure Windows Server AppFabric, please visit [Windows Server AppFabric](http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx) (<http://msdn.microsoft.com/en-us/windowsserver/ee695849.aspx>).

See Also PMN can be configured to use either ASP.NET caching or Windows Server AppFabric caching. A complete walkthrough describing how to add Windows Server AppFabric caching to PMN is available here: [PMN: Adding Caching using Velocity](http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/) (<http://channel9.msdn.com/learn/courses/VS2010/ASPNET/EnhancingAspNetMvcPlanMyNight/Exercise-1-Adding-Caching-using-Velocity/>).

Summary

In this chapter, you used a few of the new Visual Studio 2010 features to structure the data layer of the Plan My Night application using the Entity Framework version 4.0 to access a database. You also were introduced to automated entity generation using the ADO.NET Entity Framework POCO templates and to the Windows Server AppFabric caching extensions.

In the next chapter, you will explore how the ASP.NET MVC framework and the Managed Extensibility Framework can help you build great Web applications.

Chapter 9

From 2008 to 2010: Designing the Look and Feel

After reading this chapter, you will be able to

- Create an ASP.NET MVC controller that interacts with the data model
- Create an ASP.NET MVC view that displays data from the controller and validates user input
- Extend the application with an external plug-in using the Managed Extensibility Framework

Web application development in Microsoft Visual Studio has certainly made significant improvements over the years since ASP.NET 1.0 was released. Visual Studio 2008 introduced official support for AJAX-enabled Web pages, Language Integrated Query (LINQ), plus many other improvements to help developers create efficient applications that were easy to manage.

The spirit of improvement to assist developers in creating world-class applications is very much alive in Visual Studio 2010. In this chapter, we'll explore some of the new features as we add functionality to the Plan My Night companion application.



Note The companion application is an ASP.NET MVC 2 project, but a Web developer has a choice in Visual Studio 2010 to use this new form of ASP.NET application or the more traditional ASP.NET (referred to in the community as *Web Forms for distinction*). ASP.NET 4.0 has many improvements to help developers and is still a very viable approach to creating Web applications.

We'll be using a modified version of the companion application's solution to work our way through this chapter. If you installed the companion content in the default location, the correct solution can be found at Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 9\ in a folder called UserInterface-Start.

Introducing the PlanMyNight.Web Project



Note ASP.NET MVC 1.0 Framework is available as an extension to Visual Studio 2008; however, this chapter was written in the context of the user having a default installation of Visual Studio 2008, which only had support for ASP.NET Web Forms 3.5 projects.

The user interface portion of Plan My Night in Visual Studio 2010 was developed as an ASP.NET MVC application, the layout of which differs from what a developer might be accustomed to when developing an ASP.NET Web Forms application in Visual Studio 2008. Some items in the project (as seen in Figure 9-1) will look familiar (such as Global.asax), but others are completely new, and some of the structure is required by the ASP.NET MVC framework.

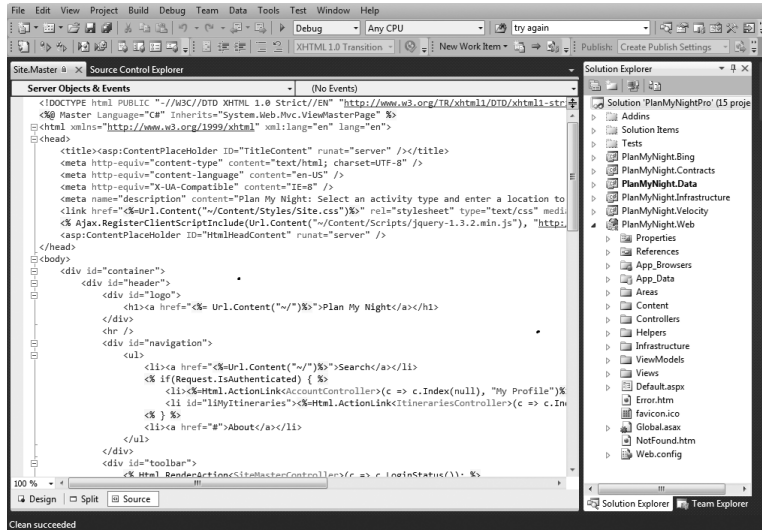


FIGURE 9-1 PlanMyNight.Web project view

Here are the items required by ASP.NET MVC:

- **Areas** This folder is used by the ASP.NET MVC framework to organize large Web applications into smaller components, without using separate solutions or projects. This feature is not used in the Plan My Night application but is called out because this folder is created by the MVC project template.
- **Controllers** During request processing, the ASP.NET MVC framework looks for controllers in this folder to handle the request.
- **Views** The Views folder is actually a structure of folders. The layer immediately inside the Views folder is named for each of the classes found in the Controllers folder, plus a Shared folder. The Shared subfolder is for common views, partial views, master pages, and anything else that will be available to all controllers.

See Also More information about ASP.NET MVC components, as well as how its request processing differs from ASP.NET Web Forms, can be found at <http://asp.net/mvc>.

In most cases, the web.config file is the last file in a project's root folder. However, it has received a much-needed update in Visual Studio 2010: Web.config Transformation. This feature allows for a base web.config file to be created but then to have build-specific web.config files override the settings of the base at build, deployment, and run times. These files appear under the base web.config file, as seen in Figure 9-2.

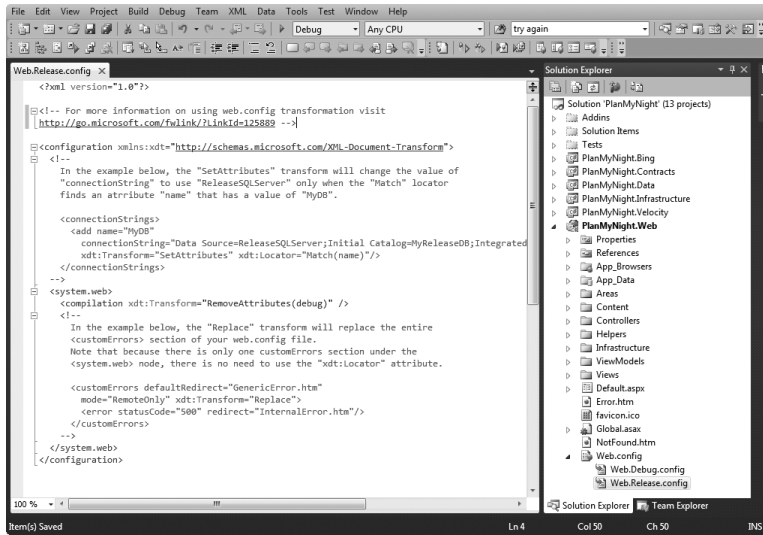


FIGURE 9-2 A web.config file with build-specific files expanded

Visual Studio 2008 When working on a project in Visual Studio 2008, do you recall needing to remember not to overwrite the web.config file with your debug settings? Or needing to remember to update web.config when it was published for a retail build with the correct settings? This is no longer an issue in Visual Studio 2010. The settings in the web.Release.config.retail file will be used during release builds to override the values in web.config, and the same goes for web.Debug.config in debug builds.

Other sections of the project include the following:

- **Content** A collection of folders containing images, scripts, and style files
- **Helpers** Includes miscellaneous classes, containing a number of extension methods, that add functionality to types used in the project
- **Infrastructure** Contains items related to dealing with the lower level infrastructure of ASP.NET MVC (for example, caching and controller factories)
- **ViewModels** Contains data entities filled out by controller classes and used by views to display data

Running the Project

If you compile and run the project, you should see a screen similar to Figure 9-3.

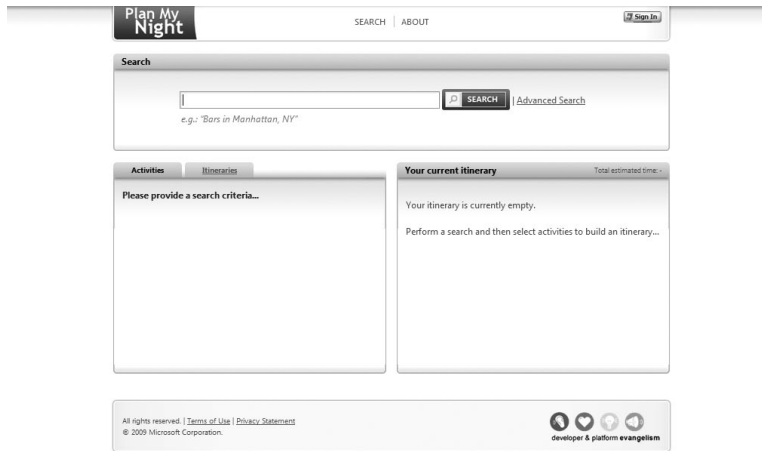


FIGURE 9-3 Default page of the Plan My Night application

The searching functionality and the ability to organize an initial list of itinerary items all work, but if you attempt to save the itinerary you are working on, or if you log in with Windows Live ID, the application will return a 404 Not Found error screen (as shown in Figure 9-4).

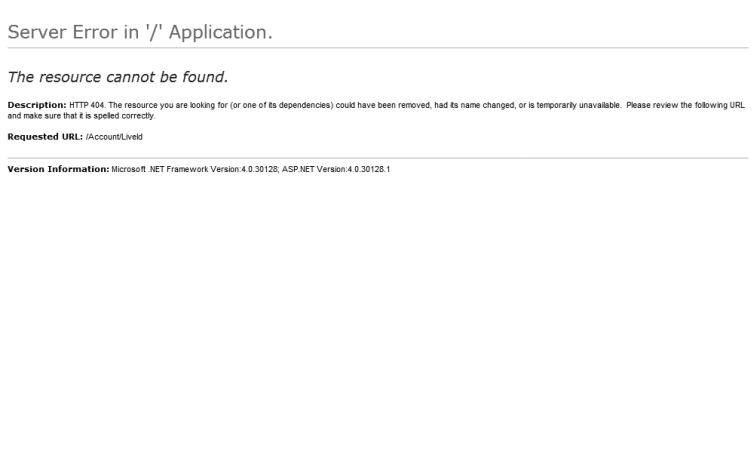


FIGURE 9-4 Error screen returned when logging into the Plan My Night application

You get this error message because currently the project does not include an account controller to handle these requests.

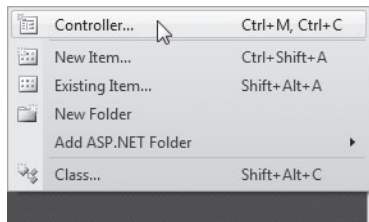
Creating the Account Controller

The *AccountController* class provides some critical functionality to the companion Plan My Night application:

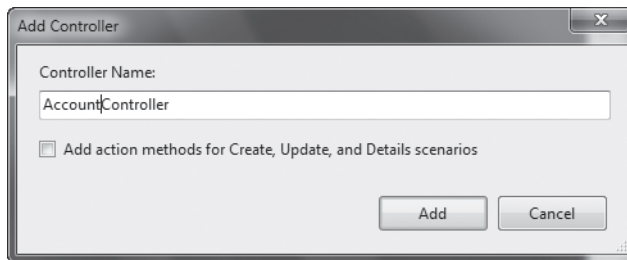
- It handles signing users in and out of the application (via Windows Live ID).
- It provides actions for displaying and updating user profile information.

To create a new ASP.NET MVC controller:

1. Use Solution Explorer to navigate to the Controllers folder in the PlanMyNight.Web project, and click the right mouse button.
2. Open the Add submenu and select the Controller item.



3. Fill in the name of the controller as *AccountController*.



Note Leave the Add Action Methods For Create, Update And Details Scenarios check box blank. Selecting the box inserts some "starter" action methods, but because you will not be using the default methods, there is no reason to create them.

After you click the Add button in the Add Controller dialog box, you should have a basic *AccountController* class open, with a single *Index* method in its body:

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

```
using System.Web.Mvc;

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    public class AccountController : Controller
    {
        //
        // GET: /Account/

        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Visual Studio 2008 A difference to be noted from developing ASP.NET Web Forms applications in Visual Studio 2008 is that ASP.NET MVC applications do not have a companion code-behind file for each of their .aspx files. Controllers like the one you are currently creating perform the logic required to process input and prepare output. This approach allows for a clear separation of display and business logic, and it's a key aspect of ASP.NET MVC.

Implementing the Functionality

To communicate with any of the data layers and services (the Model), you'll need to add some instance fields and initialize them. Before that, you need to add some namespaces to your using block:

```
using System.IO;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using System.Collections.Specialized;
using WindowsLiveId;
```

Now, let's add the instance fields. These fields are interfaces to the various sections of your Model:

```
public class AccountController : Controller
{
    private readonly IWindowsLiveLogin windowsLogin;
    private readonly IMembershipService membershipService;
    private readonly IFormsAuthentication formsAuthentication;
    private readonly IReferenceRepository referenceRepository;
    private readonly IActivitiesRepository activitiesRepository;
    .
    .
    .
```



Note Using interfaces to interact with all external dependencies allows for better portability of the code to various platforms. Also, during testing, dependencies can be mimicked much easier when using interfaces, making for more efficient isolation of a specific component.

As mentioned, these fields represent parts of the model this controller will interact with to meet its functional needs. Here are the general descriptions for each of the interfaces:

- **IWindowsLiveLogin** Defines a functionality contract for interacting with the Windows Live ID service.
- **IMembershipService** Defines user profile information and authorization methods. In your companion application, it is an abstraction of the ASP.NET Membership Service.
- **IFormsAuthentication** Defines functionality for interacting with ASP.NET Forms Authentication abstraction.
- **IRepository** Defines reference resources, such as lists of states and other model-specific information.
- **IActivitiesRepository** An interface for retrieving and updating activity information.

You'll add two constructors to this class: one for general run-time use, which uses the *ServiceFactory* class to get references to the needed interfaces, and one to enable tests to inject specific instances of the interfaces to use.

```
public AccountController() :
    this(
        new ServiceFactory().GetMembershipService(),
        new WindowsLiveLogin(true),
        new FormsAuthenticationService(),
        new ServiceFactory().GetReferenceRepositoryInstance(),
        new ServiceFactory().GetActivitiesRepositoryInstance())
{
}
public AccountController(
    IMembershipService membershipService,
    IWindowsLiveLogin windowsLogin,
    IFormsAuthentication formsAuthentication,
    IReferenceRepository referenceRepository,
    IActivitiesRepository activitiesRepository)
{
    this.membershipService = membershipService;
    this.windowsLogin = windowsLogin;
    this.formsAuthentication = formsAuthentication;
    this.referenceRepository = referenceRepository;
    this.activitiesRepository = activitiesRepository;
}
```

Authenticating the User

The first real functionality you'll implement in this controller is that of signing in and out of the application. Most of the methods you'll implement later require authentication, so this is a good place to start.

The companion application uses a few technologies together at the same time to give the user a smooth authentication experience: Windows Live ID, ASP.NET Forms Authentication, and ASP.NET Membership Services. These three technologies are used in the `LiveID` action you'll implement next.

Start by creating the following method, in the `AccountController` class:

```
public ActionResult LiveId()
{
    return Redirect("~/");
}
```

This method will be the primary action invoked when interacting with the Windows Live ID services. Right now, if it is invoked, it will just redirect the user to the root of the application.



Note The call to `Redirect` returns `RedirectResult`, and although this example uses a string to define the target of the redirection, various overloads can be used for different situations.

A few different types of actions can be taken when Windows Live ID returns a user to your application. The user can be signing into Windows Live ID, signing out, or clearing the Windows Live ID cookies. Windows Live ID uses a query string parameter called `action` on the URL when it returns a user, so you'll use a switch to branch the logic depending on the value of the parameter.

Add the following to the `LiveId` method above the return statement:

```
string action = Request.QueryString["action"];
switch (action)
{
    case "logout":
        this.formsAuthentication.SignOut();
        return Redirect("~/");

    case "clearcookie":
        this.formsAuthentication.SignOut();
        string type;
        byte[] content;
        this.windowsLogin.GetClearCookieResponse(out type, out content);
        return new FileStreamResult(new MemoryStream(content), type);
}
```

See Also Full documentation of the Windows Live ID system can be found on the <http://dev.live.com/> Web site.

The code you just added handles the two sign-out actions for Windows Live ID. In both cases, you use the *IFormsAuthentication* interface to remove the ASP.NET Forms Authentication cookie so that any future http requests (until the user signs in again) will not be considered authenticated. In the second case, you went one step further to clear the Windows Live ID cookies (the ones that remember your login name but not your password).

Handling the sign-in scenario requires a bit more code because you have to check whether the authenticating user is in your Membership Database and, if not, create a profile for the user. However, before that, you must pass the data that Windows Live ID sent you to your Windows Live ID interface so that it can validate the information and give you a *WindowsLiveLogin.User* object:

```
default:
    // login
    NameValueCollection tokenContext;
    if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
    {
        tokenContext = Request.Form;
    }
    else
    {
        tokenContext = new NameValueCollection(Request.QueryString);
        tokenContext["stoken"] =
            System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
    }

    var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
```

At this point in the case for logging in, either *liveIdUser* will be a reference to an authenticated *WindowsLiveLogin.User* object or it will be *null*. With this in mind, you can add your next section of the code, which takes action when the *liveIdUser* value is not *null*:

```
if (liveIdUser != null)
{
    var returnUrl = liveIdUser.Context;
    var userId = new Guid(liveIdUser.Id).ToString();
    if (!this.membershipService.ValidateUser(userId, userId))
    {
        this.formsAuthentication.SignIn(userId, false);
        this.membershipService.CreateUser(userId, userId, string.Empty);
        var profile = this.membershipService.CreateProfile(userId);
        profile.FullName = "New User";
        profile.State = string.Empty;
        profile.City = string.Empty;
        profile.PreferredActivityTypeId = 0;
        this.membershipService.UpdateProfile(profile);

        if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
        return RedirectToAction("Index", new { returnUrl = returnUrl });
    }
    else
    {

```

```

        this.formsAuthentication.SignIn(userId, false);
        if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
        return Redirect(returnUrl);
    }
}
break;

```

The call to the *ValidateUser* method on the *IMembershipService* reference allows the application to check whether the user has been to this site before and whether there will be a profile for the user. Because the user is authenticated with Windows Live ID, you are using the user's ID value (which is a GUID) as both the user name and password to the ASP.NET Membership Service.

If the user does not have a user record with the application, you create one by calling the *CreateUser* method and then also create a user settings profile via *CreateProfile*. The profile is filled with some defaults and saved back to its store, and the user is redirected to the primary input page so that he can update the information.



Note *Controller.RedirectToAction* determines which URL to create based on the combination of input parameters. In this case, you want to redirect the user to the *Index* action of this controller, as well as pass the current return URL value.

The other action that takes place in this code is that the user is signed into ASP.NET Forms authentication so that a cookie will be created, providing identity information on future requests that require authentication.

The settings profile is managed by ASP.NET Membership Services as well and is declared in the web.config file of the application:

```

<system.web>
...
<profile enabled="true">
  <properties>
    <add name="FullName" type="string" />
    <add name="State" type="string" />
    <add name="City" type="string" />
    <add name="PreferredActivityTypeId" type="int" />
  </properties>

  <providers>
    <clear />
    <add name="AspNetSqlProfileProvider"
type="System.Web.Profile.SqlProfileProvider,
      System.Web, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f7f11d50a3a"
      connectionStringName="ApplicationServices"
      applicationName="/" />
  </providers>
</profile>
...
</system.web>

```

At this point, the *LiveID* method is complete and should look like the following code. The application can now take authentication information from Windows Live ID, prepare an ASP.NET MembershipService profile, and create an ASP.NET Forms Authentication ticket.

```
public ActionResult LiveId()
{
    string action = Request.QueryString["action"];
    switch (action)
    {
        case "logout":
            this.formsAuthentication.SignOut();
            return Redirect("~/");

        case "clearcookie":
            this.formsAuthentication.SignOut();
            string type;
            byte[] content;
            this.windowsLogin.GetClearCookieResponse(out type, out content);
            return new FileStreamResult(new MemoryStream(content), type);

        default:
            // login
            NameValueCollection tokenContext;
            if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
            {
                tokenContext = Request.Form;
            }
            else
            {
                tokenContext = new NameValueCollection(Request.QueryString);
                tokenContext["stoken"] =
                    System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
            }

            var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);

            if (liveIdUser != null)
            {
                var returnUrl = liveIdUser.Context;
                var userId = new Guid(liveIdUser.Id).ToString();
                if (!this.membershipService.ValidateUser(userId, userId))
                {
                    this.formsAuthentication.SignIn(userId, false);
                    this.membershipService.CreateUser(userId, userId, string.Empty);
                    var profile = this.membershipService.CreateProfile(userId);
                    profile.FullName = "New User";
                    profile.State = string.Empty;
                    profile.City = string.Empty;
                    profile.PreferredActivityTypeId = 0;
                    this.membershipService.UpdateProfile(profile);

                    if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
                    return RedirectToAction("Index", new { returnUrl = returnUrl });
                }
            }
    }
}
```

```

        else
        {
            this.formsAuthentication.SignIn(userId, false);
            if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
            return Redirect(returnUrl);
        }
    }
    break;
}
return Redirect("~/");
}

```

Of course, the user has to be able to get to the Windows Live ID login page in the first place before logging in. Currently in the Plan My Night application, there is a Windows Live ID login button. However, there are cases where the application will want the user to be redirected to the login page from code. To cover this scenario, you need to add a small method called *Login* to your controller:

```

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
    return Redirect(redirect);
}

```

This method simply retrieves the login URL for Windows Live and redirects the user to that location. This also satisfies a configuration value in your web.config file for ASP.NET Forms Authentication in that any request requiring authentication will be redirected to this method:

```

<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" name="XAUTH" timeout="2880" path="~/\" />
</authentication>

```

Retrieving the Profile for the Current User

Now with the authentication methods defined, which satisfies your first goal for this controller—signing users in and out in the application—you can move on to retrieving data for the current user.

The *Index* method, which is the default method for the controller based on the URL mapping configuration in Global.asax, will be where you retrieve the current user's data and

return a view displaying that data. The *Index* method that was initially created when the *AccountController* class was created should be replaced with the following:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Index(string returnUrl)
{
    var profile = this.membershipService.GetCurrentProfile();
    var model = new ProfileViewModel
    {
        Profile = profile,
        ReturnUrl = returnUrl ?? this.GetReturnUrl()
    };

    this.InjectStatesAndActivityTypes(model);

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

Visual Studio 2008 Attributes, such as *[Authorize()]*, might not have been in common use in Visual Studio 2008; however, ASP.NET MVC makes use of them often. Attributes allow for metadata to be defined about the target they decorate. This allows for the information to be examined at run time (via reflection) and for action to be taken if deemed necessary.

The *Authorize* attribute is very handy because it declares that this method can be invoked only for http requests that are already authenticated. If a request is not authenticated, it will be redirected to the ASP.NET Forms Authentication configured login target, which you just finished setting up. The *AcceptVerbs* attribute also restricts how this method can be invoked, by specifying which http verbs can be used. In this case, you are restricting this method to HTTP GET verb requests. You've added a string parameter, *returnUrl*, to the method signature so that when the user is finished viewing or updating her information, she can be returned to what she was looking at previously.



Note This highlights a part of the ASP.NET MVC framework called Model Binding, details of which are beyond the scope of this book. However, you should know that it attempts to find a source for *returnUrl* (a form field, routing table data, or query string parameter with the same name) and binds it to this value when invoking the method. If the Model Binder cannot find a suitable source, the value will be null. This behavior can cause problems for value types that cannot be null, because it will throw an *InvalidOperationException*.

The main portion of this method is straightforward: it takes the return of the *GetCurrentProfile* method on the ASP.NET Membership Service interface and sets up a view model object for the view to use. The call to *GetReturnUrl* is an example of an extension method defined in the *PlanMyNight.Infrastructure* project. It's not a member of the

Controller class, but in the development environment it makes for much more readable code. (See Figure 9-5.)

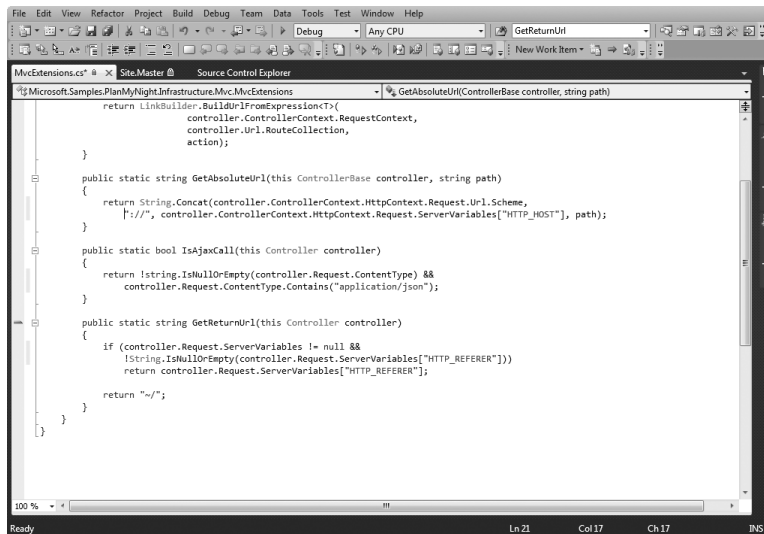


FIGURE 9-5 Example of extension methods in *MvcExtensions.cs*

InjectStatesAndActivityTypes is a method you need to implement in the *AccountController* class. It gathers data from the reference repository for names of states and the activity repository. It makes two collections of *SelectListItem* (an HTML class for MVC): one for the list of states, and the other for the list of different activity types available in the application. It also sets the respective value.

```
private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null && o.Id ==
                profile.PreferredActivityTypeId)
        }).ToList();

    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null && o.Abbreviation ==
                profile.State)
        }).ToList();

    states.Insert(0, new SelectListItem {
```

```
        Text = "Any state",
        Value = string.Empty
    });

    model.PreferredActivityTypes = types;
    model.States = states;
}
}
```

Updating the Profile Data

Having completed the infrastructure needed to retrieve data for the current profile, you can move on to updating the data in the model from a form submission by the user. After this, you can create your view pages and see how all this ties together. The *Update* method is simple; however, it does introduce some new features not seen yet:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
public ActionResult Update(UserProfile profile)
{
    var returnUrl = Request.Form["returnUrl"];
    if (!ModelState.IsValid)
    {
        // validation error
        return this.IsAjaxCall() ? new JsonResult { JsonRequestBehavior =
            JsonRequestBehavior.AllowGet, Data = ModelState }
            : this.Index(returnUrl);
    }

    this.membershipService.UpdateProfile(profile);
    if (this.IsAjaxCall())
    {
        return new JsonResult { JsonRequestBehavior = JsonRequestBehavior.AllowGet,
            Data = new { Update = true, Profile = profile, ReturnUrl = returnUrl } };
    }
    else
    {
        return RedirectToAction("UpdateSuccess", "Account", new { returnUrl =
            returnUrl });
    }
}
}
```

The *ValidateAntiForgeryToken* attribute ensures that the form has not been tampered with. To use this feature, you need to add an *AntiForgeryToken* to your view's input form. The check on the *ModelState* to see whether it is valid is your first look at input validation. This is a look at the server-side validation, and ASP.NET MVC offers an easy-to-use feature to make sure that incoming data meets some rules. The *UserProfile* object that is created for input to this method, via MVC Model Binding, has had one of its properties decorated with a

System.ComponentModel.DataAnnotations.Required attribute. During Model Binding, the MVC framework evaluates *DataAnnotation* attributes and marks the *ModelState* as valid only when all of the rules pass.

In the case where the *ModelState* is not valid, the user is redirected to the *Index* method where the *ModelState* will be used in the display of the input form. Or, if the request was an AJAX call, a *JsonResult* is returned with the *ModelState* data attached to it.

Visual Studio 2008 Because in ASP.NET MVC requests are routed through controllers rather than pages, the same URL can handle a number of requests and respond with the appropriate view. In Visual Studio 2008, a developer would have to create two different URLs and call a method in a third class to perform the functionality.

When the *ModelState* is valid, the profile is updated in the membership service and a JSON result is returned for AJAX requests with the success data, or in the case of "normal" requests, the user is redirected to the *UpdateSuccess* action on the Account controller. The *UpdateSuccess* method is the final method you need to implement to finish off this controller:

```
public ActionResult UpdateSuccess(string returnUrl)
{
    var model = new ProfileViewModel
    {
        Profile = this.membershipService.GetCurrentProfile(),
        ReturnUrl = returnUrl
    };
    return View(model);
}
```

The method is used to return a success view to the browser, display some of the updated data, and provide a link to return the user to where she was when she started the profile update process.

Now that you've reached the end of the Account controller implementation, you should have a class that resembles the following listing:

```
using System;
using System.Collections.Specialized;
using System.IO;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Microsoft.Samples.PlanMyNight.Data;
using Microsoft.Samples.PlanMyNight.Entities;
using Microsoft.Samples.PlanMyNight.Infrastructure;
using Microsoft.Samples.PlanMyNight.Infrastructure.Mvc;
```



```
using Microsoft.Samples.PlanMyNight.Web.ViewModels;
using WindowsLiveId;

namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    [HandleErrorWithContentType()]
    [OutputCache(NoStore = true, Duration = 0, VaryByParam = "")]
    public class AccountController : Controller
    {
        private readonly IWindowsLiveLogin windowsLogin;
        private readonly IMembershipService membershipService;
        private readonly IFormsAuthentication formsAuthentication;
        private readonly IReferenceRepository referenceRepository;
        private readonly IActivitiesRepository activitiesRepository;

        public AccountController() :
            this(
                new ServiceFactory().GetMembershipService(),
                new WindowsLiveLogin(true),
                new FormsAuthenticationService(),
                new ServiceFactory().GetReferenceRepositoryInstance(),
                new ServiceFactory().GetActivitiesRepositoryInstance())
        {
        }

        public AccountController(IMembershipService membershipService,
                                IWindowsLiveLogin windowsLogin,
                                IFormsAuthentication formsAuthentication,
                                IReferenceRepository referenceRepository,
                                IActivitiesRepository activitiesRepository)
        {
            this.membershipService = membershipService;
            this.windowsLogin = windowsLogin;
            this.formsAuthentication = formsAuthentication;
            this.referenceRepository = referenceRepository;
            this.activitiesRepository = activitiesRepository;
        }

        public ActionResult LiveId()
        {
            string action = Request.QueryString["action"];
            switch (action)
            {
                case "logout":
                    this.formsAuthentication.SignOut();
                    return Redirect("~/");
                case "clearcookie":
                    this.formsAuthentication.SignOut();
                    string type;
                    byte[] content;
                    this.windowsLogin.GetClearCookieResponse(out type, out content);
            }
        }
    }
}
```

```

        return new FileStreamResult(new MemoryStream(content), type);
    default:
        // login
        NameValueCollection tokenContext;
        if ((Request.HttpMethod ?? "GET").ToUpperInvariant() == "POST")
        {
            tokenContext = Request.Form;
        }
        else
        {
            tokenContext = new NameValueCollection(Request.QueryString);
            tokenContext["stoken"] =
                System.Web.HttpUtility.UrlEncode(tokenContext["stoken"]);
        }

        var liveIdUser = this.windowsLogin.ProcessLogin(tokenContext);
        if (liveIdUser != null)
        {
            var returnUrl = liveIdUser.Context;
            var userId = new Guid(liveIdUser.Id).ToString();
            if (!this.membershipService.ValidateUser(userId, userId))
            {
                this.formsAuthentication.SignIn(userId, false);
                this.membershipService.CreateUser(
                    userId, userId, string.Empty);
                var profile =
                    this.membershipService.CreateProfile(userId);
                profile.FullName = "New User";
                profile.State = string.Empty;
                profile.City = string.Empty;
                profile.PreferredActivityTypeId = 0;
                this.membershipService.UpdateProfile(profile);
                if (string.IsNullOrEmpty(returnUrl)) returnUrl = null;
                return RedirectToAction("Index", new { returnUrl =
                    returnUrl });
            }
            else
            {
                this.formsAuthentication.SignIn(userId, false);
                if (string.IsNullOrEmpty(returnUrl)) returnUrl = "~/";
                return Redirect(returnUrl);
            }
        }
        break;
    }
    return Redirect("~/");
}

public ActionResult Login(string returnUrl)
{
    var redirect = HttpContext.Request.Browser.IsMobileDevice ?
        this.windowsLogin.GetMobileLoginUrl(returnUrl) :
        this.windowsLogin.GetLoginUrl(returnUrl);
}

```

```
        return Redirect(redirect);
    }

    [Authorize()]
    [AcceptVerbs(HttpVerbs.Get)]
    public ActionResult Index(string returnUrl)
    {
        var profile = this.membershipService.GetCurrentProfile();
        var model = new ProfileViewModel
        {
            Profile = profile,
            ReturnUrl = returnUrl ?? this.GetReturnUrl()
        };

        this.InjectStatesAndActivityTypes(model);
        return View("Index", model);
    }

    [Authorize()]
    [AcceptVerbs(HttpVerbs.Post)]
    [ValidateAntiForgeryToken()]
    public ActionResult Update(UserProfile profile)
    {
        var returnUrl = Request.Form["returnUrl"];
        if (!ModelState.IsValid)
        {
            // validation error
            return this.IsAjaxCall() ?
                new JsonResult { JsonRequestBehavior =
                    JsonRequestBehavior.AllowGet, Data = ModelState }
                : this.Index(returnUrl);
        }
        this.membershipService.UpdateProfile(profile);
        if (this.IsAjaxCall())
        {
            return new JsonResult {
                JsonRequestBehavior = JsonRequestBehavior.AllowGet,
                Data = new {
                    Update = true,
                    Profile = profile,
                    ReturnUrl = returnUrl } };
        }
        else
        {
            return RedirectToAction("UpdateSuccess",
                "Account", new { returnUrl = returnUrl });
        }
    }
    public ActionResult UpdateSuccess(string returnUrl)
    {
        var model = new ProfileViewModel
        {
            Profile = this.membershipService.GetCurrentProfile(),
```

```
        returnUrl = returnUrl
    };
    return View(model);
}

private void InjectStatesAndActivityTypes(ProfileViewModel model)
{
    var profile = model.Profile;
    var types = this.activitiesRepository.RetrieveActivityTypes()
        .Select(o => new SelectListItem { Text = o.Name,
            Value = o.Id.ToString(),
            Selected = (profile != null &&
                o.Id == profile.PreferredActivityTypeId) })
        .ToList();
    types.Insert(0, new SelectListItem { Text = "Select...", Value = "0" });
    var states = this.referenceRepository.RetrieveStates().Select(
        o => new SelectListItem {
            Text = o.Name,
            Value = o.Abbreviation,
            Selected = (profile != null &&
                o.Abbreviation == profile.State) })
        .ToList();

    states.Insert(0,
        new SelectListItem { Text = "Any state",
            Value = string.Empty });
    model.PreferredActivityTypes = types;
    model.States = states;
}
}
```

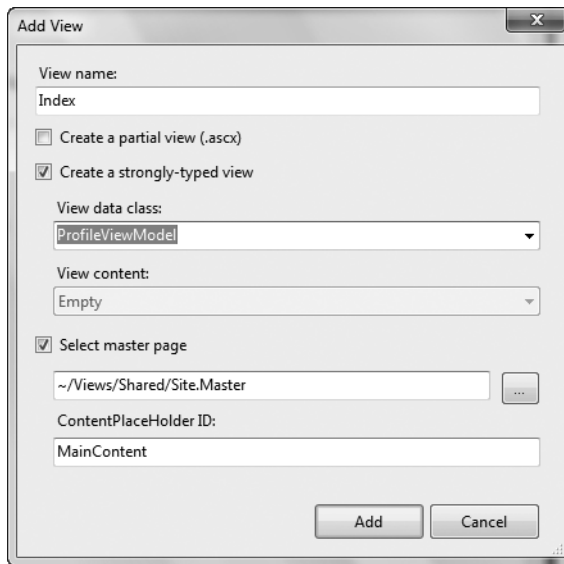
Creating the Account View

In the previous section, you created a controller with functionality that allows a user to update her information and view it. In this section you're going to walk through the Visual Studio 2010 features that enable you to create the views that display this functionality to the user.

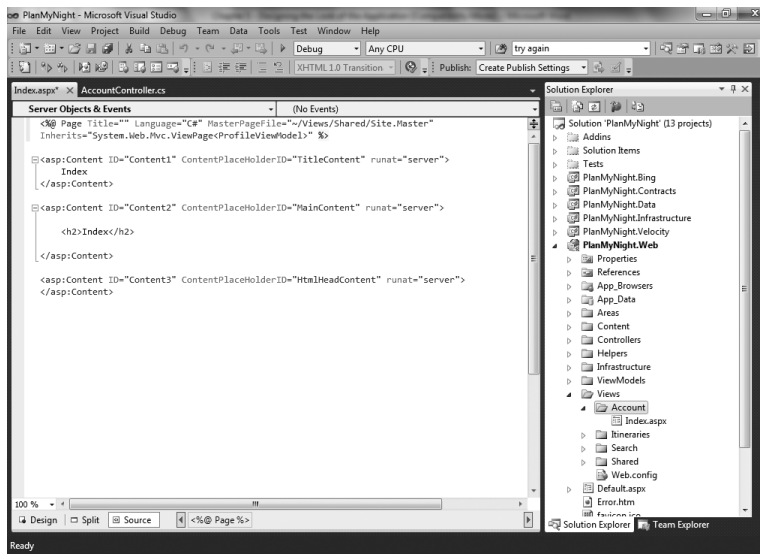
To create the Index view for the Account controller:

1. Navigate to the Views folder in the PlanMyNight.Web project.
2. Click the right mouse button on the Views folder, expand the Add submenu, and select New Folder.
3. Name the new folder **Account**.

- Click the right mouse button on the new Account folder, expand the Add submenu, and select View.
- Fill out the Add View dialog box as shown here:



- Click Add. You should see an HTML page with some `<asp:Content>` controls in the markup:



You might notice that it doesn't look much different from what you are used to seeing in Visual Studio 2008. By default, ASP.NET MVC 2 uses the ASP.NET Web Forms view engine, so there will be some commonality between MVC and Web Forms pages. The primary differences at this point are that the *page* class derives from *System.Web.Mvc.ViewPage<ProfileViewModel>* and there is no code-behind file. MVC does not use code-behind files, like ASP.NET Web Forms does, to enforce a strict separation of concerns. MVC pages are generally edited in markup view; the designer view is primarily for ASP.NET Web Forms applications.

For this page skeleton to become the main view for the Account controller, you should change the title content to be more in line with the other views:

```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Plan My Night - Profile
</asp:Content>
```

Next you need to add the client scripts you are going to use in the content placeholder for the *HtmlHeadContent*:

```
<asp:Content ID="Content3" ContentPlaceHolderID="HtmlHeadContent" runat="server">
  <% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery-1.3.2.min.js"),
    "http://ajax.microsoft.com/ajax/jquery/jquery-1.3.2.min.js"); %>
  <% Ajax.RegisterClientScriptInclude(
    Url.Content("~/Content/Scripts/jquery.validate.js"),
    "http://ajax.microsoft.com/ajax/jquery.validate/1.5.5/jquery.validate.min.js"); %>
  <% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/MicrosoftMvcJQueryValidation.js"), "pmn"); %>
  <% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.common.js"), "pmn"); %>
  <% Ajax.RegisterCombinedScriptInclude(
    Url.Content("~/Content/Scripts/ajax.profile.js"), "pmn"); %>
  <%= Ajax.RenderClientScripts() %>
</asp:Content>
```

This script makes use of extension methods for the *System.Web.Mvc.AjaxHelper*, which are found in the PlanMyNight.Infrastructure project, under the MVC folder.

With the head content set up, you can look at the main content of the view:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <div class="panel" id="profileForm">
    <div class="innerPanel">
      <h2><span>My Profile</span></h2>
      <% Html.EnableClientValidation(); %>
      <% using (Html.BeginForm("Update", "Account")) %>
      <% { %>
      <%=Html.AntiForgeryToken()%>
      <div class="items">
```

```

<fieldset>
  <p>
    <label for="FullName">Name:</label>
    <%=Html.EditorFor(m => m.Profile.FullName)%>
    <%=Html.ValidationMessage("Profile.FullName",
      new { @class = "field-validation-error-wrapper" })%>
  </p>
  <p>
    <label for="State">State:</label>
    <%=Html.DropDownListFor(m => m.Profile.State, Model.States)%>
  </p>
  <p>
    <label for="City">City:</label>
    <%=Html.EditorFor(m => m.Profile.City, Model.Profile.City)%>
  </p>
  <p>
    <label for="PreferredActivityTypeId">Preferred activity:</label>
    <%=Html.DropDownListFor(m =>
      m.Profile.PreferredActivityTypeId,
      Model.PreferredActivityTypes)%>
  </p>
</fieldset>
<div class="submit">
  <%=Html.Hidden("returnUrl", Model.ReturnUrl)%>
  <%=Html.SubmitButton("submit", "Update")%>
</div>
<div class="toolbox"></div>
<% } %>
</div>
</div>
</asp:Content>

```

Aside from some inline code, this looks to be fairly normal HTML markup. We're going to focus our attention on the inline code pieces to demonstrate the power they bring (as well as the simplicity).

Visual Studio 2008 In Visual Studio 2008, it was more commonplace to use server-side controls to display data, and other display-time logic. However, because ASP.NET MVC view pages do not have a code-behind file, server-side logic executed in the view at render time must be done in the same file with the markup. ASP.NET Web Forms controls can still be used. Our example makes use of the `<asp:Content>` control. However, the functionality of ASP.NET Web Forms controls is generally limited because there is no code-behind file.

MVC makes a lot of use of what is known as HTML helpers. The methods contained under *System.Web.Mvc.HtmlHelper* emit small, standards-compliant HTML tags for various uses. This requires the MVC developer to type more markup than a Web Forms developer in some cases, but the developer has more direct control over the output. The strongly typed version

of this extension class (*HtmlHelper<TModel>*) can be referenced in the view markup via the *ViewPage<TModel>.Html* property.

These are the *HTML* methods used in this form, which are only a fraction of what is available by default:

- *Html.EnableClientValidation* enables data validation to be performed on the client side based on the strongly typed *ModelState* dictionary.
- *Html.BeginForm* places a `<form>` tag in the markup and closes the form at the end of the *using* section. It takes various parameters for options, but the most common parameter is the name of the action and the controller to invoke that action on. This allows the MVC framework to generate the specific URL to target the form to at run time, rather than having to input a string URL into the markup.
- *Html.AntiForgeryToken* places a hidden field in the form with a check value that is also stored in a cookie in the visitor's browser and validated when the target of the form has the *ValidateAntiForgeryToken* attribute. Remember that you added this attribute to the *Update* method in the controller.
- *Html.EditorFor* is an overloaded method that inserts a text box into the markup. This is the strongly typed version of the *Html.Editor* method.
- *Html.DropDownListFor* is an overloaded method that places a drop-down list into the markup. This is the strongly typed version of the *Html.DropDownList* method.
- *Html.ValidationMessage* is a helper that will display a validation error message when a given key is present in the *ModelState* dictionary.
- *Html.Hidden* places a hidden field in the form, with the name and value that is passed in.
- *Html.SubmitButton* creates a Submit button for the form.

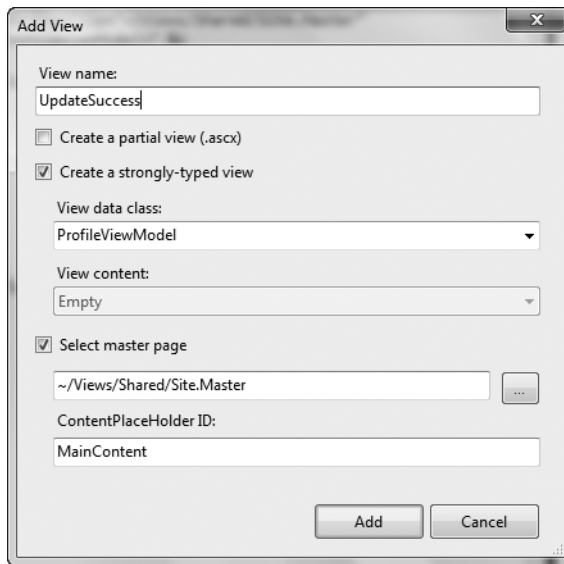


Note With the Index view markup complete, you only need to add the view for the *UpdateSuccess* action before you can see your results.

To create the *UpdateSuccess* view:

1. Expand the *PlanMyNight.Web* project in Solution Explorer, and then expand the Views folder.
2. Click the right mouse button on the Account folder.
3. Open the Add submenu, and click View.

4. Fill out the Add View dialog box so that it looks like this:



After the view page is created, fill in the title content so that it looks like this:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">Plan My Night - Profile Updated</asp:Content>
```

And the placeholder for *MainContent* should look like this:

```
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
<div class="panel" id="profileForm">
  <div class="innerPanel">
    <h2><span>My Profile</span></h2>
    <div class="items">
      <p>Your profile has been successfully updated.</p>
      <h3>> <a href="<%=Html.AttributeEncode(Model.ReturnUrl ??
        Url.Content("~/"))%>">Continue</a></h3>
    </div>
  </div>
</div>
```

To see the views created, you must perform an edit to the *Site.Master* file (located in the *Views/Shared* folder from the Web project's root). Line 33 of the file is commented out, and the comment tags should be removed so that it matches the following example:

```
<%=Html.ActionLink<AccountController>(c => c.Index(null), "My Profile")%>
```

With this last view created, you can now compile and launch the application. Click the Sign In button, as seen in the top right corner of Figure 9-6, and sign in to Windows Live ID.

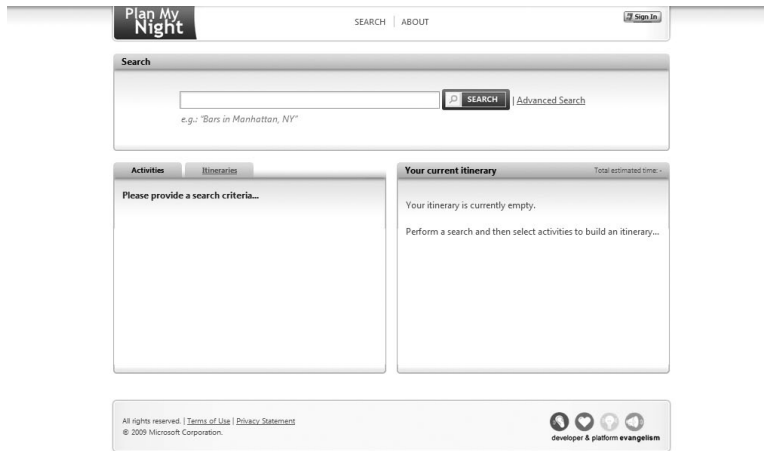


FIGURE 9-6 Plan My Night default screen

After you've signed in, you should be redirected to the Index view of the Account controller you created, shown in Figure 9-7.

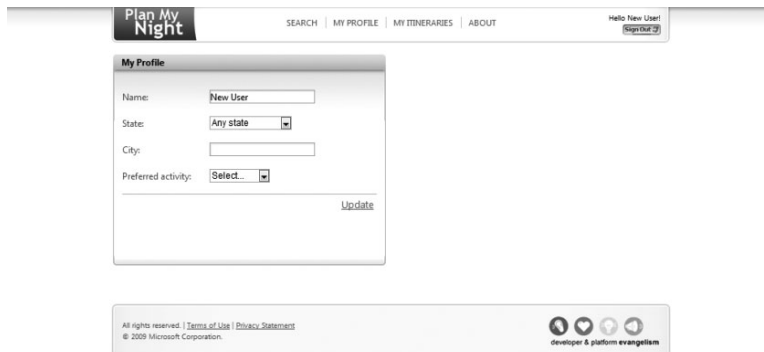
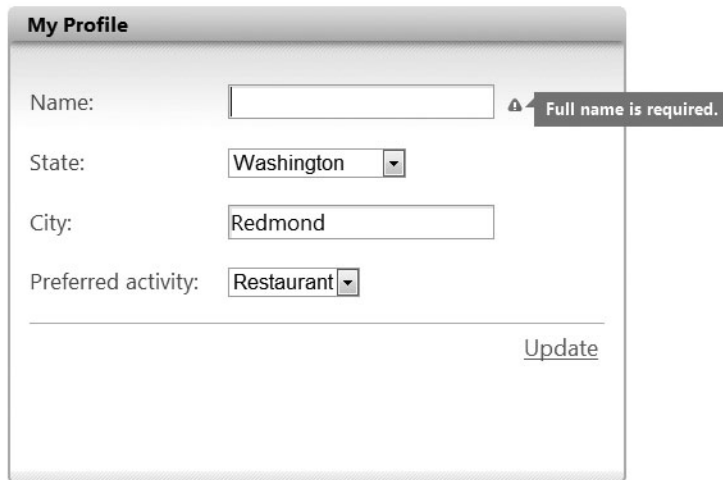


FIGURE 9-7 Profile settings screen returned from the *Index* method of the Account controller

If instead you are returned to the search page, just click the My Profile link, located in the links at the center and top of the interface. To see the new data-validation features at work,

try to save the form without filling in the Full Name field. You should get a result that looks like Figure 9-8.



The screenshot shows a web form titled "My Profile" with the following fields: "Name:" (text input), "State:" (dropdown menu with "Washington" selected), "City:" (text input with "Redmond" entered), and "Preferred activity:" (dropdown menu with "Restaurant" selected). A red error message box with a warning icon is positioned over the "Name" field, displaying the text "Full name is required." Below the form fields is an "Update" button.

FIGURE 9-8 Example of failed validation during Model Binding checks

Because you enabled client-side validation, there was no postback. To see the server-side validation work, you would have to edit the `Index.aspx` file in the Account folder and comment out the call to `Html.EnableClientValidation`. The tight integration and support of AJAX and other JavaScript in MVC applications allows for server-side operations such as validation to be moved to the client side much more easily than they were previously.

Visual Studio 2008 In ASP.NET MVC applications, the value of the ID attribute for a particular HTML element is not transformed, like they are in ASP.NET Web Forms 3.5. In Visual Studio 2008, a developer would have to make sure to set the *UniqueID* of a control/element into a JavaScript variable so that it could be accessed by external JavaScript. This was done to make sure the ID was unique. However, it was always an extra layer of complexity to the interaction between ASP.NET 3.5 Web Forms controls and JavaScript. In MVC, this transformation does not happen, but it is up to the developers to ensure uniqueness of the ID. It should also be noted that ASP.NET 4.0 Web Forms now supports disabling the ID transformation on a per-control basis, if the developer so wishes.

With the completed Account controller and related views, you have filled in the missing "core" functionality of Plan My Night, while taking a brief tour of some new features in Visual Studio 2010 and MVC 2.0 applications. But MVC is not the only choice for Web developers.

ASP.NET Web Forms has been the primary application type for ASP.NET since it was released, and it continues to be improved upon in Visual Studio 2010. In the next section, we'll explore creating an ASP.NET Web Form with the Visual Designer to be used in the MVC application.

Using the Designer View to Create a Web Form

Applications will encounter an unexpected condition at some point in their lifetime of use. The companion application is no different, and when it does encounter an unexpected condition, it returns an error screen like that shown in Figure 9-9.

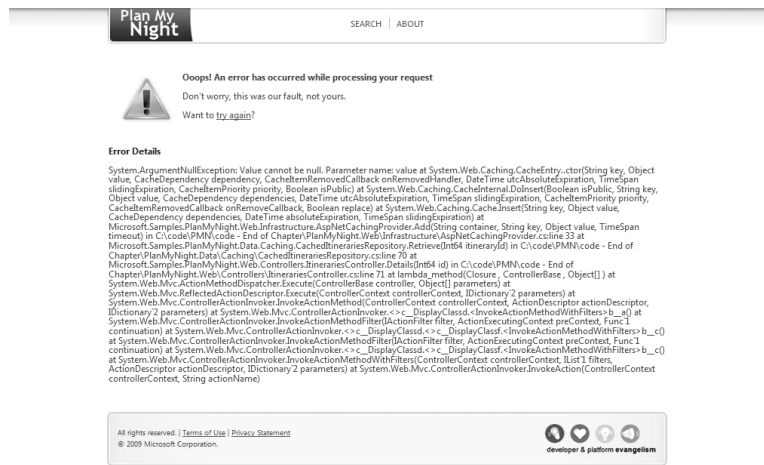
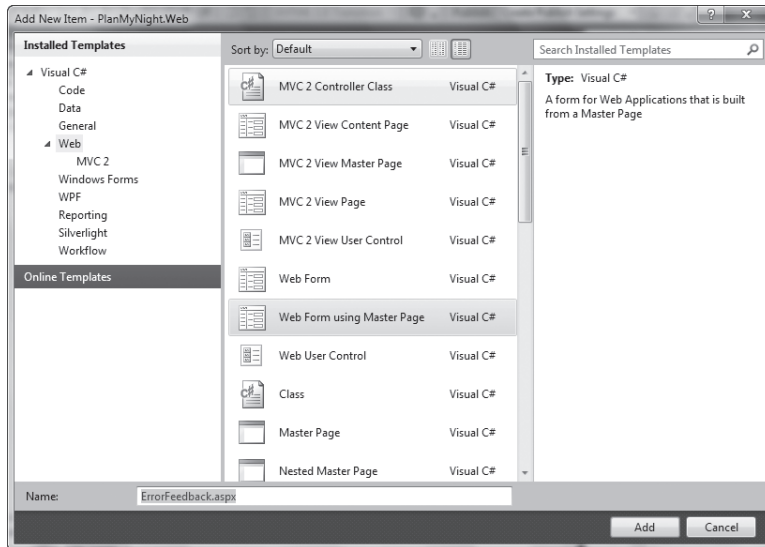


FIGURE 9-9 Example of an error screen in the Plan My Night application

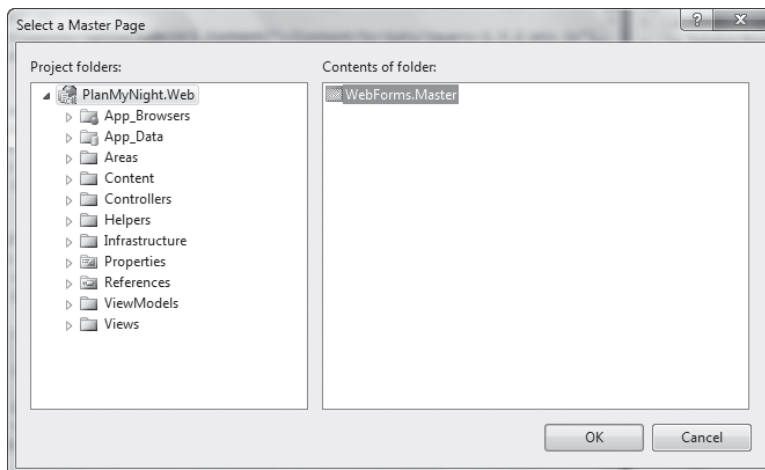
Currently, a user who sees this screen really has only the option of trying his action again or using the navigation links along the top area of the application. (Of course, that might also cause another error.) Adding an option for the user to provide feedback allows the developers to gain information about the situation that might not be apparent by using the standard exception message and stack trace. To show a different way to create a user interface component for Plan My Night, the error feedback page is going to be created as an ASP.NET Web Form using primarily the Designer view in Visual Studio. Before you can begin designing the form, you need to create a base form file to work from.

To create a new Web form:

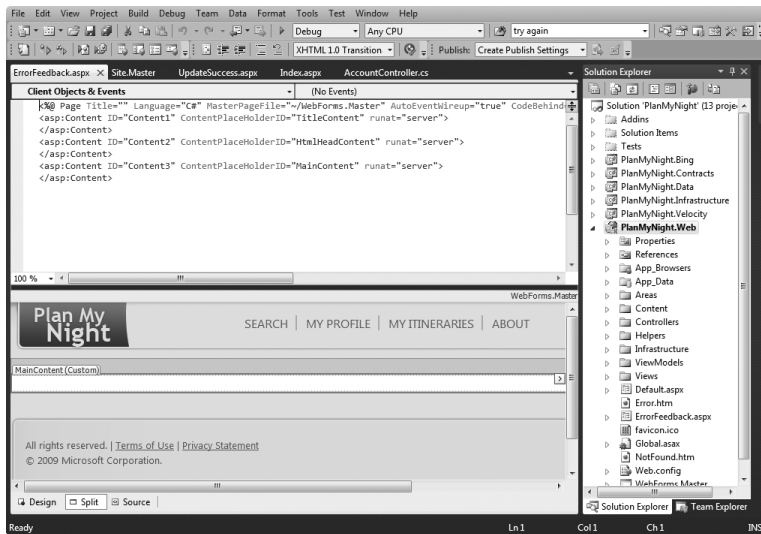
1. Open the context menu on the PlanMyNight.Web project (by clicking the right mouse button), open the Add submenu, and select New Item.
2. In the Add New Item dialog box, select Web Form Using Master Page and call the item **ErrorFeedback.aspx** in the Name field.



3. The dialog screen to associate a master page with this Web form will appear. On the Project Folders side, ensure that the main PlanMyNight.Web folder is selected and then select the WebForms.Master item on the right.



- The resulting page can be shown in the source mode (or Design view) instead of Split view. Switch the view to Split (located at the bottom of the window, just like in previous Visual Studio versions). When you are done, the screen should look similar to this:



Note Split view is recommended so that you can see the source the designer is generating and to add extra markup as needed.

It's a good idea to pin the control toolbox open on the screen because you'll be dragging controls and elements to the content area during this section. The toolbox, if not present already, can be found under the View menu.

Start by dragging a `div` element (under the HTML group) from the toolbox into the `MainContent` section of the designer. A `div` tab will appear, indicating that the new element you added is the currently selected element. Open the context menu for the `div`, and choose Properties (which can also be opened by pressing the F4 key). With the Properties window open, edit the (*Id*) property to have a value of `profileForm`. (Casing is important.) Also, change the `Class` property to have a value of `panel`. After editing the values, the size of your content area will have changed, because CSS is applied in the Design view.

Drag another `div` inside the first one, and set its `class` property to `innerPanel`. In the markup panel, add the following markup to the `innerPanel`:

```
<h2><span>Error Feedback</span></h2>
```

After the close of the `<h2>` tag, add a new line and open the context menu. Choose Insert Snippet, and follow the click path of ASP.NET > form. This will create a server-side form tag for you to insert Web controls into. Inside the form tag, place a div tag with the class attribute set to *items* and then a fieldset tag inside the div tag.

Next drag a TextBox control (found under Standard) from the toolbox and drop it inside the fieldset tag. Set the ID of the text box to *FullName*. Add a `<label>` tag before this control in the markup view, set its *for* property to the ID of the text box, and set its value to **Full Name:** (making sure to include the colon). To set the value of a `<label>` tag, place the text between the `<label>` and `</label>` tags. Surround these two elements with a `<p>`, and you should have something like Figure 9-10 in the Design view.

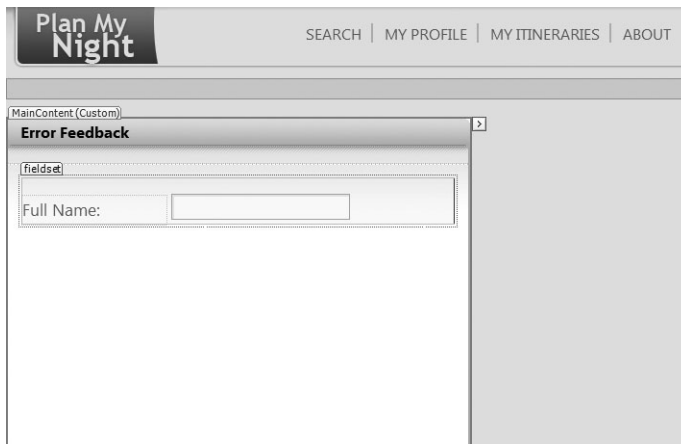


FIGURE 9-10 Current state of ErrorFeedback.aspx in the Design view

Add another text box and label it in a similar manner as the first, but set the ID of the text box to **EmailAddress** and the label value to **Email Address:** (making sure to include the colon). Repeat the process a third time, setting the TextBox ID and label value to **Comments**. There should now be three labels and three single-line TextBox controls in the Design view. The Comments control needs multiline input, so open its property page and set *TextMode* to *Multiline*, *Rows* to *5*, and *Columns* to *40*. This should create a much wider text box in which the user can enter comments.

Use the Insert Snippet feature again, after the Comments text box, and insert a "div with class" tag (`HTML>div`). Set the class of the div tag to *submit*, and drag a Button control from the toolbox into this div. Set the Button's *Text* property to *Send Feedback*.

The designer should show something similar to what you see in Figure 9-11, and at this point you have a page that will submit a form.

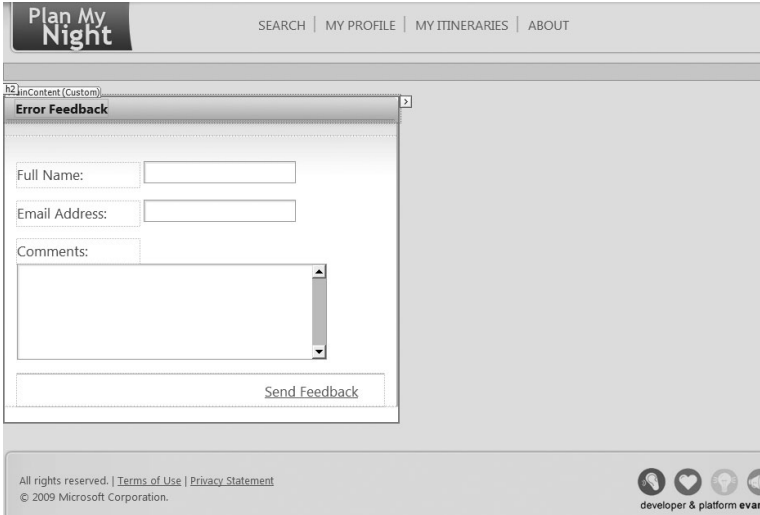
The image shows a web browser window displaying a form titled "Error Feedback". The form is contained within a window titled "InContent (Custom)". The form has three input fields: "Full Name:" with a text box, "Email Address:" with a text box, and "Comments:" with a larger text area. Below the text area is a "Send Feedback" button. The page header includes "Plan My Night" and navigation links for "SEARCH", "MY PROFILE", "MY ITINERARIES", and "ABOUT". The footer contains copyright information for 2009 Microsoft Corporation and social media icons for developer & platform evang.

FIGURE 9-11 The ErrorFeedback.aspx form with a complete field set

However, it does not perform any validation on the data being submitted. To do this, you'll take advantage of some of the validation controls present in ASP.NET. You'll make the Full Name and Comments boxes required fields and perform a regex validation of the e-mail address to ensure that it matches the right pattern.

Under the Validation group of the toolbox are some premade validation controls you'll use. Drag a *RequiredFieldValidator* object from the toolbox, and drop it to the right of the Full Name text box. Open the properties for the validation control, and set the *ControlToValidate* property to *FullName*. (It's a combo box of controls on the page.) Also, set the *CssClass* to

field-validation-error. This changes the display of the error to a red triangle used elsewhere in the application. Finally, change the Error Message property to **Name is Required**. (See Figure 9-12.)

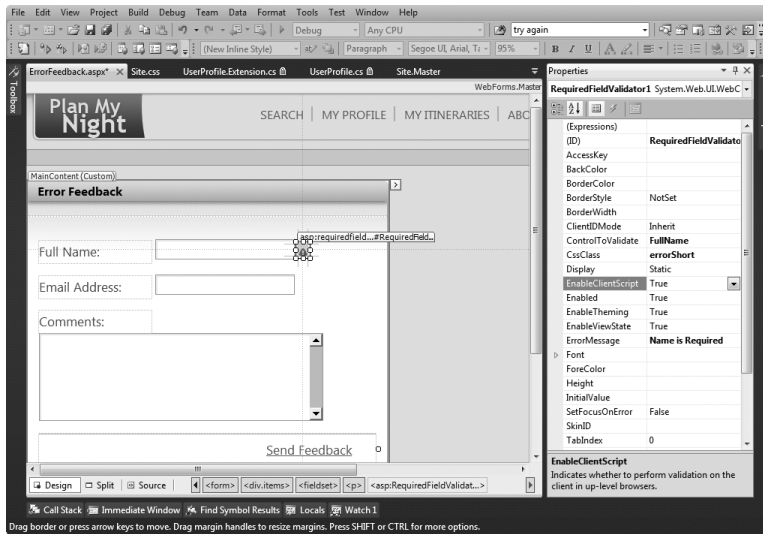


FIGURE 9-12 Validation control example

Repeat these steps for the Comments box, but substitute the *ErrorMessage* and *ControlToValidate* property values as appropriate.

For the Email Address field, you want to make sure the user types in a valid e-mail address, so for this field drag a *RegularExpressionValidator* control from the toolbox and drop it next to the Email Address text box. The property values are similar for this control in that you set the *ControlToValidate* property to *EmailAddress* and the *CssClass* property to *field-validation-error*. However, with this control you define the regular expression to be applied to the input data. This is done with the *ValidationExpression* property, and it should be set like this:

```
[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}
```

The error message for this validator should say something like "Must enter a valid e-mail address."

4. With the error screen visible, click the link to go to the feedback form. Try to submit the form with invalid data.

The screenshot shows a web form titled "Error Feedback". It contains three input fields: "Full Name:" with an empty text box and a red error message "Name is Required" pointing to it; "Email Address:" with a text box containing "someone@somewhere.com"; and "Comments:" with a text area containing "this is some feedback". At the bottom right is a "Send Feedback" button.

ASP.NET uses client-side script (when the browser supports it) to perform the validation, so no postbacks occur until the data passes. On the server side, when the server does receive a postback, a developer can check the validation state with the *Page.IsValid* property in the code-behind. However, because you used client-side validation (which is on by default), this will always be *true*. The only code in the code-behind that needs to be added is to redirect the user on a postback (and check the *Page.IsValid* property, in case client validation missed something):

```
protected void Page_Load(object sender, EventArgs e)
{
    if (this.IsPostBack && this.IsValid)
    {
        this.Response.Redirect("/", true);
    }
}
```

This really isn't very useful to the user, but our goal in this section was to work with the designer to create an ASP.NET Web Form. This added a new interface to the PlanMyNight.Web project, but what if you wanted to add new functionality to the application in a more modular sense, such as some degree of functionality that can be added or removed without having to compile the main application project. This is where an extensibility framework like the Managed Extensibility Framework (MEF) can show the benefits it brings.

Extending the Application with MEF

A new technology available in Visual Studio 2010 as part of the .NET Framework 4 is the Managed Extensibility Framework (MEF). The Managed Extensibility Framework provides developers with a simple (yet powerful) mechanism to allow their applications to be extended by third parties after the application has been shipped. Even within the same application, MEF allows developers to create applications that completely isolate components, allowing them to be managed or changed independently. It uses a resolution container to map components that provide a particular function (exporters) and components that require that functionality (importers), without the two concrete components having to know about each other directly. Resolutions are done on a contract basis only, which easily allows components to be interchanged or introduced to an application with very little overhead.

See Also MEF's community Web site, containing in-depth details about the architecture, can be found at <http://mef.codeplex.com>.

The companion Plan My Night application has been designed with extensibility in mind, and it has three "add-in" module projects in the solution, under the Addins solution folder. (See Figure 9-13.)

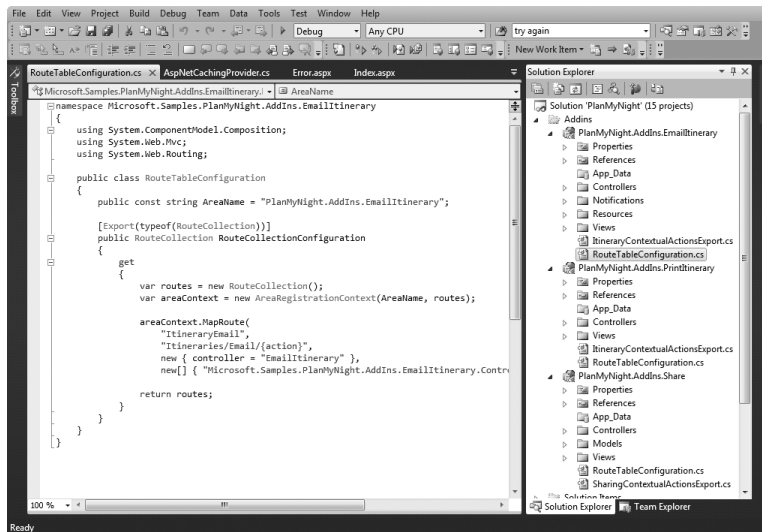


FIGURE 9-13 The Plan My Night application add-ins

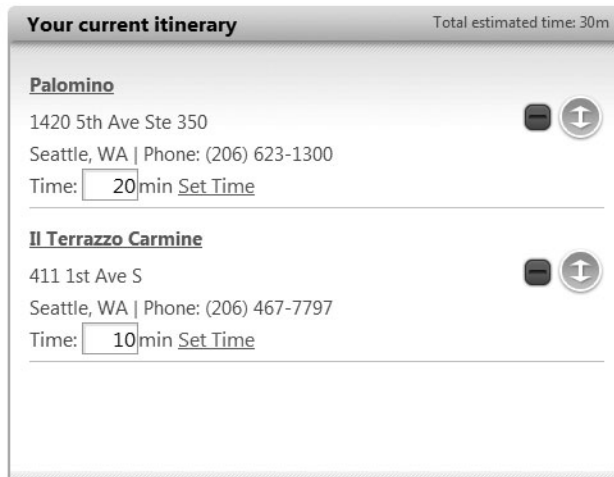
PlanMyNight.Addins.EmailItinerary adds the ability to e-mail itinerary lists to anyone the user sees fit to receive them. PlanMyNight.Addins.PrintItinerary provides a printer-friendly view

of the itinerary. Lastly, PlanMyNight.Addins.Share adds in social-media sharing functions (so that the user can post a link to an itinerary) as well as URL-shortening operations. None of these projects reference the main PlanMyNight.Web application or are referenced by it. They do have references to the PlanMyNight.Contracts and PlanMyNight.Infrastructure projects, so they can export (and import in some cases) the correct contracts via MEF as well as use any of the custom extensions in the infrastructure project.



Note Before doing the next step, if the Web application is not already running, launch the PlanMyNight.Web project so that the UI is visible to you.

To add the modules to your running application, run the DeployAllAddins.bat file, found in the same folder as the PlanMyNight.sln file. This will create new folders under the Areas section of the PlanMyNight.Web project. These new folders, one for each plug-in, will contain the files needed to add their functionality to the main Web application. The plug-ins appear in the application as extra options under the current itinerary section of the search results page and on the itinerary details page. After the batch file is finished running, go to the interface for Plan My Night, search for an activity, and add it to the current itinerary. You should notice some extra options under the itinerary panel other than just New and Save. (See Figure 9-14.)



[New](#) | [Save](#) | [Email](#) | [Print](#)

FIGURE 9-14 Location of the e-mail add-in in the UI

The social sharing options will show in the interface only after the itinerary is saved and marked public. (See Figure 9-15.)

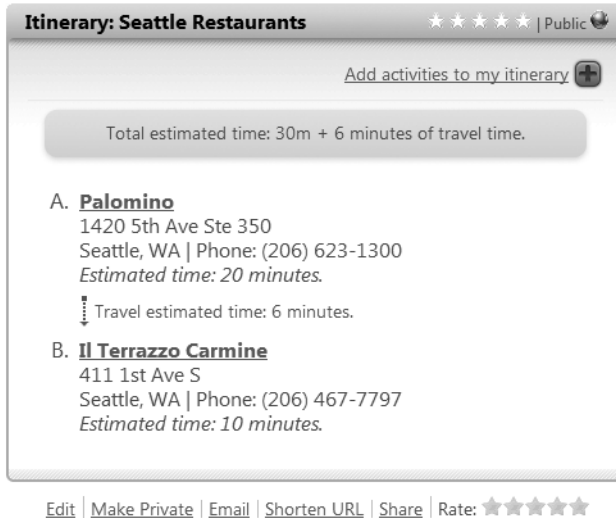


FIGURE 9-15 Location of the social sharing add-in in the UI

Visual Studio 2008 Visual Studio 2008 does not have anything that compares to MEF. To support "plug-ins," a developer would have to either write the plug-in framework from scratch or purchase a commercial package. Either of the two options led to proprietary solutions an external developer would have to understand in order to create a component for them. Adding MEF to the .NET Framework helps to cut down the entry barriers to producing extendable applications and the plug-in modules for them.

Print Itinerary Add-in Explained

To demonstrate how these plug-ins wire into the application, let's have a look at the PrintItinerary.Addin project. When you expand the project, you should see something like the structure shown in Figure 9-16.

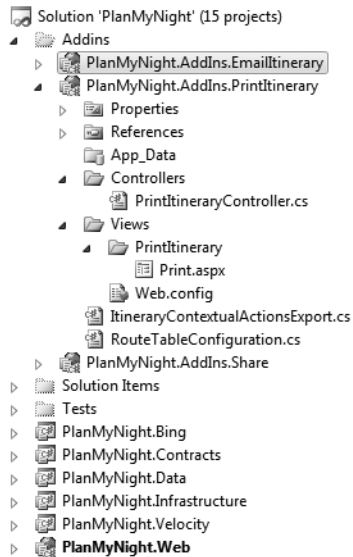


FIGURE 9-16 Structure of the PrintItinerary project

Some of this structure is similar to the PlanMyNight.Web project (Controllers and Views). That's because this add-in will be placed in an MVC application as an area. If you look more closely at the PrintItineraryController.cs file in the Controller folder, you can see it is similar in structure to the controller you created earlier in this chapter (and similar to any of the other controllers in the Web application). However, some key differences set it apart from the controllers that are compiled in the primary PlanMyNight.Web application.

Focusing on the class definition, you'll notice some extra attributes:

```
[Export("PrintItinerary", typeof(IController))]
[PartCreationPolicy(CreationPolicy.NonShared)]
```

These two attributes describe this type to the MEF resolution container. The first attribute, *Export*, marks this class as providing an *IController* under the contract name of *PrintItinerary*. The second attribute declares that this object supports only nonshared creation and cannot be created as a shared/singleton object. Defining these two attributes are all you need to do to have the type used by MEF. In fact, *PartCreationPolicy* is an optional attribute, but it should be defined if the type cannot handle all the creation policy types.

Further into the `PrintItineraryController.cs` file, the constructor is decorated with an *ImportingConstructor* attribute:

```
[ImportingConstructor]
public PrintItineraryController(IServiceFactory serviceFactory) :
this(
    serviceFactory.GetItineraryContainerInstance(),
    serviceFactory.GetItinerariesRepositoryInstance(),
    serviceFactory.GetActivitiesRepositoryInstance())
{
}
```

The *ImportingConstructor* attribute informs MEF to provide the parameters when creating this object. In this particular case, MEF provides an instance of *IServiceFactory* for this object to use. Where the instance comes from is of no concern to the *this* class and really assists with creating modular applications. For our purposes, the *IServiceFactory* contracted is being exported by the `ServiceFactory.cs` file in the `PlanMyNight.Web` project.

The `RouteTableConfiguration.cs` file registers the URL route information that should be directed to the `PrintItineraryController`. This route, and the routes of the other add-ins, are registered in the application during the *Application_Start* method in the `Global.asax.cs` file of `PlanMyNight.Web`:

```
// MEF Controller factory
var controllerFactory = new MefControllerFactory(container);
ControllerBuilder.Current.SetControllerFactory(controllerFactory);

// Register routes from Addins
foreach (RouteCollection routes in container.GetExportedValues<RouteCollection>())
{
    foreach (var route in routes)
    {
        RouteTable.Routes.Add(route);
    }
}
```

The *controllerFactory*, which was initialized with an MEF container containing path information to the `Areas` subfolder (so that it enumerated all the plug-ins), is assigned to be the controller factory for the lifetime of the application. This allows controllers imported via MEF to be usable anywhere in the application. The routes these plug-ins respond to are then retrieved from the MEF container and registered in the MVC routing table.

The `ItineraryContextualActionsExport.cs` file exports information to create the link to this plug-in, as well as metadata for displaying it. This information is used in the `ViewModelExtensions.cs` file, in the `PlanMyNight.Web` project, when building a view model for display to the user:

```
// get addin links and toolboxes
var addinBoxes = new List<RouteValueDictionary>();
var addinLinks = new List<ExtensionLink>();

addinBoxes.AddRange(AddinExtensions.GetActionsFor("ItineraryToolbox", model.Id == 0 ? null :
new { id = model.Id }));

addinLinks.AddRange(AddinExtensions.GetLinksFor("ItineraryLinks", model.Id == 0 ? null : new
{ id = model.Id }));
```

The call to `AddinExtensions.GetLinksFor` enumerates over exports in the MEF Export provider and returns a collection of them to be added to the local `addinLinks` collection. These are then used in the view to display more options when they are present.

Summary

In this chapter, we explored a few of the many new features and technologies found in Visual Studio 2010 that were used to create the companion Plan My Night application. We walked through creating a controller and its associated view and how the ASP.NET MVC framework offers Web developers a powerful option for creating Web applications. We also explored how using the Managed Extensibility Framework in application design can allow plug-in modules to be developed external to the application and loaded at run time. In the next chapter, we'll explore how debugging applications has been improved in Visual Studio 2010.

Chapter 10

From 2008 to 2010: Debugging an Application

After reading this chapter, you will be able to

- Use the new debugger features of Microsoft Visual Studio 2010
- Create unit tests and execute them in Visual Studio 2010
- Compare what was available to you or see what was different for you as a developer in Visual Studio 2008

As we were writing this book, we realized how much the debugging tools and developer aids have evolved over the last three versions of Visual Studio. Focusing on debugging an application and writing unit tests just increases the opportunities we have to work with Visual Studio 2010.

Visual Studio 2010 Debugging Features

In this chapter, you'll go through the different debugging features using a modified Plan My Night application. If you installed the companion content at the default location, you'll find the modified Plan My Night application at the following location: %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 10 \Code. Double-click the PlanMyNight.sln file.

First, before diving into the debugging session itself, you'll need to set up a few things:

1. In Solution Explorer, ensure that PlanMyNight.Web is the startup project. If the project name is not in bold, right-click on PlanMyNight.Web and select Set As StartUp Project.
2. To get ready for the next steps, in the PlanMyNight.Web solution open the Global.asax.cs file by clicking the triangle beside the Global.asax folder and then double-clicking the Global.asax.cs file, as shown in Figure 10-1.

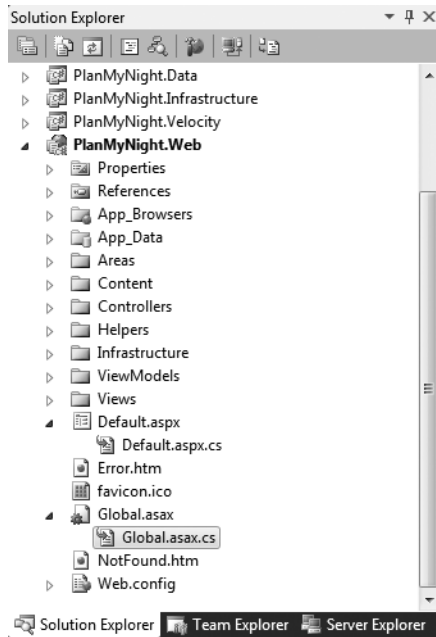


FIGURE 10-1 Solution Explorer before opening the file Global.asax.cs

Managing Your Debugging Session

Using the Plan My Night application, you'll examine how a developer can manage and share breakpoints. And with the use of new breakpoint enhancements, you'll learn how to inspect the different data elements in the application in a much faster and more efficient way. You'll also look at new minidumps and the addition of a new intermediate language (IL) interpreter that allows you to evaluate managed code properties and functions during minidump debugging.

New Breakpoint Enhancements

At this point, you have the Global.ascx.cs file opened in your editor. The following steps walk you through some ways to manage and share breakpoints:

1. Navigate to the *Application_BeginRequest(object sender, EventArgs e)* method, and set a breakpoint on the line that reads *var url = HttpContext.Current.Request.Url;* by clicking in the left margin or pressing F9. Look at Figure 10-2 to see this in action.

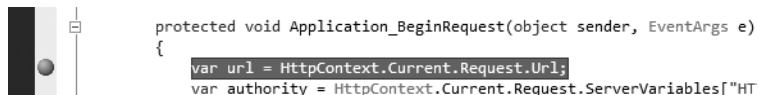


FIGURE 10-2 Creating a breakpoint

- Press F5 to start the application in debug mode. You should see the developer Web server starting in the system tray and a new browser window opening. The application should immediately stop at the breakpoint you just created. The Breakpoints window might not be visible even after starting the application in debug mode. If that is the case, you can make it visible by going to the Debug menu and selecting Windows and then Breakpoints, or you can use the following keyboard shortcut: Ctrl+D+B.

You should now see the Breakpoints window as shown in Figure 10-3.

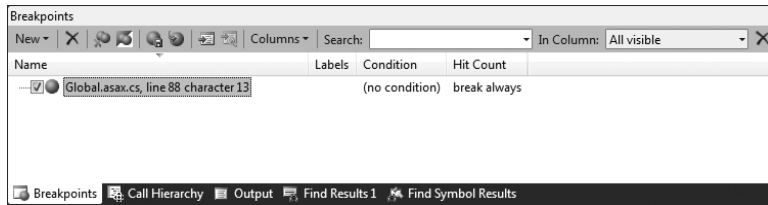


FIGURE 10-3 Breakpoints window

- In the same method, add three more breakpoints so that the editor and the Breakpoints window look like those shown in Figure 10-4.

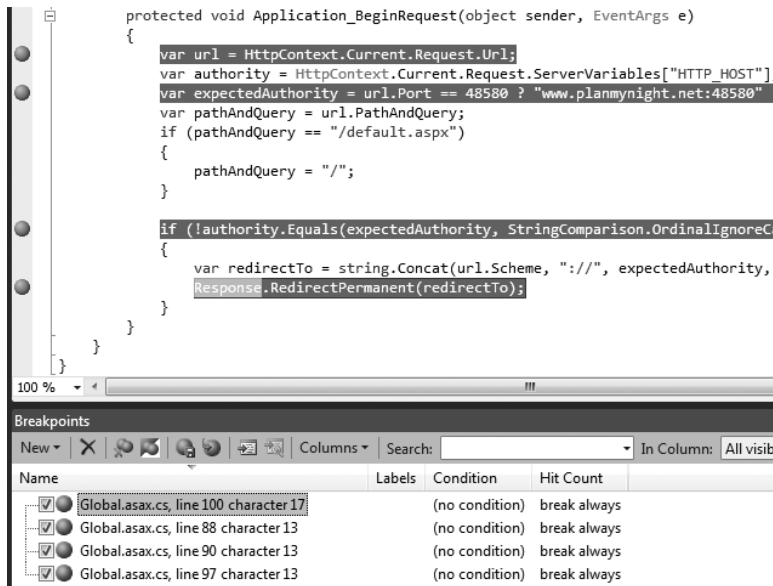


FIGURE 10-4 Code editor and Breakpoints window with three new breakpoints

Visual Studio 2008 As a reader and a professional developer who used Visual Studio 2008 often, you probably noticed a series of new buttons as well as new fields in the Breakpoints window in this exercise. As a reminder, take a look at Figure 10-5 for a quick comparison of what it looks like in Visual Studio 2008.

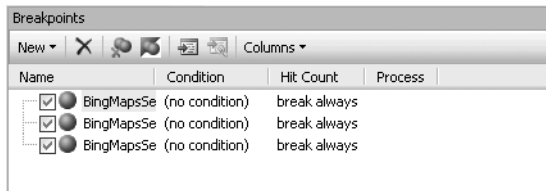


FIGURE 10-5 Visual Studio 2008 Breakpoints window

- Notice that the Labels column is now available to help you index and search breakpoints. It is a really nice and useful feature that Visual Studio 2010 brings to the table. To use this feature, you simply right-click on a breakpoint in the Breakpoints window and select Edit Labels or use the keyboard shortcut Alt+F9, L, as shown in Figure 10-6.

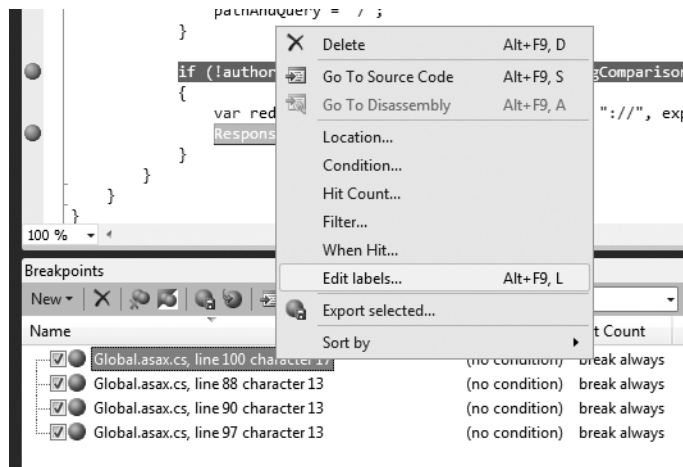


FIGURE 10-6 Edit Labels option

- In the Edit Breakpoint Labels window, add labels for the selected breakpoint (which is the first one in the Breakpoints window). Type **ContextRequestUrl** in the Type A New Label text box, and click Add. Repeat this operation on the next breakpoint, and

type a label name of **url**. When you are done, click OK. You should see a window that looks like Figure 10-7 while you are entering them, and to the right you should see the Breakpoints window after you are done with those two operations.

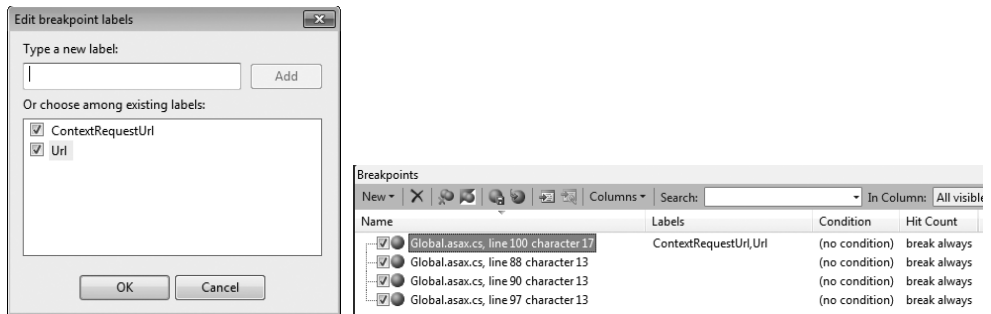


FIGURE 10-7 Adding labels that show up in the Breakpoints window



Note You can also right-click on the breakpoint in the left margin and select Edit Labels to accomplish the same tasks just outlined.



Note You'll see that when adding labels to a new breakpoint you can choose any of the existing labels you have already entered. You'll find these in the Or Choose Among Existing Labels area, which is shown in the Edit Breakpoint Labels dialog box on the left in the preceding figure.

- Using any of the ways you just learned, add labels for each of the breakpoints and make sure your Breakpoints window looks like Figure 10-8 after you're done.

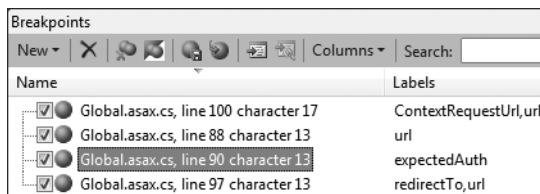


FIGURE 10-8 Breakpoints window with all labels entered





When you have a lot of code and are in the midst of a debugging session, it would be great to be able to filter the displayed list of breakpoints. That's exactly what the new Search feature in Visual Studio 2010 allows you to do.

7. To see the Search feature in action, just type **url** in the search text box and you'll see the list of breakpoint is filtered down to breakpoints containing *url* in one of their labels.

In a team environment where you have many developers and testers working together, often two people at some point in time are working on the same bugs. In Visual Studio 2008, the two people needed to sit near each other, send one another screen shots, or send one another the line numbers of where to put breakpoints to refine where they should look while debugging a particular bug.



Important One of the great new additions to breakpoint management in Visual Studio 2010 is that you can now export breakpoints to a file and then send them to a colleague, who can then import them into his own environment. Another scenario that this feature is useful for is to share breakpoints between machines. We'll see how to do that next.

8. In the Breakpoints window, click the Export button  to export your breakpoints to a file, and then save the file on your desktop. Name the file **breakexports.xml**.
9. Delete all the breakpoints either by clicking the Delete All Breakpoints Matching The Current Search Criteria button  or by selecting all the breakpoints and clicking the Delete The Selected Breakpoints button . The only purpose of deleting them is to simulate two developers sharing them or one developer sharing breakpoints between two machines.
10. You'll now import your breakpoints by clicking the Import button  and loading them from your desktop. Notice that all your breakpoints with all of their properties are back and loaded in your environment. For the purposes of this chapter, delete all the breakpoints.

Visual Studio 2008 Starting in Visual Studio 2008 and continuing in Visual Studio 2010, you are getting great support for JavaScript as well as for the latest iteration of jQuery. It was already good in Visual Studio, but the integration in Visual Studio 2010 is faster and you don't have to do anything to get it.

Inspecting the Data

When you are debugging your applications, you know how much time one can spend stepping into the code and inspecting the content of variables, arguments, and so forth. Maybe you can remember when you were learning to write code, a while ago, when debuggers weren't a reality or when they were really rudimentary. Do you remember (maybe not—you might not be as old as we are) how many *printf* or *WriteLn* statements you had to write to inspect the content of different data elements.

Visual Studio 2008 From the days of Visual Studio 2005 and continuing into the Visual Studio 2008 era, there already was a big improvement from the days of writing to the console with all kinds of statements. The improved conditions occurred because you had a real debugger with new functionalities. New data visualizers allowed you to see XML as a well-formed XML snippet and not as a long string. Furthermore, with those data visualizers, you could view arrays in a more useful way, with the list of elements and their indices, and you accomplished that by simply hovering your mouse over the object. Take a look at Figure 10-9 for an example.

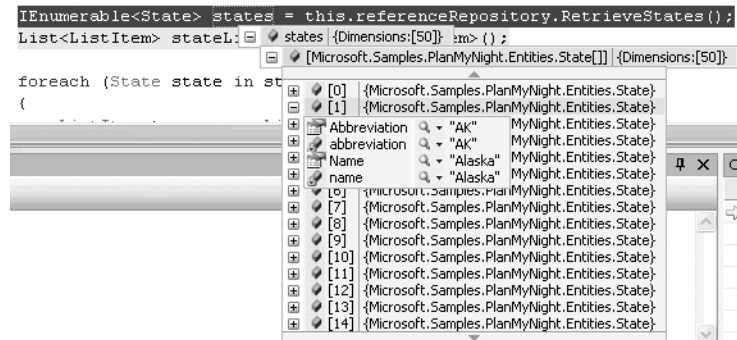
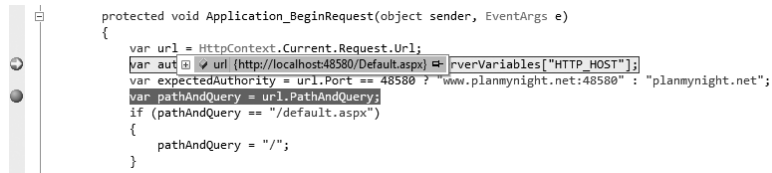


FIGURE 10-9 Collection view in the debugger in Visual Studio 2008

Visual Studio 2008 In Visual Studio 2008, there were some improvements in visualizing new types of data elements. The nicer and most noticeable improvement was the ability to view the results of a LINQ statement by using debugger elements such as DataTips, the Locals window, and the Watch or QuickWatch window. As you can with any other element—but it is so cool that you can do it as well for a LINQ query—you can copy a LINQ variable and paste it into a debugger window. Remember that to display the results of a query the debugger must evaluate it. Pay attention to things like side effects or clear differences in performance as you expand some subnodes.

Although those DataTip data visualization techniques are still available in Visual Studio 2010, a few great enhancements have been added that make DataTips even more useful. The DataTip enhancements have been added in conjunction with another new feature of Visual Studio 2010, multimonitor support. Floating DataTips can be valuable to you as a developer. Having the ability to put them on a second monitor can make your life a lot easier while debugging, because it keeps the data that always needs to be in context right there on the second monitor. The following steps demonstrate how to use these features:

1. In the Global.ascx.cs file, insert breakpoints on lines 89 and 91, lines starting with the source code `var authority` and `var pathAndQuery`, respectively.
2. You are now going to experiment with the new DataTip features. Start the debugger by pressing F5. When the debugger hits the first breakpoint, move your mouse over the word `url` and click on the pushpin as seen in Figure 10-10.



```
protected void Application_BeginRequest(object sender, EventArgs e)
{
    var url = HttpContext.Current.Request.Url;
    var authority = url (http://localhost:48580/Default.aspx) .ServerVariables["HTTP_HOST"];
    var expectedAuthority = url.Port == 48580 ? "www.planmynight.net:48580" : "planmynight.net";
    var pathAndQuery = url.PathAndQuery;
    if (pathAndQuery == "/default.aspx")
    {
        pathAndQuery = "/";
    }
}
```


FIGURE 10-10 The new DataTip pushpin feature

- To the right of the line of code, you should see the pinned DataTip (as seen in Figure 10-11 on the left). If you hover your mouse over the DataTip, you'll get the DataTip management bar (as seen in Figure 10-11 on the right):



FIGURE 10-11 On the left is the pinned DataTip, and on the right is the DataTip management bar



Note You should also see in the breakpoint gutter a blue pushpin indicating that the DataTip is pinned. The pushpin should look like this: . Because you have a breakpoint on that line, the pushpin is actually underneath it. To see the pushpin, just toggle the breakpoint by clicking on it in the gutter. Toggle once to disable the breakpoint and another time to get it back.



Note If you click the double arrow pointing down in the DataTip management bar, you can insert a comment for this DataTip, as shown in Figure 10-12. You can also remove the DataTip altogether by clicking the X button in the DataTip management bar.

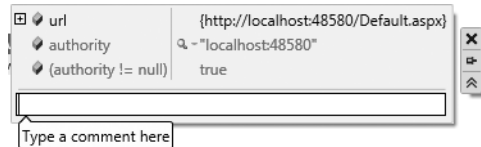


FIGURE 10-12 Inserting a comment for a DataTip

- One nice feature of the new DataTip is that you can insert any expression to be evaluated right there in your debugging session. For instance, right-click on the DataTip name, in this case on `url`, select Add Expression, type **authority**, and then add another one like this: **(authority != null)**. You'll see that the expressions are evaluated immediately and will continue to be evaluated for the rest of the debugging session every time your debugger stops on those breakpoints. At this point in the debugging session, the expression should evaluate to `null` and `false`, respectively.

- Press F10 to execute the line where the debugger stopped, and look at the url DataTip as well as both expressions. They should contain values based on the current context. Take a look at Figure 10-13 to see this in action.

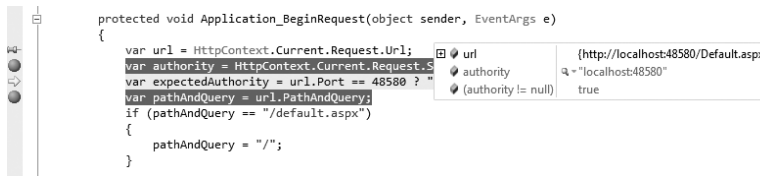


FIGURE 10-13 The url pinned DataTip with the two evaluated expressions

- Although it is nice to be able to have a mini-watch window where it matters—right there where the code is executing—you can also see that it is superimposed on the source code being debugged. Keep in mind that you can move the DataTip window anywhere you want in the code editor by simply dragging it. Take a look at Figure 10-14 for an example.

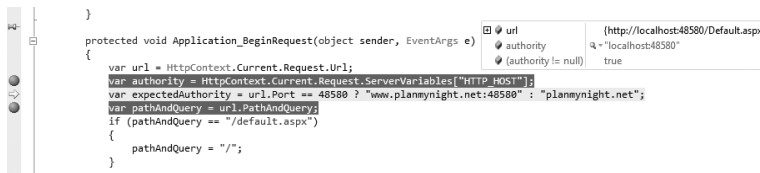



FIGURE 10-14 Moving the pinned DataTip away from the source code

- Because it is pinned, the DataTip window stays where you pinned it, so it will not be in view if you trace into another file. But in some cases, you need the DataTip window to be visible at all times. For instance, keeping it visible is interesting for global variables that are always in context or for multimonitor scenarios. To move a DataTip, you have to first unpin it by clicking the pushpin in the DataTip management bar. You'll see that it turns yellow. That indicates you can now move it wherever you want—for instance, over Solution Explorer, to a second monitor, over your desktop, or to any other window. Take a look at Figure 10-15 for an example.



FIGURE 10-15 Unpinned DataTip over Solution Explorer and the Windows desktop



Note If the DataTip is not pinned, the debugger stops in another file and method, and the DataTip contains items that are out of context, the DataTip windows will look like Figure 10-16. You can retry to have the debugger evaluate the value of an element by clicking this button: . However, if that element has no meaning in this context, it's possible that nothing happens.

```
public partial class _Default : Page
{
    public void Page_Load(object sender, System.EventArgs e)
    {
        string originalPath = Request.Path;
        HttpContext.Current.RewritePath(Request.ApplicationPath, false);
        IHttpHandler httpHandler = new MvcHttpHandler();
        httpHandler.ProcessRequest(HttpContext.Current);
        HttpContext.Current.RewritePath(originalPath, false);
    }
}
```

url	{http://localhost:48580/Default.aspx}
authority	"www.planmynight.net:48580"
(authority != null)	true
url.LocalPath	"/Default.aspx"

FIGURE 10-16 Results when the DataTip is not pinned and contains out-of-context items



Note You'll get an error message if you try to pin outside the editor, as seen in Figure 10-17.

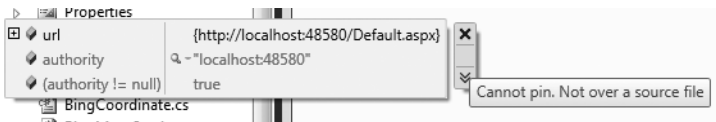


FIGURE 10-17 Error message that appears when trying to pin a DataTip outside the code editor



Note Your port number might be different than in the screen shots just shown. This is normal—it is a random port used by the personal Web server included with Visual Studio.



Note You can also pin any child of a pinned item. For instance, if you look at url and expand its content by pressing the plus sign (+), you'll see that you can also pin a child element, as seen in Figure 10-18.

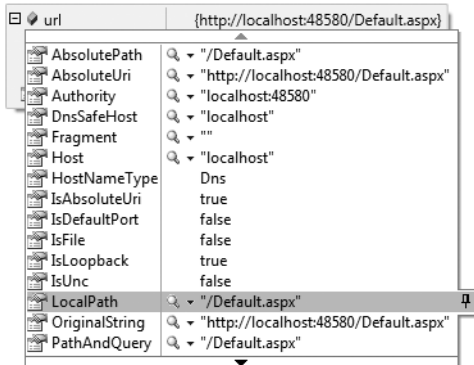



FIGURE 10-18 Pinned child element within the url DataTip

- Before stopping the debugger, go back to the Global.ascx.cs if you are not already there and re-pin the DataTip window. Then stop the debugging session by clicking the Stop Debugging button in the debug toolbar () or by pressing Shift+F5. Now if you hover your mouse over the blue pushpin in the breakpoint gutter, you'll see the values from the last debug session, which is a nice enhancement over the watch window. Take a look at Figure 10-19 for what you should see.

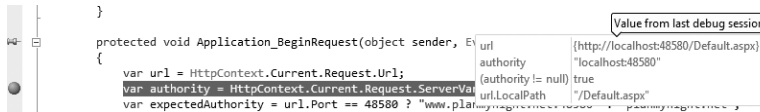


FIGURE 10-19 Values from the last debug session for a pinned DataTip



Note As with the breakpoints, you can export or import the DataTips by going to the Debug menu and selecting Export DataTips or Import DataTips, respectively.

Using the Minidump Debugger

Many times in real-world situations, you'll have access to a minidump from your product support team. Apart from their bug descriptions and repro steps, it might be the only thing you have to help debug a customer application. Visual Studio 2010 adds a few enhancements to the minidump debugging experience.

Visual Studio 2008 In Visual Studio 2008, you could debug managed application or minidump files, but you had to use an extension if your code was written in managed code. You had to use a tool called SOS and load it in the debugger using the Immediate window. You had to attach the debugger both in native and managed mode, and you couldn't expect to have information in the call stack or Locals window. You had to use commands for SOS in the Immediate window to help you go through minidump files. With application written in native code, you used normal debugging windows and tools. To read more about this or just to refresh your knowledge on the topic, you can read the *Bug Slayer* column in MSDN magazine here: <http://msdn.microsoft.com/en-us/magazine/cc164138.aspx>.

Let's see the new enhancements to the minidump debugger. First you need to create a crash from which you'll be able to generate a minidump file:

- In Solution Explorer in the PlanMyNight.Web project, rename the file Default.aspx to **DefaultA.aspx**. Note the *A* appended to the word "Default."
- Make sure you have no breakpoints left in your project. To do that, look in the Breakpoints window and delete any breakpoints left there using any of the ways you learned earlier in the chapter.

3. Press F5 to start debugging the application. Depending on your machine speed, soon after the build process is complete you should see an unhandled exception of type *HttpException*. Although the bug is simple in this case, let's go through the steps of creating the minidump file and debugging it. Take a look at Figure 10-20 to see what you should see at this point.

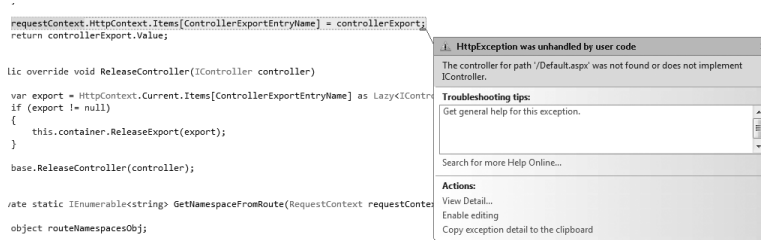


FIGURE 10-20 The unhandled exception you should expect

4. It is time to create the minidump file for this exception. Go to the Debug menu, and select Save Dump As, as seen in Figure 10-21. You should see the name of the process from which the exception was thrown. In this case, the process from which the exception was thrown was Cassini or the Personal Web Server in Visual Studio. Keep the file name proposed (WebDev.WebServer40.dmp), and save the file on your desktop. Note that it might take some time to create the file because the minidump file size will be close to 300 MB.

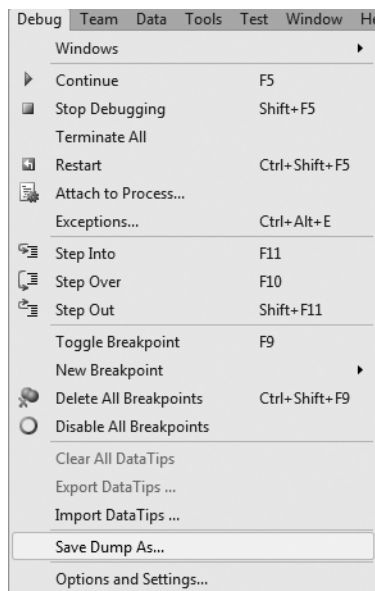


FIGURE 10-21 Saving the minidump file

5. Stop Debugging by pressing Shift+F5 or the Stop Debugging button.
6. Next, go to the File menu and close your solution.
7. In the File menu, click Open and point to the desktop to load your minidump file named WebDev.WebServer40.dmp. Doing so opens the Minidump File Summary page, which gives you some summary information about the bug you are trying to fix. (Figure 10-22 shows what you should see.) Before you start to debug, you'll get basic information from that page such as the following: process name, process architecture, operating system version, CLR version, modules loaded, as well as some actions you can take from that point. From this place, you can set the paths to the symbol files. Conveniently, the Modules list contains the version and path on disk of your module, so finding the symbols and source code is easy. The CLR version is 4.0; therefore, you can debug here in Visual Studio 2010.

The screenshot shows the 'Minidump File Summary' window for a file named 'WebDev.WebServer40.dmp'. The window is divided into several sections:

- Dump Summary:** Shows the dump file path, last write time (2/22/2010 1:58:45 AM), process name (WebDev.WebServer40.exe), process architecture (x86), exception code (0x0434F4D), and exception information (An exception came from the CLR).
- System Information:** Shows OS Version (6.1.7600) and CLR Version(s) (4.0.30128.1).
- Modules:** A table listing loaded modules with columns for Module Name, Module Version, and Module Path.
- Notifications:** Indicates 'IL Interpreter is Enabled' and 'Disable IL Interpreter'.
- Actions:** A list of actions including 'Debug with Mixed', 'Debug with Native Only', 'Set symbol paths', and 'Copy all to clipboard'.

Module Name	Module Version	Module Path
WebDev.WebServer40.exe	10.0.30128.1	C:\Program Files\Common Files\Mi
ntdll.dll	6.1.7600.16385	C:\Windows\System32\ntdll.dll
mscorlib.dll	4.0.31106.0	C:\Windows\System32\mscorlib.dll
kernel32.dll	6.1.7600.16481	C:\Windows\System32\kernel32.dll
KERNELBASE.dll	6.1.7600.16385	C:\Windows\System32\KERNELBAS
advapi32.dll	6.1.7600.16385	C:\Windows\System32\advapi32.dll
msvcrt.dll	7.0.7600.16385	C:\Windows\System32\msvcrt.dll
sechost.dll	6.1.7600.16385	C:\Windows\System32\sechost.dll
rpcrt4.dll	6.1.7600.16385	C:\Windows\System32\rpcrt4.dll
mscorlib.dll	4.0.30128.1	C:\Windows\Microsoft.NET\Frames
shlwapi.dll	6.1.7600.16385	C:\Windows\System32\shlwapi.dll
gdi32.dll	6.1.7600.16385	C:\Windows\System32\gdi32.dll
user32.dll	6.1.7600.16385	C:\Windows\System32\user32.dll
lpk.dll	6.1.7600.16385	C:\Windows\System32\lpk.dll
usp10.dll	1.626.7600.16385	C:\Windows\System32\usp10.dll
imm32.dll	6.1.7600.16385	C:\Windows\System32\imm32.dll

FIGURE 10-22 Minidump summary page

8. To start debugging, locate the Actions list on the right side of the Minidump File Summary page and click Debug With Mixed.

9. You should see almost immediately a first-chance exception like the one shown in Figure 10-23. In this case, it tells you what the bug is; however, this won't always be the case. Continue by clicking the Break button.

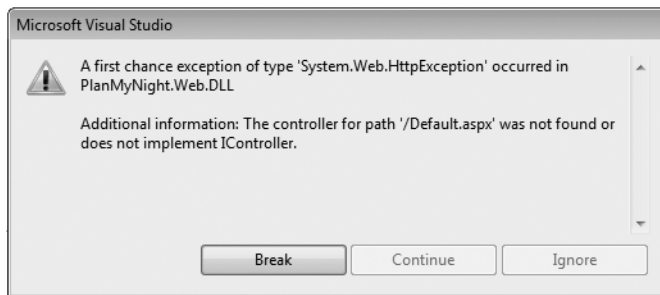


FIGURE 10-23 First-chance exception

10. You should see a green line indicating which instruction caused the exception. If you look at the source code, you'll see in your Autos window that the *controllerExport* variable is *null*, and just before that we specified that if the variable was *null* we wanted to have an *HttpException* thrown if the file to load was not found. In this case, the file to look for is *Default.aspx*, as you can see in the Locals window in the *controllerName* variable. You can glance at many other variables, objects, and so forth in the Locals and Autos windows containing the current context. Here, you have only one call that belongs to your code, so the call stack indicates that the code before and after is external to your process. If you had a deeper chain of calls in your code, you could step back and forth in the code and look at the variables. Figure 10-24 shows a summary view of all that.

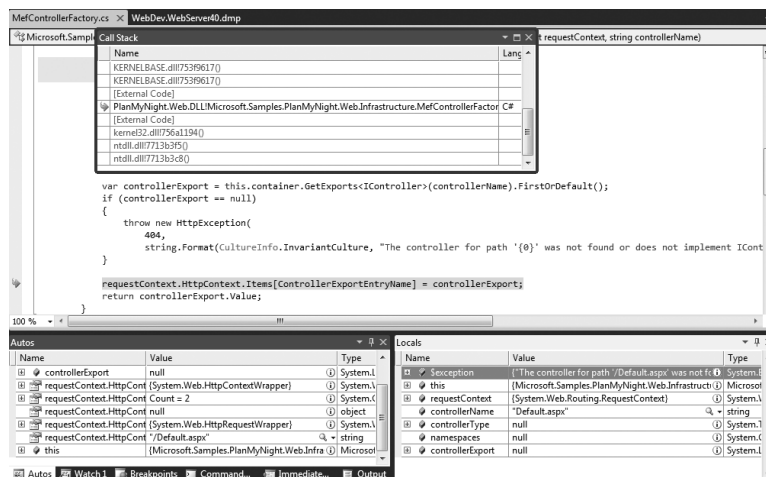


FIGURE 10-24 Autos, Locals, and Call Stack windows, and the next instruction to execute

11. OK, you found the bug, so stop the debugging by pressing Shift+F5 or clicking the Stop Debugging button. Then fix the bug by reloading the PlanMyNight solution and renaming the file back to **default.aspx**. Then rebuild the solution by going to the Build menu and selecting Rebuild Solution. Then press F5, and the application should be working again.

Web.Config Transformations

This next new feature, while small, is one that will delight many developers because it saves them time while debugging. The feature is the Web.Config transformations that allow you to have transform files that show the differences between the debug and release environments. As an example, connection strings are often different from one environment to the other; therefore, by creating transform files with the different connection strings—because ASP.NET provides tools to change (transform) web.config files—you'll always end up with the right connection strings for the right environment. To learn more about how to do this, take a look at the following article on MSDN: <http://go.microsoft.com/fwlink/?LinkId=125889>.

Creating Unit Tests

Most of the unit test framework and tools are unchanged in Visual Studio 2010 Professional. It is in other versions of Visual Studio 2010 that the change in test management and test tools is really apparent. Features such as UI Unit Tests, IntelliTrace, and Microsoft Test Manager 2010 are available in other product versions like Visual Studio 2010 Premium and Visual Studio 2010 Ultimate. To see which features are covered in the Application Lifecycle Management and for more specifics, refer to the following article on MSDN: [http://msdn.microsoft.com/en-us/library/ee789810\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/ee789810(VS.100).aspx).

Visual Studio 2008 With Visual Studio 2008 you had to own either Visual Studio 2008 Team System or Visual Studio 2008 Team Test to have the ability to create and execute tests out of the box within Visual Studio 2008. Another option back then was to go with a third-party option like NUnit.

In this part of the chapter, we'll simply show you how to add a unit test for a class you'll find in the Plan My Night application. We won't spend time defining what a unit test is or what it should contain; rather, we'll show you within Visual Studio 2010 how to add tests and execute them.

You'll add unit tests to the Plan My Night application for the Print Itinerary Add-in. To create unit tests, open the solution from the companion content folder. If you do not remember how to do this, you can look at the first page of this chapter for instructions. After you have the solution open, just follow these steps:

1. In Solution Explorer, expand the project PlanMyNight.Web and then expand the Helpers folder. Then double-click the file ViewHelper.cs to open it in the code editor. Take a look at Figure 10-25 to make sure you are at the right place.

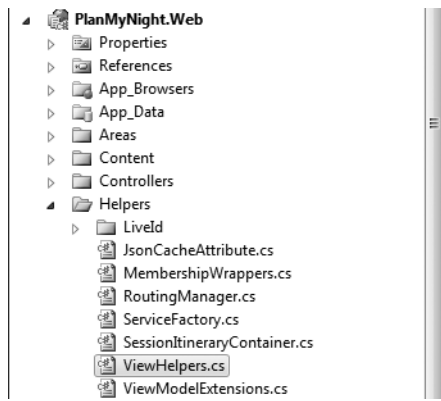


FIGURE 10-25 The PlanMyNight.Web project and ViewHelper.cs file in Solution Explorer

2. In the code editor, you can add unit tests in two different ways. You can right-click on a class name or on a method name and select Create Unit Tests. You can also go to the Test menu and select New Test. We'll explore the first way of creating unit tests. This way Visual Studio automatically generates some source code for you. Right-click the *GetFriendlyTime* method, and select Create Unit Tests. Figure 10-26 shows what it looks like.

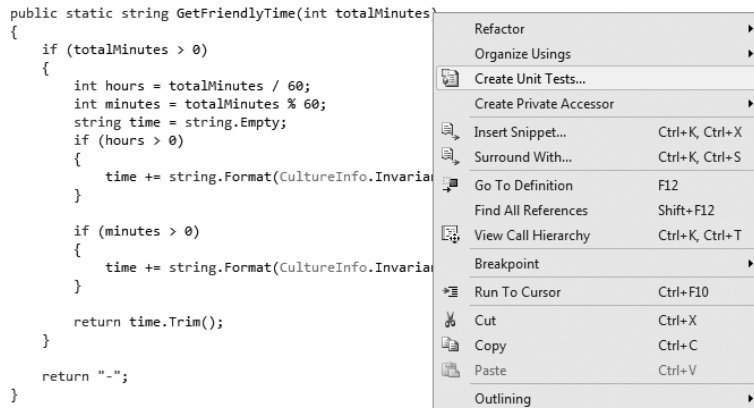


FIGURE 10-26 Contextual menu to create unit tests from by right-clicking on a class name

3. After selecting Create Unit Tests, you'll be presented with a dialog that, by default, shows the method you selected from that class. To select where you want to create the unit tests, click on the drop-down combo box at the bottom of this dialog and select PlanMyNight.Web.Tests. If you didn't have an existing location, you would have simply selected Create A New Visual C# Test Project from the list. Figure 10-27 shows what you should be seeing.

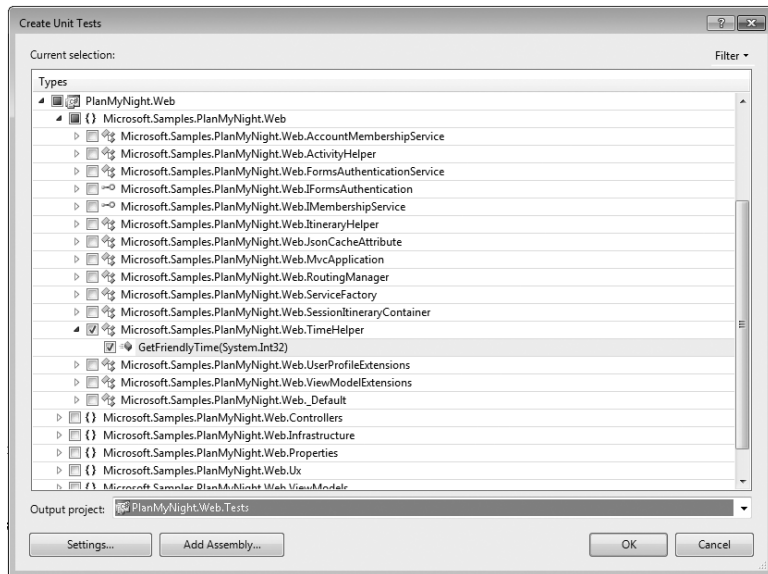


FIGURE 10-27 Selecting the method you want to create a unit test against

4. After you click OK, the dialog switches to a test-case generation mode and displays a progress bar. After this is complete, a new file is created named *TimeHelperTest.cs* that has autogenerated code stubs for you to modify.
5. Remove the method and its attributes because you'll create three new test cases for that method. Remove the following code:

```

/// <summary>
///A test for GetFriendlyTime
///</summary>
// TODO: Ensure that the UrlToTest attribute specifies a URL to an ASP.NET page (for
// example, http://.../Default.aspx). This is necessary for the unit test to be executed
// on the web server, whether you are testing a page, web service, or a WCF service.
[TestMethod()]
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("C:\\Users\\Patrice\\Documents\\Chapter 10\\code\\
PlanMyNight.Web", "/")]
[UrlToTest("http://localhost:48580/")]
public void GetFriendlyTimeTest()
{
    int totalMinutes = 0; // TODO: Initialize to an appropriate value
    string expected = string.Empty; // TODO: Initialize to an appropriate value
    string actual;
    actual = TimeHelper.GetFriendlyTime(totalMinutes);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}

```

6. Add the three simple test cases validating three key scenarios used by Plan My Night. To do that, insert the following source code right below the method attributes that were left behind when you deleted the block of code in step 5:

```
[TestMethod]
public void ZeroReturnsSlash()
{
    Assert.AreEqual("-", TimeHelper.GetFriendlyTime(0));
}
```

```
[TestMethod]
public void LessThan60MinutesReturnsValueInMinutes()
{
    Assert.AreEqual("10m", TimeHelper.GetFriendlyTime(10));
}
```

```
[TestMethod()]
public void MoreThan60MinutesReturnsValueInHoursAndMinutes()
{
    Assert.AreEqual("2h 3m", TimeHelper.GetFriendlyTime(123));
}
```

7. In the PlanMyNight.Web.Tests project, create a solution folder called **Helpers**. Then move your TimeHelperTests.cs file to that folder so that your project looks like Figure 10-28 when you are done.

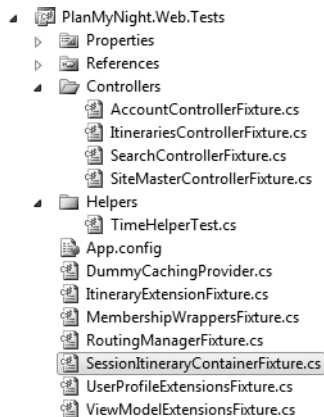


FIGURE 10-28 TimeHelperTest.cs in its Helpers folder

8. It is time to execute your newly created tests. To execute only your newly created tests, go into the code editor and place your cursor on the class named *public class TimeHelperTest*. Then you can either go to the Test menu, select Run, and finally select

Test In Current Context or accomplish the same thing using the keyboard shortcut CTRL+R, T. Look at Figure 10-29 for a reference.

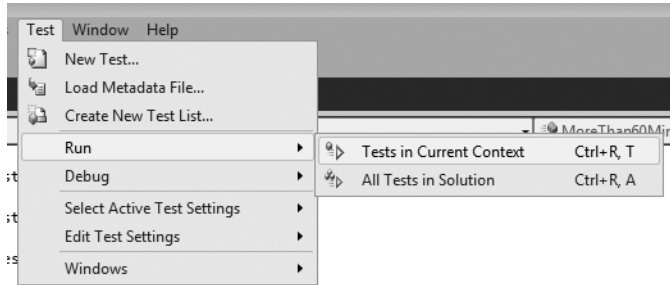


FIGURE 10-29 Test execution menu

9. Performing this action executes only your three tests. You should see the Test Results window (shown in Figure 10-30) appear at the bottom of your editor with the test results.

Result	Test Name	Project	Error Message
Passed	LessThan60Minutes>ReturnsValueInMinutes	PlanMyNight.Web.Tests	
Passed	MoreThan60Minutes>ReturnsValueInHoursAndMinutes	PlanMyNight.Web.Tests	
Passed	ZeroReturnsSlash	PlanMyNight.Web.Tests	

FIGURE 10-30 Test Results window for your newly created tests



More Info Depending on what you select, you might have a different behavior when you choose the Tests In Current Context option. For instance, if you select a test method like *ZeroReturnsSlash*, you'll execute only this test case. However, if you click outside the test class, you could end up executing every test case, which is the equivalent of choosing All Tests In Solution.

New Threads Window

The emergence of computers with multiple cores and the fact that language features give developers many tools to take advantage of those cores creates a new problem: the difficulty of debugging concurrency in applications. The new Threads window enables you, the developer, to pause threads and search the calling stack to see artifacts similar to those you see when using the famous SysInternals Process Monitor (<http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>). You can display the Threads window by going to Debug and

Finally, throughout this chapter you also saw how Visual Studio 2010 Professional has raised the bar in terms of debugging applications and has given professional developers the tools to debug today's feature-rich experiences. You saw that it is a clear improvement over what was available in Visual Studio 2008. The exercises in the chapter scratched the surface of how you'll save time and money by moving to this new debugging environment and showed that Visual Studio 2010 is more than a small iteration in the evolution of Visual Studio. It represents a huge leap in productivity for developers. The gap between Visual Studio 2008 and Visual Studio 2010 in terms of debugging is less severe than in earlier versions.

The various versions of Visual Studio 2010 give you a great list of improvements related to the debugger and testing. My personal favorites are IntelliTrace—[http://msdn.microsoft.com/en-us/library/dd264915\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(VS.100).aspx)—which is available only in Visual Studio 2010 Ultimate and Microsoft Test Manager. IntelliTrace enables test teams to have much better experiences using Visual Studio 2010 and Visual Studio 2010 Team Foundation Server—[http://msdn.microsoft.com/en-us/library/bb385901\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb385901(VS.100).aspx).

Index

A

AcceptVerbs attribute, 51, 165, 263

account controller, 2003 to 2010

- account view, creating, 59–66
- ASP.NET MVC, 43–44
- creating, 43–58
- current user profile retrieval, 51–53
- functionality implementation, 44–46
- profile data, updating, 53–58
- user authentication, 46–51

account controller, 2005 to 2010

- account view, creating, 173–180
- ASP.NET MVC, 157–158
- creating, 157–173
- current user profile retrieval, 165–167
- functionality implementation, 158–160
- profile data, updating, 167–173
- user authentication, 160–165

account controller, 2008 to 2010

- account view, creating, 270–278
- ASP.NET MVC, 255–256
- creating, 255–270
- current user profile retrieval, 262–265
- functionality implementation, 256–287
- profile data, updating, 265–270
- user authentication, 258–262

account view, creating, 59–66, 173–180, 270–278

Add method, 27, 32, 143, 146, 244, 246

Add Web Service Reference, 32–33, 146

add-in module projects, 75–77, 188–190, 286–288

ADO.NET Entity Framework (EF).
See Entity Framework

ADO.NET POCO entity generator, 23–25, 138–140, 222, 239–241

AJAX, 54, 168–169, 266

AntiForgeryToken, 54, 168, 265

AppFabric caching, 36–37, 150–151, 248–249

architecture, 3–5, 117–119, 217–219

areas folder, 40, 154, 252

ASP.NET Forms Authentication, 46–51, 160–165, 258–262

ASP.NET Membership Service, 46–51, 160–165, 258–262

ASP.NET MVC, 2003 to 2010

- account controller, 43–44
- caching, 37
- component folders, 40–41
- Model Binding, 52, 54
- overview, 39
- requests, 54
- Web Forms versus, 44

ASP.NET MVC, 2005 to 2010

- account controller, 157–158
- caching, 151
- component folders, 154–155
- Model Binding, 166, 168
- overview, 153
- requests, 168
- Web Forms versus, 158

ASP.NET MVC, 2008 to 2010

- account controller, 255–256
- caching, 249
- component folders, 252–253
- Model Binding, 263, 265–266
- overview, 251
- requests, 266
- Web Forms versus, 256

ASP.NET Web Forms. *See* Web Forms

attributes

- 2003 to 2010, 51, 54, 66
- 2005 to 2010, 165, 168, 179
- 2008 to 2010, 263, 265, 277

authenticating user, 46–51, 160–165, 258–262

Authorize attribute, 51, 165, 263

B

Bing Maps Web services data, 32–34, 146–148

breakpoint enhancements, 82–87, 196–200, 294–298

business logic and data, from 2003 to 2010

- AppFabric caching, 36–37
- architecture, 3–5

- Bing Maps Web services data, 32–34
- database data retrieval, 27–32
- Entity Framework data, 6–26
- parallel programming, 35–36
- Plan My Night data, 5–6

business logic and data, from 2005 to 2010

- AppFabric caching, 150–151
- architecture, 117–119
- Bing Maps Web services data, 146–148
- database data retrieval, 142–146
- Entity Framework data, 121–142
- parallel programming, 149–150
- Plan My Night data, 119–121

business logic and data, from 2008 to 2010

- AppFabric caching, 248–249
- architecture, 217–219
- database data retrieval, 243–247
- Entity Framework data, 222–243
- parallel programming, 247–248
- Plan My Night data, 219–221

C

caching, 36–37, 150–151, 248–249

code-behind files, 60, 62, 174, 176, 272, 273

code-only support, 222

component folders, 40–41, 154–155, 252–253

content folders, 41, 155, 253

contract interface components, 4, 118, 218

contracts project, entity classes, moving to, 25–26, 140–142, 241–243

Controller.RedirectToAction, 48, 162, 260

controllers, 40, 154, 252

Copy Project deployment method, 105–106

CreateProfile, 48, 162, 260

CreateUser method, 48, 162, 260

CSS application, 69, 183, 280

current user profile retrieval, 51–53, 165–167, 262–265

D

data inspection, 87–91, 200–205, 298–303

data visualizers, 87–88, 201, 299

database, connecting to existing, 8, 123, 224

database data retrieval, 27–32, 142–146, 243–247

database scripts, 20–22, 135–138, 236–239

datacenters/servers, deploying applications to, 104

DataTip enhancements, 88–91, 201–205, 299–303

debugging an application, from 2003 to 2010

- breakpoint enhancements, 82–87
- data inspection, 87–91
- minidump debugger, 92–95
- overview, 81–82
- session management, 82–100
- Threads window, 100
- unit testing, 95–100
- web.config transformations, 95

debugging an application, from 2005 to 2010

- breakpoint enhancements, 196–200
- data inspection, 200–205
- minidump debugger, 205–208
- overview, 195–196
- session management, 196–213
- Threads window, 213
- unit testing, 208–213
- web.config transformations, 208

debugging an application, from 2008 to 2010

- breakpoint enhancements, 294–298
- data inspection, 298–303
- minidump debugger, 303–307
- overview, 293–294
- session management, 294–311
- Threads window, 311–312
- unit testing, 307–311
- web.config transformations, 307

deploying an application, from 2003 to 2010

- to enterprise datacenters/servers, 104
- to hosting company, 103–104
- one-click publish, 112–113
- web deployment packages, 104–111

Designer view, 66–74, 180–187, 278–285

designing application look and feel, from 2003 to 2010

- account controller, 43–58
- account view, creating, 59–66
- designer view, 66–74
- extending application with MEF, 74–79
- overview, 39
- PlanMyNight.Web project, 39–42

designing application look and feel, from 2005 to 2010

- account controller, 157–173
- account view, creating, 173–180
- designer view, 180–187
- extending application with MEF, 188–193
- overview, 153
- PlanMyNight.Web project, 153–156

designing application look and feel, from 2008 to 2010

- account controller, 255–270
- account view, creating, 270–278
- designer view, 278–285
- extending application with MEF, 286–291
- overview, 251
- PlanMyNight.Web project, 251–254

E

enterprise datacenters/servers, deploying applications to, 104

entity classes, moving to contracts project, 25–26, 140–142, 241–243

Entity Data Model (EDM) Designer, 2003 to 2005

- ADO.NET POCO entity generator, 23–25
- existing database import model, 7–16
- POCO templates, 22–23
- wizard modifications, 8–9, 16–20

Entity Data Model (EDM) Designer, 2005 to 2010

- ADO.NET POCO entity generator, 138–140
- existing database import model, 122–131
- manual modifications, 126–129
- POCO templates, 138
- wizard modifications, 123–124, 131–142

Entity Data Model (EDM) Designer, 2008 to 2010

- ADO.NET POCO entity generator, 222, 239–241
- existing database import model, 222–232

manual modifications, 227–230

POCO templates, 222, 239

wizard modifications, 224–225, 232–243

Entity Framework (EF), 2003 to 2005

- data with, 6–26
- existing database, importing, 7–16
- Model First approach, importing, 16–26
- overview, 6
- PlanMyNight existing solutions, 7

Entity Framework (EF), 2005 to 2010

- data with, 121–142
- existing database, importing, 122–130
- Model First approach, importing, 131–142
- overview, 121
- PlanMyNight existing solutions, 122

Entity Framework (EF), 2008 to 2010

- data with, 222–243
- existing database, importing, 222–232
- Model First approach, importing, 222, 232–243
- overview, 222
- PlanMyNight existing solutions, 223
- Visual Studio 2008 and, 222

EntitySet, 127, 225, 228

EntityType, 127, 225, 228

exception handling, 66–74, 180–187, 278–285. *See also headings starting with “debugging”*

exporting breakpoints, 86, 200, 298

exporting DataTips, 91, 205, 303

extending application with MEF, 74–79, 188–193, 286–291

extension methods, 52, 166

F

filtering breakpoints, 86, 200, 297

foreign key associations, 235

function imports, 15–16, 130–131, 231–232

G

Generate Database Wizard, 21–22, 136–138, 237–239

generics, Visual Studio 2003 and, 29

GetCurrentProfile, 52, 166, 263

GetReturnUrl, 52, 166, 263

H

helpers, 41, 155, 253
 hosting companies, deploying applications to, 103–104
 HTML methods, 62–63, 176, 273–274

I

IActivitiesRepository interface, 4, 45, 118, 159, 218, 257
 ICachingProvider interface, 4, 118, 218
 ID attribute, 66, 179, 277
 IFormsAuthentication, 45, 159, 257
 ItinerariesRepository interface, 4, 27, 118, 142, 218, 243
 IMembershipService, 45, 159, 257
 importing breakpoints, 87, 200, 298
 importing data from existing database, 2003 to 2010
 fixing generated data model, 9–14
 function imports, 15–16
 initial process, 7–9
 stored procedure, 15–16
 importing data from existing database, 2005 to 2010
 fixing generated data model, 124–131
 function imports, 130–131
 initial process, 122–124
 stored procedure, 130–131
 importing data from existing database, 2008 to 2010
 fixing generated data model, 226–230
 function imports, 231–232
 initial process, 222–226
 stored procedure, 231–232
 importing DataTips, 91, 205, 303
 independent key associations, 235
 Index method, 51, 165, 262–263
 Index view, 59–63, 173–176, 270–274
 infrastructure components, 41, 155, 253
 InjectStatesAndActivityTypes method, 52–53, 167, 264–265
 InstallShield, 109
 instance fields, 45, 159, 256–257
 interfaces
 IActivitiesRepository, 4, 45, 118, 159, 218, 257
 ICachingProvider, 4, 118, 218

IFormsAuthentication, 45, 159, 257
 ItinerariesRepository, 4, 27, 118, 142, 218, 243
 IMembershipService, 45, 159, 257
 IReferenceRepository, 45, 159, 257
 IWindowsLiveLogin, 45, 159, 257
 IReferenceRepository, 45, 159, 257
 ItineraryActivities navigation property, 9, 124–125, 225–226
 IWindowsLiveLogin, 45, 159, 257

J

jQuery, 87, 298
 JsonResult, 54, 168–169, 266

L

labels, debug breakpoint feature, 84–86, 198–199, 296–297
 Language Integrated Query (LINQ), 28, 143, 299
 lazy loading, 222
 LiveID method, 49–50, 163–164, 261–262
 Login method, 50, 164, 262

M

Managed Extensibility Framework (MEF), 74, 188, 286
 minidump debugger, 92–95, 205–208, 303–307
 Model Binding, 52, 54, 166, 168, 263, 265–266
 Model First approach, 2003 to 2005
 ADO.NET POCO entity generator, 23–25
 generating database script, 20–22
 initial process, 16–20
 moving entity classes to contracts project, 25–26
 POCO templates, 22–23
 Model First approach, 2005 to 2010
 ADO.NET POCO entity generator, 138–140
 generating database script, 135–138
 initial process, 131–135

moving entity classes to contracts project, 140–142
 POCO templates, 138
 Model First approach, 2008 to 2010
 ADO.NET POCO entity generator, 222, 239–241
 generating database script, 236–239
 initial process, 232–236
 moving entity classes to contracts project, 241–243
 POCO templates, 222, 239
 Visual Studio 2008 and, 222
 MSBuild, 104, 107, 108, 113
 msdeploy tool, 109, 113
 multicore computers, 100, 213, 311
 multimonitor support for debugging, 88, 201, 299
 multithreaded applications, 100, 213, 311

N

namespaces, 44, 158, 256
 naming, singular vs. plural, 225
 navigation properties, 9, 20, 124–125, 135, 226, 236
 NavigationProperty, 225
 .NET Framework 4.0, 2003 to 2010
 AppFabric caching, 37
 Managed Extensibility Framework (MEF), 74
 parallel programming, 35–36
 PLINQ libraries, 36, 100
 .NET Framework 4.0, 2005 to 2010
 AppFabric caching, 151
 Managed Extensibility Framework (MEF), 188
 parallel programming, 149–150
 PLINQ libraries, 150
 .NET Framework 4.0, 2008 to 2010
 AppFabric caching, 249
 Managed Extensibility Framework (MEF), 286
 parallel programming, 247–248
 PLINQ libraries, 248

O

one-click publish, 112–113

P

parallel programming, 35–36, 149–150, 247–248
 Plain-Old CLR Objects. *See* POCO templates

Plan My Night application, 2003 to 2010

- add-in module projects, 75–77
- AppFabric caching, 36–37
- architecture, 3–5
- ASP.NET caching, 37
- Bing Maps Web services data, 32–34
- data, 5–6
- database data retrieval, 27–32
- Entity Framework data, 6–26
- existing projects, 7
- parallel programming, 35–36
- Visual Studio 2003 and, 5–6

Plan My Night application, 2005 to 2010

- add-in module projects, 188–190
- AppFabric caching, 150–151
- architecture, 117–119
- ASP.NET caching, 151
- Bing Maps Web services data, 146–148
- data, 119–121
- database data retrieval, 142–146
- Entity Framework data, 121–142
- existing projects, 122
- parallel programming, 149–150
- Visual Studio 2005 and, 119–121

Plan My Night application, 2008 to 2010

- add-in module projects, 286–288
- AppFabric caching, 248–249
- architecture, 217–219
- ASP.NET caching, 249
- data, 219–221
- database data retrieval, 243–247
- existing projects, 223
- parallel programming, 247–248

Plan My Night application, deploying application, 103–113

PlanMyNight.Web project, 39–42, 153–156, 251–254

PLINQ libraries, 36, 100, 150, 248

plug-ins support, 77, 190, 288

POCO templates, 22–23, 138, 222, 239

PrintItinerary.Addin project, 77–79, 190–193, 288–291

profiles

- CreateProfile, 48, 162, 260
- retrieving current user data, 51–53, 165–167, 262–265
- updating data, 53–58, 167–173, 265–270
- UserProfile, 10–14, 126–129, 227–230

Q

query writing, 28, 143

R

requests, ASP.NET MVC, 54, 168, 266

S

SearchByActivity, 27, 28–30, 142, 143–144, 244–245

SearchByRadius, 27, 31–32, 143, 145–146, 244, 246

SearchByZipCode, 27, 30–31, 142, 144–145, 244, 245

self-tracking entities, 222

server-side controls, 62, 176, 273

SOS tool, 92, 205, 303

stored procedure, 15–16, 130–131, 231–232

T

T4 templates (Text Template Transformation Toolkit), 22–23, 25, 138, 140, 222, 239, 241

testing, unit, 95–100, 208–213, 307–311

TFSBuild, 104, 111, 113

Threads window, 100, 213, 311–312

U

unexpected conditions, 66–74, 180–187, 278–285

unit testing, 95–100, 208–213, 307–311

Update method, 53–54, 168, 265

UpdateSuccess view, 63–66, 177–180, 274–278

user authentication, 46–51, 160–165, 258–262

user data retrieval, 51–53, 165–167, 262–265

UserProfile, 10–14, 126–129, 227–230

V

ValidateAntiForgeryToken attribute, 54, 168, 265

ValidateUser method, 48, 162, 260

ViewModels, 41, 155, 253

views, 40, 154, 252

Visual Studio 2003

- Add Web Service Reference, 32–33
- add-ins to generate code, 25
- attributes, 51, 66
- breakpoint window, 84
- CSS application, 69
- debugging, 83, 87, 88, 92
- extension methods, 52
- generics and, 29
- InjectStatesAndActivityTypes method, 52–53
- jQuery, 87
- Plan My Night data and, 5–6
- plug-ins support, 77
- query writing and, 28
- requests, 54
- server-side controls, 62
- SOS tool, 92
- unit testing, 96
- Web Forms and, 44
- web.config file, 40–41

Visual Studio 2005

- Add Web Service Reference, 146
- attributes, 165, 179
- breakpoint window, 198
- CSS application, 183
- debugging, 198, 201, 205, 213, 312
- extension methods, 166
- InjectStatesAndActivityTypes method, 167
- Plan My Night application, 2005 to 2010 and, 119–121
- Plan My Night data, 119–121
- plug-ins support, 190
- query writing and, 143
- requests, 168
- server-side controls, 176
- SOS tool, 205
- unit testing, 209
- Web Forms and, 158
- web.config file, 155
- XSD processing, 140

Visual Studio 2008

- attributes, 263, 277
- breakpoint window, 296
- debugging, 296, 299, 303
- Entity Framework

 - enhancements, 222
 - foreign key associations, 235

- jQuery, 298
- LINQ debugging, 299
- plug-ins support, 288
- requests, 266
- server-side controls, 273
- singular vs. plural naming, 225
- SOS tool, 303

unit testing, 307
Web Forms and, 256
web.config file, 253
XSD processing, 241

W

WCF. *See* Windows Communication Foundation (WCF)
Web Application Project, 107
web deployment packages, 104–111
Web Forms
 ASP.NET MVC applications versus, 44, 158, 256
 using Designer View, 66–74, 180–187, 278–285
 Visual Studio 2003 and, 44
 Visual Studio 2005 and, 158
 Visual Studio 2008 and, 256
Web service data retrieval, 32–34, 146–148
Web Service Proxy, 34, 148
Web Setup Project deployment method, 104–106
web.config file, 40–41, 154–155, 253
web.config transformations, 95, 208, 307
Windows Communication Foundation (WCF), 33–34, 147–148

Windows Live ID authentication, 46–51, 160–165, 258–262
Windows Server AppFabric. *See* AppFabric caching
WindowsLiveLogin.User object, 47–48, 161–162, 259–260
Wix Toolset, 109, 113

X

XCOPY deployment, 104–111
XSD processing, 140, 241

About the Authors

Ken Haines is a software development engineer at Microsoft, working in the Consumer and Online Division. He has a passion for distributed applications in the cloud and strives to help customers and partners find the right solution for their needs.

For the past 12 years, he has worked in various roles in a wide range of industries, including Internet service providers, satellite telecommunications, and network monitoring and Web analytics. Some of these jobs have taken him to extremely remote locations, such as the Nunavut and Northwest Territories of Canada.

When not at Microsoft or developing software in his spare time, he enjoys hiking, mountain biking, nature photography, reading, and spending time with family. He resides with his family in Monroe, Washington.

Pascal Paré has worked at Microsoft since 2006, where he has held positions as a software engineer on both development and testing teams. He is currently a software development engineer in the Consumer and Online Division.

He graduated 13 years ago from Université Laval in Québec, Canada, as a computer engineer. He worked as a tester and developer in different companies and in a variety of industries (including fiber optics test equipment, telecommunications, and medical software) before joining Microsoft.

For leisure, he enjoys hiking, cycling, and skiing in the Pacific Northwest, as well as cooking and traveling. His favorite pastime is to take his Lotus Elise out for a spirited drive in the back roads around Puget Sound. He is married and currently lives in the Seattle area.

Patrice Pelland is a principal development manager at Microsoft, working in the Consumer and Online Division. He leads a development team that is focused on innovation and incubation across all Microsoft consumer products. He has a passion for complex distributed systems, for mobile development, and for helping consumers and partners around the world get the most out of Microsoft products.

For the past 17 years, he has worked in software development as both an individual contributor and a manager in various industries, including Web development, developer tools, fiber optics telecommunications, aviation, and coffee and dairy companies. He also spent three years teaching computer science and software development at a college in Canada.

When he is not developing software at Microsoft—and for fun in his spare time—he enjoys spending time with family and friends having great dinners with good food and fine drinks, traveling, cooking, reading books, reading about Porsche cars, watching hockey and football, and training at the gym. He resides with his family in Sammamish, Washington.

