Microsoft

ПРЕДВАРИТЕЛЬНАЯ ВЕРСИЯ

Завершенная книга будет доступна до конца 2010

Переход на Visual Studio 2010

Ресурсы Microsoft для технических специалистов

www.msdn.com/ru-ru

www.vs2010.ru

msdn

Патрис Пелланд,
Паскаль Паре, Кен Хэйнз

Глава 8

От 2008 к 2010: бизнес-логика и данные

В данной главе рассматривается

- Применение Entity Framework для создания слоя доступа к данным с использованием существующей базы данных или модель-ориентированного подхода
- Создание типов сущностей в дизайнере Entity Data Model (Модель сущность-данные) с использованием POCO шаблонов в ADO.NET Entity Framework
- Кэширование данных с использованием Windows Server AppFabric (ранее известной под кодовым названием Velocity)

Архитектура приложения

С помощью приложения PlanMyNight пользователь может составлять маршруты своих поездок и делиться ими с остальными. Данные сохраняются в базу данных SQL Server. Маршруты составляются на основании обращений к Веб-сервисам Bing Maps.

Рассмотрим исходную блок-схему модели данных приложения (рис. 8-1).

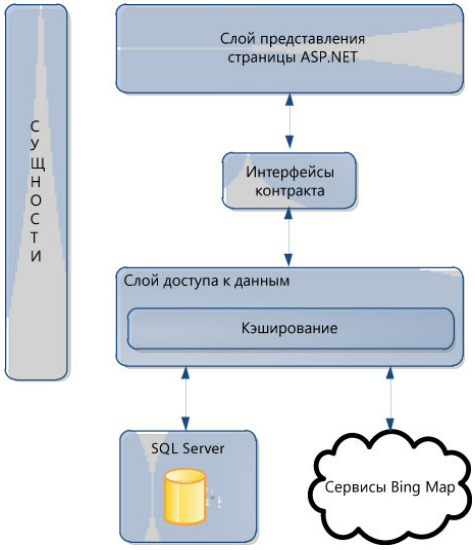


Рис. 8-1 Архитектурная диаграмма приложения PlanMyNight

Описание контрактов и классов сущностей, свободных от каких-либо ограничений, налагаемых методом хранения, позволит объединять их в сборки, не зависящие от метода

хранения. Это обеспечит четкое разделение слоев представления и доступа к данным.

Опишем интерфейсы контрактов основных компонентов приложения PMN:

- IItinerariesRepository (Хранилище маршрутов) будет интерфейсом хранилища данных поездки (база данных Microsoft SQL Server).
- IActivitiesRepository (Хранилище действий) позволит выполнять поиск действий (Веб-сервисы Bing Map).
- ICachingProvider (Поставщик кэширования) обеспечивает интерфейс кэширования данных (кэширование ASP.NET или кэширование AppFabric Windows Server).

Примечание Это неполный список контрактов, реализованных в приложении PMN.

PMN сохраняет маршруты пользователя в базе данных SQL. Все пользователи смогут комментировать и рейтинговать маршруты друг друга. На рис. 8-2 представлены таблицы, используемые приложением PMN.

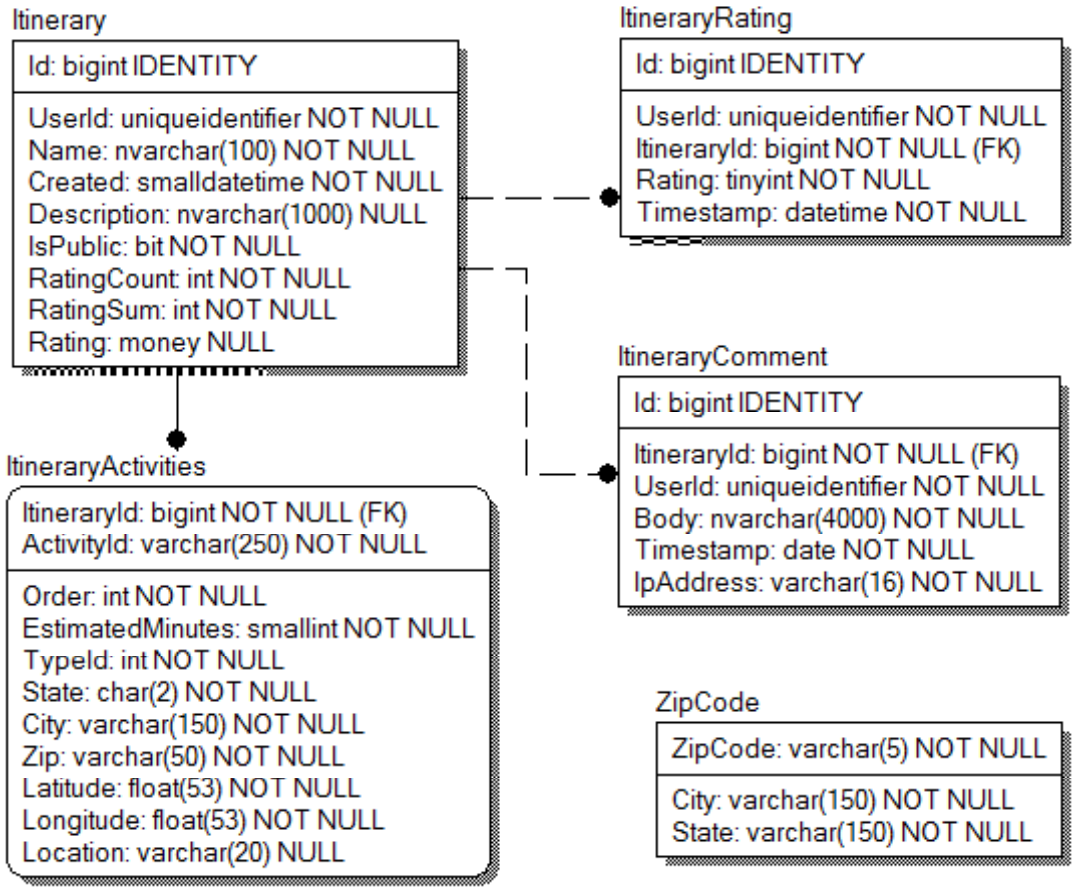


Рис. 8-2 Схема базы данных PlanMyNight

Важно В приложении PlanMyNight безопасное хранение учетных данных пользователей обеспечивается посредством возможностей членства ASP.NET. Таблицы хранилища пользователя на рис. 8-2 не показаны. Более подробную информацию об этих возможностях можно найти на сайте MSDN в разделе ASP.NET 4 – Introduction to Membership (ASP.NET 4 – введение в членство).

Данные PlanMyNight в Microsoft Visual Studio 2008

Visual Studio 2008 предоставляет все необходимые инструменты для создания приложения, поэтому написание PlanMyNight не составило бы никакого труда. Однако некоторые применяемые тогда технологии требовали написания намного большего объема кода для достижения тех же целей.

Рассмотрим, как создавался бы слой доступа к данным в Visual Studio 2008. Один из подходов – непосредственное использование DataSet (Множество данных) или DataReader (Средство чтения данных) ADO.NET. Такое решение предлагает большую гибкость, поскольку обеспечивает полный контроль над доступом к базе данных. С другой стороны, оно имеет определенные недостатки:

- Необходимо знать синтаксис SQL.
- Все запросы специализированы. Изменение в запросе или в таблицах приводит к необходимости обновления всех соответствующих запросов в коде.

PlanMyNightDatabase.cs | PlanMyNightDatabase.xsd

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;
    }
}
```

Рис. 8-3 Запрос Insert ADO.NET Insert

- Приходится сопоставлять свойства классов сущностей, используя имя столбца, что довольно утомительно и чревато многочисленными ошибками.
- Приходится самостоятельно управлять отношениями между таблицами.

Другой подход – использование дизайнера DataSet, предоставляемого Visual Studio 2008. При наличии базы данных с таблицами PMN, можно было бы использовать TableAdapter Configuration Wizard (Мастер настройки TableAdapter) для импорта таблиц базы данных, как показано на рис. 8-4. Автоматически сформированный код предлагает типизированный DataSet. К преимуществам относится контроль типов во время разработки, что обеспечивает автоматическое завершение выражений средствами редактора исходного кода. Но, тем не менее, данный подход имеет и недостатки:

- По-прежнему требуется знание синтаксиса SQL, хотя прямо из дизайнера DataSet имеется возможность доступа к построителю запросов.
- По-прежнему приходится создавать специализированные SQL-запросы, чтобы выполнять все требования контрактов данных.
- Нет возможности управления автоматически сформированными классами. Например, добавление или удаление запроса к таблице из DataSet приведет к повторной сборке автоматически сформированных классов TableAdapter и может изменить индекс, используемый для запроса. Это усложняет задачу по написанию предсказуемого кода с использованием таких автоматически формируемых классов.

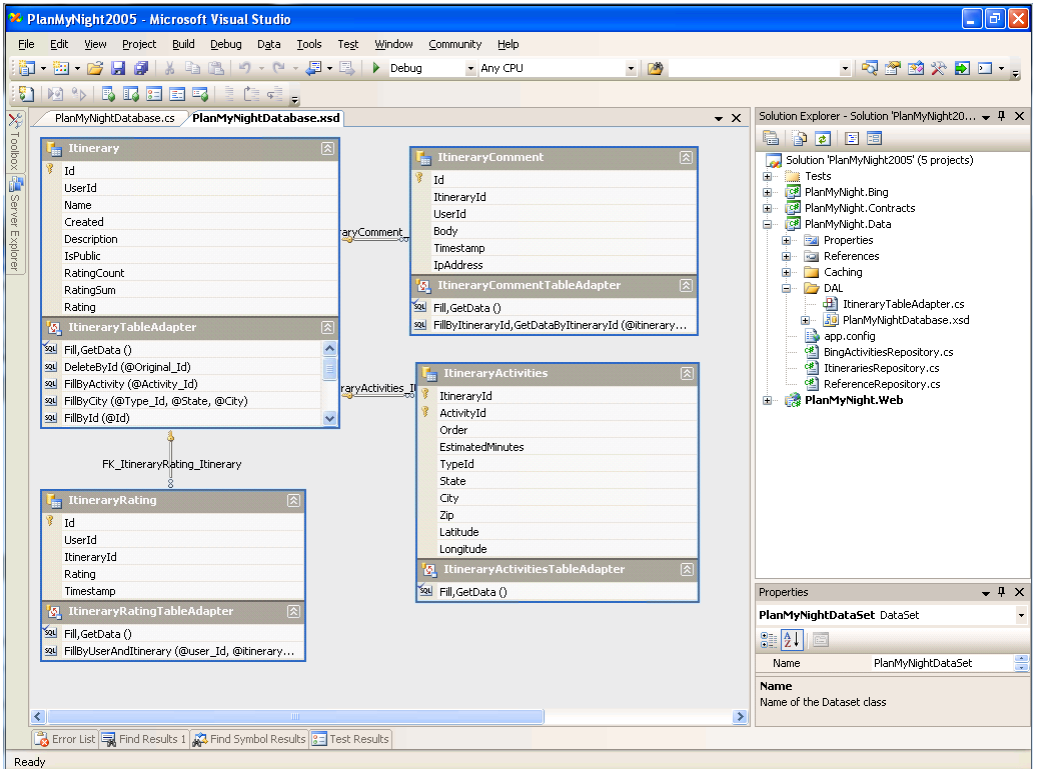


Рис. 8-4 Дизайнер DataSet в Visual Studio 2008

- Автоматически формируемые классы, ассоциированные с таблицами, включают код, зависящий от метода хранения, поэтому придется создавать другое множество простых сущностей и копировать данные туда. Это означает больший объем обработки и более активное использование памяти.

Еще одна технология, доступная в Visual Studio 2008 – LINQ to SQL (L2S). Объектно-реляционный дизайнер (Object Relational Designer) для L2S позволяет без труда добавлять привязки для необходимых приложению таблиц базы данных. Этот подход обеспечивает доступ к строго типизированным объектам и к LINQ для создания запросов для доступа к данным, т.е. знание синтаксиса SQL не требуется. Недостатки такого подхода:

- LINQ to SQL применим только для СУБД SQL Server.
- Ограниченные возможности управления создаваемыми сущностями. Не обеспечивается возможность простого обновления базы данных при изменении ее схемы.
- Автоматически создаваемые сущности включают код, зависящий от метода хранения.

Примечание Для .NET 4.0 Майкрософт рекомендует Entity Framework в качестве решения для сценариев использования LINQ с реляционными базами данных.

В следующих разделах данной главы будут рассмотрены некоторые новые возможности, предоставляемые Visual Studio 2010. Благодаря им создание слоя доступа к данным связано с меньшим объемом написания кода, предоставляется больший контроль над автоматически формируемым кодом, обслуживать и расширять его становится намного проще.

Работа с данными в Visual Studio 2010 с использованием Entity Framework

ADO.NET Entity Framework (EF) позволяет без труда создавать слой доступа к данным приложения через абстрагирование работы с данными от базы данных и создание модели, приближенной к бизнес-требованиям приложения. EF была существенно улучшена в выпущенной версии .NET Framework 4.

Будем использовать приложение PlanMyNight в качестве примера использования некоторых возможностей EF. В следующих двух разделах демонстрируются два разных подхода автоматического формирования модели данных PMN. В первом случае EF создаст Entity Data Model (EDM) из существующей базы данных. Во втором – используется модель-ориентированный подход, при котором сначала в дизайнере EF создаются сущности, а затем для создания базы данных, в которой может храниться EDM, автоматически формируются сценарии на языке описания данных (Data Definition Language, DDL).

Visual Studio 2008 Первая версия Entity Framework была выпущена с Visual Studio 2008 Service Pack 1. Вторая версия EF, вошедшая в состав .NET Framework 4.0, предлагает множество новых возможностей, которые облегчат создание приложений для работы с данными. Назовем некоторые из них:

- Шаблоны формирования кода T4, которые можно настроить соответственно собственным требованиям.
- Возможность определения собственных POCO (Plain Old CLR Objects[1]), что гарантирует разделение сущностей и технологии хранения.
- Модель-ориентированная разработка, т.е. сначала создается модель сущностей, на базе которой Visual Studio 2010 формирует базу данных.
- Поддержка подхода – только код, что позволяет работать с Entity Framework, используя сущности POCO, и без файла EDMX.
- Отложенная загрузка для связанных сущностей, что обеспечивает их загрузку из базы данных только по мере надобности.
- Самоотслеживающиеся сущности, которые регистрируют собственные изменения на клиенте и передают их в хранилище базы данных.

В следующих разделах мы рассмотрим некоторые из этих новых возможностей. Большое количество ресурсов по ADO.NET Entity Framework в .NET 4 также предлагается на сайте MSDN в разделе Data Developer Center (Центр решений по работе с данными).

EF: импорт существующей базы данных

Мы будем работать с уже существующим решением, в котором описаны основные проекты приложения PMN. Если сопроводительные материалы данной книги установлены в каталог по умолчанию, интересующее нас решение находится по адресу %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ExistingDatabase. Щелкните двойным щелчком файл PlanMyNight.sln.

Это решение включает все проекты, показанные на рис. 8-5.

- PlanMyNight.Data: слой доступа к данным приложения.
- PlanMyNight.Contracts: сущности и контракты.
- PlanMyNight.Bing: сервисы Bing Map.
- PlanMyNight.Web: слой представления.
- PlanMyNight.AppFabricCaching: кэширование AppFabric.

[1] Обычные CLR-объекты (*прим. переводчика*).

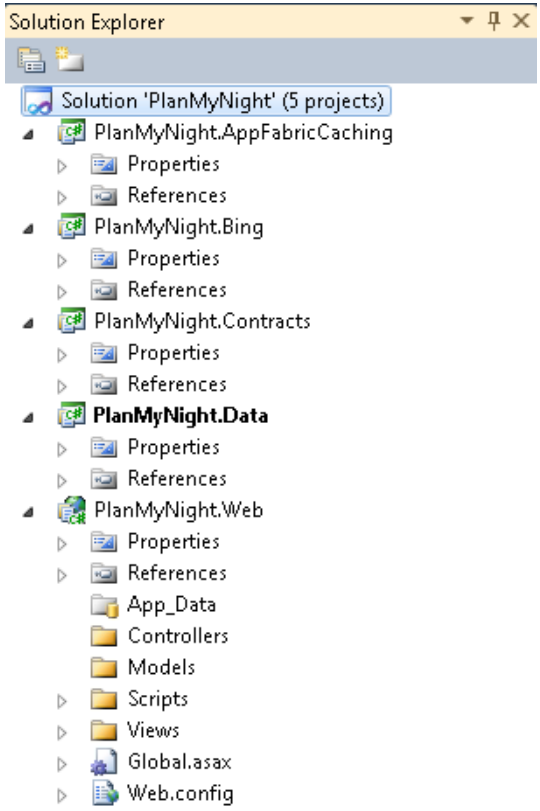


Рис. 8-5 Решение PlanMyNight

EF позволяет без труда импортировать существующую базу данных. Рассмотрим этот процесс поэтапно.

Первый шаг – добавление EDM в проект PlanMyNight.Data. Щелкаем правой кнопкой мыши проект PlanMyNight.Data, выбираем Add (Добавить) и New Item… (Новый элемент…). Выбираем элемент ADO.NET Entity Data Model и меняем его имя на PlanMyNight.edmx, как показано на рис. 8-6.

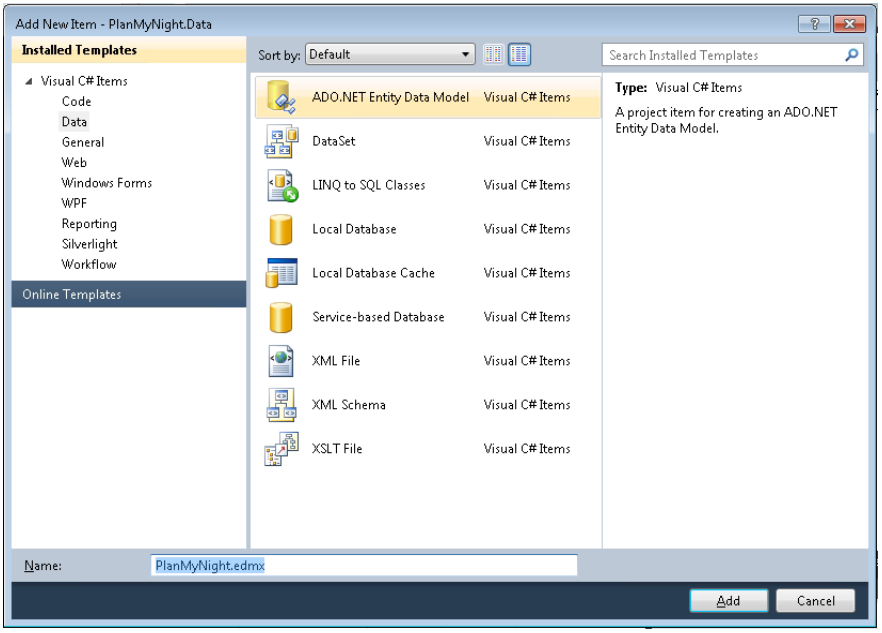


Рис. 8-6 Диалоговое окно Add New Item с ADO.NET Entity Data Model

Первое диалоговое окно мастера Entity Data Model Wizard позволяется выбрать содержимое модели. Мы собираемся создать модель из существующей базы данных. Выбираем Generate From Database (Создать из базы данных) и щелкаем Next (Далее).

Необходимо подключиться к существующему файлу базы данных. Щелкаем New Connection… (Создать подключение…) и выбираем файл %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 5\ExistingDatabase\PlanMyNight.Web\AppData\PlanMyNight.mdf.

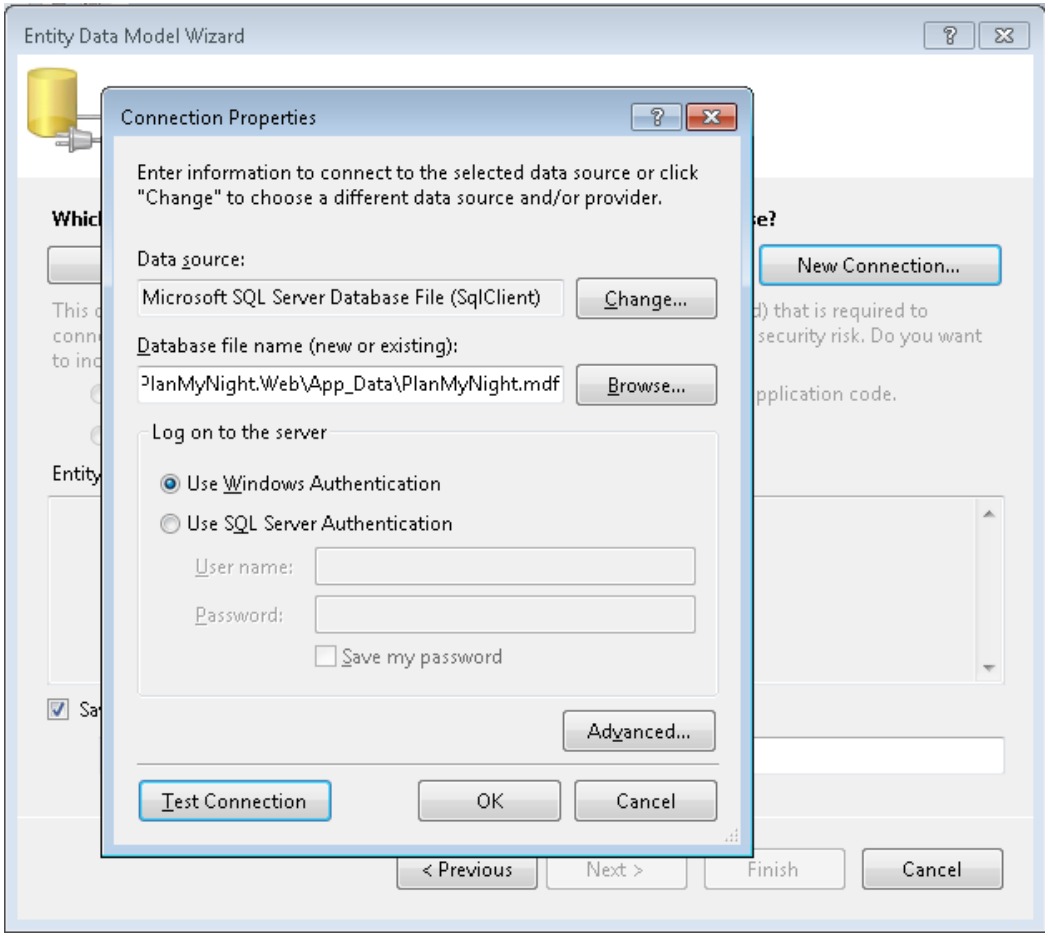


Рис. 8-7 Мастер EDM, диалог создания подключения к базе данных

Все остальные поля формы пока оставляем без изменений и щелкаем Next.

В диалоговом окне Chose Your Database Objects (Выберите объекты своей базы данных) выберите таблицы Itinerary (Маршрут), ItineraryActivities (Действия на маршруте), ItineraryComment (Комментарий к маршруту), ItineraryRating (Рейтинг маршрута) и ZipCode (Почтовый индекс), а также представление UserProfile (Профиль пользователя). Выберите хранимую процедуру RetrieveItinerariesWithinArea (Извлечение маршрутов по определенному району). В поле Model Namespace (Пространство имен модели) зададим Entities (Сущности), как показано на рис. 8-8.

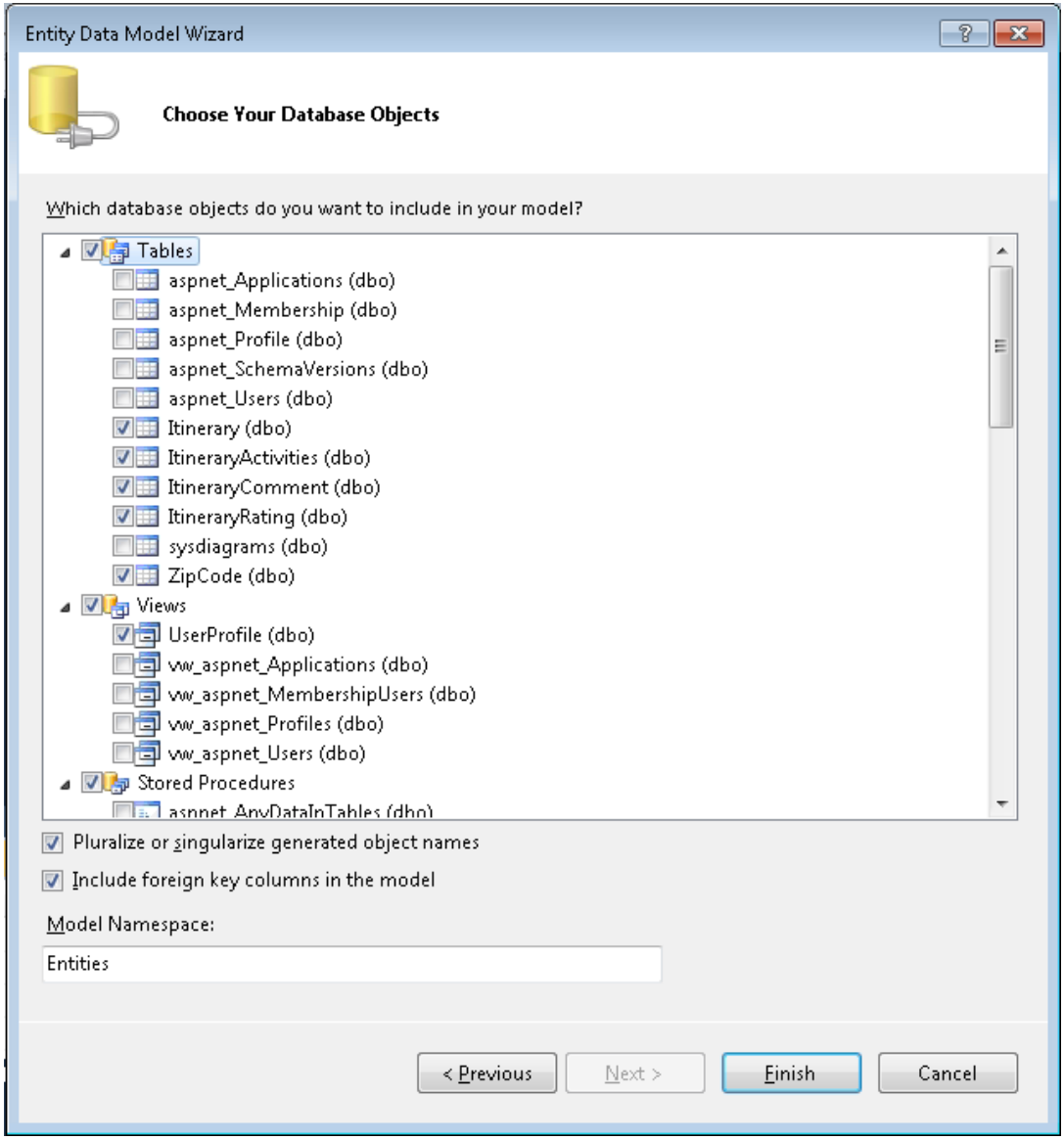


Рис. 8-8 Мастер EDM, выбор объектов базы данных

Щелкаем Finish (Готово), чтобы сформировать EDM.

> **Visual Studio 2008** В первой версии EF при создании модели из базы данных имена, ассоциированные с EntityType (Тип сущности), EntitySet (Множество сущностей) и NavigationProperty (Свойство навигации), часто были неверными, потому что они создавались на основании имени таблицы базы данных. Наверняка, нам не нужен экземпляр сущности с именем во множественном числе, *ItineraryActivities*, скорее

всего, имя должно быть в единственном числе: *ItineraryActivity*. Кнопка-флажок *Pluralize or singularize generated object names* (Образовывать форму единственного или множественного числа для формируемых имен объектов), показанная на рис. 8-8, позволяет управлять тем, в какой форме будут создаваться имена объектов.

Корректировка сформированной модели данных

Теперь мы имеем модель, которая представляет собой множество сущностей, соответствующее используемой базе данных. Мастер сформировал все навигационные свойства, ассоциированные с внешними ключами базы данных.

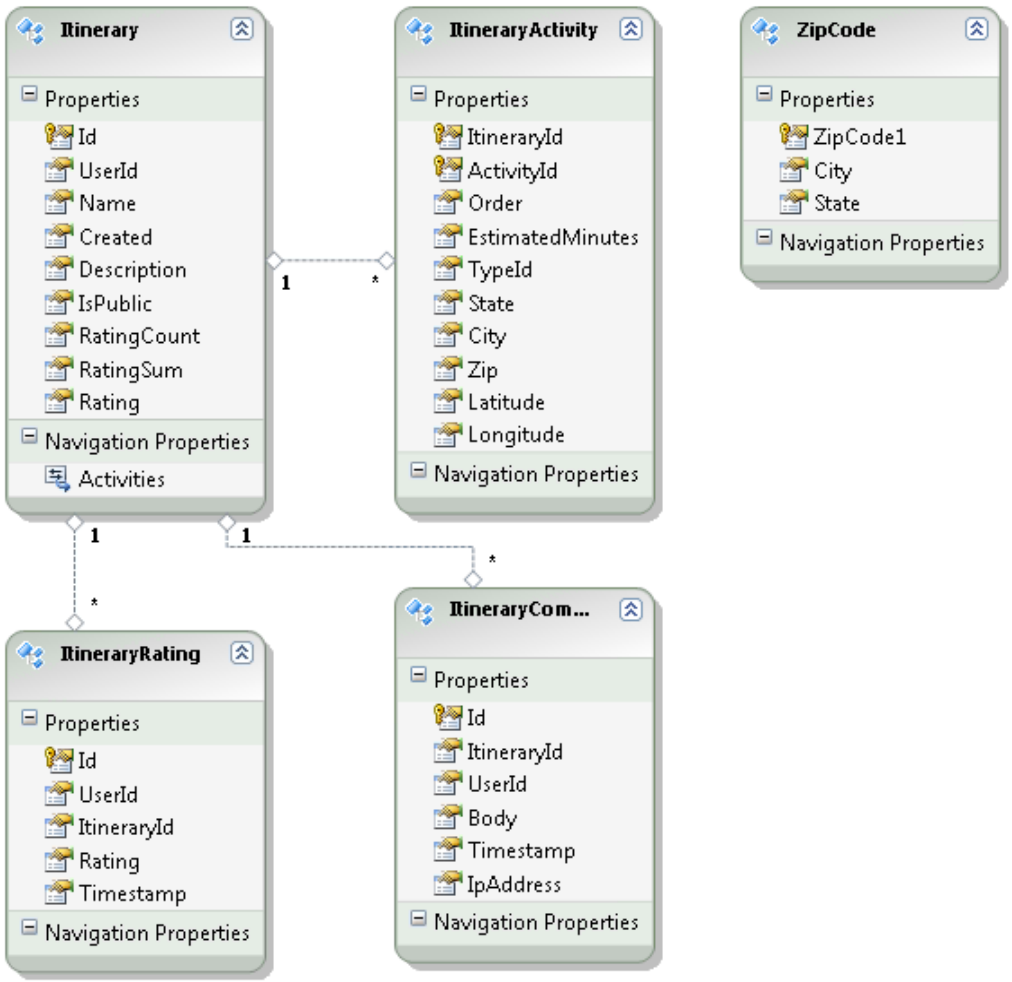


Рис. 8-9 Модель, импортированная из базы данных PlanMyNight

Приложению PMN необходимо только навигационное свойство, связанное с таблицей ItineraryActivies, поэтому все остальные навигационные свойства можно смело удалить. Также придется переименовать навигационное свойство ItineraryActivities в Activities. Обновленная модель представлена на рис. 8-9.

Можно заметить, что одному из свойств сущности ZipCode присвоено имя ZipCode1. Чтобы исправить это имя, щелкнем по нему двойным щелчком. Зададим имя Code, как показано на рис. 8-10.

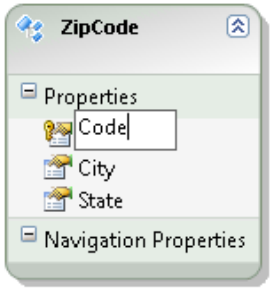


Рис. 8-10 Сущность ZipCode

Выполним сборку решения, нажав Ctrl+Shift+B. В окне вывода можно увидеть два сообщения о сформированной EDM. Первое можно сразу удалить, поскольку столбец Location (Местоположение) в PMN не нужен. Второе сообщение гласит:

> *Для таблицы/представления 'dbo.UserProfile' первичный ключ не задан и не может быть сформирован. Данная таблица/представление не включена в модель. Для использования этой сущности необходимо извлечь схему, добавить соответствующие ключи и раскомментировать ее.*

Взглянув на представление UserProfile можно заметить, что первичный ключ не задан явно, даже несмотря на то что значения столбца UserName (Имя пользователя) уникальны.

Придется вручную подкорректировать EDM, исправив сопоставление для представления UserProfile.

В обозревателе проекта щелкните правой кнопкой мыши файл PlanMyNight.edmx и выберите Open With… (Открыть в…). В диалоговом окне Open With выберите XML (Text) Editor ((Текстовый) редактор XML), как показано на рис. 8-11. Это обеспечит открытие XML-файла, связанного с нашей моделью[2].

[2] Этот XML-файл содержит описание всей созданной модели (*прим. технического редактора*)

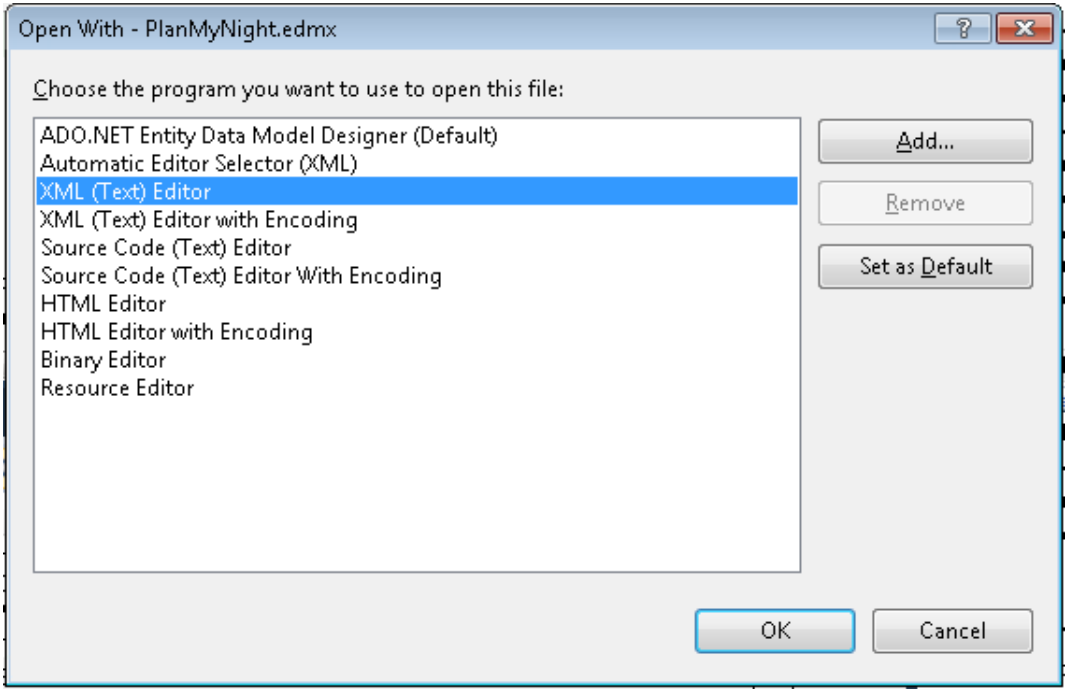


Рис. 8-11 Открываем PlanMyNight.emdx в редакторе XML

> **Примечание** На экран будет выведено сообщение о том, что файл PlanMyNight.edmx уже открыт. Щелкните Yes, чтобы закрыть его.

Механизм формирования кода закомментировал созданный код из-за отсутствия первичного ключа. Чтобы использовать представление UserProfile из дизайнера, необходимо раскомментировать тип сущности UserProfile и добавить в него тег Key (Ключ). Найдите UserProfile в файле. Раскомментируйте тип сущности, добавьте тег ключа, задайте для него имя **UserName** и сделайте свойство UserName не допускающим значение null. Обновленный EntityType представлен в листинге 8-1.

Листинг 8-1 XML-описание типа сущности UserProfile

```
<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>
```

Если закрыть XML-файл и попытаться открыть дизайнер EDM, возникнет такая ошибка:

Error 102: Entity type 'UserProfile' has no entity set.[3]

Необходимо задать множество сущностей для типа UserProfile, чтобы тип сущности мог проецироваться на схему хранилища. Для этого открываем файл PlanMyNight.edmx в редакторе XML. Вверху файла, прямо над множеством сущностей Itinerary, добавляем XML-код, представленный в листинге 8-2.

Листинг 8-2 XML-описание множества сущностей для UserProfile

```
<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>
```

Сохраним XML-файл EDM и повторно откроем дизайнер EDM. На рис. 8-12 показано представление UserProfile в разделе Entities.Store браузера модели (Model Browser).

Подсказка Model Browser можно открыть из меню View (Вид), щелкнув Other Windows (Другие окна) и выбрав Entity Data Model Browser.

[3] Для типа сущности 'UserProfile' не задано множество сущностей (*прим. переводчика*).

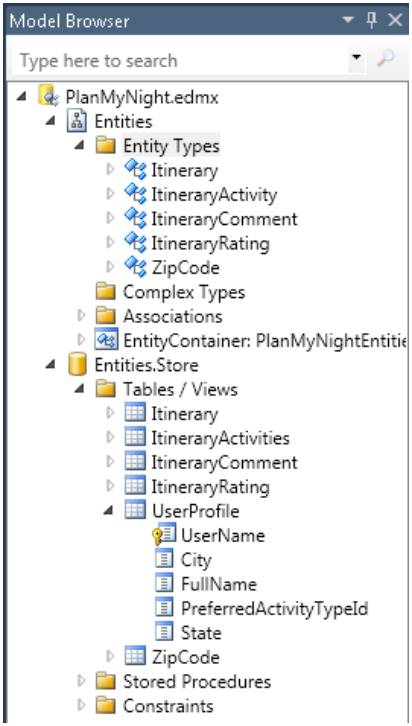


Рис. 8-12 Представление UserProfile в Model Browser

Теперь, когда представление доступно в метаданных хранилища, добавим сущность UserProfile и сопоставим ее с представлением UserProfile. Щелкаем правой кнопкой мыши фон дизайнера EDM, выбираем Add и затем Entity...

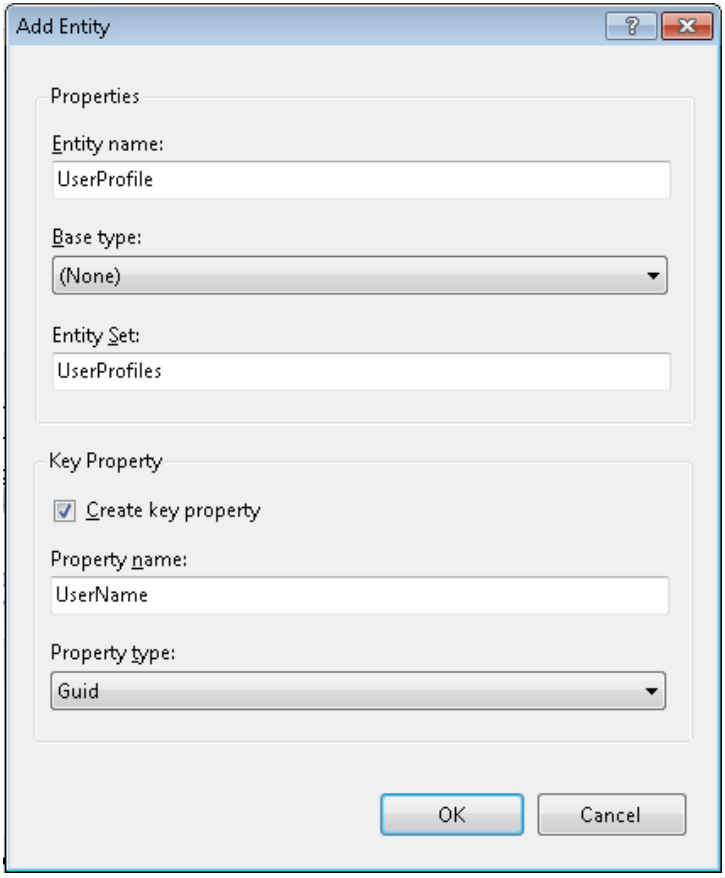


Рис. 8-13 Диалоговое окно для добавления сущности UserProfile

Заполните диалоговое окно, как показано на рис. 8-13, и щелкните OK, чтобы создать сущность. После этого понадобиться добавить остальные свойства: *City* (Город), *State* (Страна) и *PreferedActiviyTypeId* (Идентификатор предпочтительного типа действия). Для этого щелкаем правой кнопкой мыши сущность UserProfile, выбираем Add и Scalar Property (Скалярное свойство). Как только свойство добавлено, задаем значения полей *Type* (Тип), *Max Length* (Максимальная длина) и *Unicode*. В табл. 8-1 представлены ожидаемые значения каждого из полей.

Таблица 8-1 Свойства сущности UserProfile

Name	Type	Max Length	Unicode
FullName	String	500	False
City	String	500	False
State	String	500	False
PreferedActivityTypeId	Int32	Недоступно	Недоступно

Теперь, когда сущность UserProfile создана, ее требуется сопоставить с представлением UserProfile. Щелкаем правой кнопкой мыши сущность UserProfile и выбираем Table Mapping (Сопоставление таблиц), как показано на рис. 8-14.

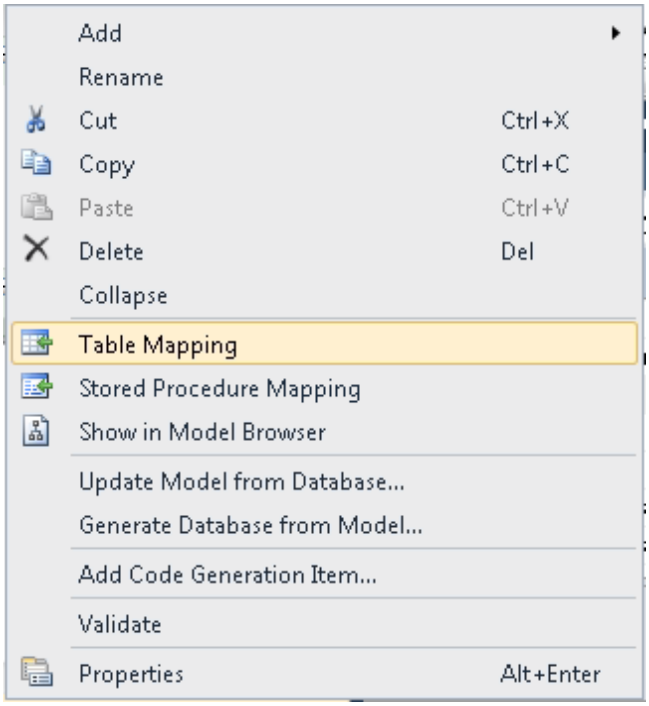


Рис. 8-14 Пункт меню Table Mapping

Затем выбираем представление UserProfile из раскрывающегося списка, как показано на рис. 8-15. Убедитесь, что все столбцы и свойства сущности сопоставлены правильно. Теперь представление UserProfile нашего хранилища доступно из кода через сущность UserProfile.

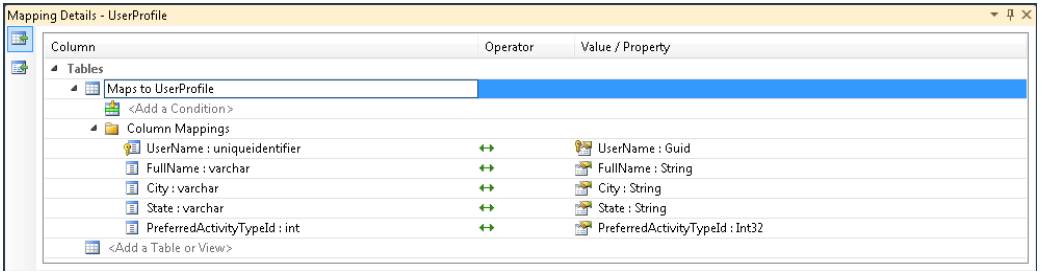


Рис. 8-15 Детали сопоставления UserProfile

Хранимая процедура и импортированные функции

Мастер Entity Data Model Wizard (Мастер модели сущность-данные) создал в модели хранилища запись для хранимой процедуры *RetrieveItinerariesWithinArea*, которая была выбрана в мастере в заключительном шаге. Необходимо создать соответствующую запись в концептуальной модели, добавив Function Import (Импортированная функция).

В разделе Entities.Store браузера модели откройте папку Stored Procedures (Хранимые процедуры). Щелкните правой кнопкой мыши *RetrieveItineraryWithinArea* и выберите Add Function Import… (Добавить импортированную функцию). Диалоговое окно Add Function Import представлено на рис. 8-16. Задайте возвращаемый тип, выбрав Entities, и элемент Itinerary в окне раскрывающегося списка. Щелкните OK.

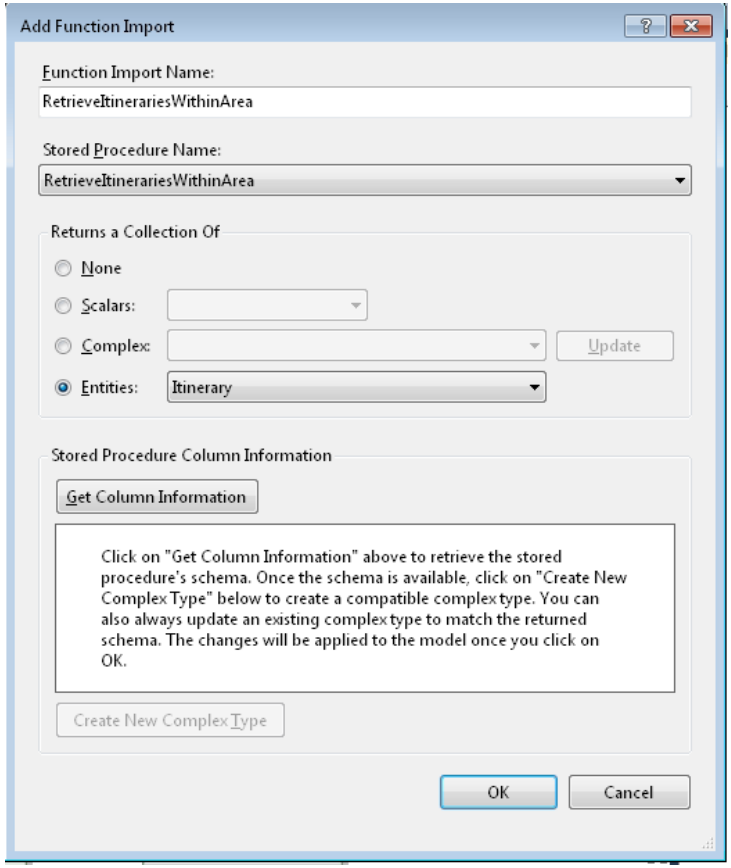


Рис. 8-16 Диалоговое окно Add Function Import

В браузер модели добавлена импортированная функция *RetrieveItinerariesWithinArea*, как показано на рис. 8-17.

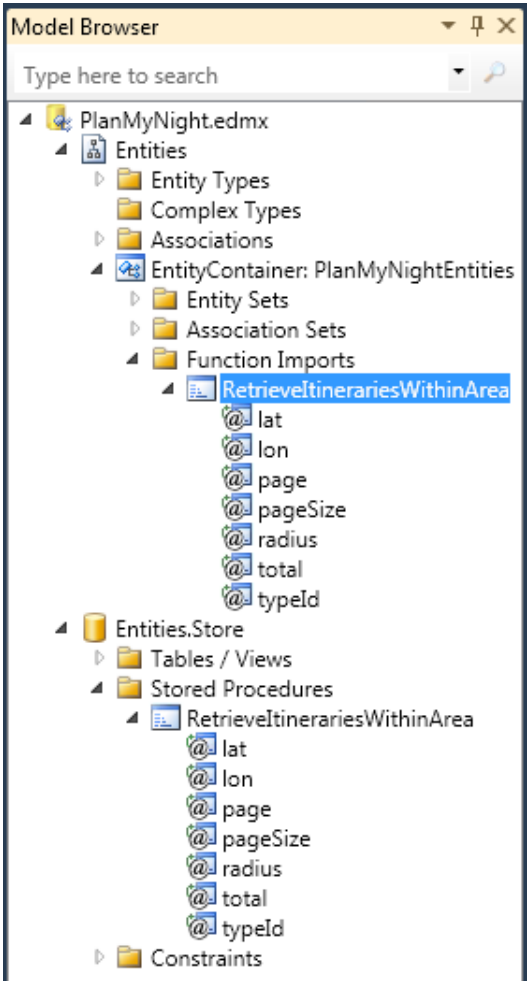


Рис. 8-17 Импортированные функции и браузере модели

Теперь можно проверить правильность EDM, щелкнув правой кнопкой мыши рабочую область дизайнера и выбрав Validate (Проверить). При этом не должно появиться никаких ошибок или предупреждений.

EF: сначала модель[4]

В предыдущем разделе мы увидели, как использовать дизайнер EF для создания модели путем импорта существующей базы данных. Дизайнер EF в Visual Studio 2010 также поддерживает возможность автоматического формирования файла Data Definition Language (DDL), который позволит создать базу данных из модели сущностей.

Можно начать с пустой модели. Для этого в мастере Entity Data Model Wizard выбираем опцию Empty model (Пустая модель).

[4] Модель-ориентированный подход (*прим. технического редактора*)

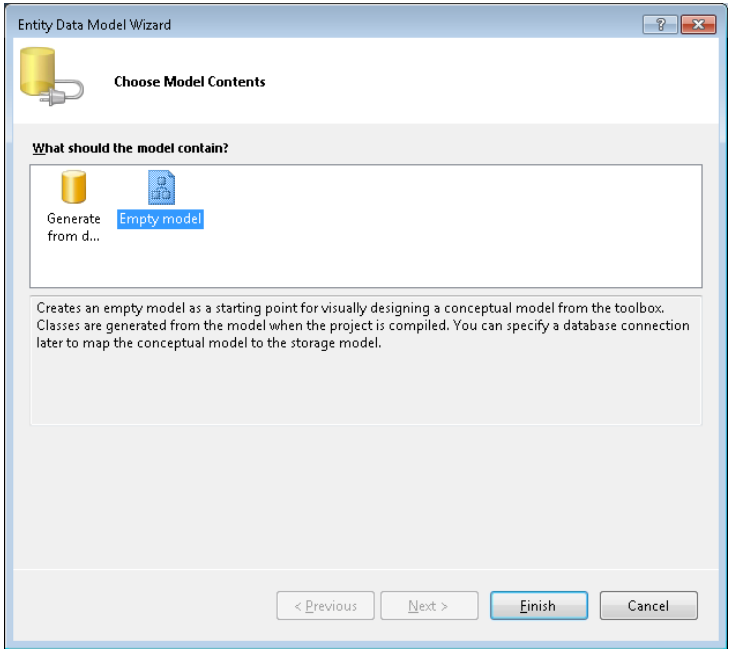


Рис. 8-18 Мастер EDM: модель сущностей

Открываем решение PMN по адресу %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ModelFirst, щелкнув двойным щелчком файл PlanMyNight.sln.

Проект PlanMyNight.Data этого решения уже содержит EDM-файл PlanMyNight.edmx с некоторыми уже созданными сущностями. Эти сущности соответствуют схеме данных, представленной на рис. 8-2.

Дизайнер Entity Model позволяет без труда добавлять сущности в модель данных. Добавим в нашу модель недостающую сущность ZipCode. Из панели инструментов перенесите в дизайнер элемент Entity, как показано на рис. 8-19. Присвойте этой сущности имя ZipCode, свойству *Id* – имя Code и задайте для него тип String.

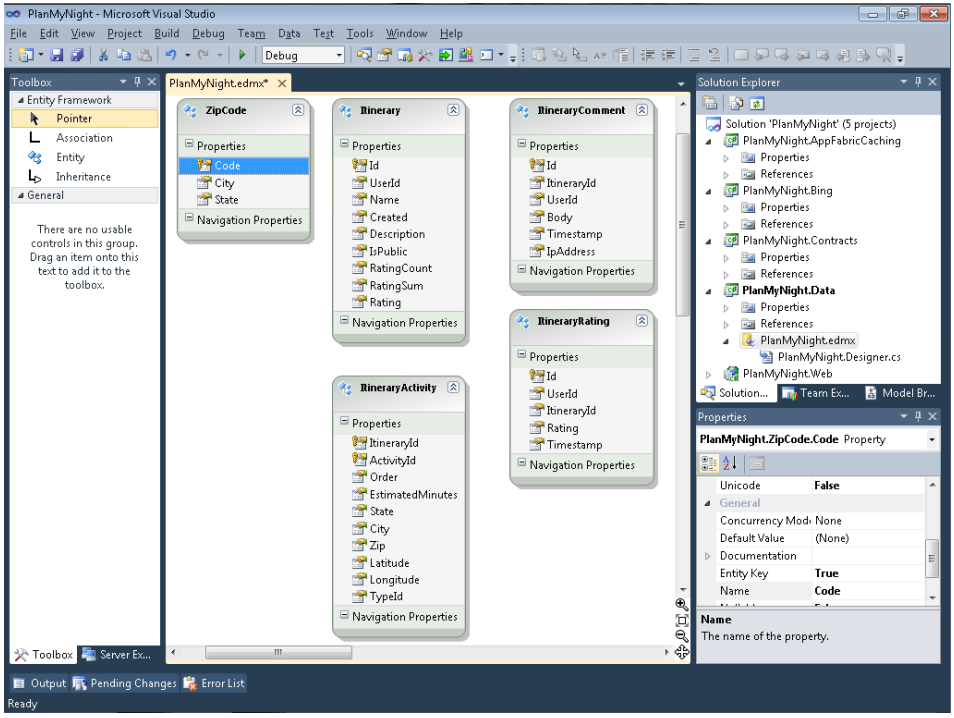


Рис. 8-19 Дизайнер модели сущностей

Теперь необходимо добавить в сущность свойства *City* и *State*. Щелкаем правой кнопкой мыши сущность ZipCode, выбираем Add и затем Scalar Property. Убеждаемся, что значения всех свойств соответствуют приведенным в табл. 8-2.

Таблица 8-2 Свойства сущности ZipCode

Name	Type	Fixed Length	Max length	Unicode
Code	String	False	5	False
City	String	False	150	False
String	String	False	150	False

Добавим отношения между ItineraryComment и сущностями Itinerary. Щелкаем правой кнопкой мыши поверхность дизайнера, выбираем Add и затем Association… (Связь).

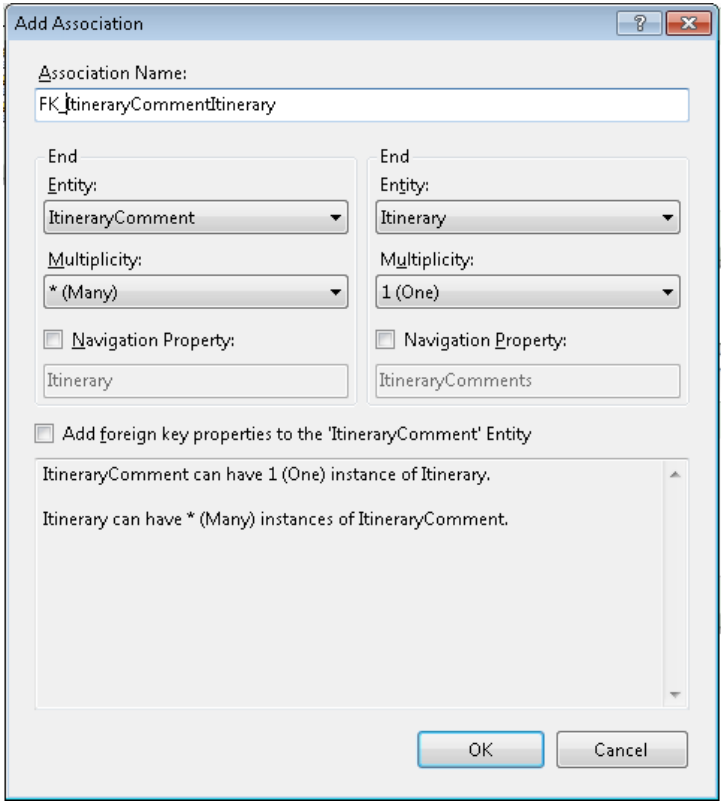


Рис. 8-20 Диалоговое окно Add Association для FK_ItineraryCommentItinerary

> **Visual Studio 2008** В версию Entity Framework .NET 4.0 включены Foreign Key Associations (Связи внешних ключей). Это позволяет задавать для сущностей внешние свойства. Foreign Key Associations теперь является типом по умолчанию для связи, но поддерживаемый в .NET 3.5 тип Independent Associations (Независимые связи) по-прежнему доступен.

Задайте имя связи FK_ItineraryCommentItinerary, выберите сущность и кратность для каждого конца связи, как показано на рис. 8-20. Как только связь создана, щелкните двойным щелчком строку связи, чтобы задать Referential Constraint (Справочное ограничение), как показано на рис. 8-21.

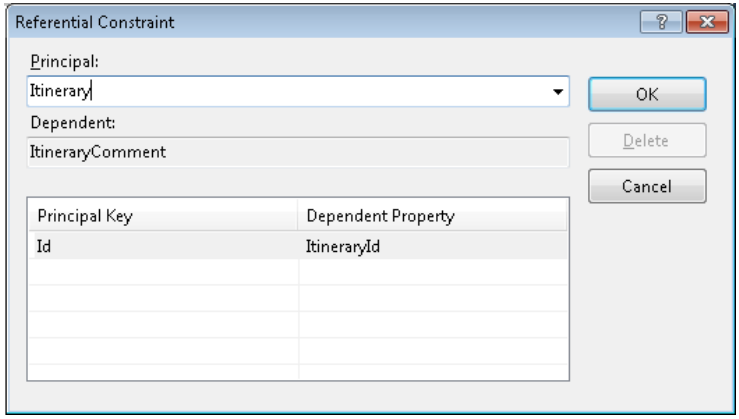


Рис. 8-21 Диалоговое окно Association Referential Constraint

Те же действия повторите для связи FK_ItineraryItineraryRating, задав ItineraryRating как первой конец связи.

Для связи FK_ItineraryItineraryActivity также надо создать свойство навигации и назвать его Activities, как показано на рис. 8-22.

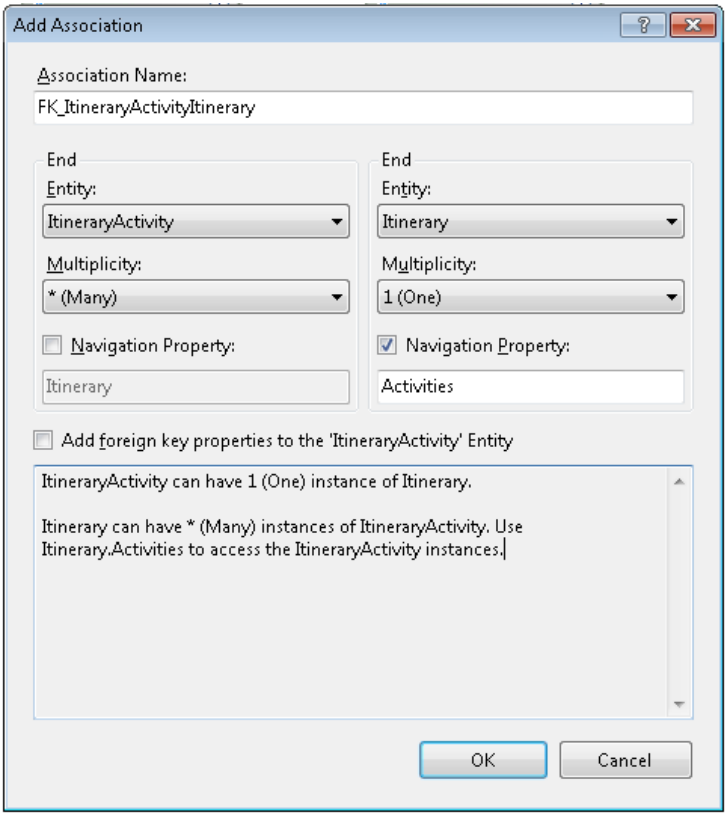


Рис. 8-22 Диалоговое окно Add Association для FK_ItineraryActivityItinerary

Автоматическое формирование сценария базы данных из модели

Модель данных готова, но с ней не ассоциировано никакое хранилище или сопоставление. Дизайнер EF предлагает возможность автоматического формирования сценария базы данных из модели.

Щелкните правой кнопкой мыши дизайнер и выберите пункт меню Generate Database From Model… (Сформировать базу данных из модели), как показано на рис. 8-23.

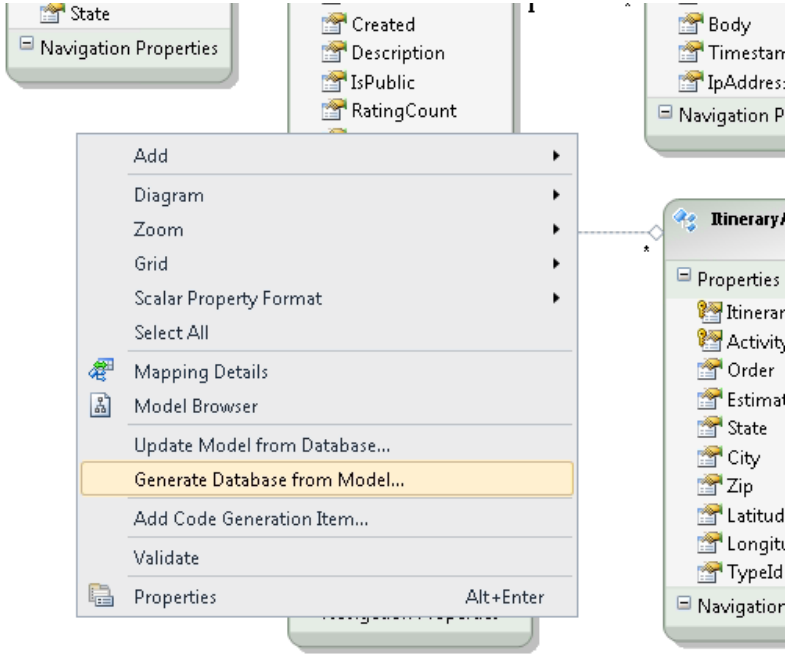


Рис. 8-23 Пункт меню Generate Database from Model

Мастер Generate Database Wizard требует подключения к данным. Этот мастер будет использовать данные подключения для трансляции типов модели в тип базы данных и для формирования DDL-сценария для этой базы данных.

Выберите New Connection… и убедитесь, что в качестве Data Source (Источник данных) задан Microsoft SQL Server File. Нажмите Browse… (Обзор) и выберите файл базы данных, находящийся по адресу %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ModelFirst\Data\PlanMyNight.mdf.

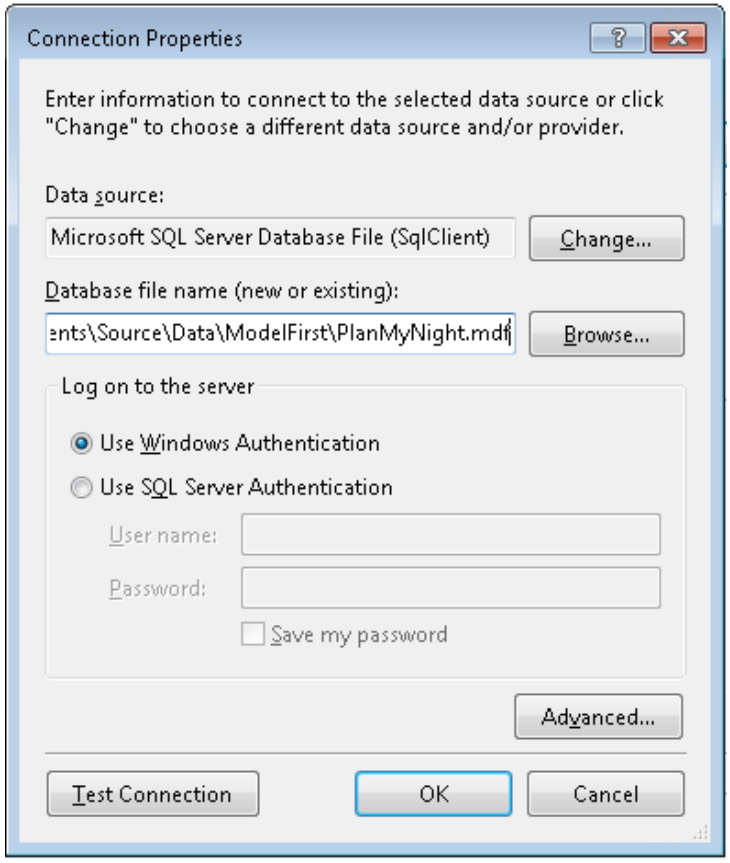


Рис. 8-24 Подключение базы данных для формирования сценария

Как только подключение настроено, щелкните Next, чтобы перейти к последнему окну мастера. По нажатию Finish сформированный файл T-SQL PlanMyNight.edmx.sql добавляется в проект. DDL-сценарий сформирует ограничения первичного и внешнего ключей для модели.

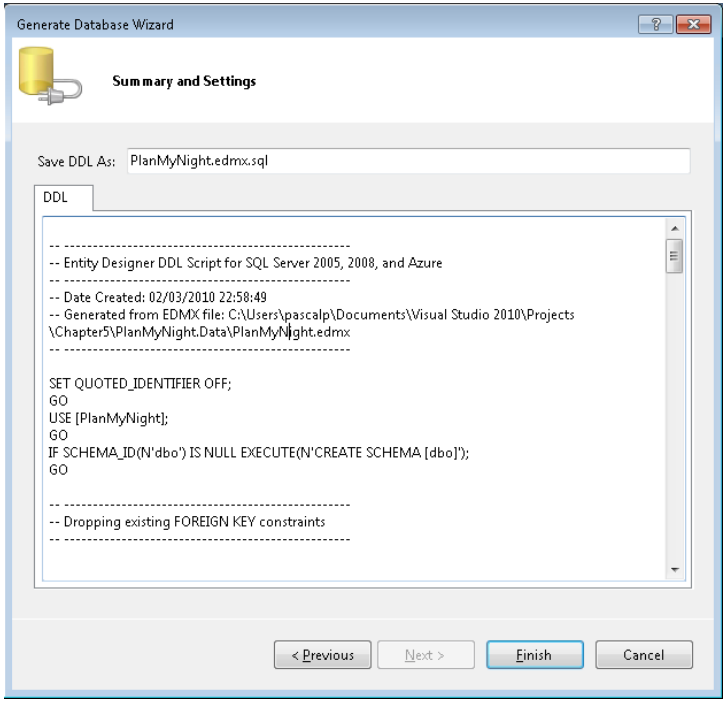


Рис. 8-25 Сформированный файл Generated T-SQL

EDM также обновляется, чтобы вновь созданное хранилище гарантированно сопоставлялось с сущностями. Теперь сформированный DDL-сценарий может использоваться для добавления таблиц в базу данных, и мы получили слой доступа к данным, предоставляющий строго типизированные сущности, которые могут использоваться в приложении.

> **Важно** Для создания полной базы данных PMN потребовалось бы добавить оставшиеся таблицы, хранимые процедуры и триггеры, используемые приложением. Вместо того чтобы осуществлять все эти операции, вернемся к решению, каким оно было в конце раздела «EF: импорт существующей базы данных».

POCO шаблоны

Для формирования кода сущностей дизайнер EDM использует шаблоны T4. До сих пор мы позволяли дизайнеру создавать сущности на базе шаблонов по умолчанию. Автоматически сформированный код можно увидеть в файле PlanMyNight.Designer.cs, ассоциированном с PlanMyNight.edmx. Созданные в нем сущности типа EntityObject и снабжены атрибутами, которые обеспечивают возможность EF управлять ими во время выполнения.

> **Примечание** T4 расшифровывается как Text Template Transformation Toolkit (Набор инструментов для преобразования текстовых шаблонов). Поддержка T4 в Visual Studio 2010 позволяет без труда создавать собственные шаблоны и формировать

любые текстовые файлы (Веб, ресурсы или источники). Более подробные сведения об автоматическом формировании кода в Visual Studio 2010 можно найти в статье «Code Generation and Text Templates» (Автоматическое формирование кода и текстовые шаблоны).

EF также поддерживает типы сущностей POCO. POCO-классы – это простые объекты без атрибутов или базового класса инфраструктуры. (Листинг 8-3 в следующем разделе представляет POCO-класс для сущности ZipCode.) EF использует имена типов и свойства этих объектов для их сопоставления с моделью во время выполнения.

Примечание POCO расшифровывается как Plain-Old CLR Objects.

Шаблон ADO.NET POCO Entity Generator

Повторно откроем файл %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\ExistingDatabase\PlanMyNight.sln.

Выберем файл PlanMyNight.edmx, щелкнем правой кнопкой мыши дизайнер и выберем Add Code Generation Item... (Добавить элемент формирования кода). При этом откроется диалоговое окно, показанное на рис. 8-26, которое предоставляет возможность выбрать необходимый шаблон. Выберем шаблон ADO.NET POCO Entity Generator[5] и назовем его PlanMyNight.tt. После этого щелкнем кнопку Add.

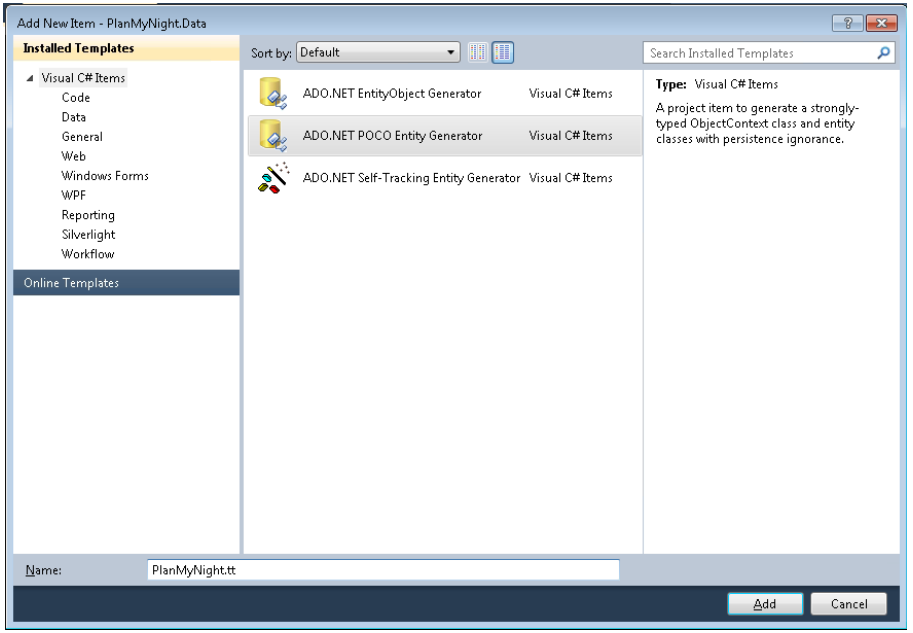


Рис. 8-26 Диалоговое окно Add New Item

[5] По-умолчанию данный шаблон не установлен. Если у Вас его нет, выберите страницу Online Templates (он-лайн шаблоны), а далее введите слово POCO в поле Search Online Templates (Поиск он-лайн шаблонов) и нажмите Enter, через некоторое время шаблон будет найден, его можно будет установить и использовать (*прим. технического редактора*)

В проект были добавлены два файла (рис. 8-27). Эти файлы заменяют шаблон формирования кода по умолчанию, и код больше не формируется в файле PlanMyNight.Designer.cs.

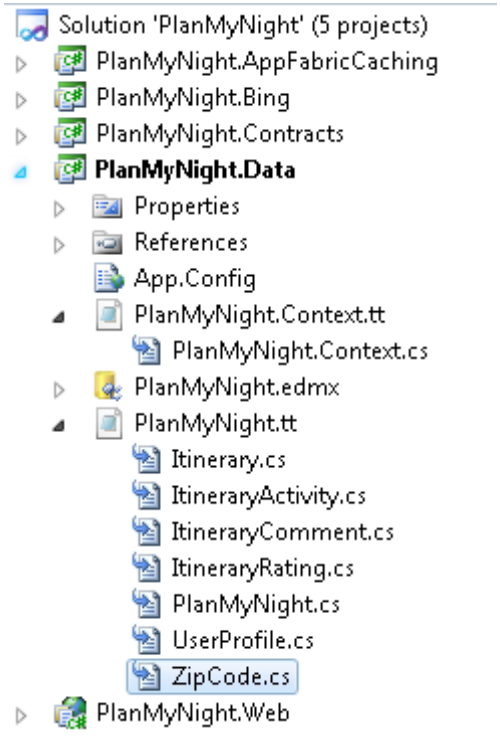


Рис. 8-27 Добавленные шаблоны

Шаблон PlanMyNight.tt создает по файлу класса для каждой сущности в модели. Листинг 8-3 представляет POCO-версию класса *ZipCode*.

Листинг 8-3 POCO-версия класса *ZipCode*

```
namespace Microsoft.Samples.PlanMyNight.Data
{
    public partiall class ZipCode
    {
        #region Primitive Properties
        public virtuall string Code
        {
            get;
            set;
        }
        public virtuall string City
        {
```

```
            get;
            set;
        }
        public virtuall string State
        {
            get;
            set;
        }
        #endregion
    }
}
```

Другой файл, PlanMyNight.Context.cs, формирует объект ObjectContext (Контекст объекта) для модели PlanMyNight.edmx. Этот объект мы будем использовать для взаимодействия с базой данных.

> **Подсказка** Шаблоны POCO будут автоматически обновлять сформированные классы соответственно внесенным в модель изменениям при сохранении файла .edmx.

Перенос классов сущностей в проект контрактов

Архитектура приложения PMN спроектирована таким образом, что слой представления гарантированно безразличен к методу хранения. Это реализовано путем перемещения контрактов и классов сущностей в сборку, которая не имеет ссылки на хранилище.

> **В Visual Studio 2008** существовала возможность расширения обработки XSD с помощью инструментов формирования кода, но это было связано с определенными сложностями и требовало обслуживания этих инструментов. EF использует шаблоны T4 для формирования и схемы базы данных, и кода. Эти шаблоны можно без труда настроить соответственно конкретным требованиям.

Шаблоны POCO ADO.NET выносят формирование классов сущностей в отдельный шаблон, что дает нам возможность с легкостью переносить эти сущности в другой проект.

Перенесем файл PlanMyNight.tt в проект PlanMyNight.Contracts. Щелкнув правой кнопкой мыши файл PlanMyNight.tt, выбираем Cut (Вырезать). Щелкаем правой кнопкой мыши папку Entities проекта PlanMyNight.Contracts и выбираем Paste (Вставить).

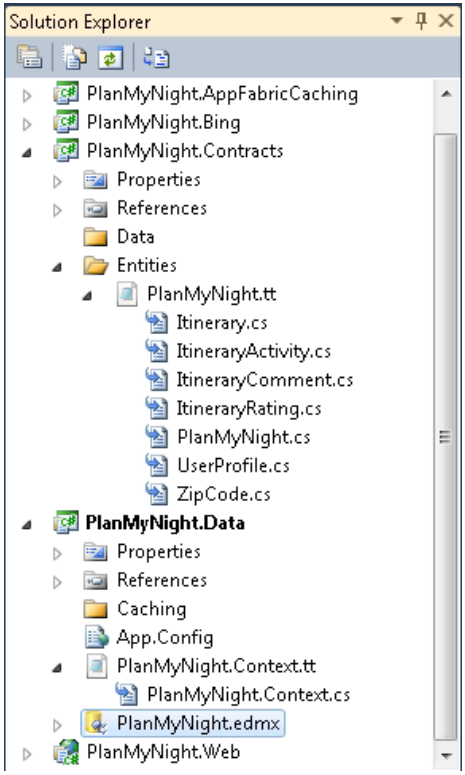


Рис. 8-28 Шаблон POCO, перенесенный в проект Contracts

Для формирования кода типа сущности шаблон PlanMyNight.tt использует метаданные модели EDM. Необходимо скорректировать относительный путь, используемый шаблоном для доступа к файлу EDMX.

Откройте шаблон PlanMyNight.tt и найдите строку:

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

Исправьте путь к файлу, чтобы он указывал на файл PlanMyNight.edmx в проекте PlanMyNight.Data:

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

Классы сущностей будут сформированы повторно, как только вы сохраните шаблон.

Также необходимо обновить шаблон PlanMyNight.Context.tt, поскольку теперь классы сущностей находятся в пространстве имен *Microsoft.Samples.PlanMyNight.Entities*, а не в *Microsoft.Samples.PlanMyNight.Data*. Откройте файл PlanMyNight.Context.tt и включите новое пространство имен в раздел директив *using*.

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

Нажав Ctrl+Shift+B, выполните сборку решения. Теперь компиляция проекта должна пройти успешно.

Сведение воедино

Теперь, когда универсальный код для взаимодействия с базой данных SQL создан, все готово для реализации специальной функциональности приложения PMN. В следующих разделах мы поэтапно рассмотрим этот процесс, вкратце остановимся на получении данных с сервисов Bing Maps и пройдем небольшой вводный курс в возможность кэширования AppFabric Windows Server, используемую в PMN.

Чтобы свести все это воедино, требуется довольно большой объем вспомогательного кода. Упростить процесс поможет использование обновленного решения, в котором уже написана большая часть кода для контрактов и сущностей, а также для работы с сервисами Bing Maps. Решение также будет включать проект PlanMyNight.Data.Test для проверки кода проекта PlanMyNight.Data.

Примечание Тестирование в Visual Studio 2010 будет рассмотрено в главе 10.

Получение данных из базы данных

В начале этой главы нами было принято решение сгруппировать операции над сущностью Itinerary в интерфейс хранилища IItinerariesRepository. Вот некоторые из этих операций:

- Поиск маршрута по действия
- Поиск маршрута по почтовому индексу
- Поиск маршрута по радиусу
- Добавление нового маршрута

Посмотрим на соответствующие методы интерфейса IItinerariesRepository:

- *SearchByActivity* обеспечит поиск маршрутов по действию и возвращение страницы данных.
- *SearchByZipCode* обеспечит поиск маршрутов по почтовому индексу и возвращение страницы данных.
- *SearchByRadius* обеспечит поиск маршрутов из определенного пункта и возвращение страницы данных.
- *Add* обеспечит добавление маршрута в базу данных.

Откроем решение PMN, располагающееся по адресу %userprofile%\Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 8\Code\Final, щелкнув двойным щелчком файл PlanMyNight.sln.

Выберем проект PlanMyNight.Data и откроем файл ItinerariesRepository.cs. Это реализация интерфейса IItinerariesRepository. Используя сформированный ранее объект контекста для работы с базой данных PlanMyNightEntities, мы можем создать запросы LINQ к модели, и EF транслирует их в обычный T-SQL, который будет применяться для работы с базой данных.

Перейдите к описанию *SearchByActivity*. Этот метод должен возвращать набор маршрутов, обозначенных как общедоступные, ID одного из действий которых соответствует заданному. Полученные в результате маршруты должны быть сортированы по полю рейтинга.

SearchByActivity реализуется с применением стандартных операторов LINQ, как показано в листинге 8-4. Добавим выделенный код в тело метода *SearchByActivity*.

Листинг 8-4 Реализация *SearchByActivity*

```
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);
    }
}
```

Примечание Разбиение результатов на страницы реализовано в методе *PageResults*.

```
private static PagingResult<Itinerary> PageResults(IQueryable<Itinera
ry> query, int page, int pageSize)
{
      int rowCount = 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;
}
```

В этот метод передается *IQueryable<Itinerary>*, что позволяет добавить разбиение на страницы в состав базового запроса. Передача *IQueryable* вместо *IEnumerable*

гарантирует, что T-SQL для запроса к хранилищу будет формироваться только при вызове *query.ToArray()*.

Метод *SearchByZipCode* аналогичен методу *SearchByActivity*, но также добавляет фильтр по почтовому индексу места выполнения действия. Опять же, поддержка LINQ упрощает реализацию, как показано в листинге 8-5. Добавим выделенный код в тело метода *SearchByZipCode*.

Листинг 8-5 Реализация *SearchByZipCode*

```
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);
    }
}
```

Функция *SearchByRadius* вызывает импортированную функцию *RetrieveItinerariesWithinArea*, которая была сопоставлена с хранимой процедурой. После этого загружаются действия для каждого найденного маршрута. Вы можете скопировать выделенный код листинга 8-6 в тело метода *SearchByRadius* в файле ItinerariesRepository.cs.

Листинг 8-6 Реализация *SearchByRadius*

```
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;

        // Хранимая процедура с выходным параметром
        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.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
        };
    }
}
```

С помощью метода *Add* маршруты добавляются в хранилище данных. Реализация этой функциональности упростилась, поскольку наш контракт и Context Object (Объект контекста) используют один и тот же объект сущности. Вставьте выделенный код листинга 8-7 в тело метода *Add*.

Листинг 8-7 Реализация *Add*

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

Готово! Мы завершили реализацию *ItinerariesRepository*, применив Context Object, сформированный дизайнером EF. Выполните все тесты решения, нажав CTRL+R, A. Все тесты, касающиеся *ItinerariesRepository*, должны быть пройдены успешно.

Разработка многопоточных приложений

С развитием многопроцессорной и многоядерной обработки данных все большее и большее значение приобретает обеспечение разработчикам возможностей для создания многопоточных приложений. Visual Studio 2010 и .NET Framework 4.0 предлагают новые пути реализации многопоточности в приложениях. Библиотека

Task Parallel Library (TPL)[6] стала частью библиотеки Base Class Library (BCL)[7] для .NET Framework. Это означает, что теперь любое .NET-приложение может использовать TPL без всяких ссылок на сборки.

Для каждого ItineraryActivity PMN сохраняет в базу данных только идентификатор действия в Bing (Bing Activity Id). Когда приходит время извлечь весь объект Bing Activity (Действие в Bing), для заполнения сущности Bing Activity данными Веб-сервиса Bing Maps используется функция, которая выбирает все ItineraryActivity текущего Itinerary.

Один из способов выполнения этой операции – последовательное обращение к сервису для каждого действия Itinerary, как показано в листинге 8-8. Эта функция, прежде чем выполнять последующий вызов RetrieveActivity() (Извлечь действие), будет ожидать завершения предыдущего вызова, что делает ее выполнение линейным по времени.

Листинг 8-8 Последовательное извлечение действий

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

В прошлом для организации многопоточной обработки этой задачи потребовалось бы использовать потоки и выполнить огромный объем работы. Теперь, с появлением TPL, для этого достаточно использовать статический метод Parallel.ForEach, который позаботиться обо всем, что связано с многопоточной обработкой, как показано в листинге 8-9.

Листинг 8-9 Параллельное извлечение действий

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

Дополнительные сведения .NET Framework 4.0 теперь включает библиотеки Parallel Linq (в System.Core.dll). PLinq представляет расширение .AsParallel() для реализации многопоточной обработки в запросах Linq. Благодаря расширениям .AsOrdered() не составляет труда инициировать обработку источника данных, как будто данные

[6] Библиотека для параллельного выполнения задач (прим. переводчика).
[7] Библиотека базовых классов (прим. переводчика).

были изначально отсортированы. Также в пространство имен System.Collections.Concurrent добавлены новые потокобезопасные коллекции. Более подробные сведения об этих новых возможностях можно найти в разделе Parallel Computing (Многопоточная обработка данных) на сайте MSDN.

Кэширование AppFabric

PMN – это управляемое данными приложение, которое получает данные из базы данных приложения и от Веб-сервисов Bing Maps. Одна из возможных сложностей при создании Веб-приложения – реализация поддержки большого числа пользователей с обеспечением необходимой производительности и времени отклика. Обращения к хранилищу данных и к сервисам для поиска действий могут значительно повысить использование ресурсов сервера для элементов, совместно используемых множеством пользователей. Например, многие пользователи имеют доступ к общедоступным маршрутам. Просмотр этих маршрутов обусловит многочисленные обращения к одним и тем же элементам в базе данных. Реализация кэширования на Веб-уровне приведет к сокращению использования ресурсов хранилища данных и снизит задержку при выполнении повторных обращений к Веб-сервисам Bing Maps. На рис. 8-29 представлена архитектура приложения, реализующего кэширование на уровне Веб-сервера.

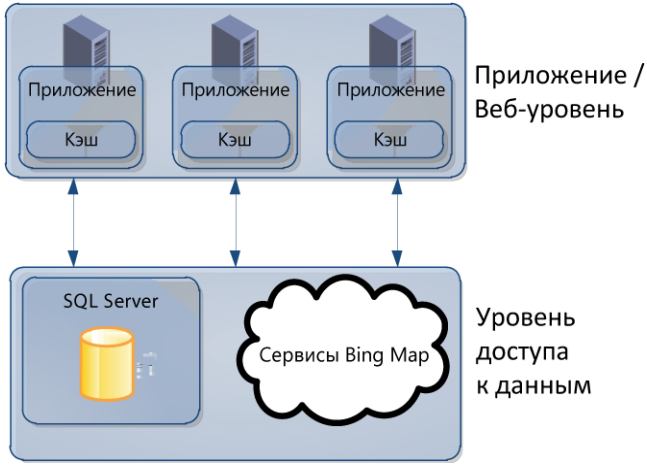


Рис. 8-29 Типовая архитектура Веб-приложения

Применение этого подхода сократит нагрузку на слой доступа к данным, но кэширование по-прежнему связано с конкретным сервером, обслуживающим запрос. Каждый сервер Веб-уровня будет иметь собственный кэш, и по-прежнему существует вероятность неравномерного распределения нагрузки между этими серверами.

Кэширование AppFabric Windows Server предлагает платформу распределенного кэширования в памяти. Клиентская библиотека AppFabric обеспечивает приложению возможность доступа к кэшу как к унифицированному событию просмотра, если кэш распределен на несколько компьютеров, как показано на рис. 8-30. API предоставляет

простые методы get и set, что позволяет без труда извлекать и сохранять сериализуемые CLR-объекты. С кэшем AppFabric вы можете добавлять компьютер для размещения кэша по мере надобности, т.е. масштабирование становится прозрачным для клиента. Другое преимущество – кэш может также разделять копии данных между кластерами, обеспечивая высокую доступность данных даже в случае сбоев.

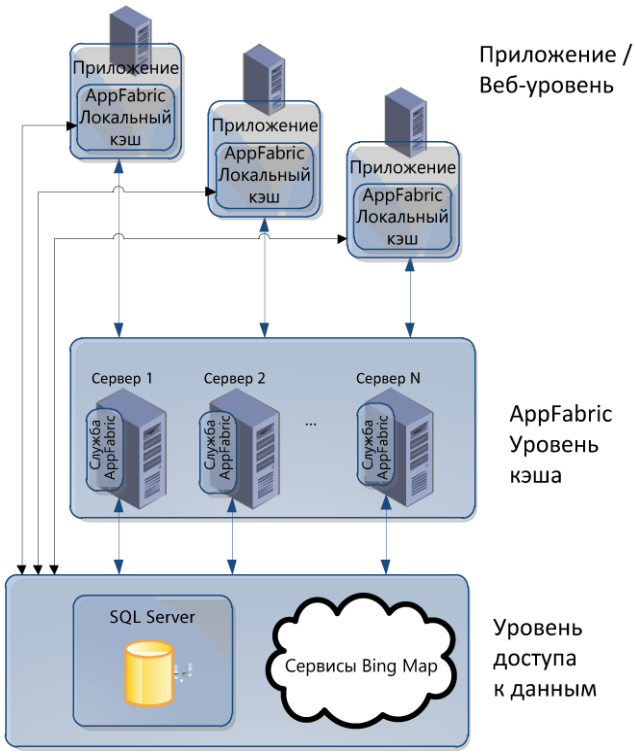


Рис. 8-30 Веб-приложение, использующее кэширование AppFabric Windows Server

Дополнительные сведения Кэширование AppFabric Windows Server предлагается как набор расширений .NET Framework 4.0. Подробные сведения о том, как найти, установить и настроить Windows Server AppFabric, можно найти на сайте Windows Server AppFabric.

Дополнительные сведения PMN может быть конфигурирован на использование либо кэширования ASP.NET, либо кэширования AppFabric Windows Server. Подробное пошаговое руководство с описанием того, как добавить кэширование AppFabric Windows Server в PMN предлагается в разделе PMN: Adding Caching using Velocity (Добавление кэширования с использованием Velocity)

Заключение

В данной главе мы рассмотрели ряд новых возможностей Visual Studio 2010 для структурирования слоя доступа к данным приложения PlanMyNight и использовали Entity Framework v4.0 для организации доступа к базе данных. Также были представлены автоматическое формирование сущностей с помощью POCO-шаблонов ADO.NET Entity Framework и расширения кэширования AppFabric Windows Server. В следующей главе мы займемся созданием замечательных Веб-приложений с помощью инфраструктуры ASP.NET MVC и Managed Extensibility Framework (Инфраструктура расширения возможностей).

Глава 9
От 2008 к 2010: проектирование восприятия и поведения

В данной главе рассматривается

- Создание контроллера ASP.NET MVC, взаимодействующего с моделью данных
- Создание представления ASP.NET MVC для отображения данных контроллера и проверки пользовательского ввода
- Расширение приложения внешним подключаемым модулем с помощью Managed Extensibility Framework

За годы, прошедшие с момента выхода ASP.NET 1.0, разработка Веб-приложений в Microsoft Visual Studio, безусловно, существенно улучшилась. В Visual Studio 2008 появилась официальная поддержка AJAX, Language Integrated Query (LINQ)[1] и многие другие дополнения для Веб-страниц, благодаря которым разработчики получили возможность создавать эффективные простые в обслуживании приложения.

Жажда улучшений, призванных помочь разработчикам в создании высококлассных приложений, не иссякла и в Visual Studio 2010. В этой главе будут рассмотрены некоторые новые возможности на примере расширения функциональности используемого нами для примера приложения Plan My Night.

> **Примечание** Приложение-пример – это проект ASP.NET MVC 2, но в Visual Studio 2010 у разработчика есть выбор использовать либо новую форму приложения ASP.NET, либо более традиционную ASP.NET (которую в сообществе разработчиков *для отличия называют Веб-формами* (Web Forms)). ASP.NET 4.0 была существенно дополнена, чтобы облегчить жизнь разработчиков и при этом обеспечить крайне эффективный подход к созданию Веб-приложений.

В данной главе будет использоваться модифицированная форма решения приложения-примера. Если сопроводительные материалы данной книги установлены в каталог по умолчанию, интересующее нас решение находится в Documents\Microsoft Press\Moving to Visual Studio 2010\Chapter 9\UserInterfaceStart.

Введение в проект PlanMyNight.Web

> **Примечание** Инфраструктура ASP.NET MVC 1.0 Framework предлагается как расширение Visual Studio 2008. Однако данная глава написана, исходя из предположения о том, что у пользователя имеется стандартная установка Visual Studio 2008, которая поддерживает только проекты ASP.NET Web Forms 3.5.

Пользовательский интерфейс Plan My Night в Visual Studio 2010 реализован как ASP.NET MVC-приложение, компоновка которого отличается от того, к чему, возможно,

[1] Язык интегрированных запросов (*прим. переводчика*).

привыкли разработчики, создавая приложения ASP.NET Web Forms в Visual Studio 2008. Некоторые элементы проекта (как видно на рис. 9-1) будут знакомыми (такие как Global.asax), но все остальные абсолютно новые. Некоторые из них являются обязательными для инфраструктуры ASP.NET MVC.

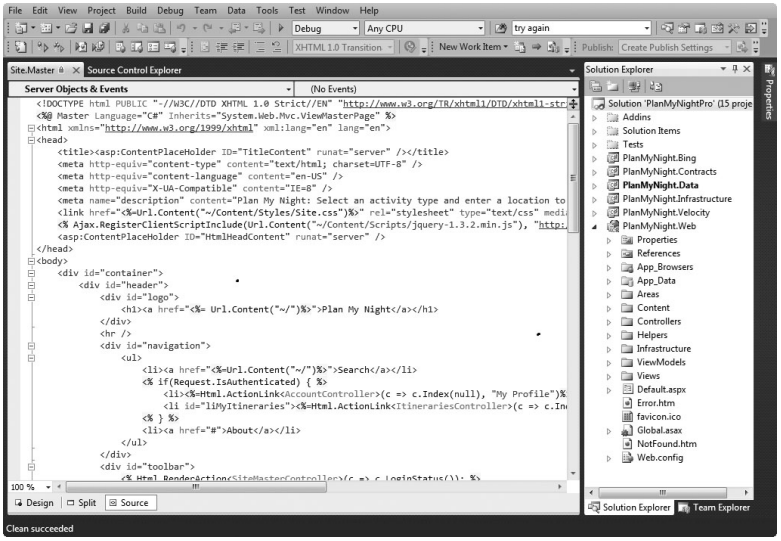


Рис. 9-1 Внешний вид проекта PlanMyNight.Web

Обязательные элементы ASP.NET MVC:

- **Areas (Области)** Эта папка используется инфраструктурой ASP.NET MVC для организации больших Веб-приложений в небольшие компоненты без разделения решений или проектов. Эта возможность не используется в приложении Plan My Night, но представлена здесь, потому что данная папка создается шаблоном MVC-проекта.
- **Controllers (Контроллеры)** Во время обработки запроса инфраструктура ASP.NET MVC ищет контроллеры для обработки запроса в этой папке.
- **Views (Представления)** Папка Views на самом деле является структурой каталогов. Его подпапки носят имена соответствующие классам в папке Controllers. Также имеется подпапка Shared (Общие). Она предназначена для представлений, частичных представлений, главных страниц и любых других ресурсов, которые должны быть доступны всем котроллерам.

Дополнительные сведения Более подробные сведения о компонентах ASP.NET MVC, а также отличия обработки запросов от того, как это происходит в ASP.NET Web Forms, представлены по адресу http://asp.net/mvc.

В большинстве случаев web.config является последним файлом в корневой папке проекта. В Visual Studio 2010 для этого файла предлагается существенное обновление: преобразование Web.config. Эта возможность позволяет создавать на базе основного web.config файлы web.config для конкретных сборок, переопределяющие настройки

базового файла во время сборки, развертывания и выполнения. Эти файлы отображаются под базовым web.config, как показано на рис. 9-2.

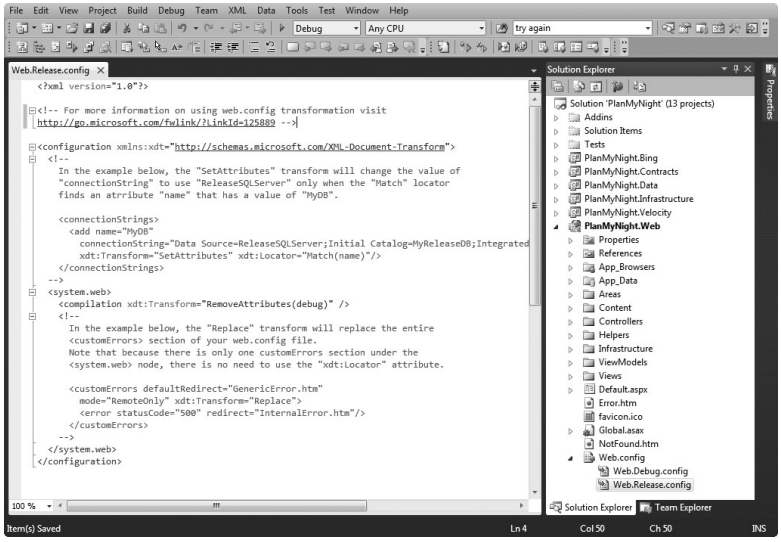


Рис. 9-2 Файл Web.config и специальные файлы конфигурации для конкретных сборок

> В Visual Studio 2008 приходилось всегда помнить о том, что не надо переопределять параметры отладки в web.config. Или вносить в web.config правильные настройки после его публикации для окончательной сборки. В Visual Studio 2010 таких проблем нет. Для переопределения значений в web.config при окончательных сборках используется набор параметров файла web.config.retail; при отладочных сборках используется файл web.config.debug.

Проект также включает разделы:

- **Content (Содержимое)** Набор папок, содержащих изображения, сценарии и файлы стилей.
- **Helpers (Вспомогательные классы)** Различные классы, включающие ряд методов расширения, которые расширяют функциональность типов, используемых в проекте.
- **Infrastructure (Инфраструктура)** В данную папку включены элементы, обеспечивающие взаимодействие с более низкоуровневой инфраструктурой ASP.NET MVC (например, фабрики кэширования и контроллеров).
- **ViewModels (Модели представлений)** Объекты данных, заполненные классами контроллеров (Controller) и используемые представлениями (Views) для отображения данных.

Запуск проекта

Если скомпилировать и запустить проект, на экране должно быть выведено следующее (рис. 9-3).

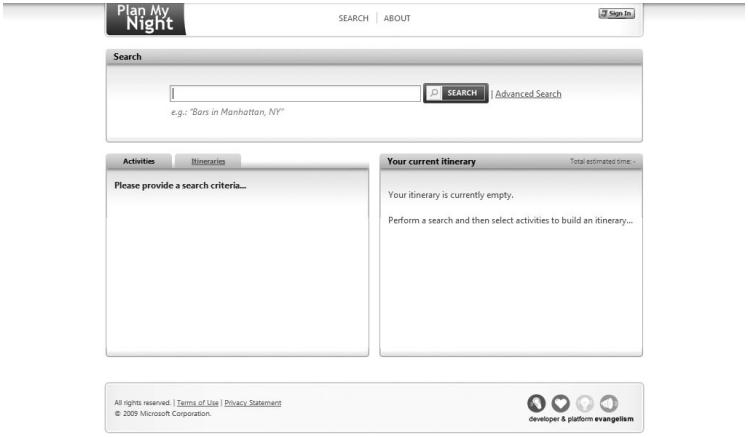


Рис. 9-3 Страница по умолчанию приложения Plan My Night

Поиск и организация исходного списка элементов маршрута функционируют, но если попытаться сохранить разрабатываемый маршрут или выполнить вход с использованием Windows Live ID, приложение возвратит ошибку 404 Not Found (Не найден) (как показано на рис. 9-4).

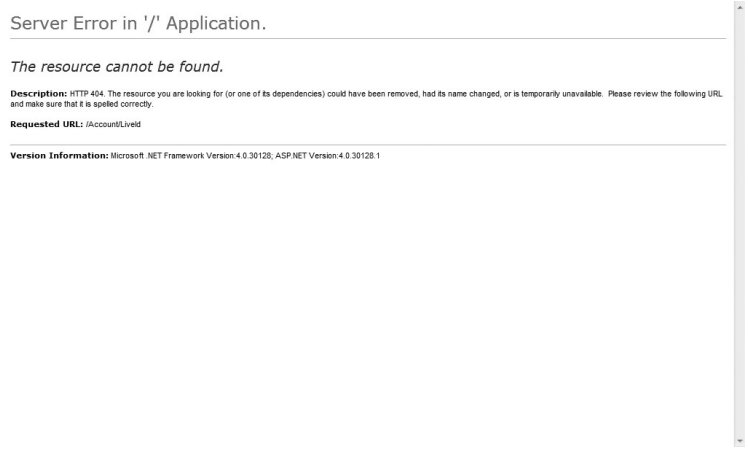


Рис. 9-4 Ошибка, возвращаемая приложением Plan My Night при попытке регистрации

Это происходит потому, что на данный момент в проекте нет контроллера учетной записи для обработки таких запросов.

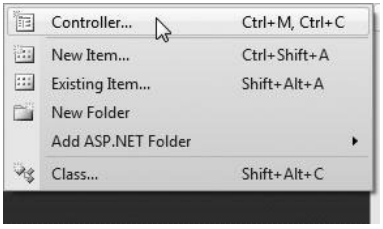
Создание контроллера учетной записи

Класс *AccountController* (Контроллер учетной записи) обеспечивает приложение Plan My Night критически важными функциями:

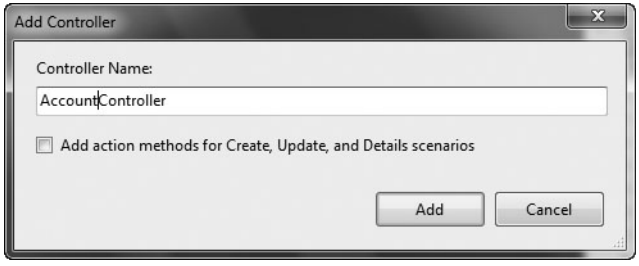
- Он обрабатывает вход и выход пользователей из приложения (с использованием Windows Live ID).
- Обеспечивает действия для отображения и обновления данных профиля пользователя.

Для создания нового контроллера ASP.NET MVC:

1. В Solution Explorer (Обозреватель решений) перейдите к папке Controllers проекта PlanMyNight.Web и щелкните ее правой кнопкой мыши.
2. Откройте подменю Add и выберите пункт Controller.



3. Введите имя контроллера, **AccountController**.



Примечание Не устанавливайте флажок Add Action Methods For Create, Update, And Delete Scenarios (Добавить методы для сценариев создания, обновления и удаления). Установка этого флажка обеспечит вставку некоторых методов-«заглушек», но поскольку мы не будем использовать методы по умолчанию, создавать их нет необходимости.

По щелчку кнопки Add в диалоговом окне Add Controller откроется базовый класс *AccountController* с единственным методом *Index ()*:

```
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 Следует отметить отличие от разработки приложений ASP.NET Web Forms в Visual Studio 2008: в приложениях ASP.NET MVC для файлов .aspx не создаются файлы выделенного кода. Котроллеры, такие как мы только что рассмотрели, осуществляют логику, необходимую для обработки ввода и подготовки вывода. Такой подход обеспечивает четкое разделение логики отображения и бизнес-логики, и это является ключевым аспектом ASP.NET MVC.

Реализация функциональности

Для взаимодействия с любым слоем доступа к данным и сервисами (Моделью) потребуется создать и инициализировать некоторые поля экземпляров. Но перед этим следует добавить ряд пространств имен в блок директив *using*:

```
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 WindowsLiveId;
```

Вот теперь добавим поля экземпляров. Эти поля являются интерфейсами различных разделов Модели:

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

Примечание Использование интерфейсов для взаимодействия со всеми внешними зависимостями обеспечивает лучшую портируемость кода на различные платформы. Также интерфейсы более эффективно изолируют отдельные компоненты, что облегчает моделирование зависимостей при тестировании.

Как уже упоминалось, эти поля представляют части Модели, с которой будет взаимодействовать данный контроллер для реализации функциональных требований. Рассмотрим общие описания каждого из этих интерфейсов:

- **IWindowsLiveLogin (Регистрация WindowsLive)** Обеспечивает функциональность для взаимодействия с сервисом Windows Live ID.
- **IMembershipService (Сервис членства)** Обеспечивает данные профиля пользователя и методы авторизации. В используемом приложении-примере это абстракция ASP.NET Membership Service (Сервис членства).
- **IFormsAuthentication (Аутентификация с помощью форм)** Абстракция ASP.NET Forms Authentication (Аутентификация с помощью форм).
- **IReferenceRepository (Хранилище справочных ресурсов)** Обеспечивает справочные ресурсы, такие как списки состояний и другие характерные для модели данные.
- **IActivitiesRepository (Хранилище действий)** Интерфейс для извлечения и обновления данных действий.

В этот класс добавим два конструктора: один – общий для времени выполнения, который использует класс *ServiceFactory* (Фабрика сервисов) для получения ссылок на необходимые интерфейсы, и другой – для обеспечения возможности вводить конкретные экземпляры интерфейсов при тестировании.

```
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;
}
```

Аутентификация пользователя

Первой настоящей функциональностью, реализованной нами в этом контроллере, будет вход и выход из приложения. Большинство методов, которые будут реализовываться в дальнейшем, требуют аутентификации, поэтому именно с нее мы и начнем.

В нашем приложении для обеспечения аутентификации используется одновременно несколько технологий: Windows Live ID, ASP.NET Forms Authentication и ASP.NET Membership Services. Эти три технологии применяются в действии LiveID, которое будет реализовано следующим.

Начнем с создания в классе *AccountController* следующего метода:

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

Это основной метод для взаимодействия с сервисами Windows Live ID. Если вызвать его прямо сейчас, он просто перенаправит пользователя к корню приложения.

> **Примечание** Вызов *Redirect* (Перенаправить) возвращает *RedirectResult* (Результат перенаправления). В данном примере для определения цели перенаправления используется строка, но в разных ситуациях могут использоваться различные перегрузки этого метода.

Когда Windows Live ID возвращает пользователя в приложение, может быть предпринято несколько разных типов действий. Пользователь может выполнять вход в Windows Live ID, выполнять выход или очищать «cookies» Windows Live ID. Когда Windows Live ID возвращает пользователя, в его URL присутствует строковый параметр запроса *action*, поэтому переключение ветвей логики выполняется на основании значения этого параметра.

Добавим следующий код в метод *LiveId*:

```
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);
}
```

Дополнительные сведения Полную документацию по системе Windows Live ID можно найти по адресу http://dev.live.com/.

В только что добавленном коде обрабатываются два действия выхода для Windows Live ID. В обоих случаях используется интерфейс *IFormsAuthentication* для удаления файла «cookie» ASP.NET Forms Authentication, чтобы все будущие http-запросы (до момента повторной регистрации пользователя) не считались аутентифицированными. Во втором случае выполняется на одну операцию больше, и очищаются файлы «cookies» Windows

Live ID (те, в которых сохраняется регистрационное имя, но не пароль).

Сценарий входа включает немного больший объем кода, поскольку требуется проверка присутствия аутентифицируемого пользователя в базе данных сервиса членства (Membership Database). Если этого пользователя там нет, для него создается новый профиль. Однако перед этим должны быть переданы данные, которые Windows Live ID переслал в интерфейс Windows Live ID, что позволит проверить их и предоставить объект *WindowsLiveLogin.User*:

```
default:
       // вход
       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);
```

На данном этапе, если пользователь выполнит вход, *liveIdUser* будет либо ссылаться на аутентифицированный объект *WindowsLiveLogin.User*, либо будет null. Исходя из этого, добавим следующий раздел кода, который обеспечивает выполнение некоторых действий, если значение *liveIdUser* не 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;
```

Вызов метода *ValidateUser* (Проверить пользователя) в *IMembershipService* позволяет приложению проверить, посещал ли данный пользователь этот сайт ранее и существует ли его профиль. Поскольку аутентификация пользователя выполняется по Windows Live ID, значение его ID (которое является GUID) используется и как имя пользователя, и как пароль для ASP.NET Membership Service.

Если в приложении нет записи для данного пользователя, она создается путем вызова метода *CreateUser* (Добавить пользователя). Затем посредством *CreateProfile* (Создать профиль) создается и профиль параметров пользователя. Профиль заполняется некоторыми значениями по умолчанию и сохраняется в хранилище. Пользователь перенаправляется на основную страницу ввода, где может обновить сведения.

> **Примечание** На основании сочетания входных параметров *Controller.RedirectToAction* определяет, какой URL должен быть создан. В данном случае требуется перенаправить пользователя к действию *Index* (Корень) данного контроллера и передать текущее значение URL-адреса возврата.

Также в данном коде реализована регистрация пользователя в сервисе аутентификации ASP.NET Forms, т.е. обеспечивается создание файла «cookie» с идентификационными данными для использования в запросах, требующих аутентификации, в будущем.

Параметры профиля также управляются сервисами ASP.NET Membership Services и объявляются в файле web.config приложения:

```
<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>
```

Теперь метод *LiveID* полностью готов и должен выглядеть, как следующий фрагмент кода. Приложение принимает сведения для аутентификации от Windows Live ID, подготавливает профиль ASP.NET Membership Service и создает маркер аутентификации ASP.NET Forms.

```
public ActionResult LiveId()
{
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:
       // вход
       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("~/");
}
```

Конечно, пользователь должен иметь возможность перед регистрацией попасть на страницу входа Windows Live ID. На данный момент в приложении Plan My Night имеется кнопка входа с помощью Windows Live ID. Но возможны ситуации, когда понадобится перенаправить пользователя на страницу входа прямо из кода. Для реализации такого сценария добавим в контроллер небольшой метод под именем *Login* (Вход):

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

Данный метод просто извлекает URL-адрес входа для Windows Live и перенаправляет пользователя по этому адресу. Это также удовлетворяет конфигурации для аутентификации ASP.NET Forms в web.config с той точки зрения, что любой запрос, требующий аутентификации, будет перенаправлен в этот метод:

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

Извлечение профиля для текущего пользователя

Теперь, когда методы аутентификации описаны, что реализует первую цель данного контролера – обработка входа и выхода пользователей из приложения – можно переходить к извлечению данных текущего пользователя.

Метод *Index*, который является методом по умолчанию для контроллера, будет располагаться там, где должны извлекаться данные текущего пользователя и возвращаться представление, отображающее эти данные. Метод *Index*, изначально описывавшийся при создании класса *AccountController*, необходимо заменить следующим:

```
[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);
}
```

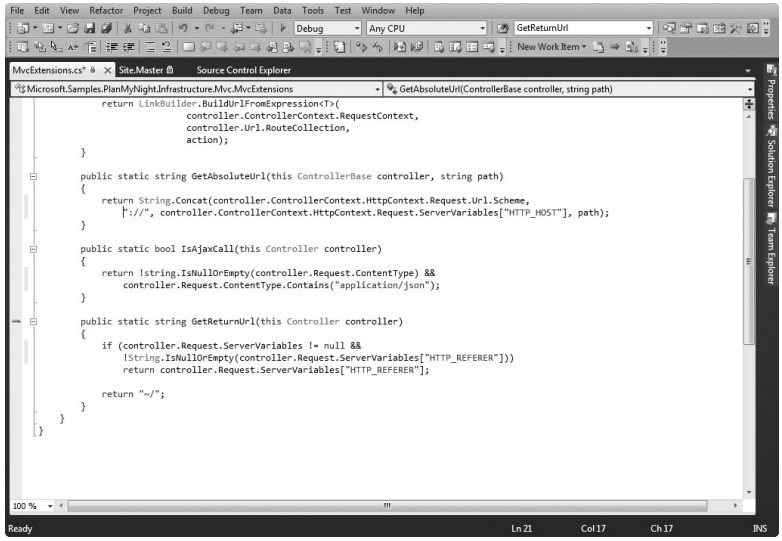
Visual Studio 2008 Атрибуты не имели такого широкого распространения в Visual Studio 2008, но в ASP.NET MVC они часто используются. Атрибуты позволяют описывать метаданные цели, к которой они относятся. Благодаря этому появляется возможность проверять данные во время выполнения (через отражение) и предпринимать действия в случае необходимости.

Очень удобен атрибут *Authorize* (Авторизовать). Он показывает, что данный метод может вызываться только для уже аутентифицированных http-запросов. Если запрос не аутентифицирован, он будет перенаправлен в заданную цель входа ASP.NET Forms Authentication, которая только что была настроена. Атрибут *AcceptVerbs* (Допустимые команды) также ограничивает способы вызова этого метода, задавая допустимые Http-команды. В данном случае этот метод вызывается по запросу HTTP GET. В сигнатуру метода добавлен строковый параметр *returnUrl*. Он обеспечивает возможность возвращения пользователя на исходный адрес по завершении просмотра или обновления его сведений.

Примечание Это приводит нас к части инфраструктуры ASP.NET MVC под названием *Привязка модели* (*Model Binding*), подробное рассмотрение которой выходит за рамки данной книги. Однако необходимо знать, что она пытается найти источник *returnUrl* (поле формы, данные таблицы маршрутизации или параметр строки запроса с таким же именем) и выполняет его привязку к этому значению при вызове данного метода. Если средству привязки модели не удается найти подходящего источника, значение будет null. Такое поведение может обусловить проблемы для

типов значений, которые не могут быть null, потому что приведет к формированию исключения *InvalidOperationException* (Недопустимая операция).

В целом, этот метод прост: он принимает возвращаемое значение метода *GetCurrentProfile* (Получить профиль текущего пользователя) интерфейса ASP.NET Membership Service и настраивает объект модели представления, который будет использоваться представлением. Вызов *GetReturnUrl* (Получить URL возврата) – это пример метода расширения, описанного в проекте PlanMyNight.Infrastructure. Он не является членом класса *Controller*, но в среде разработки обеспечивает намного более надежный код.



InjectStatesAndActivityTypes (Ввести состояния и типы действий) – метод, который требуется реализовать. Он собирает данные имен состояния из хранилища справочных ресурсов и данные о типах действий из хранилища действий. Он создает две коллекции *SelectListItem* (HTML-класс для MVC): одну для списка состояний, и другую для списка разных типов действий, доступных в приложении. Также он задает соответствующее значение.

```
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;
}
```

Обновление данных профиля

Закончив с инфраструктурой для извлечения данных текущего профиля, перейдем к обновлению данных модели посредством формы, передаваемой пользователем. После этого можно создавать собственные страницы представления и видеть, как все это сочетается. Метод *Update* прост, однако, в нем появляются некоторые новые до сих пор не встречавшиеся нам возможности:

```
[Authorize()]
[AcceptVerbs(HttpVerbs.Post)]
[ValidateAntiForgeryToken()]
public ActionResult Update(UserProfile profile)
{
    var returnUrl = Request.Form["returnUrl"];
    if (!ModelState.IsValid)
    {
        // ошибка проверки
        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 });
    }
}
```

Атрибут *ValidateAntiForgeryToken* (Проверить маркер, препятствующий фальсификации) подтверждает то, что форма не была сфальсифицирована. Для использования этой возможности необходимо добавить *AntiForgeryToken* (Маркер, препятствующий фальсификации) в форму ввода представления. Проверка действительности *ModelState* (Состояние модели) – наш первый опыт проверки ввода. Это проверка на стороне сервера, и ASP.NET MVC предлагает простую в использовании возможность убедиться в том, что поступающие данные удовлетворяют некоторым правилам. Одно из свойств объекта *UserProfile*, созданного для обеспечения ввода в этот метод через привязку модели MVC, имеет атрибут *System.ComponentModel.DataAnnotations.Required*. При привязке модели инфраструктура MVC проверяет атрибуты *DataAnnotation* и объявляет *ModelState* действительным только в случае выполнения всех правил.

В случае, когда *ModelState* недействительный, пользователь перенаправляется в метод *Index*, где *ModelState* используется для отображения формы ввода. Или, если запрос был AJAX-вызовом, возвращается *JsonResult* с прикрепленными к нему данными *ModelState*.

> **Visual Studio 2008** В ASP.NET MVC запросы маршрутизируются не через страницы, а через контроллеры. Поэтому один URL может обрабатывать множество разных запросов и отвечать на них соответствующим представлением. В Visual Studio 2008 для реализации этой функциональности разработчику пришлось бы создавать два разных URL и вызывать метод в третьем классе.

Если *ModelState* действительный, сервис членства обновляет профиль, и для AJAX-запросов возвращается JSON-результат со сведениями об успешности операции. Для «обычных» запросов пользователь перенаправляется к действию *UpdateSuccess* (Успешное обновление) контроллера Account. Метод *UpdateSuccess* – последний метод, необходимый для завершения выполнения данного контроллера:

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

Этот метод используется для возвращения представления успешного выполнения в браузер, отображения некоторых обновленных данных и обеспечения ссылки для возвращения пользователя туда, где он находился перед началом процесса обновления профиля.

Теперь, когда контроллер Account полностью реализован, полученный класс должен выглядеть следующим образом:

```
namespace Microsoft.Samples.PlanMyNight.Web.Controllers
{
    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;

    [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:
                    // вход
                    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)
            {
                // ошибка проверки
                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;
        }
    }
}
```

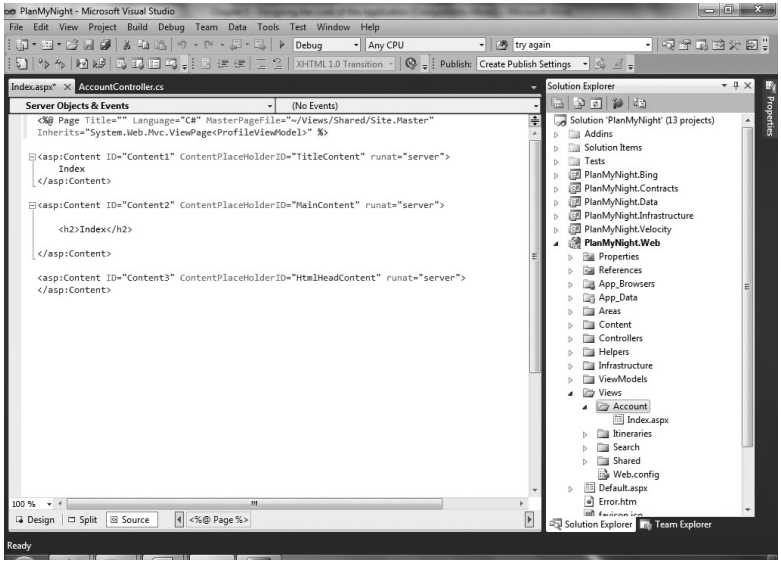
Создание представления учетной записи

В предыдущем разделе был создан контроллер, реализующий функциональность для обновления и просмотра данных пользователя. В этом разделе мы поэтапно рассмотрим возможности Visual Studio 2010, позволяющие создавать представления, которые будут отображать эту функциональность пользователю.

Чтобы создать представление Index контроллера Account:

1. Перейдем к папке Views проекта PlanMyNight.Web.
2. Щелкнем папку Views правой кнопкой мыши, раскроем подменю Add и выберем New Folder.
3. Назовем новую папку **Account**.
4. Щелкнем правой кнопкой мыши новую папку Account, раскроем подменю Add и выберем View.
5. Заполним поля диалогового окна Add View (Добавить представление), как показано на рисунке:

6. Щелкнем OK. Мы должны получить HTML-страницу с несколькими элементами управления *<asp:Content>*:



Можно заметить, здесь нет особых отличий от того, что мы привыкли видеть в Visual Studio 2008. По умолчанию ASP.NET MVC 2 использует механизм формирования представлений ASP.NET Web Forms, поэтому страницы MVC и Web Forms будет иметь некоторое сходство. Основное отличие на этом этапе в том, что класс *page* наследуется от *System.Web.Mvc.ViewPage<ProfileViewModel>*, и отсутствует файл выделенного кода. MVC не использует файлы выделенного кода, в отличие от ASP.NET Web Forms, чтобы обеспечить четкое разделение функциональных областей. Редактирование страниц MVC, как правило, осуществляется через разметку. Конструктор применяется преимущественно для приложений ASP.NET Web Forms.

Чтобы этот каркас страницы стал основным представлением контроллера Account, необходимо изменить содержимое заголовка, обеспечив его единообразие с остальными представлениями:

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

Далее требуется добавить клиентские сценарии в *HtmlHeadContent* (Содержимое HTML-заголовка):

```
<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>
```

Этот сценарий использует метод расширения *System.Web.Mvc.AjaxHelper*, описанный в проекте PlanMyNight.Infrastructure в папке MVC.

Когда содержимое заголовка настроено, можно заняться основным содержимым представления:

```
<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>
```

Если не обращать внимания на некоторый встроенный код, все это выглядит как обычная HTML-разметка. Рассмотрим фрагменты встроенного кода и продемонстрируем, какую мощь они обеспечивают (и простоту при этом).

> **Visual Studio 2008** В Visual Studio 2008 более типичным для отображения данных было использовать серверные элементы управления и логику времени отображения. Но поскольку страницы представлений ASP.NET MVC не имеют файла выделенного кода, эта логика должна быть описана в одном файле с разметкой. По-прежнему могут применяться элементы управления ASP.NET Web Forms. В нашем примере используется элемент управления <*asp:Content*>. Однако, как правило, функциональность элементов управления ASP.NET Web Forms ограничена из-за отсутствия файла выделенного кода.

В MVC широко применяются вспомогательные классы HTML. Методы, содержащиеся в *System.Web.Mvc.HtmlHelper*, генерируют компактные соответствующие стандартам HTML-теги для использования в различных целях. Для этого MVC-разработчику в некоторых случаях приходится писать больше кода разметки, чем разработчику Web Forms, но он получает непосредственный контроль над формированием вывода. Строго типизированная версия этого класса расширения (*HtmlHelper<TModel>*) может использоваться в разметке представления через свойство *ViewPage<TModel>.Html*.

В данной форме используются такие *HTML*-методы (и это лишь часть того, что доступно по умолчанию):

- *Html.EnableClientValidation()* (Включить проверку на стороне клиента) обеспечивает проверку данных на стороне клиента на основании строго типизированного словаря ModelState.
- *Html.BeginForm* (Начать форму) помещает в разметку тег <form> и закрывает форму в конце раздела директив *using*. Принимает различные параметры, но наиболее часто используемыми являются имя действия и контроллер, для которого это действие должно быть вызвано. Благодаря этому инфраструктура MVC может автоматически формировать URL конкретной формы во время выполнения, что избавляет от необходимости вводить строку URL в разметку.
- *Html.AntiForgeryToken* размещает в форме скрытое поле с проверочным значением, которое также сохраняется на стороне сервера и проверяется, если цель формы имеет атрибут *ValidateAntiForgeryToken*. Мы добавили этот атрибут в методе *Update* контроллера.

- *Html.EditorFor* (Редактор для) – перегруженный метод, обеспечивающий вставку текстового поля в разметку. Это строго типизированная версия метода *Html.Editor*.
- *Html.DropDownListFor* (Раскрывающийся список для) – перегруженный метод, обеспечивающий вставку раскрывающегося списка в разметку. Это строго типизированная версия метода *Html.DropDownList*.
- *Html.ValidationMessage* (Сообщение проверки) – вспомогательный метод, который обеспечит вывод на экран сообщения об ошибке проверки, если заданный ключ уже имеется в словаре ModelState.
- *Html.Hidden* (Скрытый) помещает в форму скрытое поле с переданными именем и значением.
- *Html.SubmitButton* (Кнопка Передать) создает в форме кнопку Submit (Передать).

Примечание Когда разметка представления Index готова, остается лишь добавить представления для действия *UpdateSuccess* , и можно будет просматривать результаты.

Чтобы создать представление UpdateSuccess:

1. Раскройте проект PlanMyNight.Web в Solution Explorer и затем раскройте папку Views.
2. Щелкните правой кнопкой мыши папку Account.
3. Откройте подменю Add и щелкните View.
4. Заполните поля диалогового окна Add View, чтобы оно выглядело следующим образом:

После создания страницы представления заполните содержимое заголовка следующим образом:

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

И содержимое *MainContent* должно выглядеть так:

```
<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>
```

Теперь, когда это последнее представление создано, можно откомпилировать и выполнить приложение. Щелкните кнопку Sign In (Вход), которая показана в верхнем правом углу формы на рис. 9-6, и выполните вход с помощью Windows Live ID.

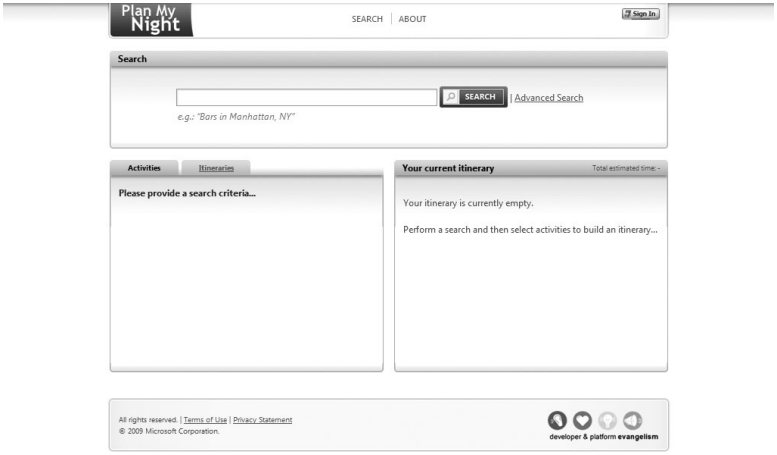


Рис. 9-6 Окно по умолчанию Plan My Night

После того, как вход выполнен, вы должны быть перенаправлены к созданному представлению Index контроллера Account (рис. 9-7).

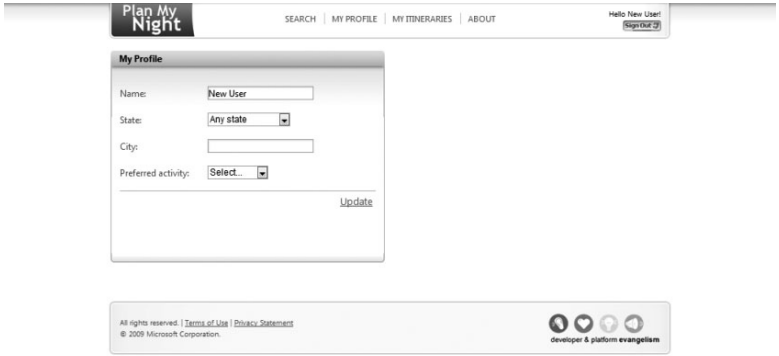


Рис. 9-7 Окно настроек профиля, возвращенное методом Index контроллера Account

Если вместо этого вы оказались на странице поиска, просто щелкните ссылку My Profile (Мой профиль), которая находится в центре вверху интерфейса рядом с остальными ссылками. Чтобы увидеть новые возможности проверки данных в действии, попробуйте сохранить форму, не заполнив поле Name. На экране должно появиться следующее сообщение (рис. 9-8).

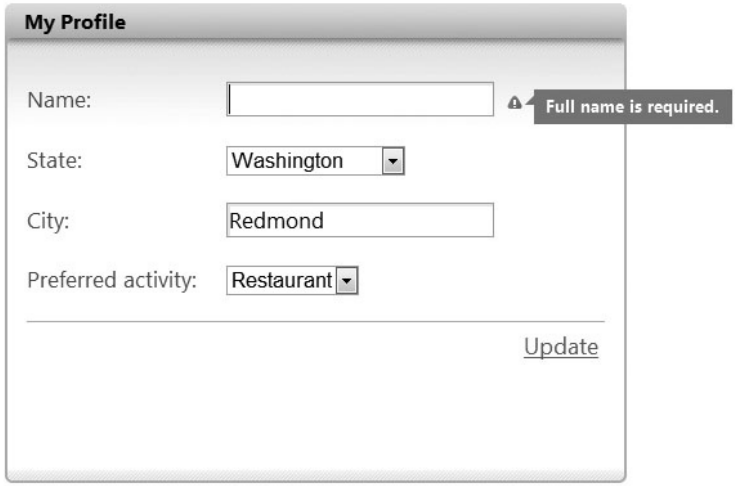


Рис. 9-8 Пример сбоя при проведении проверки Model Binding

Поскольку в приложении включена проверка на стороне клиента, обратного вызова не было. Чтобы увидеть, как работает проверка на стороне сервера, потребовалось бы внести изменения в файл Index.aspx в папке Account, закомментировав вызов *Html.EnableClientValidation*. В приложениях MVC благодаря тесной интеграции и поддержке

AJAX и JavaScript намного упрощается перенос на сторону клиента таких серверных операций, как проверка, по сравнению с тем, как это было ранее.

> **Visual Studio 2008** В приложениях ASP.NET MVC значение атрибута ID отдельного HTML-элемента не преобразовывается, как это происходит в ASP.NET Web Forms 3.5. В Visual Studio 2008 разработчик должен был сохранять *UniqueID* элемента управления/элемента в переменной JavaScript, чтобы обеспечить возможность доступа к нему внешнего JavaScript. Это делалось, чтобы гарантировать уникальность ID, но всегда вводило дополнительный уровень сложности во взаимодействие между элементами управления ASP.NET 3.5 Web Forms и JavaScript. В MVC такого преобразования нет, но обеспечение уникальности ID – сфера ответственности разработчиков. Также следует отметить, что теперь ASP.NET 4.0 Web Forms поддерживает отключение преобразования ID на уровне элемента управления по желанию разработчика.

Контроллер Account и связанные представления и являются той недостающей «базовой» функциональностью Plan My Night. В ходе работы над ними мы рассмотрели некоторые новые возможности приложений Visual Studio 2010 и MVC 2.0. Но MVC не единственный доступный выбор для Веб-разработчиков. ASP.NET Web Forms являются основным типом приложений ASP.NET с момента ее создания, и были улучшены в Visual Studio 2010. В следующем разделе Веб-форма ASP.NET для приложения MVC будет создана в визуальном дизайнере.

Использование дизайнера для создания Веб-формы

Все приложения рано или поздно сталкиваются с непредвиденными условиями, и наше приложение-пример не исключение. При возникновении непредвиденной ситуации оно возвращает сообщение об ошибке, такое как показано на рис. 9-9.

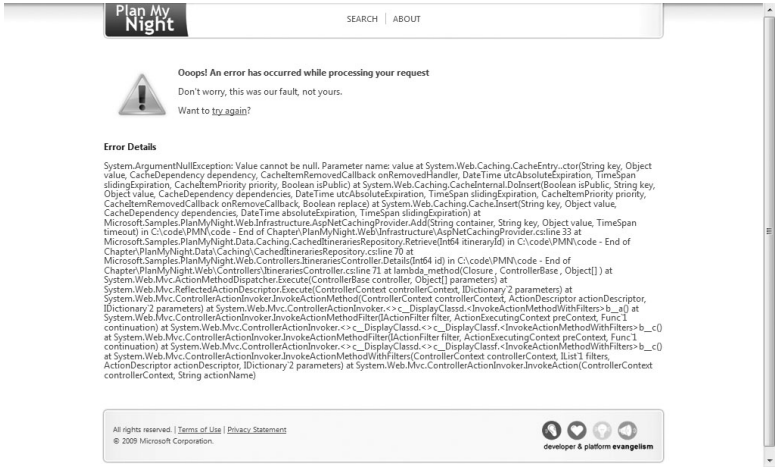
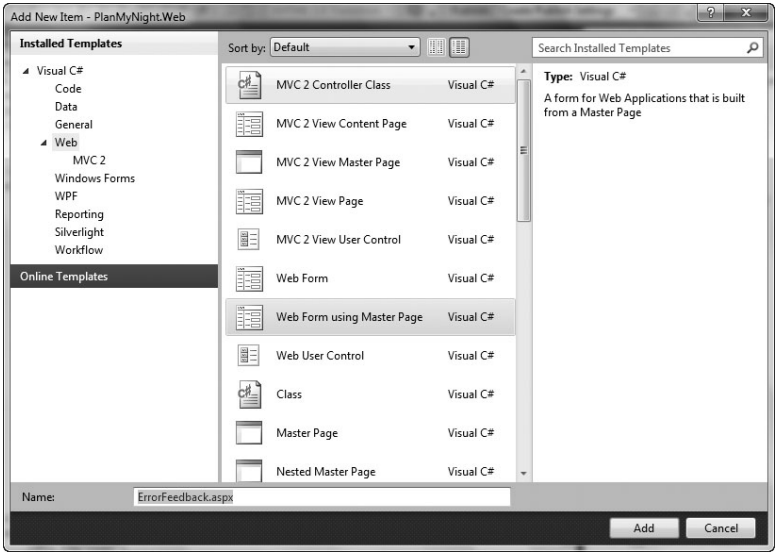


Рис. 9-9 Пример сообщения об ошибке приложения Plan My Night

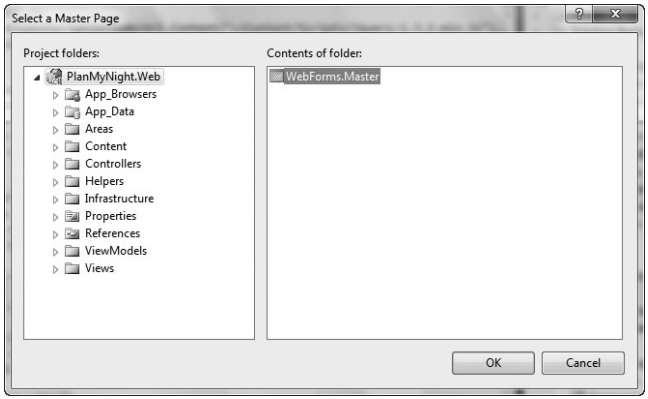
В настоящее время у пользователя, получившего такое сообщение, небогатый выбор: либо попытаться повторить это действие, либо использовать навигационные ссылки в верхней части окна приложения. (Конечно, это может привести к еще одной ошибке.) Предоставление пользователю возможности обратной связи позволит разработчикам получать сведения о ситуации, которых могут не обеспечивать стандартное сообщение об ошибке и трассировка стека. Чтобы рассмотреть другой способ создания компонента пользовательского интерфейса для приложения Plan My Night, спроектируем страницу обратной связи в случае возникновения ошибки как Веб-форму ASP.NET, преимущественно используя для этого дизайнер Visual Studio. Прежде чем приступать к дизайну формы, необходимо создать файл формы.

Чтобы создать новую Веб-форму:

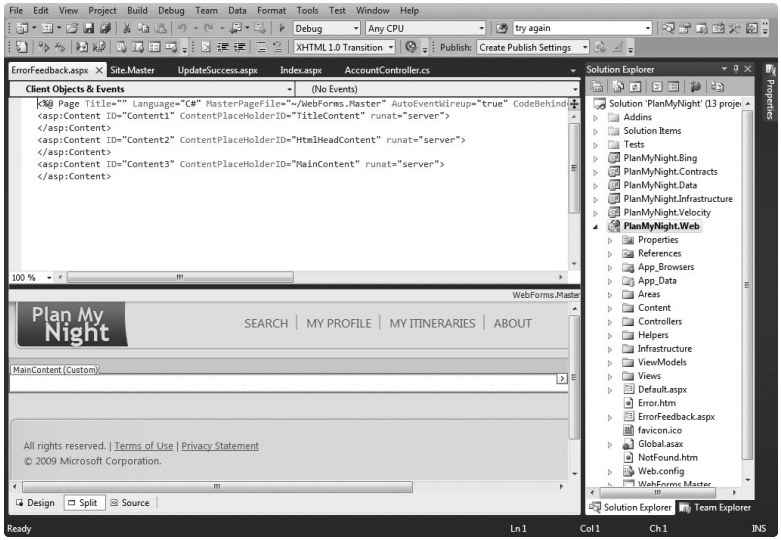
1. Откроем контекстное меню проекта PlanMyNight.Web (щелкнув его правой кнопкой мыши), откроем подменю Add и выберем New Item.
2. В диалоговом окне Add New Item выберем Web Form using Master Page (Веб-форма, использующая главную страницу) и в поле Name введем имя элемента: **ErrorFeedback.aspx**.



3. Появится диалоговое окно, которое позволяет ассоциировать главную страницу с этой Веб-формой. Убедитесь, что в части Project Folders (Папки проекта) окна выбрана основная папка PlanMyNight.Web, и затем выберите элемент WebForms.Master в правой части окна.



4. Результирующая страница может быть представлена не в режиме разделенного представления (Split view), а в режиме просмотра исходного кода (или в режиме дизайнера (Design view)). Перейдите к разделенному представлению (соответствующая кнопка располагается внизу окна, как и в предыдущих версиях Visual Studio). После этого экран должен выглядеть следующим образом:



Примечание Рекомендуется использовать разделенное представление, поскольку оно позволяет видеть исходный код, формируемый дизайнером, и добавлять разметку в случае необходимости.

Раскройте панель элементов управления и оставьте ее на экране, поскольку в ходе работы над формой будете постоянно брать из нее элементы управления и элементы и переносить в область содержимого. Если эта панель еще не вынесена на экран, ее можно найти в меню View (Вид).

Начнем с переноса методом drag-and-drop элемента div (из группы элементов HTML) из панели элементов управления в раздел MainContent (Основное содержимое) дизайнера. При этом появится вкладка div, свидетельствующая о том, что добавленный новый элемент выбран в настоящий момент. Откроем контекстное меню div и выберем Properties (Свойства) (которые также можно открыть, нажав клавишу F4). В раскрывшемся окне Properties задаем свойству *(Id)* значение *profileForm* (Форма профиля) (регистр имеет значение). Также изменим значение свойства *Class* (Класс) на *panel* (панель). После редактирования этих значений размер области содержимого изменится, потому что к представлению дизайнера применены CSS.

Поместим другой div внутрь первого и зададим его свойству *class* значение *innerPanel* (Внутренняя панель). На панели разметки добавим в *innerPanel* следующую разметку:

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

Закрыв тег <h2>, добавим новую строку и откроем контекстное меню. Выберем Insert Snippet (Вставить фрагмент) и последовательно щелкнем ASP.NET > formr. Это обеспечит создание тега серверной формы, которая будет служить контейнером для Веб-элементов управления. В форму поместите div, атрибуту class которого присвоено значение *items* (элементы), и тег fieldset (набор полей).

Далее перетягиваем в тег fieldset элемент управления TextBox (находится в разделе Standard (Стандартные) панели элементов управления). Задаем ID этого текстового поля значение **FullName** (Полное имя). Добавляем тег <label> (метка) перед этим элементом управления в представлении разметки, в качестве значения его свойства *for* задаем значение ID текстового поля, т.е. **Full Name:** (двоеточие обязательно). Поместите эти два элемента в тег <p>, и представление дизайнера должно выглядеть, как показано на рис. 9-10.

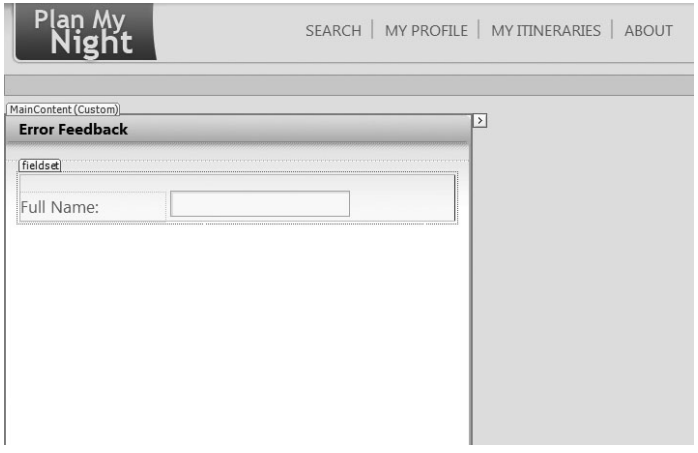


Рис. 9-10 Текущее состояние ErrorFeedback.aspx в дизайнере

Вышеописанным способом добавим еще одно текстовое поле и метку, но в качестве значения ID текстового поля зададим **EmailAddress** (Адрес электронной почты) и в качестве значения метки – **Email Address:**. Повторим весь этот процесс третий раз и зададим TextBox ID и метку **Comments** (Комментарии). Теперь в дизайнере должно находиться три метки и три однострочных элемента управления TextBox. Для элемента управления Comments необходимо обеспечить многострочный ввод, поэтому открываем его свойства и задаем *TextMode* (Текстовый режим) значение *Multiline* (Многострочный), *Rows* (Строки) значение *5*и *Columns* (Столбцы) значение *40*. После этого текстовое поле станет намного шире, что позволит пользователю вводить его комментарии.

Снова воспользуемся возможностью Insert Snippet и после текстового поля Comments вставим тег «div with class» (HTML>divc). В качестве класса тега div зададим *submit*. Из панели инструментов перетащим в этот div элемент управления Button (Кнопка). Зададим свойству *Text* (Текст) кнопки значение *Send Feedback* (Отправить отзыв).

В дизайнере должно отображаться нечто похожее представленному на рис. 9-11. На данный момент мы получили страницу, которая будет передавать форму.

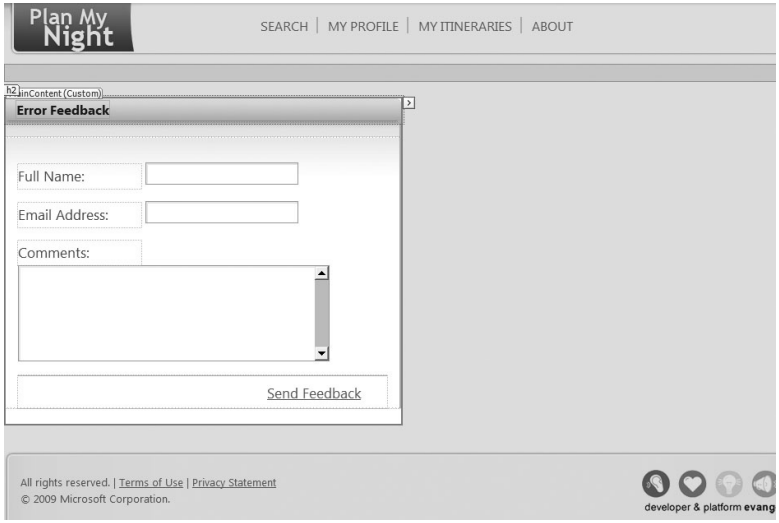
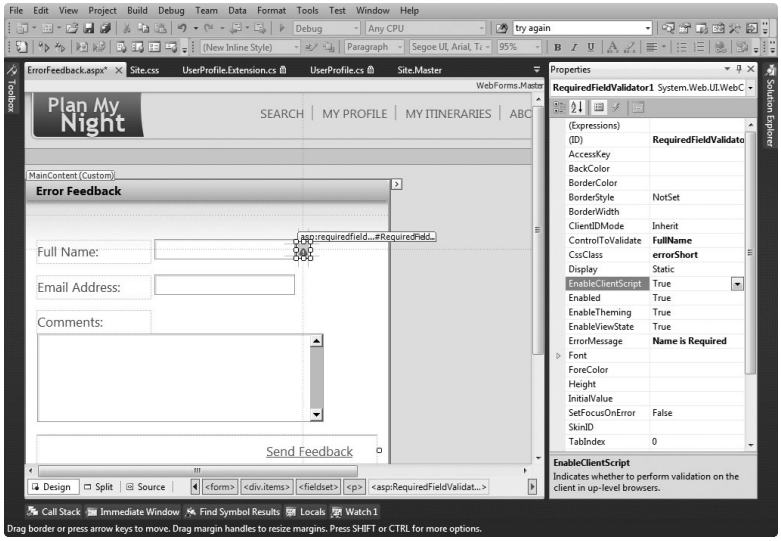


Рис. 9-11 Форма ErrorFeedback.aspx с полным набором полей

Но эта форма не выполняет никакой проверки передаваемых ею данных. Для реализации этой функциональности воспользуемся некоторыми элементами управления проверки, предлагаемыми ASP.NET. Сделаем поля Full Name и Comments обязательными для заполнения и осуществим проверку адреса электронной почты при помощи регулярных выражений, чтобы гарантировать его соответствие заданному шаблону.

В группе Validation (Проверка) панели инструментов имеются некоторые готовые элементы управления проверки. Возьмите на панели инструментов объект *RequiredFieldValidator* (Средство проверки обязательного поля) и поместите его справа от текстового поля Full Name. Откройте свойства этого элемента управления проверки и задайте свойству *ControlToValidate* (Проверяемый элемент управления) значение

FullName. (Для удобства предлагается раскрывающийся список элементов управления страницы.) Также задайте свойству *CssClass* (Класс CSS) значение *field-validation-error* (ошибка проверки поля). Это обусловит то, что все ошибки в приложении будут обозначаться красным треугольником. Наконец, свойству Error Message (Сообщение об ошибке) задаем значение «Name is Required» (Необходимо указать имя).



Эти же шаги повторим для поля Comments, но зададим соответствующие значения *language* (язык) и *property* (свойство).

Пользователь должен гарантированно вводить действительный адрес электронной почты в поле Email Address, поэтому возьмем на панели инструментов элемент управления *RegularExpressionValidator* и поместим его рядом с этим текстовым полем. Значения свойствам этого элемента управления задаем по используемой ранее схеме, т.е. свойству *ControlToValidate* присваиваем значение *EmailAddress* и свойству *CssClass* – значение *errorShort*. Но для этого элемента управления также задается регулярное выражение, которое должно применяться к вводимым данным. Это осуществляется посредством свойства *ValidationExpression* (Выражение проверки), значение которого должно быть определено следующим образом:

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

Сообщение об ошибке этого средства проверки должно быть сформулировано примерно так: «Введите действительный адрес электронной почты».

Наконец, сообщения о возникающих в ходе проверки формы ошибках должны где-то отображаться. Обо всем этом позаботиться элемент управления ValidationSummary (Сводка проверки). Найдите этот элемент управления в панели инструментов и поместите его в начало формы.

Форма готова. Чтобы увидеть ее в приложении, необходимо добавить опцию обратной связи при возникновении ошибки для пользователя. В Solution Explorer перейдите по дереву проекта PlanMyNight.Web к папке Views и затем к подпапке Shared. Откройте

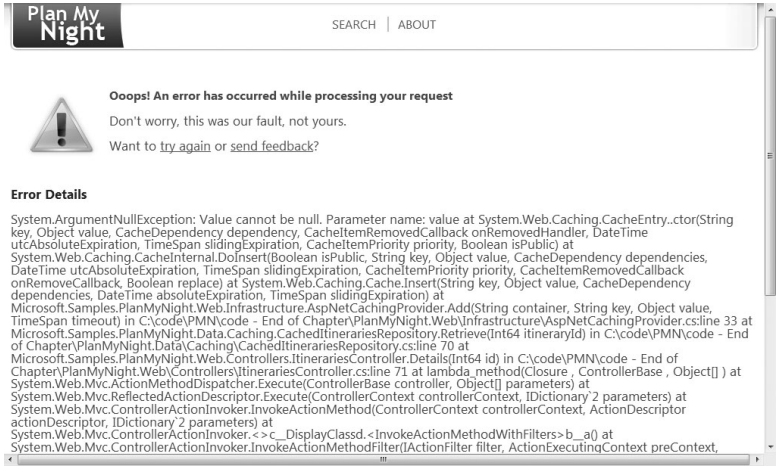
файл Error.aspx в средстве просмотра разметки и перейдите к строке 35. В этой строке обрабатывается предложение пользователю сделать еще одну попытку выполнить действие, и сюда мы поместим опцию для отправки отзыва. После текста вопроса в тот же абзац добавьте следующую разметку:

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

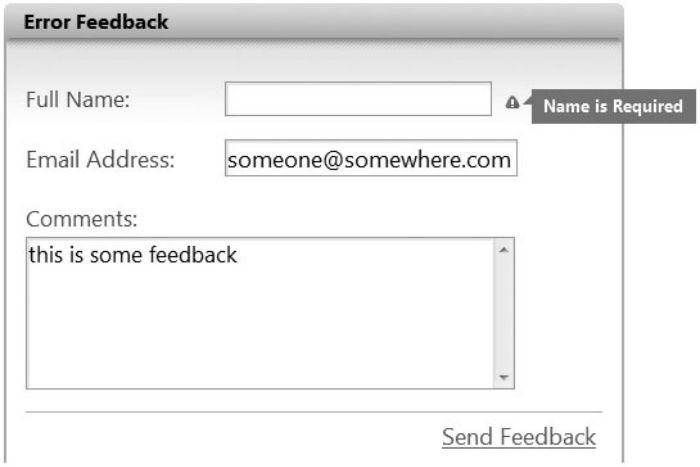
Это обеспечит добавление опции перехода к только что созданной нами форме при возникновении в приложении MVC любой ошибки общего характера. Чтобы увидеть эту форму, необходимо спровоцировать ошибку в приложении.

Чтобы спровоцировать возникновение ошибки в приложении Plan My Night:

1. Запустите приложение.
2. Увидев на экране стандартную страницу поиска, введите в адресную строку браузера следующий адрес: http://www.planmynight.net:48580/Itineraries/Details/38923828.
3. Очень маловероятно, что в базе данных имеется маршрут с таким ID, поэтому на экране появится страница с сообщением об ошибке.



4. На странице с сообщением об ошибке щелкните ссылку для перехода к форме обратной связи. Попробуйте передать форму, введя в нее недействительные данные.



Для осуществления проверки ASP.NET использует сценарий на стороне клиента (если браузер поддерживает это), поэтому пока данные вводятся, никаких обратных вызовов не выполняется. Когда же сервер получает введенные данные, разработчик может проконтролировать состояние проверки на стороне сервера с помощью свойства *Page.IsValid* в файле выделенного кода. Однако поскольку использовалась проверка на стороне клиента (по умолчанию), значение этого свойства всегда будет *true*. В файл выделенного кода необходимо добавить лишь код, обеспечивающий перенаправление пользователя при обратном вызове (и проверку свойства *Page.IsValid* в случае, если что-то было упущено при проверке на стороне клиента):

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

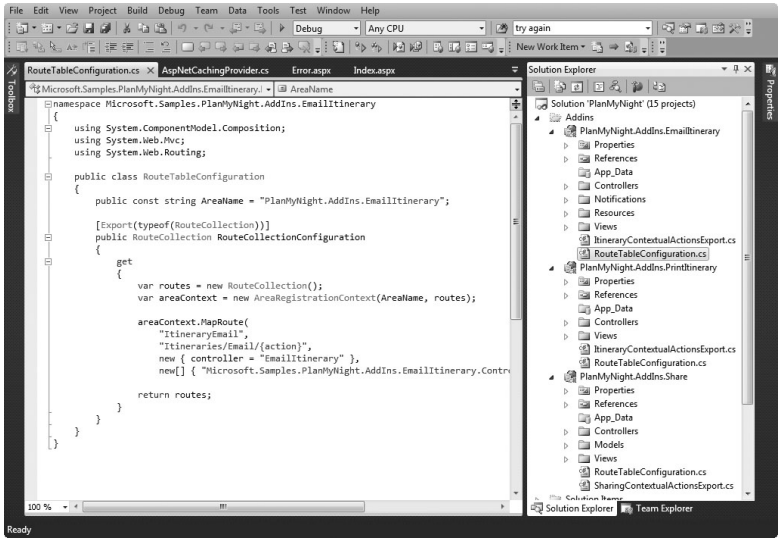
В этом нет особого смысла для пользователя, но нашей целью в данном разделе было создание Веб-формы ASP.NET с помощью дизайнера. Поэтому в проекте PlanMyNight.Web появился новый интерфейс. Но что, если бы потребовалось обеспечить большую модульность функциональности, скажем, реализовать некую функцию, которую можно было бы добавлять или удалять без компиляции основного проекта приложения. Вот здесь-то и может продемонстрировать свои преимущества такая инфраструктура расширения как Managed Extensibility Framework (MEF).

Расширение приложения с помощью MEF

Visual Studio 2010 предлагает новую технологию в составе .NET Framework 4 – Managed Extensibility Framework (MEF). Managed Extensibility Framework обеспечивает разработчикам простой, но при этом мощный механизм, с помощью которого сторонние производители получают возможность расширять приложения уже после их поставки. Благодаря MEF даже в рамках одного приложения разработчики могут создавать полностью изолированные компоненты, что обеспечивает возможность их независимого обслуживания или замены. Для этого MEF использует контейнер разрешений, который позволяет сопоставлять компоненты, обеспечивающие конкретную функцию (экспортеры), и компоненты, нуждающиеся в этой функциональности (импортеры), и при этом двум конкретным компонентам даже не надо ничего знать друг о друге. Разрешения выполняются только на контрактной основе, что делает компоненты взаимозаменяемыми или позволяет вводить их в приложения с очень небольшими издержками.

> **Примечание** Веб-сайт сообщества разработчиков MEF, на котором представлена детальная информация об этой архитектуре, можно найти по адресу http://mef.codeplex.com.

Приложение Plan My Night создавалось с учетом возможности расширения. В папке Addins (Надстройки) его решения имеется три проекта модулей «надстроек».

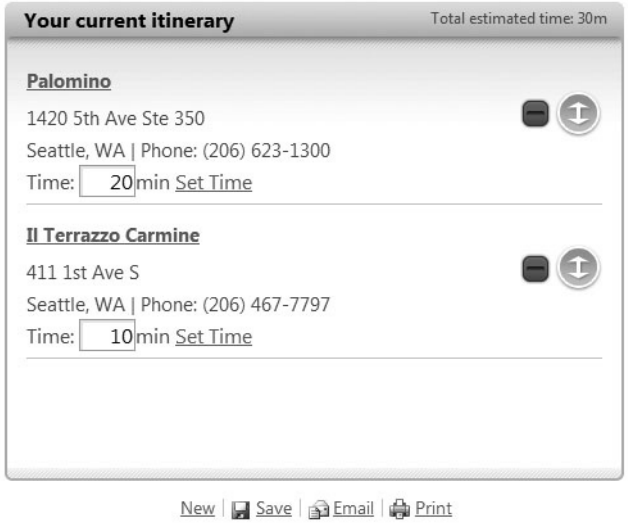


PlanMyNight.Addins.EmailItinerary добавляет возможность отправлять списки маршрутов всем, кому пожелает пользователь. PlanMyNight.Addins.PrintItinerary обеспечивает версию для печати маршрута. Наконец, PlanMyNight.Addins.Share добавляет функции публикации на социальных сайтах (что позволяет пользователю публиковать ссылку на маршрут), а также операции «обрезки» URL. Ни один из этих проектов не ссылается на основное приложение и не упоминается в нем. Да, в них имеются ссылки на проекты

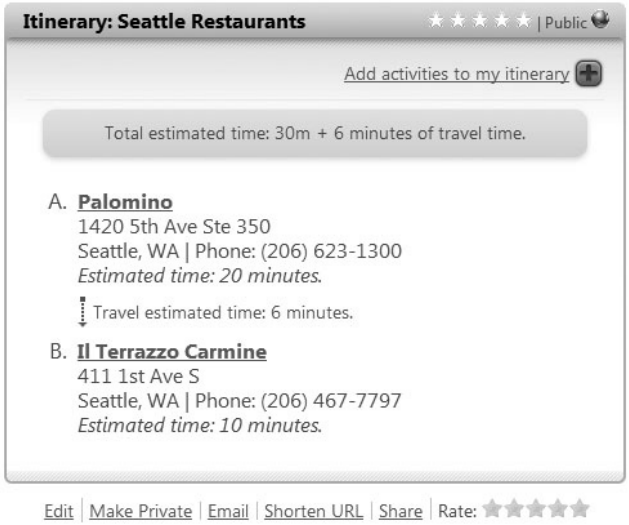
PlanMyNight.Contracts и PlanMyNight.Infrastructure, что позволяет экспортировать (и импортировать в некоторых случаях) соответствующие контракты через MEF, а также использовать любые специальные расширения проекта инфраструктуры.

> **Примечание** Если Веб-приложение еще не выполняется, прежде чем переходить к следующему этапу, запустите проект PlanMyNight.Web, чтобы видеть пользовательский интерфейс.

Чтобы добавить модули в выполняющееся приложение, запустим файл DeployAllAddins.bat, располагающийся в одной папке с файлом PlanMyNight.sln. Это обеспечит создание новых папок в разделе Areas проекта PlanMyNight.Web. Новые папки, по одной для каждой надстройки, будут включать файлы, необходимые для добавления их функциональности в основное Веб-приложение. Надстройки появляются в приложении как дополнительные опции на странице результатов поиска и на странице данных маршрута под разделом текущего маршрута. После того как командный файл закончит выполнение, перейдите к интерфейсу PlanMyNight, выполните поиск действия и добавьте его в текущий маршрут. Теперь под панелью маршрута, кроме New (Создать) и Save (Сохранить), должны появиться дополнительные опции.

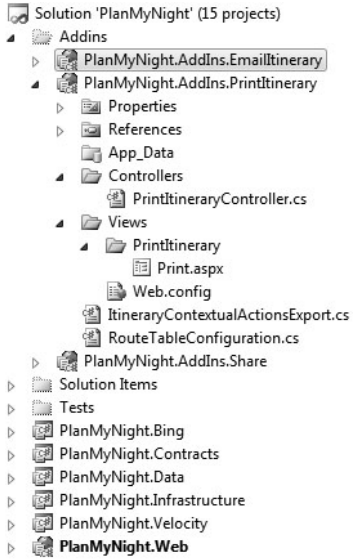


Опции публикации на социальных сайтах начнут отображаться в интерфейсе только после того, как маршрут будет сохранен и отмечен как общедоступный.



Visual Studio 2008 В Visual Studio 2008 нет ничего похожего на MEF. Для обеспечения поддержки подключаемых модулей разработчику приходилось создавать инфраструктуру подключаемого модуля с нуля или приобретать коммерческий продукт. Любой из этих вариантов приводил к созданию собственных решений, в которых внешнему разработчику приходилось разбираться в случае необходимости создания компонента для них. Добавление MEF в .NET Framework помогает убрать барьеры для перехода к разработке расширяемых приложений и подключаемых модулей для них.

Надстройка для создания печатной версии маршрута



Рассмотрим использование подключаемых модулей в приложении на примере проекта PrintItinerary.Addin. Дерево проекта в развернутом виде должно быть похожим на структуру, представленную на рисунке.

Некоторые моменты данной структуры аналогичны проекту PlanMyNight.Web (Controllers и Views), и поэтому эта надстройка будет помещена в приложение MVC как Area. Если более внимательно посмотреть на файл PrintItineraryController.cs в папке Controller, можно заметить, что его структура очень похожа на структуру контроллера, созданного нами ранее в этой главе (и любого другого контроллера Веб-приложения), но при этом имеются и существенные отличия от контроллеров, компилируемых в основном приложении PlanMyNight.Web.

В описании класса присутствуют дополнительные атрибуты:

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

Эти два атрибута описывают данный тип контейнера разрешений MEF. Первый атрибут, *Export* (Экспорт), помечает этот класс как предоставляющий *IController* с именем контракта *PrintItinerary*. Второй атрибут объявляет, что этот объект поддерживает только индивидуальное создание и не может создаваться как совместно используемый/singleton-объект. Задание этих двух атрибутов – это все, что необходимо для создания типа, используемого MEF. На самом деле, *PartCreationPolicy* (Политика создания части) является необязательным атрибутом, но он должен быть

определен на случай, если тип не может обрабатывать все типы политики создания.

Далее в файле PrintItineraryController.cs можно заметить, что конструктор снабжен атрибутом *ImportingConstructor* (Конструктор импорта):

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

Атрибут *ImportingConstructor* указывает MEF о необходимости предоставления параметров при создании этого объекта. В данном конкретном случае MEF обеспечивает экземпляр *IServiceFactory* для использования этим объектом. Классу *this* не важно, откуда поступает экземпляр, участвующий в создании модульных приложений. Для наших целей оговоренный контрактом *IServiceFactory* экспортируется файлом ServiceFactory.cs в проект PlanMyNight.Web.

Файл RouteTableConfiguration.cs регистрирует сведения о маршруте URL, которые должны быть направлены в PrintItineraryController. Этот маршрут, и маршруты других надстроек, регистрируются в приложении в ходе выполнения метода *Application_Start*, описанного в файле Global.asax.cs приложения PlanMyNight.Web:

```
// Фабрика MEF Controller
var controllerFactory = new MefControllerFactory(container);
ControllerBuilder.Current.SetControllerFactory(controllerFactory);

// Регистрация маршрутов из надстроек
foreach (RouteCollection routes in container.GetExportedValues<RouteC
ollection>())
{
    foreach (var route in routes)
    {
        RouteTable.Routes.Add(route);
    }
}
```

controllerFactory – это контейнер MEF, включающий путь к подпапке Areas (в которой находятся все надстройки). Он назначен фабрикой контроллера на весь срок жизни приложения. Благодаря этому контроллеры, импортированные через MEF, могут использоваться в любом месте приложения. Маршруты этих надстроек извлекаются из контейнера MEF и регистрируются в таблице маршрутизации MVC.

Файл ItineraryContextualActionsExport.cs экспортирует сведения для создания ссылки на этот подключаемый модуль, а также создания метаданных для отображения. Эти сведения используются в файле ViewModelExtensions.cs проекта PlanMyNight.Web при построении модели представления для отображения пользователю:

```
// получаем ссылки и панели инструментов надстроек
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 }));
```

Вызов *AddinExtensions.GetLinksFor* обеспечивает перечисление экспортированных элементов в поставщике экспорта MEF и возвращает их коллекцию, которая может быть добавлена в локальную коллекцию *addinLinks* (Ссылки на надстройки). Впоследствии эта коллекция используется в представлении для отображения большего числа опций, если они имеются.

Заключение

В данной главе мы обсудили лишь несколько возможностей из большого числа новых технологий, предлагаемых Visual Studio 2010. Поэтапно рассмотрели создание контроллера и его связанного представления, а также мощную возможность, предлагаемую инфраструктурой ASP.NET MVC Веб-разработчикам для создания Веб-приложений. Мы познакомились, как с помощью Managed Extensibility Framework можно создавать внешние модули надстроек и подключать их к приложению во время выполнения. В следующей главе будет показано, какие улучшения предлагает Visual Studio 2010 для отладки приложений.

Microsoft® Visual Studio® 2010

Microsoft® Visual Studio® 2010 – пути сертификации

Независимо от того, являетесь ли вы новичком в области ИТ, ищущим работу, или опытным ИТ-специалистом, планирующим свой карьерный рост, сертификация Microsoft поможет вам достичь цель. Почувствуйте уверенность в своих знаниях, протестировав их с помощью сертификационного экзамена Microsoft, и одновременно продемонстрируйте своим заказчикам, работодателю и коллегам свою готовность совершенствовать свои навыки и решать более трудные задачи.

Сертификация Microsoft помогает вам не только при найме на новую работу или при продвижении в вашей компании, она обеспечивает доступ к сообществу сертифицированных специалистов и ресурсам, которые помогут вам преуспеть на работе на любой ступеньке вашей карьеры.

Пути сертификации

обязательные экзамены
обязательные сертификации

ПУТЬ К MCPD
MICROSOFT CERTIFIED PROFESSIONAL DEVELOPER

MCPD
Windows Developer 4

EXAM 70-518
PRO: Desining and Developing Windows Application Using Microsoft .NET Framework 4

MCTS (3 CERTIFICATIONS)
.NET Framework 4: Windows Applications (1); Data Access (2); and Service Communication Application (3)

MCPD
Web Developer 4

EXAM 70-519
PRO: Designing and Developing Web Applications Using Microsoft .NET Framework 4

MCTS (3 CERTIFICATIONS)
.NET Framework 4: Web Applications (1); Data Access (2); and Service Communication Applications (3)

ПУТЬ К MCTS
MICROSOFT CERTIFIED TECHNOLOGY SPECIALIST

MCTS
.NET Framework 4, Windows Applications

EXAM 70-511
TS: Windows Applications Development with Microsoft .NET Framework 4

MCTS
.NET Framework 4, Service Communication Applications

EXAM 70-513
TS: Windows Communication Foundation Development with Microsoft .NET Framework 4

MCTS
.NET Framework 4, Data Access

EXAM 70-516
TS: Accessing Data with Microsoft .NET Framework 4

MCTS
.NET Framework 4, Web Applications

EXAM 70-515
TS: Web Applications Development with Microsoft .NET Framework 4

ПУТИ ОБНОВЛЕНИЯ СЕРТИФИКАЦИИ
FROM VISUAL STUDIO 2008 (.NET FRAMEWORK 3.5)

MCPD Windows Developer 4
MCTS (3) .NET Framework 4: Windows Applications (1); Service Communication Applications (2); Data Access (3)

EXAM 70-521
UPGRADE: Transition your MCPD Windows Developer 3.5, or Enterprise Application Developer 3.5, Skills to MCPD Windows Developer 4

MCPD
Windows Developer 3.5 or Enterprise Application Developer 3.5

MCPD Web Developer 4
MCTS (3) .NET Framework 4: Web Applications (1); Service Communication Applications (2); Data Access(3)

EXAM 70-523
UPGRADE: Transition your MCPD ASP.NET Developer 3.5, or Enterprise Application Developer 3.5, Skills to MCPD Web Developer 4

MCPD
ASP.NET Developer 3.5 or Enterprise Application Developer 3.5

Рекомендованные аудиторные курсы для подготовки к указанным выше сертификационным экзаменам

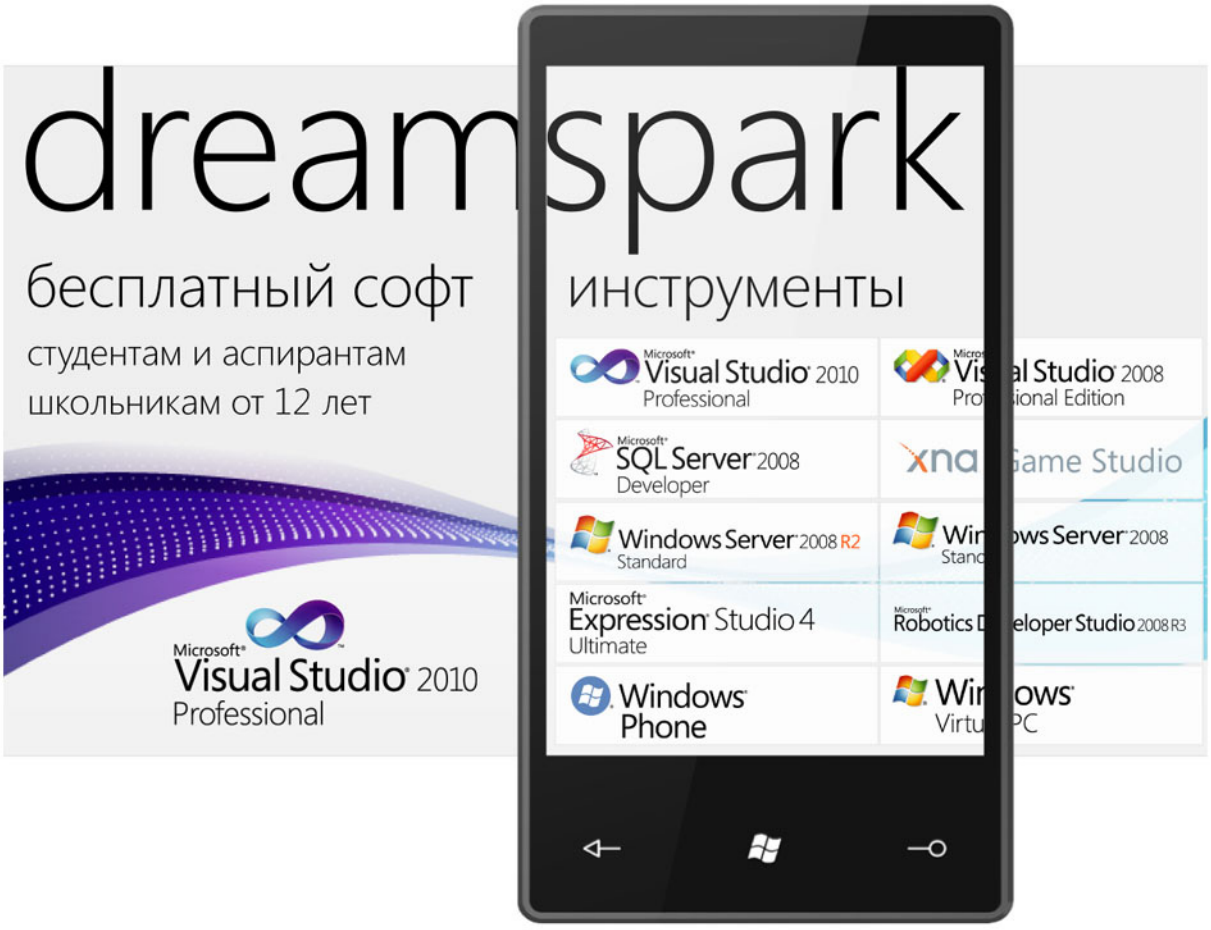
Экзамен 70-511	Курс 10262A: Developing Windows® Applications with Microsoft® Visual Studio® 2010 (5 дней)	Экзамен 70-513	Курс 10263A: Developing Windows® Communication Foundation Solutions with Microsoft® Visual Studio® 2010 (3 дня)
Экзамен 70-515	Курс 10264A: Developing Web Applications with Microsoft® Visual Studio® 2010 (5 дней) Курс 10267A: Introduction to Web Development with Microsoft® Visual Studio® 2010 (5 дней)	Экзамен 70-516	Курс 10265A: Developing Data Access Solutions with Microsoft® Visual Studio® 2010 (5 дней)

Воспользуйтесь скидками и специальными предложениями Microsoft по обучению и сертификации!

Информация о специальных предложениях - на веб-сайте Microsoft Learning: www.microsoft.com\rus\learning

Microsoft | Learning

www.microsoft.com/rus/learning



Microsoft DreamSpark™

Microsoft DreamSpark — это программа для студентов, аспирантов и школьников старше 12 лет, в рамках которой вы можете бесплатно получить инструменты Microsoft для дизайна и разработки для некоммерческого использования.

В рамках программы предоставляются такие продукты, как Visual Studio 2010, Expression Studio 4, SQL Server 2008, Windows Server 2008 и 2008 R2, XNA Game Studio и Robotics Developer Studio.

Также через программу DreamSpark вы можете получить бесплатный доступ к Windows Phone 7 Marketplace для разработчиков — и зарабатывать деньги на своих решениях.

Подробности и регистрации в программе: http://dreamspark.ru.

Microsoft