The Entity Framework 4.0 and ASP.NET Web Forms: Getting Started

Tom Dykstra

Step-by-Step

Microsoft

The Entity Framework 4.0 and ASP.NET Web Forms: Getting Started

Tom Dykstra

Summary: In this book, you'll learn the basics of using Entity Framework Database First to display and edit data in an ASP.NET Web Forms application.

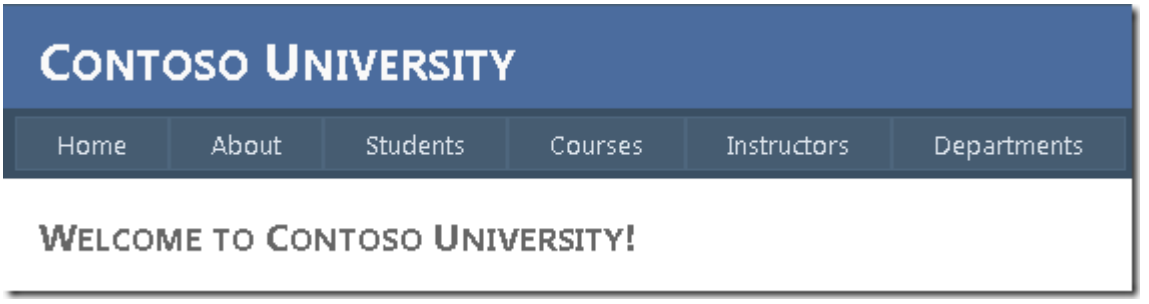
Category: Step-By-Step
Applies to: ASP.NET 4.0, ASP.NET Web Forms, Entity Framework 4.0, Visual Studio 2010
Source: ASP.NET site (link to source content)
E-book publication date: June 2012

Microsoft

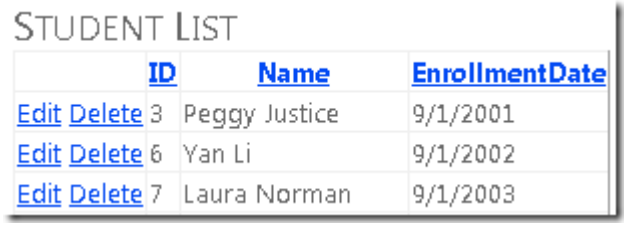
Contents

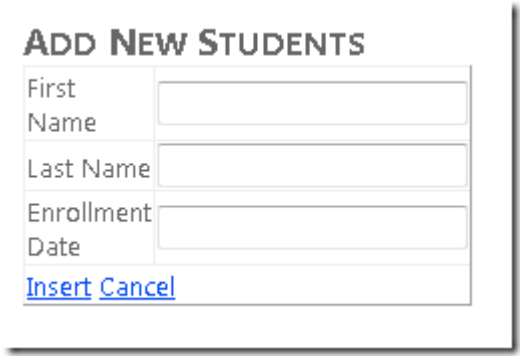
Introduction

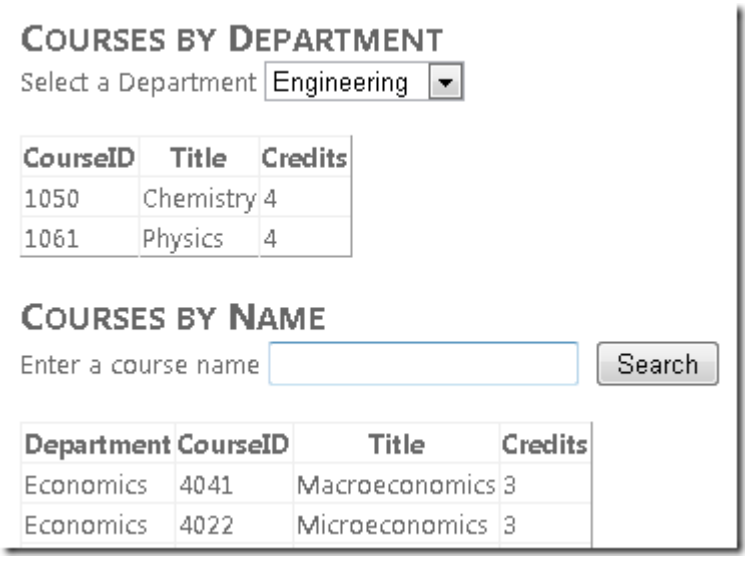
The application you'll be building in these tutorials is a simple university website.

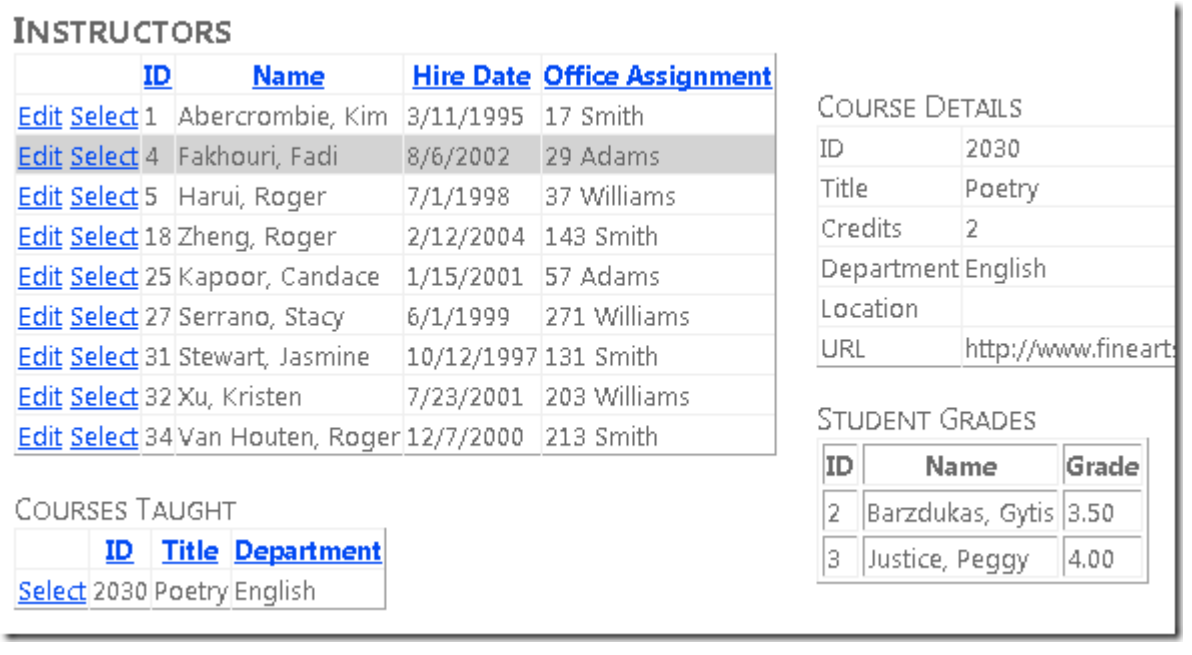


Users can view and update student, course, and instructor information. A few of the screens you'll create are shown below.



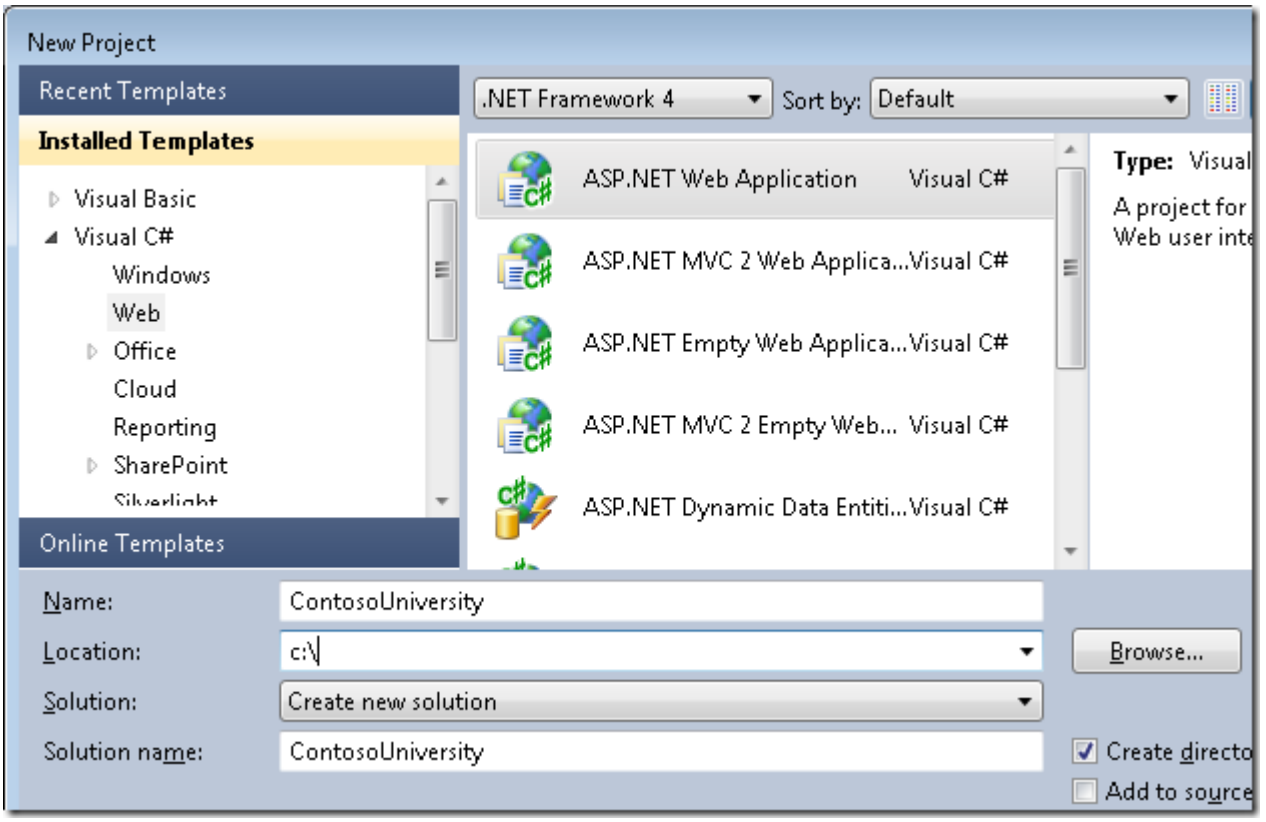




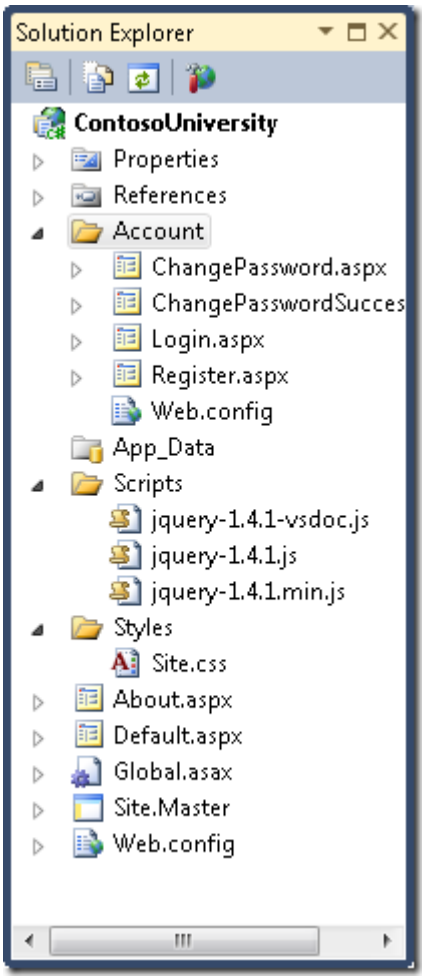


Creating the Web Application

To start the tutorial, open Visual Studio and then create a new ASP.NET Web Application Project using the **ASP.NET Web Application** template:



This template creates a web application project that already includes a style sheet and master pages:



Open the *Site.Master* file and change "My ASP.NET Application" to "Contoso University".

```
<h1>
    Contoso University
</h1>
```

Find the *Menu* control named `NavigationMenu` and replace it with the following markup, which adds menu items for the pages you'll be creating.

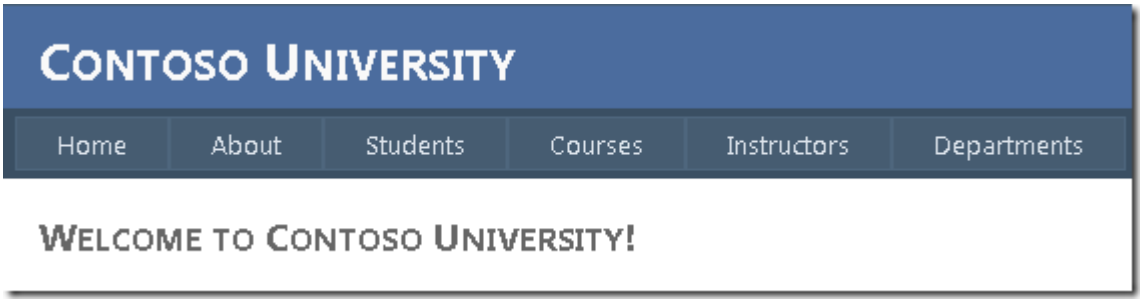
```
<asp:MenuID="NavigationMenu"runat="server"CssClass="menu"EnableViewState="false"
IncludeStyleBlock="false"Orientation="Horizontal">
<Items>
<asp:MenuItemNavigateUrl="/Default.aspx"Text="Home"/>
<asp:MenuItemNavigateUrl="/About.aspx"Text="About"/>
```

```
<asp:MenuItemNavigateUrl="/Students.aspx"Text="Students">
<asp:MenuItemNavigateUrl="/StudentsAdd.aspx"Text="Add Students"/>
</asp:MenuItem>
<asp:MenuItemNavigateUrl="/Courses.aspx"Text="Courses">
<asp:MenuItemNavigateUrl="/CoursesAdd.aspx"Text="Add Courses"/>
</asp:MenuItem>
<asp:MenuItemNavigateUrl="/Instructors.aspx"Text="Instructors">
<asp:MenuItemNavigateUrl="/InstructorsCourses.aspx"Text="Course Assignments"/>
<asp:MenuItemNavigateUrl="/OfficeAssignments.aspx"Text="Office Assignments"/>
</asp:MenuItem>
<asp:MenuItemNavigateUrl="/Departments.aspx"Text="Departments">
<asp:MenuItemNavigateUrl="/DepartmentsAdd.aspx"Text="Add Departments"/>
</asp:MenuItem>
</Items>
</asp:Menu>
```

Open the *Default.aspx* page and change the `Content` control named `BodyContent` to this:

```
<asp:ContentID="BodyContent"runat="server"ContentPlaceHolderID="MainContent">
<h2>
        Welcome to Contoso University!
</h2>
</asp:Content>
```

You now have a simple home page with links to the various pages that you'll be creating:



Creating the Database

For these tutorials, you'll use the Entity Framework data model designer to automatically create the data model based on an existing database (often called the *database-first* approach). An alternative that's not covered in this tutorial series is to create the data model manually and then have the designer generate scripts that create the database (the *model-first* approach).

For the database-first method used in this tutorial, the next step is to add a database to the site. The easiest way is to first download the project that goes with this tutorial. Then right-click the *App_Data* folder, select **Add Existing Item**, and select the *School.mdf* database file from the downloaded project.

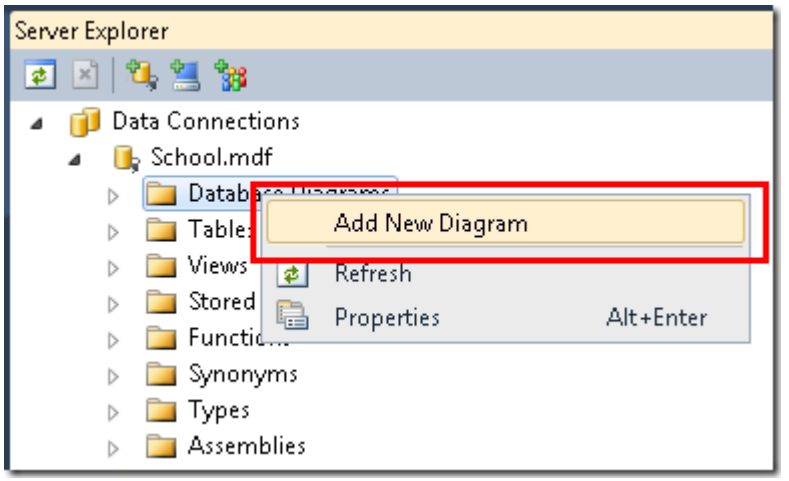
An alternative is to follow the instructions at Creating the School Sample Database. Whether you download the database or create it, copy the *School.mdf* file from the following folder to your application's *App_Data* folder:

```
%PROGRAMFILES%\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\DATA
```

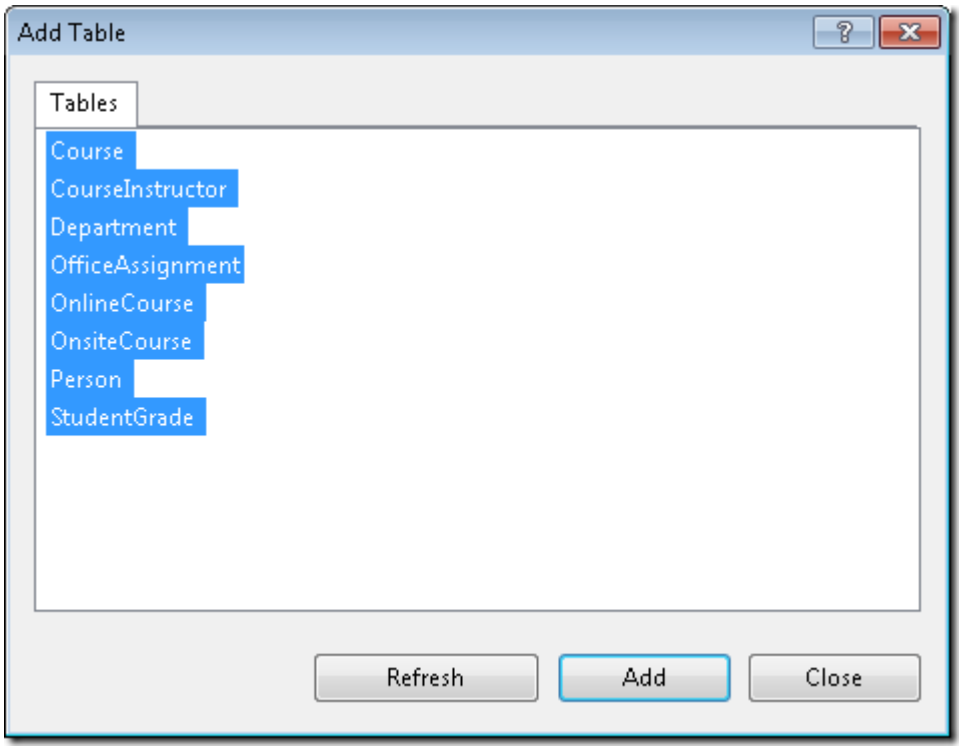
(This location of the *.mdf* file assumes you're using SQL Server 2008 Express.)

If you create the database from a script, perform the following steps to create a database diagram:

1. In **Server Explorer**, expand **Data Connections**, expand *School.mdf*, right-click **Database Diagrams**, and select **Add New Diagram**.



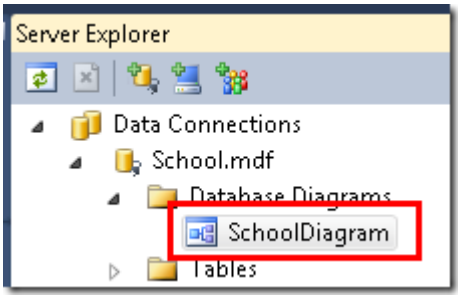
2. Select all of the tables and then click **Add**.



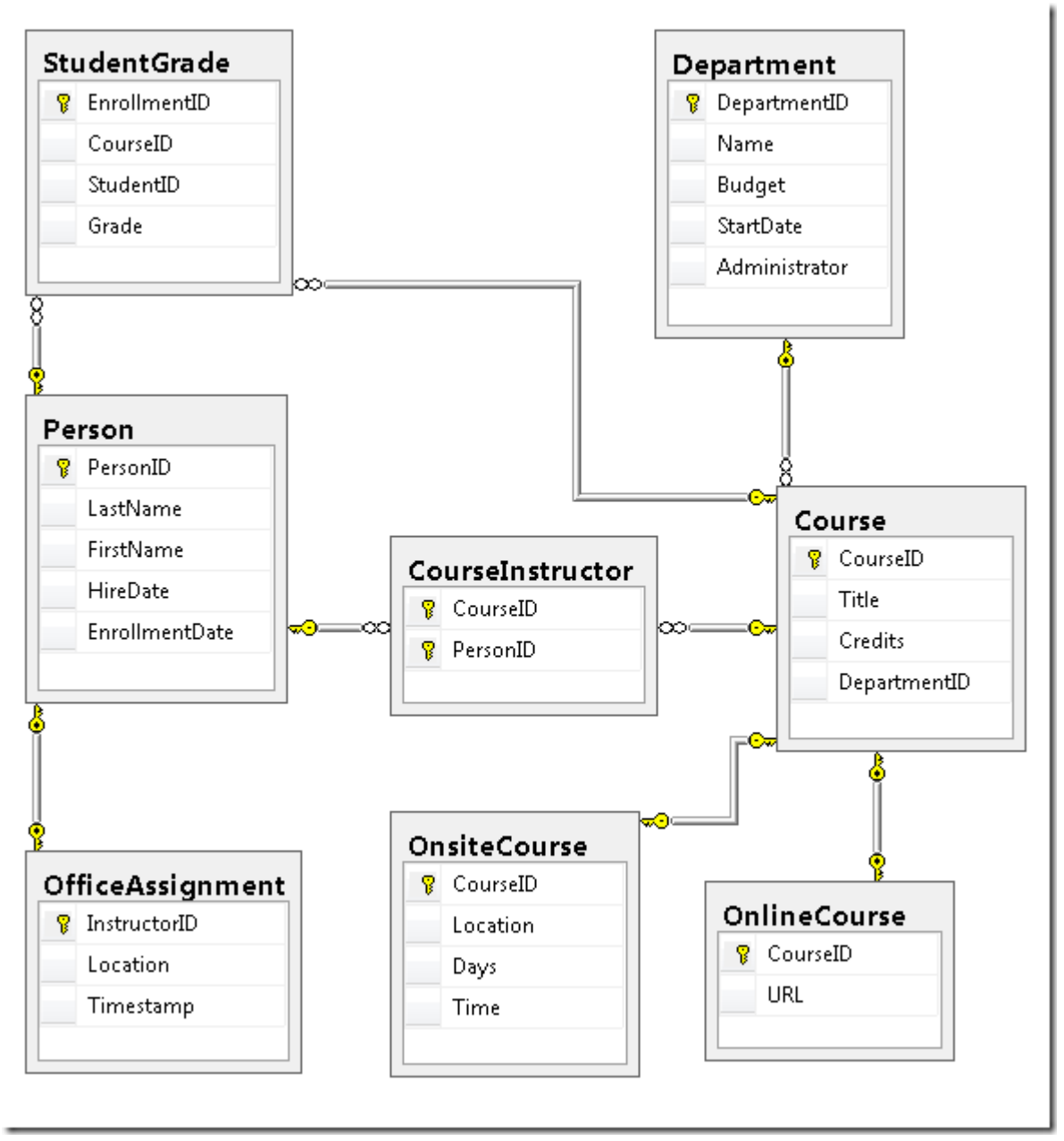
SQL Server creates a database diagram that shows tables, columns in the tables, and relationships between the tables. You can move the tables around to organize them however you like.

3. Save the diagram as "SchoolDiagram" and close it.

If you download the *School.mdf* file that goes with this tutorial, you can view the database diagram by double-clicking **SchoolDiagram** under **Database Diagrams** in **Server Explorer**.



The diagram looks something like this (the tables might be in different locations from what's shown here):

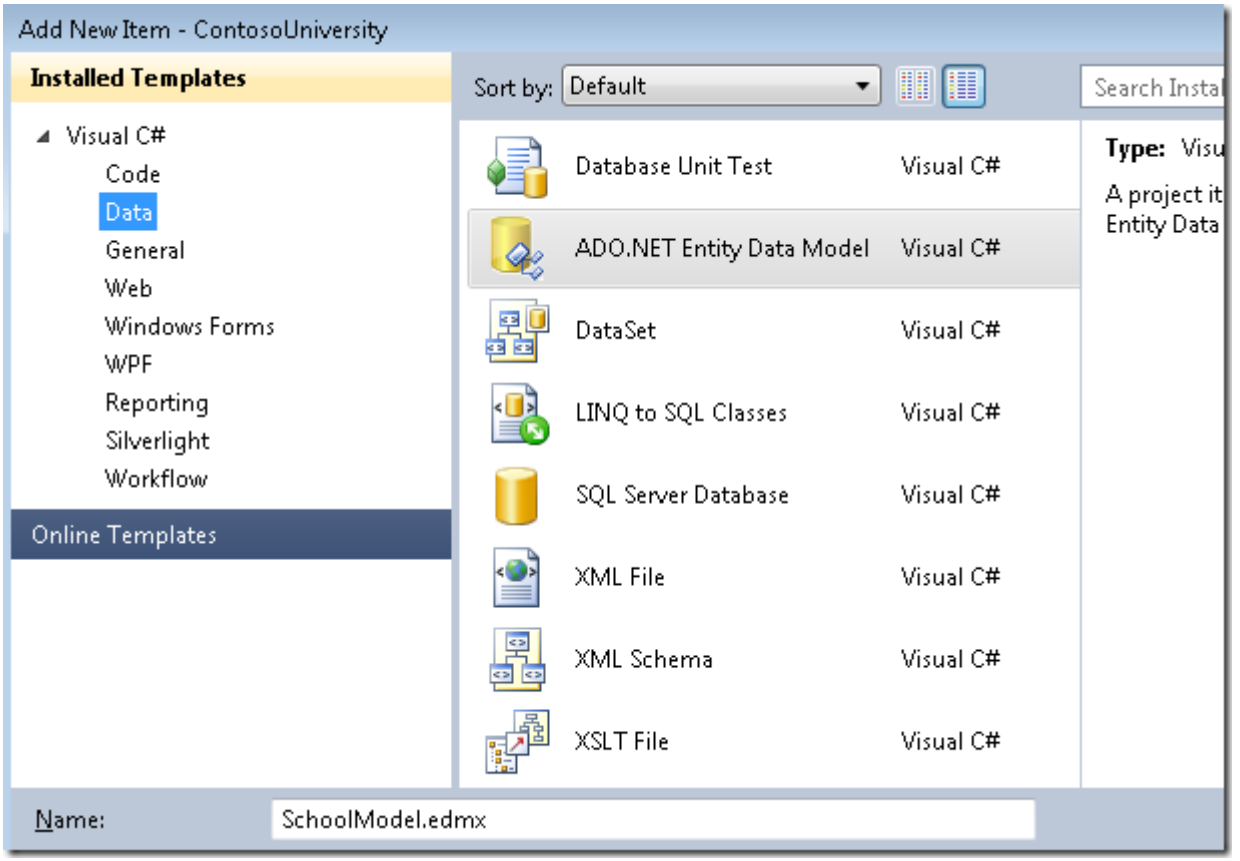


Creating the Entity Framework Data Model

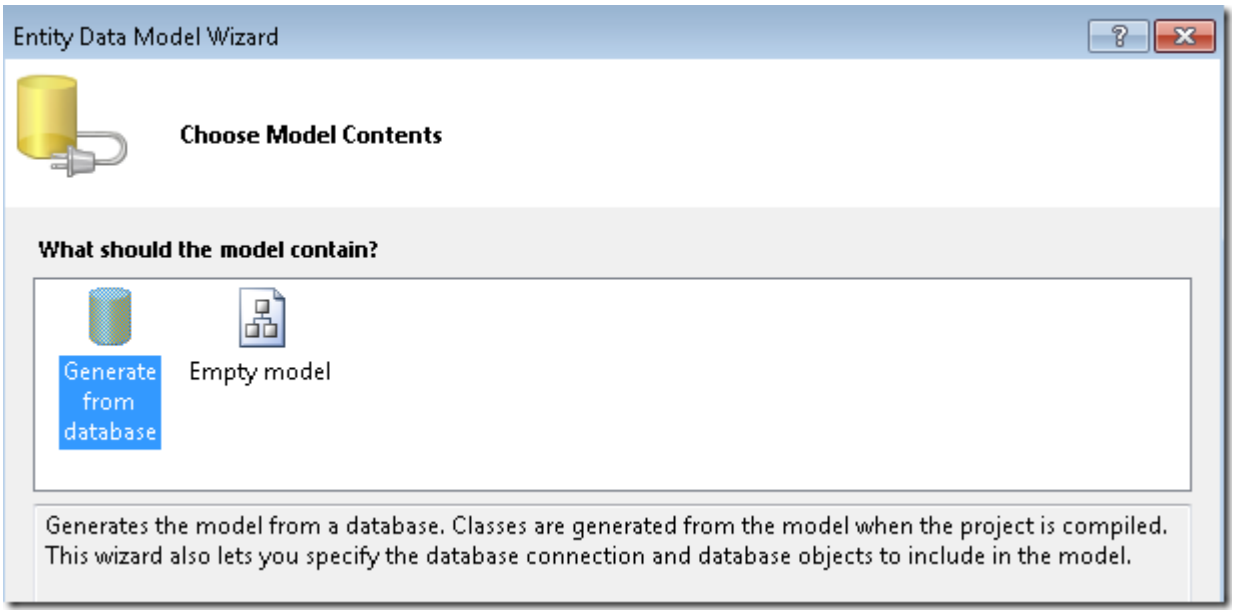
Now you can create an Entity Framework data model from this database. You could create the data model in the root folder of the application, but for this tutorial you'll place it in a folder named *DAL* (for Data Access Layer).

In **Solution Explorer**, add a project folder named *DAL* (make sure it's under the project, not under the solution).

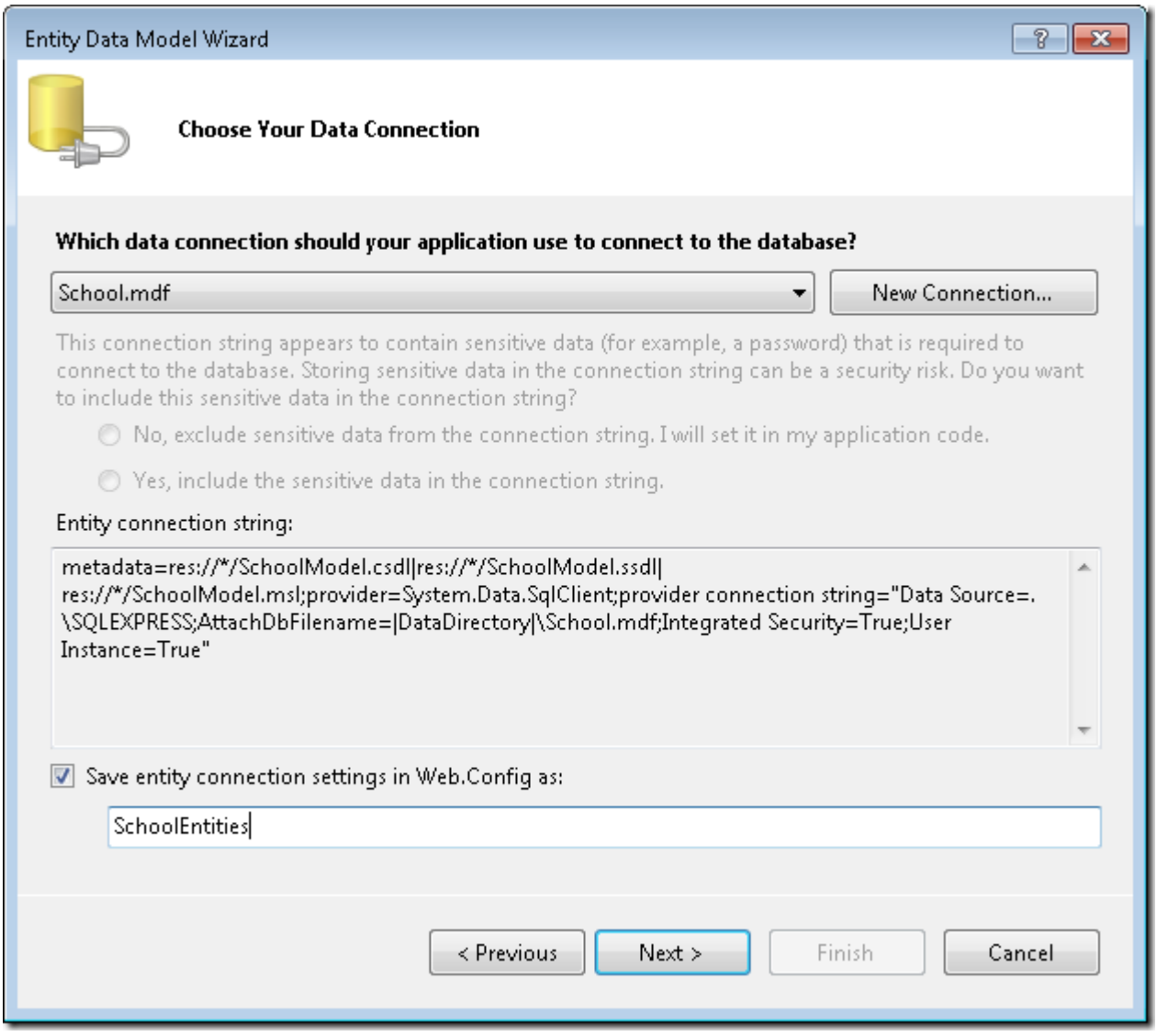
Right-click the *DAL* folder and then select **Add** and **New Item**. Under **Installed Templates**, select **Data**, select the **ADO.NET Entity Data Model** template, name it *SchoolModel.edmx*, and then click **Add**.



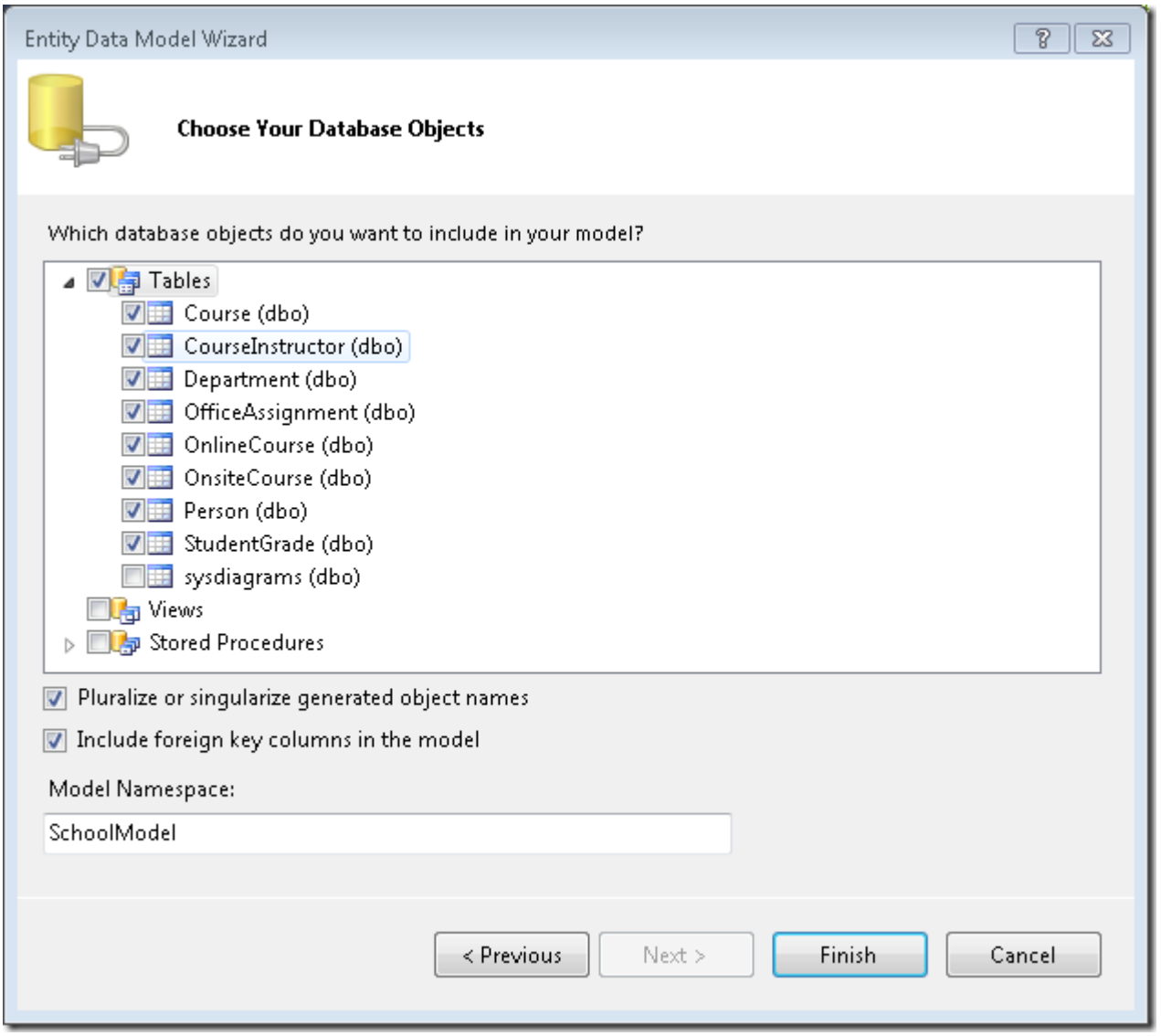
This starts the Entity Data Model Wizard. In the first wizard step, the **Generate from database** option is selected by default. Click **Next**.



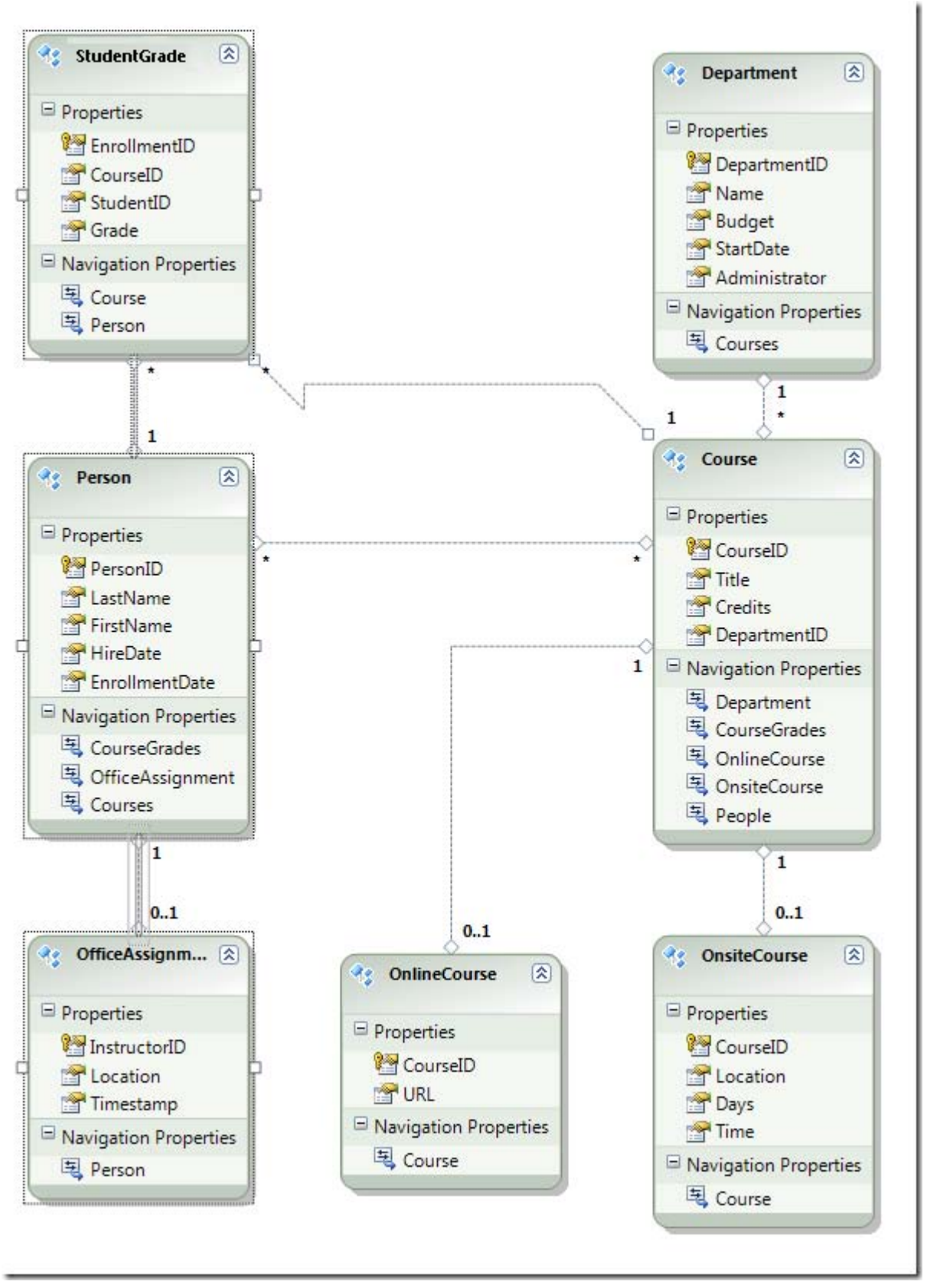
In the **Choose Your Data Connection** step, leave the default values and click **Next**. The School database is selected by default and the connection setting is saved in the *Web.config* file as **SchoolEntities**.



In the **Choose Your Database Objects** wizard step, select all of the tables except `sysdiagrams` (which was created for the diagram you generated earlier) and then click **Finish**.

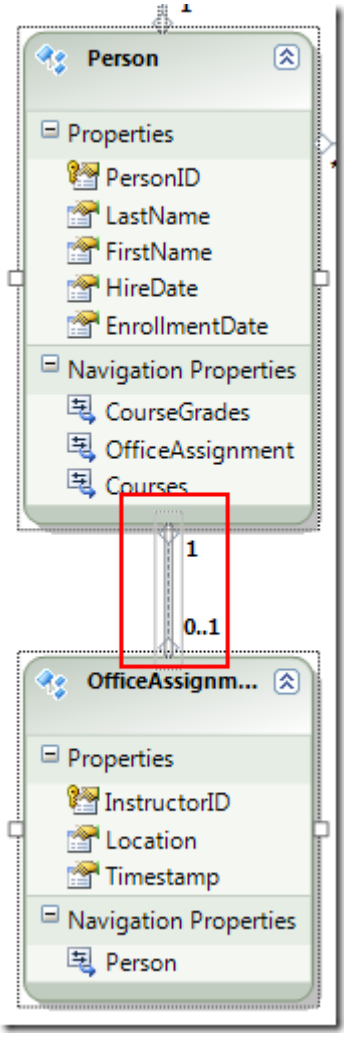


After it's finished creating the model, Visual Studio shows you a graphical representation of the Entity Framework objects (entities) that correspond to your database tables. (As with the database diagram, the location of individual elements might be different from what you see in this illustration. You can drag the elements around to match the illustration if you want.)


StudentGrade
Properties
EnrollmentID
CourseID
StudentID
Grade
Navigation Properties
Course
Person
Department
Properties
DepartmentID
Name
Budget
StartDate
Administrator
Navigation Properties
Courses
Person
Properties
PersonID
LastName
FirstName
HireDate
EnrollmentDate
Navigation Properties
CourseGrades
OfficeAssignment
Courses
Course
Properties
CourseID
Title
Credits
DepartmentID
Navigation Properties
Department
CourseGrades
OnlineCourse
OnsiteCourse
People
OfficeAssignm...
Properties
InstructorID
Location
Timestamp
Navigation Properties
Person
OnlineCourse
Properties
CourseID
URL
Navigation Properties
Course
OnsiteCourse
Properties
CourseID
Location
Days
Time
Navigation Properties
Course
1
*
0..1

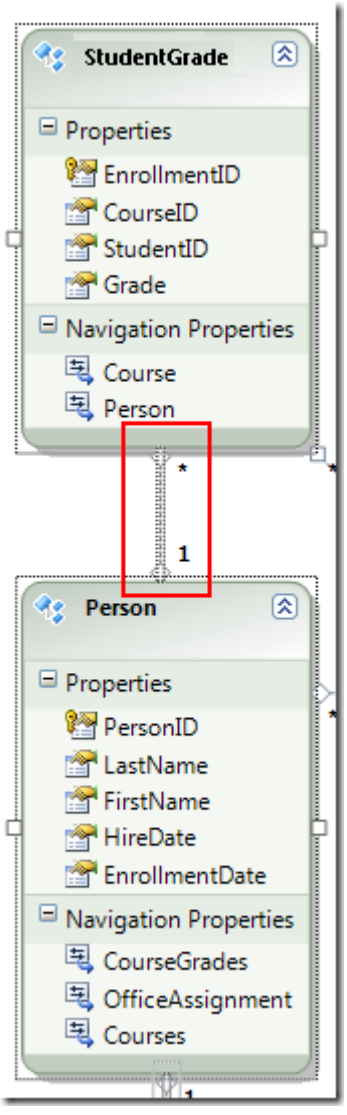
You can see that the entity diagram looks very similar to the database diagram, with a couple of differences. One difference is the addition of symbols at the end of each association that indicate the type of association (table relationships are called entity associations in the data model):

- A one-to-zero-or-one association is represented by "1" and "0..1".



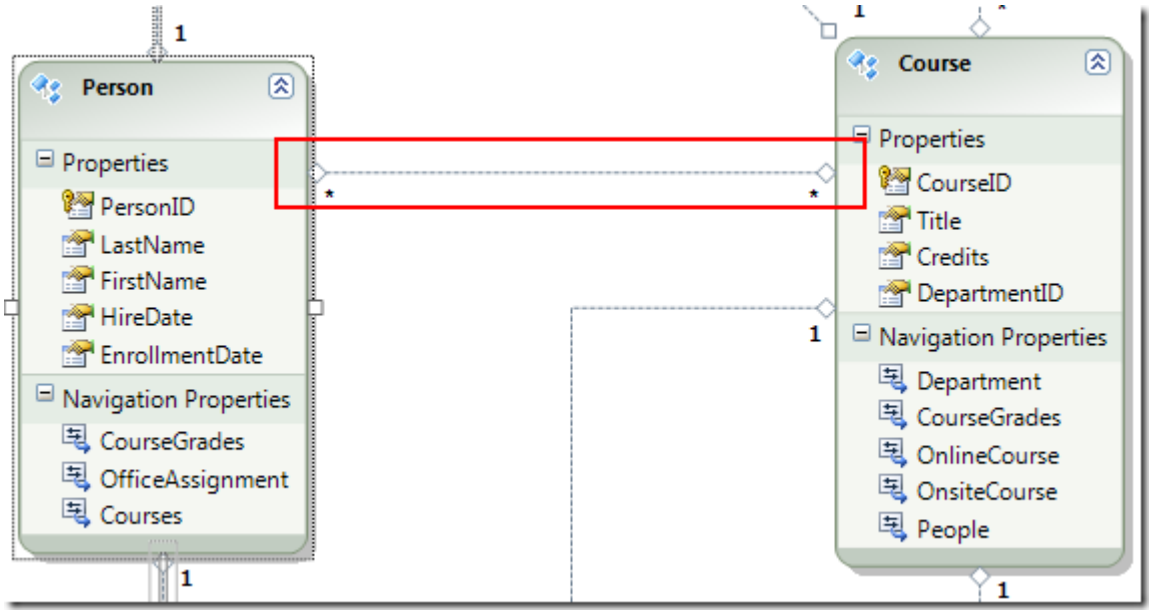
In this case, a `Person` entity may or may not be associated with an `OfficeAssignment` entity. An `OfficeAssignment` entity must be associated with a `Person` entity. In other words, an instructor may or may not be assigned to an office, and any office can be assigned to only one instructor.

- A one-to-many association is represented by "1" and "*".



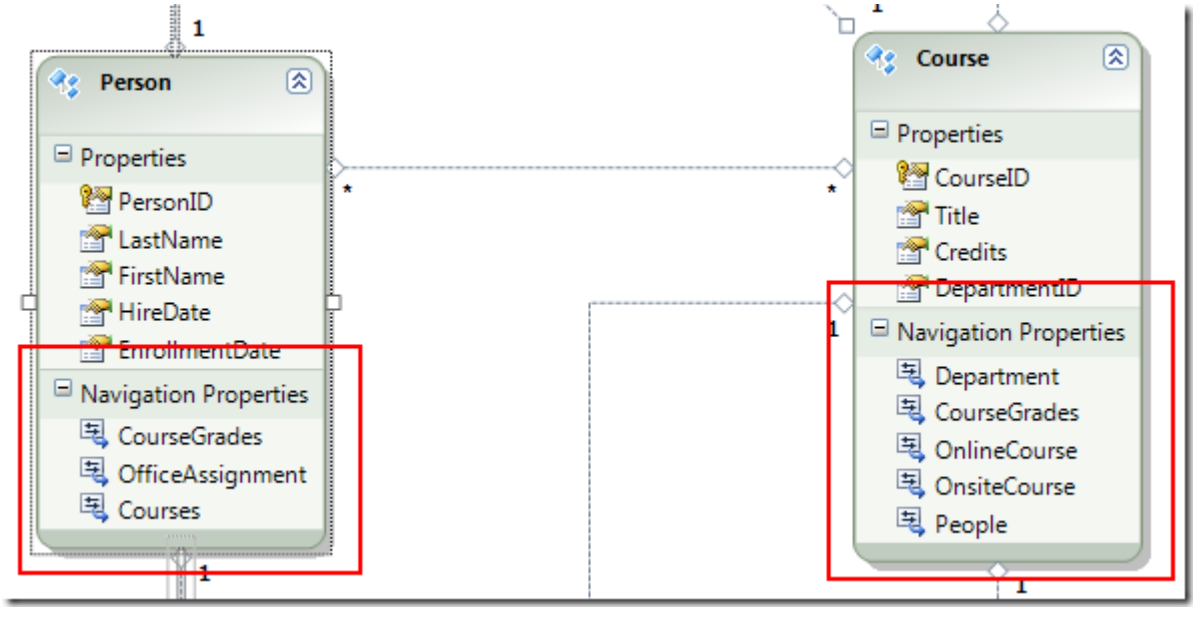
In this case, a `Person` entity may or may not have associated `StudentGrade` entities. A `StudentGrade` entity must be associated with one `Person` entity. `StudentGrade` entities actually represent enrolled courses in this database; if a student is enrolled in a course and there's no grade yet, the `Grade` property is null. In other words, a student may not be enrolled in any courses yet, may be enrolled in one course, or may be enrolled in multiple courses. Each grade in an enrolled course applies to only one student.

- A many-to-many association is represented by "*" and "*".

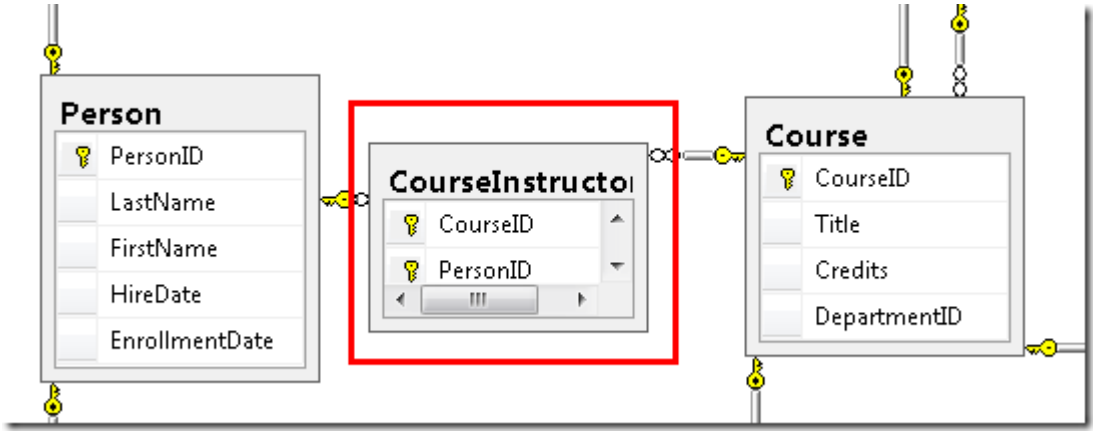


In this case, a `Person` entity may or may not have associated `Course` entities, and the reverse is also true: a `Course` entity may or may not have associated `Person` entities. In other words, an instructor may teach multiple courses, and a course may be taught by multiple instructors. (In this database, this relationship applies only to instructors; it does not link students to courses. Students are linked to courses by the StudentGrades table.)

Another difference between the database diagram and the data model is the additional **Navigation Properties** section for each entity. A navigation property of an entity references related entities. For example, the `Courses` property in a `Person` entity contains a collection of all the `Course` entities that are related to that `Person` entity.

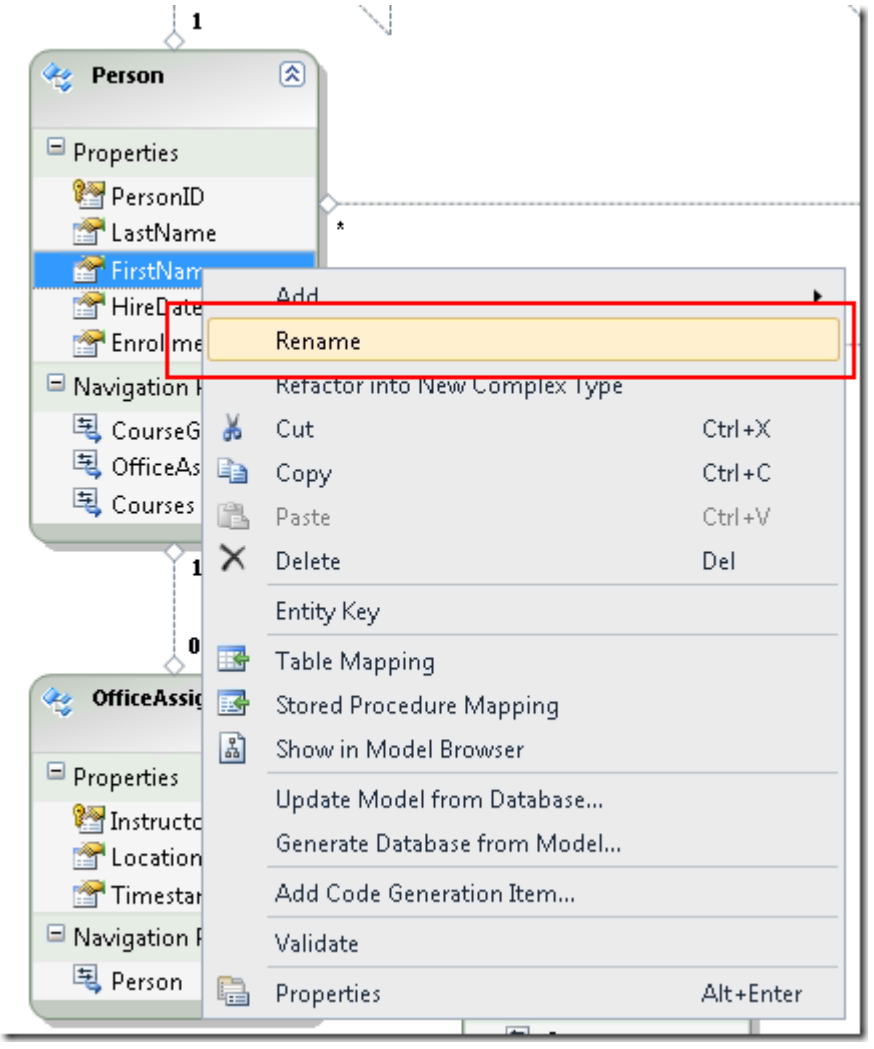


Yet another difference between the database and data model is the absence of the `CourseInstructor` association table that's used in the database to link the `Person` and `Course` tables in a many-to-many relationship. The navigation properties enable you to get related `Course` entities from the `Person` entity and related `Person` entities from the `Course` entity, so there's no need to represent the association table in the data model.

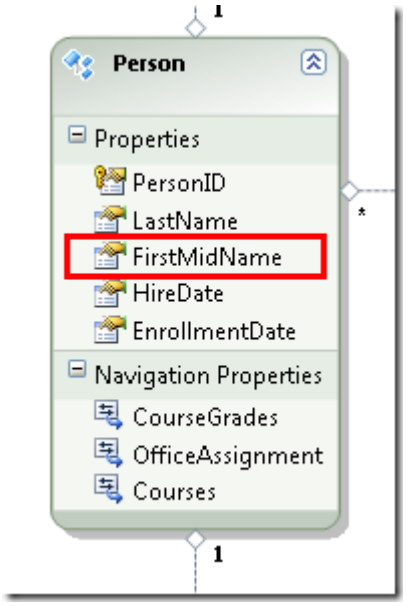


For purposes of this tutorial, suppose the `FirstName` column of the `Person` table actually contains both a person's first name and middle name. You want to change the name of the field to reflect this, but the database administrator (DBA) might not want to change the database. You can change the name of the `FirstName` property in the data model, while leaving its database equivalent unchanged.

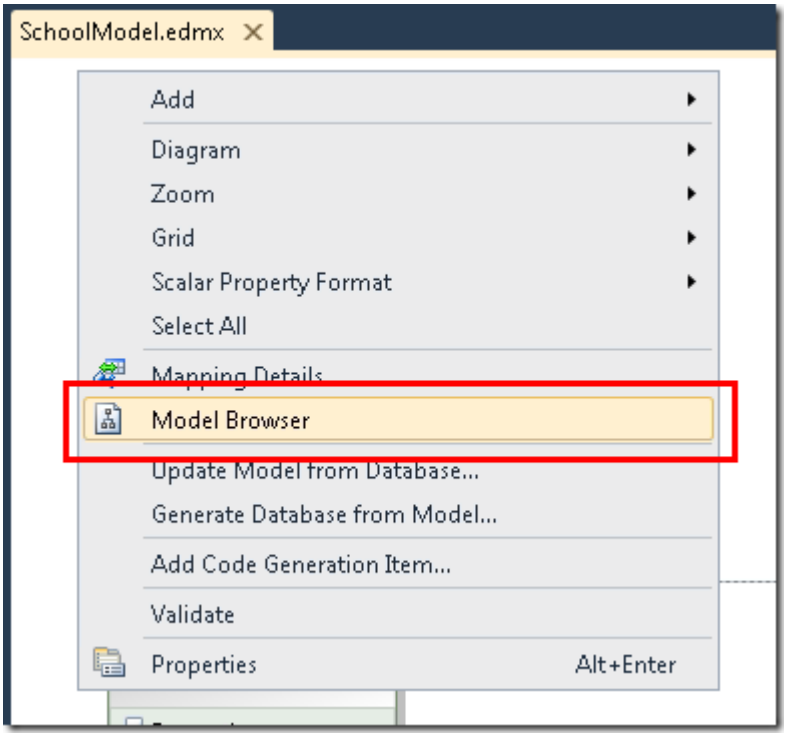
In the designer, right-click **FirstName** in the `Person` entity, and then select **Rename**.



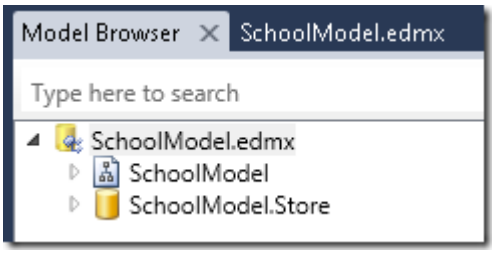
Type in the new name "FirstMidName". This changes the way you refer to the column in code without changing the database.



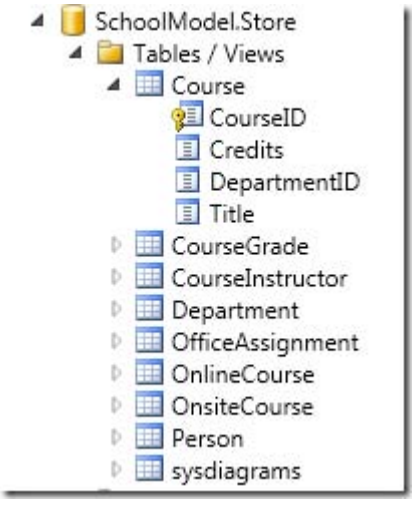
The model browser provides another way to view the database structure, the data model structure, and the mapping between them. To see it, right-click a blank area in the entity designer and then click **Model Browser**.



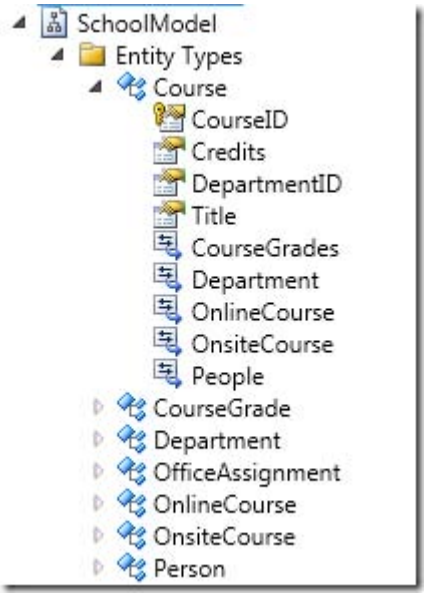
The **Model Browser** pane displays a tree view. (The **Model Browser** pane might be docked with the **Solution Explorer** pane.) The **SchoolModel** node represents the data model structure, and the **SchoolModel.Store** node represents the database structure.



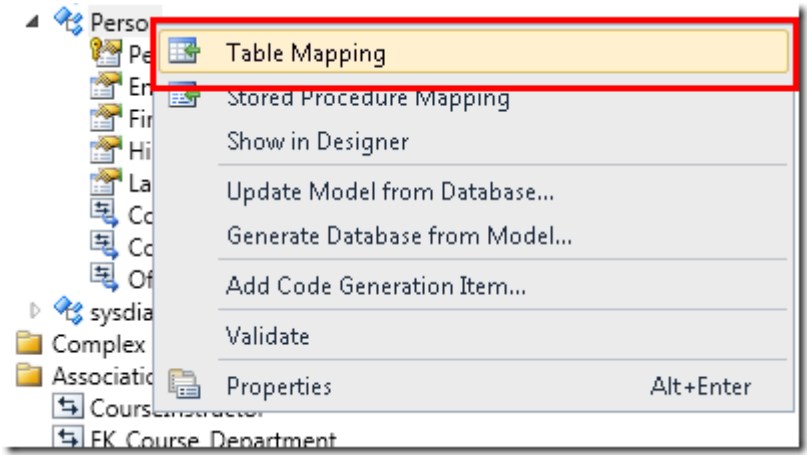
Expand **SchoolModel.Store** to see the tables, expand **Tables / Views** to see tables, and then expand **Course** to see the columns within a table.



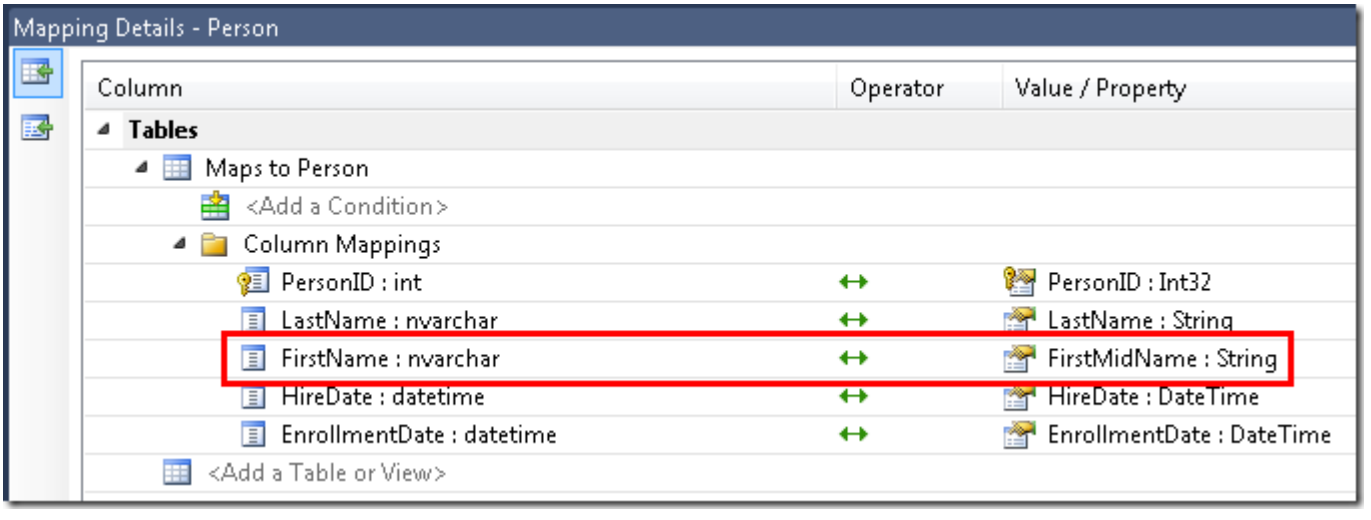
Expand **SchoolModel**, expand **Entity Types**, and then expand the **Course** node to see the entities and the properties within the entities.



In either the designer or the **Model Browser** pane you can see how the Entity Framework relates the objects of the two models. Right-click the `Person` entity and select **Table Mapping**.



This opens the **Mapping Details** window. Notice that this window lets you see that the database column `FirstName` is mapped to `FirstMidName`, which is what you renamed it to in the data model.

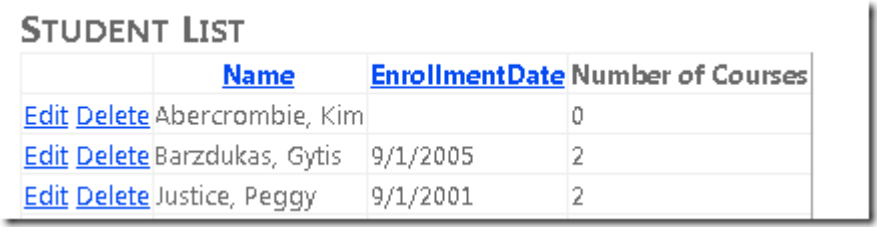


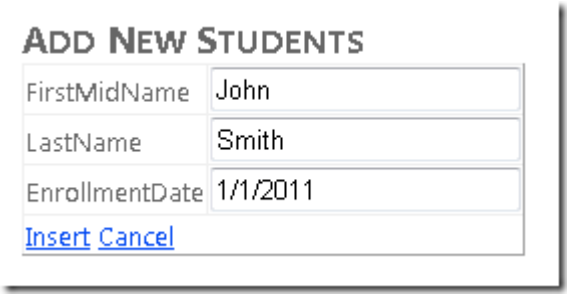
The Entity Framework uses XML to store information about the database, the data model, and the mappings between them. The *SchoolModel.edmx* file is actually an XML file that contains this information. The designer renders the information in a graphical format, but you can also view the file as XML by right-clicking the *.edmx* file in **Solution Explorer**, clicking **Open With**, and selecting **XML (Text) Editor**. (The data model designer and an XML editor are just two different ways of opening and working with the same file, so you cannot have the designer open and open the file in an XML editor at the same time.)

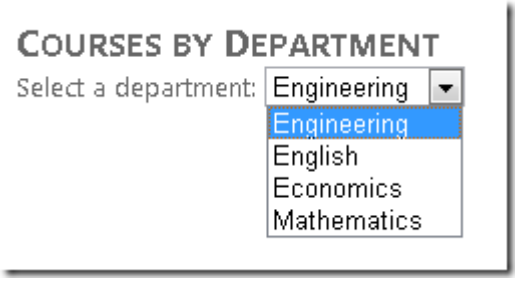
You've now created a website, a database, and a data model. In the next walkthrough you'll begin working with data using the data model and the ASP.NET `EntityDataSource` control.

The EntityDataSource Control

In the previous tutorial you created a web site, a database, and a data model. In this tutorial you work with the `EntityDataSource` control that ASP.NET provides in order to make it easy to work with an Entity Framework data model. You'll create a `GridView` control for displaying and editing student data, a `DetailsView` control for adding new students, and a `DropDownList` control for selecting a department (which you'll use later for displaying associated courses).







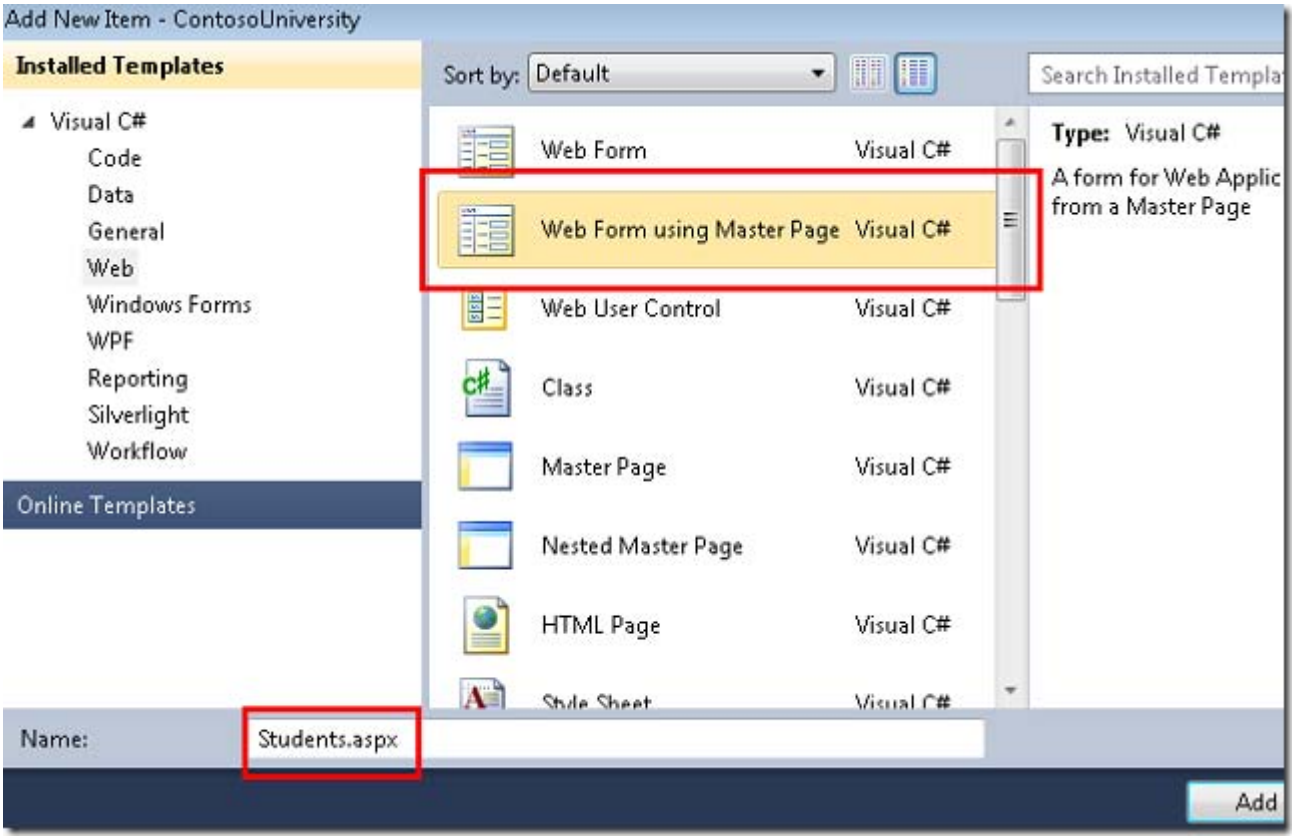
Note that in this application you won't be adding input validation to pages that update the database, and some of the error handling will not be as robust as would be required in a production application. That keeps the tutorial focused on the Entity Framework and keeps it from getting too long. For details about how to add these features to your application, see Validating User Input in ASP.NET Web Pages and Error Handling in ASP.NET Pages and Applications.

Adding and Configuring the EntityDataSource Control

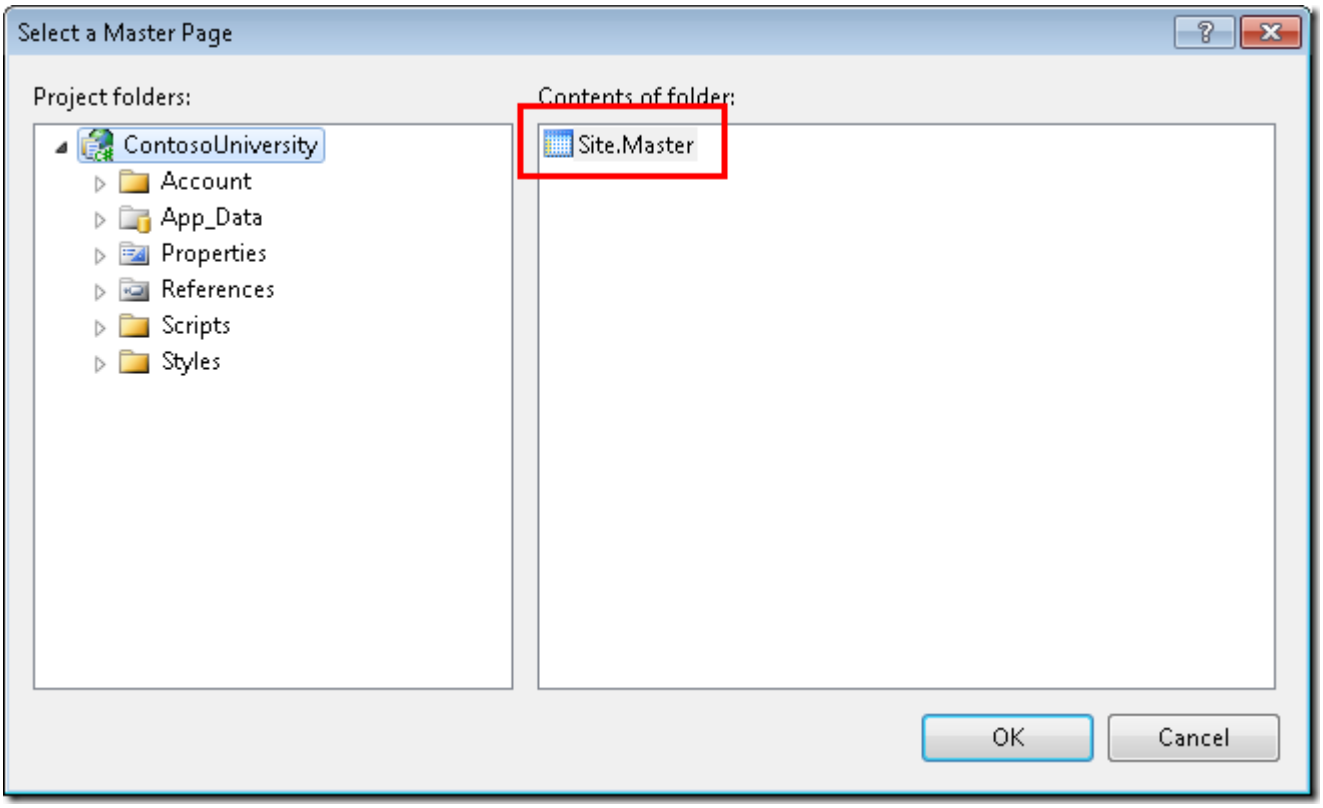
You'll begin by configuring an `EntityDataSource` control to read `Person` entities from the `People` entity set.

Make sure you have Visual Studio open and that you're working with the project you created in part 1. If you haven't built the project since you created the data model or since the last change you made to it, build the project now. Changes to the data model are not made available to the designer until the project is built.

Create a new web page using the **Web Form using Master Page** template, and name it *Students.aspx*.



Specify *Site.Master* as the master page. All of the pages you create for these tutorials will use this master page.



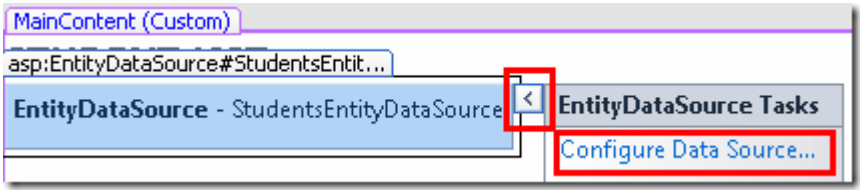
In **Source** view, add an `h2` heading to the `Content` control named `Content2`, as shown in the following example:

```
<asp:ContentID="Content2"ContentPlaceHolderID="MainContent"runat="server">
<h2>Student List</h2>
</asp:Content>
```

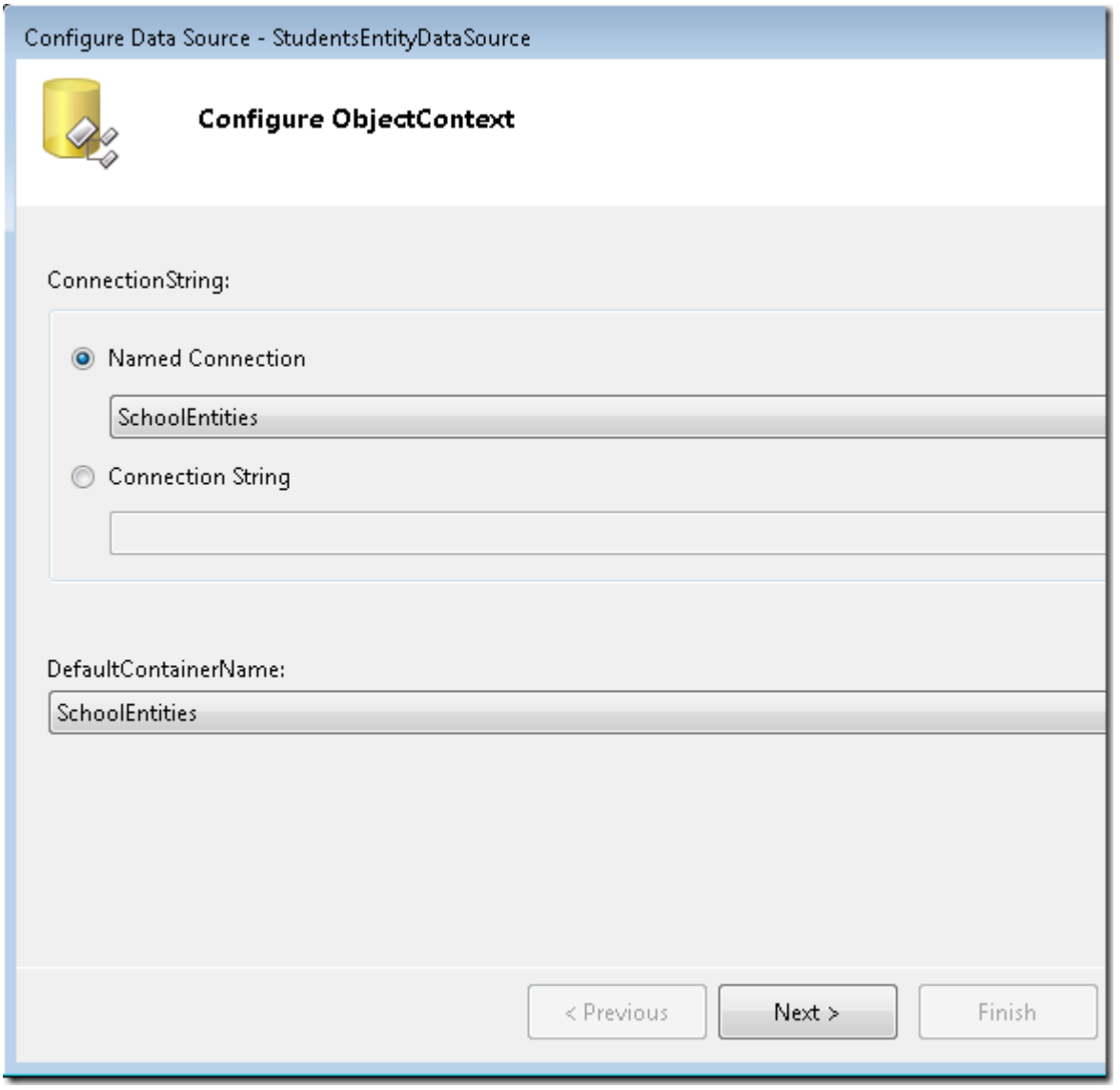
From the **Data** tab of the **Toolbox**, drag an `EntityDataSource` control to the page, drop it below the heading, and change the ID to `StudentsEntityDataSource`:

```
<asp:ContentID="Content2"ContentPlaceHolderID="MainContent"runat="server">
<h2>Student List</h2>
<asp:EntityDataSourceID="StudentsEntityDataSource"runat="server">
</asp:EntityDataSource>
</asp:Content>
```

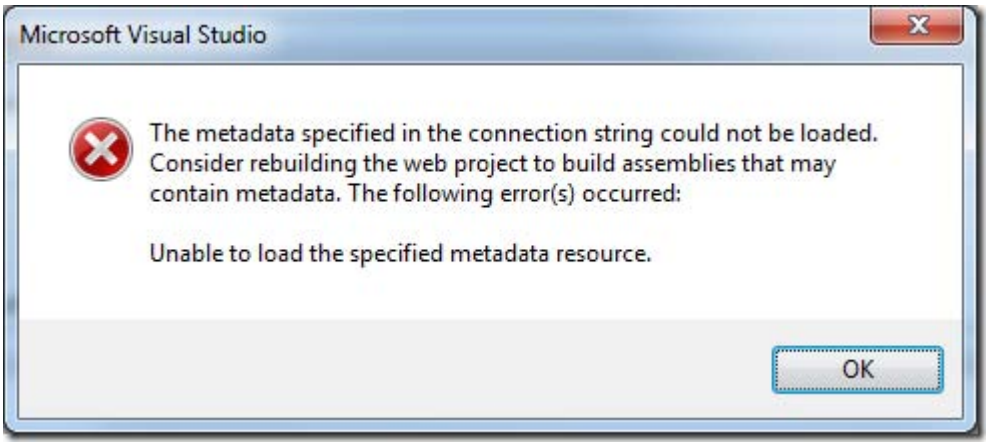
Switch to **Design** view, click the data source control's smart tag, and then click **Configure Data Source** to launch the **Configure Data Source** wizard.



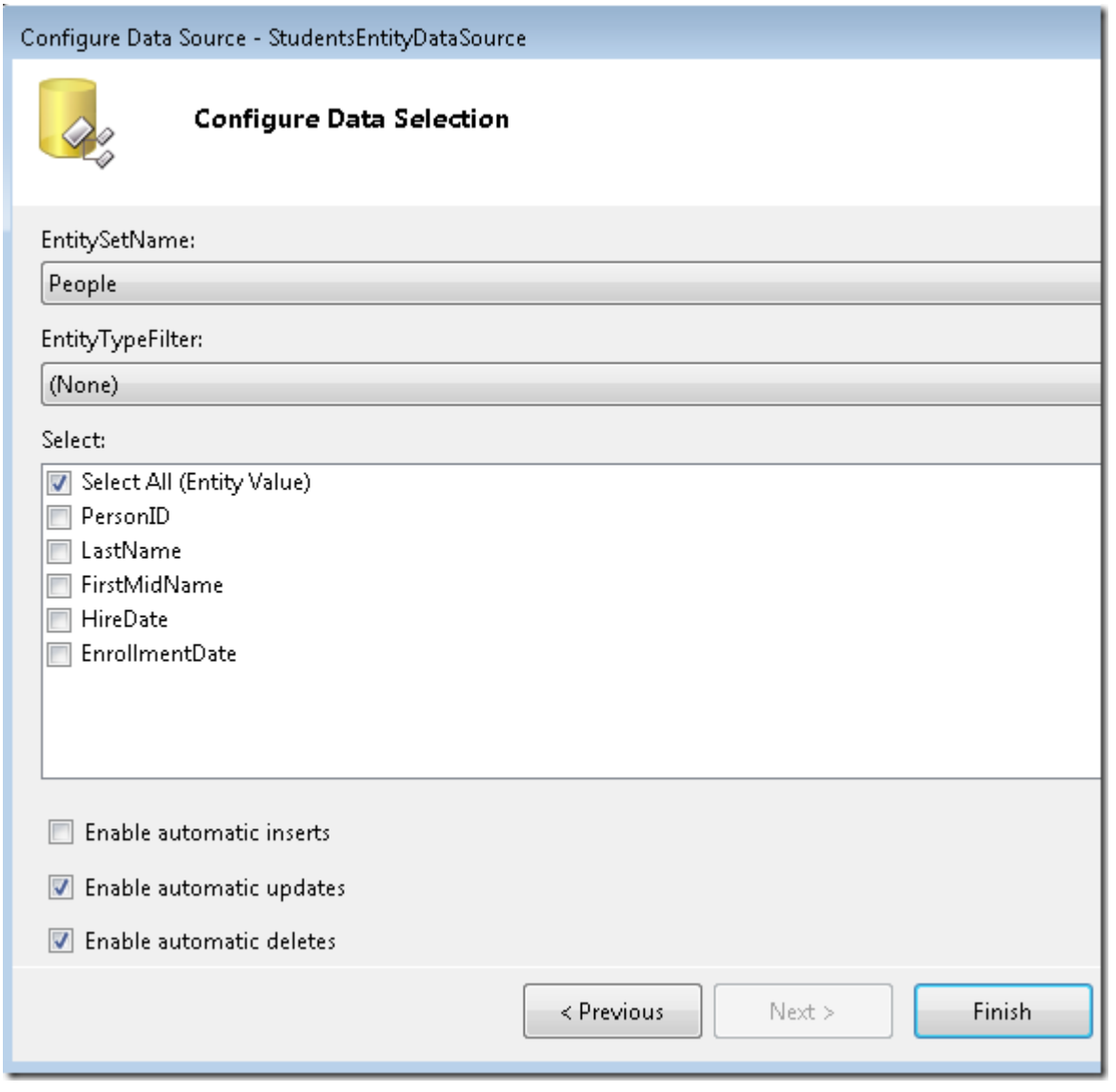
In the **Configure ObjectContext** wizard step, select **SchoolEntities** as the value for **Named Connection**, and select **SchoolEntities** as the **DefaultContainerName** value. Then click **Next**.



Note: If you get the following dialog box at this point, you have to build the project before proceeding.



In the **Configure Data Selection** step, select **People** as the value for **EntitySetName**. Under **Select**, make sure the **Select A**ll check box is selected. Then select the options to enable update and delete. When you're done, click **Finish**.

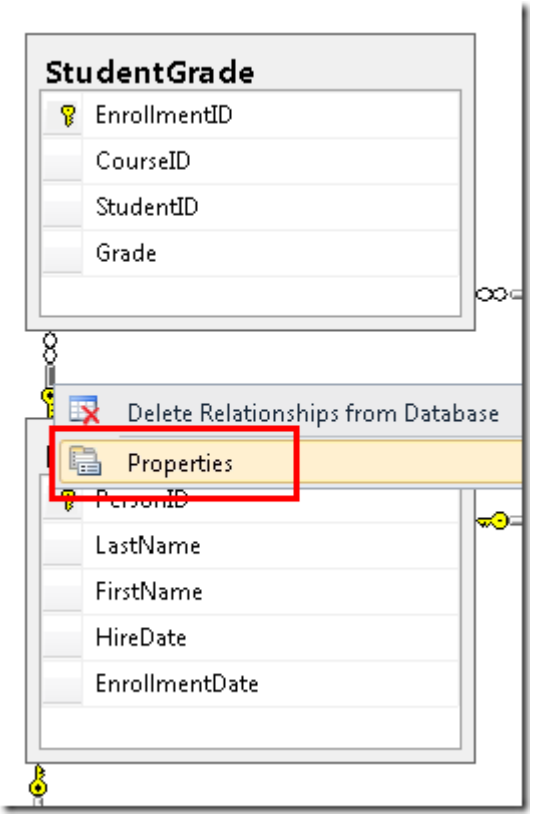


Configuring Database Rules to Allow Deletion

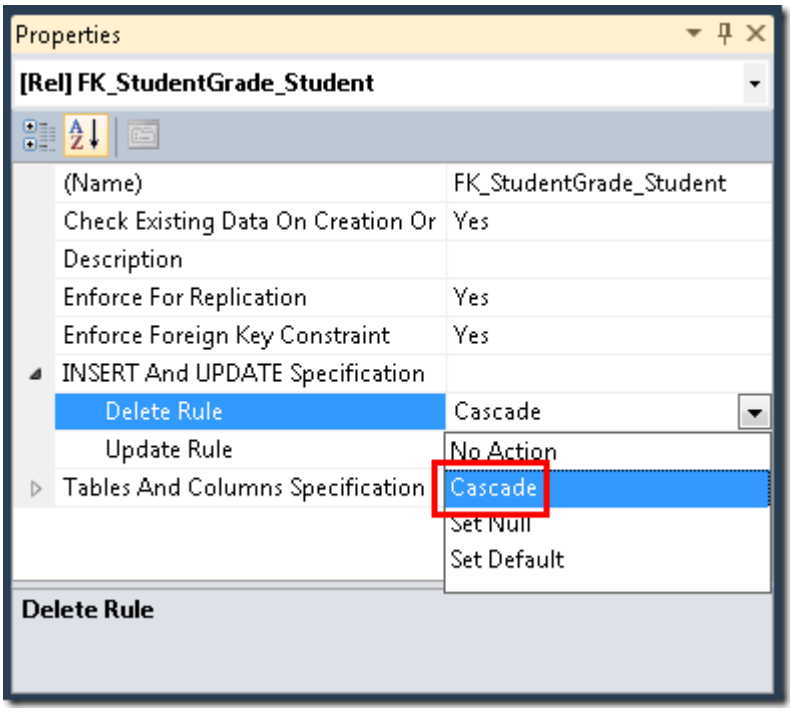
You'll be creating a page that lets users delete students from the `Person` table, which has three relationships with other tables (`Course`, `StudentGrade`, and `OfficeAssignment`). By default, the database will prevent you from deleting a row in `Person` if there are related rows in one of the other tables. You can manually delete the related rows first, or you can configure the database to delete them automatically when you delete a `Person` row. For student records in this tutorial, you'll configure the database to delete the related data automatically. Because students can have related rows only in the `StudentGrade` table, you need to configure only one of the three relationships.

If you're using the *School.mdf* file that you downloaded from the project that goes with this tutorial, you can skip this section because these configuration changes have already been done. If you created the database by running a script, configure the database by performing the following procedures.

In **Server Explorer**, open the database diagram that you created in part 1. Right-click the relationship between `Person` and `StudentGrade` (the line between tables), and then select **Properties**.

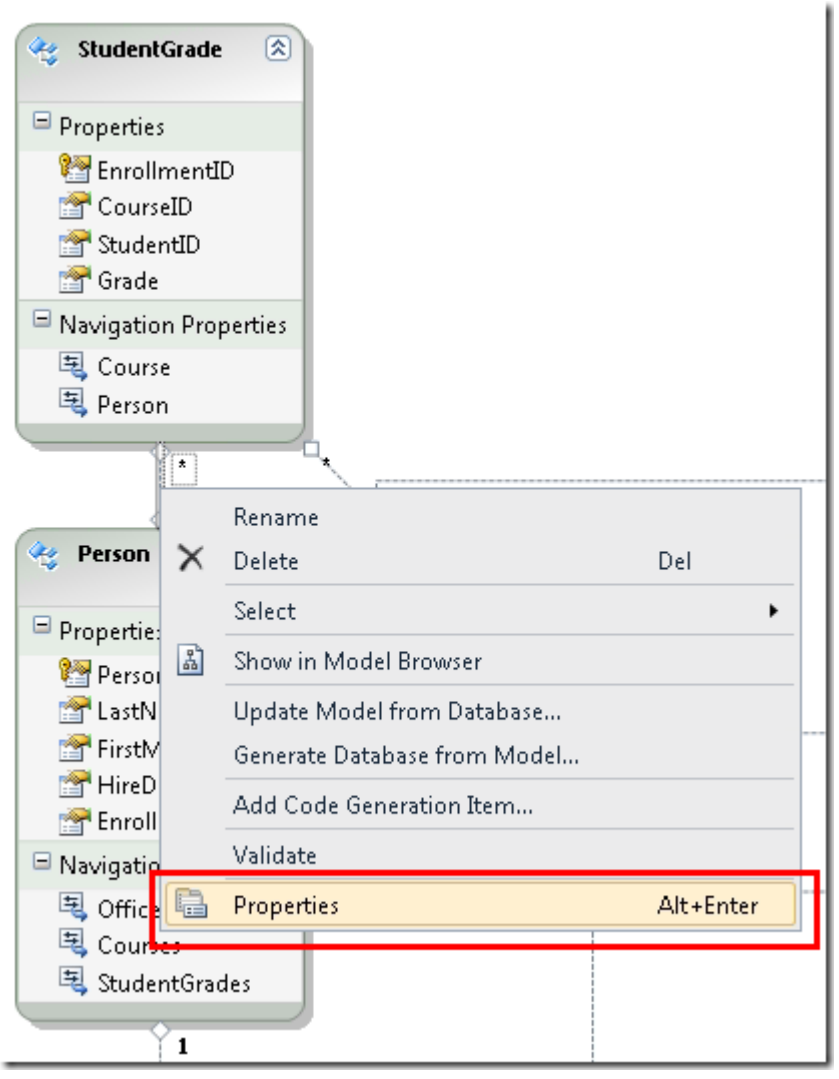


In the **Properties** window, expand **INSERT and UPDATE Specification** and set the **DeleteRule** property to **Cascade**.

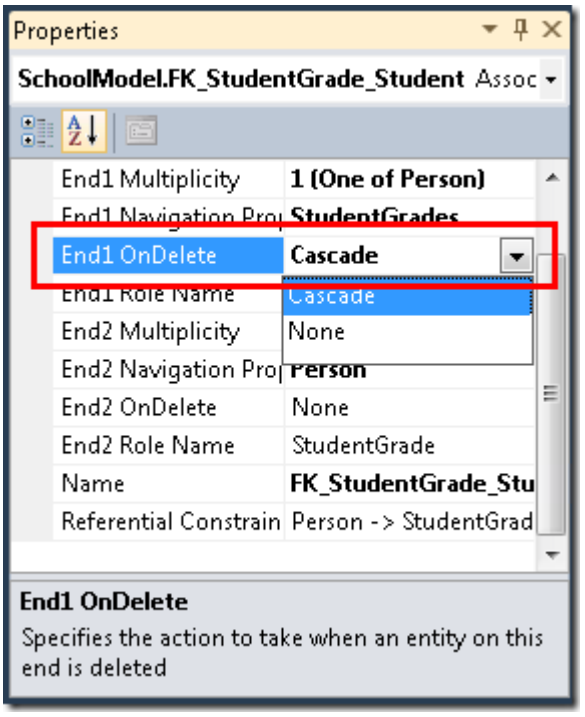


Save and close the diagram. If you're asked whether you want to update the database, click **Yes**.

To make sure that the model keeps entities that are in memory in sync with what the database is doing, you must set corresponding rules in the data model. Open *SchoolModel.edmx*, right-click the association line between `Person` and `StudentGrade`, and then select **Properties**.



In the **Properties** window, set **End1 OnDelete** to **Cascade**.



Save and close the *SchoolModel.edmx* file, and then rebuild the project.

In general, when the database changes, you have several choices for how to sync up the model:

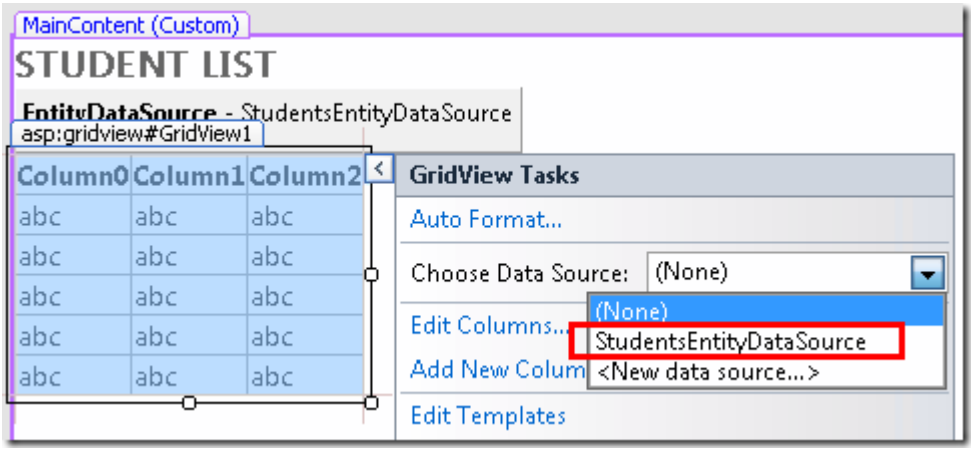
- For certain kinds of changes (such as adding or refreshing tables, views, or stored procedures), right-click in the designer and select **Update Model from Database** to have the designer make the changes automatically.
- Regenerate the data model.
- Make manual updates like this one.

In this case, you could have regenerated the model or refreshed the tables affected by the relationship change, but then you'd have to make the field-name change again (from `FirstName` to `FirstMidName`).

Using a GridView Control to Read and Update Entities

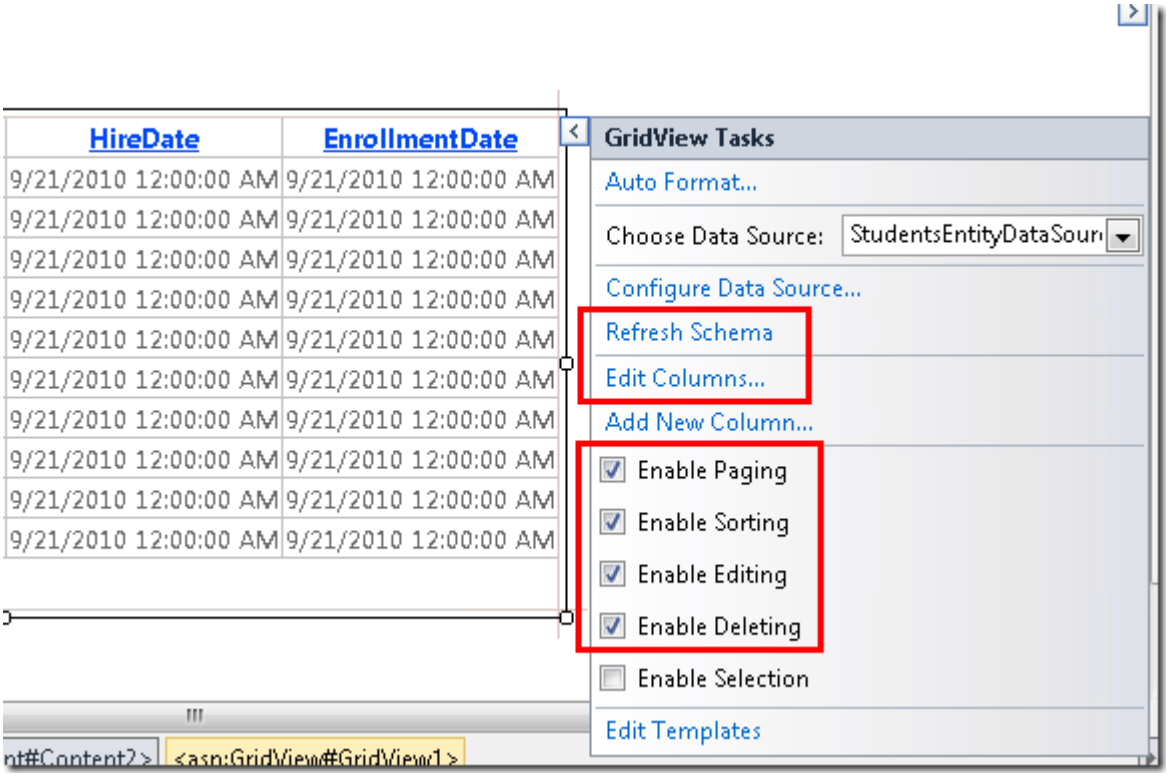
In this section you'll use a `GridView` control to display, update, or delete students.

Open or switch to *Students.aspx* and switch to **Design** view. From the **Data** tab of the **Toolbox**, drag a `GridView` control to the right of the `EntityDataSource` control, name it `StudentsGridView`, click the smart tag, and then select **StudentsEntityDataSource** as the data source.

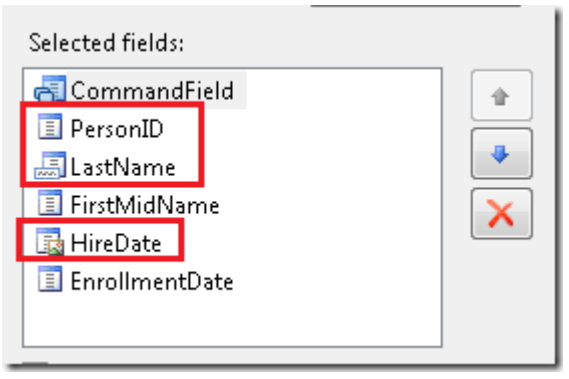


Click **Refresh Schema** (click **Yes** if you're prompted to confirm), then click **Enable Paging**, **Enable Sorting**, **Enable Editing**, and **Enable Deleting**.

Click **Edit Columns**.

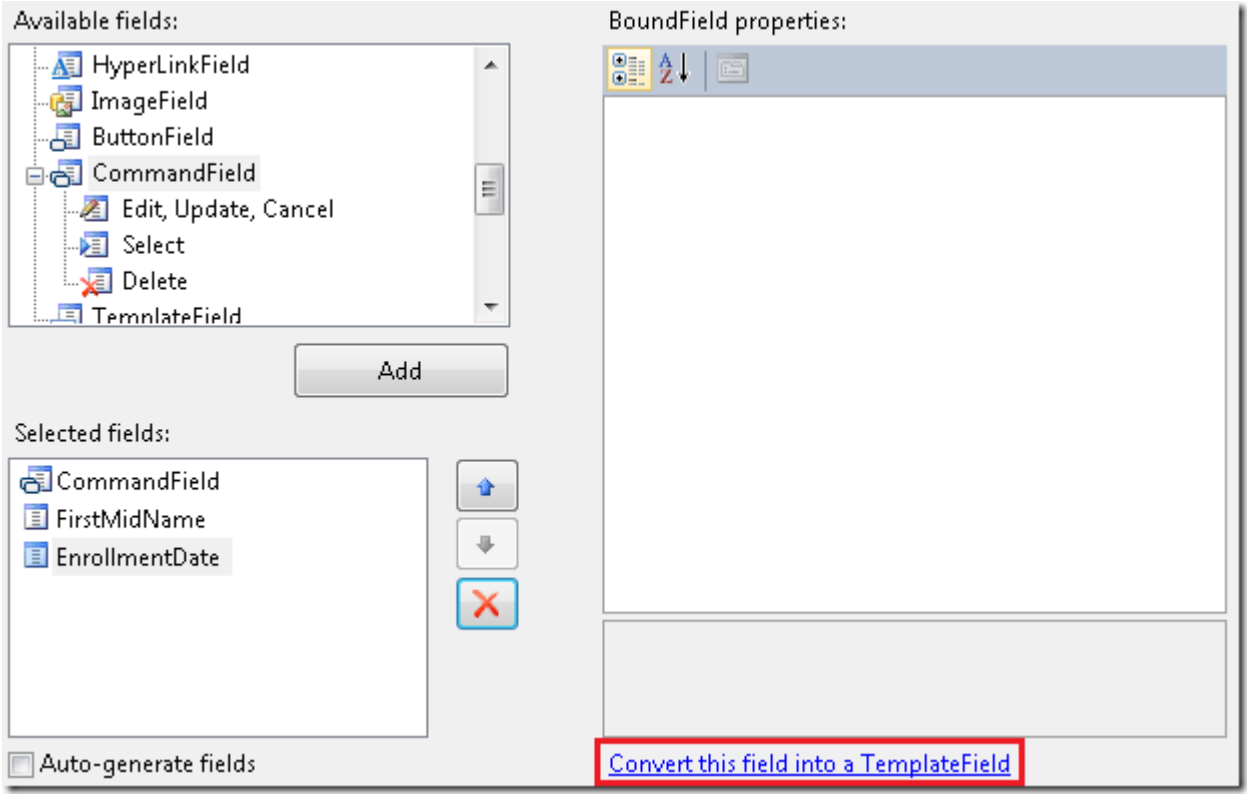


In the **Selected fields** box, delete **PersonID**, **LastName**, and **HireDate**. You typically don't display a record key to users, hire date is not relevant to students, and you'll put both parts of the name in one field, so you only need one of the name fields.)



Select the **FirstMidName** field and then click **Convert this field into a TemplateField**.

Do the same for **EnrollmentDate**.



Click **OK** and then switch to **Source** view. The remaining changes will be easier to do directly in markup. The `GridView` control markup now looks like the following example.

```
<asp:GridViewID="StudentsGridView"runat="server"AllowPaging="True"
AllowSorting="True"AutoGenerateColumns="False"DataKeyNames="PersonID"
DataSourceID="StudentsEntityDataSource">
```

```
<Columns>
<asp:CommandFieldShowDeleteButton="True"ShowEditButton="True"/>
<asp:TemplateFieldHeaderText="FirstMidName"SortExpression="FirstMidName">
<EditItemTemplate>
<asp:TextBox ID="TextBox1" runat="server" Text='<%# Bind("FirstMidName")
%>'></asp:TextBox>
</EditItemTemplate>
<ItemTemplate>
<asp:Label ID="Label1" runat="server" Text='<%# Bind("FirstMidName") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="EnrollmentDate"SortExpression="EnrollmentDate">
<EditItemTemplate>
<asp:TextBox ID="TextBox2" runat="server" Text='<%# Bind("EnrollmentDate")
%>'></asp:TextBox>
</EditItemTemplate>
<ItemTemplate>
<asp:Label ID="Label2" runat="server" Text='<%# Bind("EnrollmentDate")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```

The first column after the command field is a template field that currently displays the first name. Change the markup for this template field to look like the following example:

```
<asp:TemplateFieldHeaderText="Name"SortExpression="LastName">
<EditItemTemplate>
<asp:TextBox ID="LastNameTextBox" runat="server" Text='<%# Bind("LastName")
%>'></asp:TextBox>
<asp:TextBox ID="FirstNameTextBox" runat="server" Text='<%# Bind("FirstMidName")
%>'></asp:TextBox>
</EditItemTemplate>
<ItemTemplate>
<asp:Label ID="LastNameLabel" runat="server" Text='<%# Eval("LastName")
%>'></asp:Label>,
```

```
<asp:Label ID="FirstNameLabel" runat="server" Text='<%# Eval("FirstMidName")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
```

In display mode, two `Label` controls display the first and last name. In edit mode, two text boxes are provided so you can change the first and last name. As with the `Label` controls in display mode, you use `Bind` and `Eval` expressions exactly as you would with ASP.NET data source controls that connect directly to databases. The only difference is that you're specifying entity properties instead of database columns.

The last column is a template field that displays the enrollment date. Change the markup for this field to look like the following example:

```
<asp:TemplateFieldHeaderText="Enrollment Date"SortExpression="EnrollmentDate">
<EditItemTemplate>
<asp:TextBox ID="EnrollmentDateTextBox" runat="server" Text='<%#
Bind("EnrollmentDate", "{0:d}") %>'></asp:TextBox>
</EditItemTemplate>
<ItemTemplate>
<asp:Label ID="EnrollmentDateLabel" runat="server" Text='<%# Eval("EnrollmentDate",
"{0:d}") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
```

In both display and edit mode, the format string "{0,d}" causes the date to be displayed in the "short date" format. (Your computer might be configured to display this format differently from the screen images shown in this tutorial.)

Notice that in each of these template fields, the designer used a `Bind` expression by default, but you've changed that to an `Eval` expression in the `ItemTemplate` elements. The `Bind` expression makes the data available in `GridView` control properties in case you need to access the data in code. In this page you don't need to access this data in code, so you can use `Eval`, which is more efficient. For more information, see Getting your data out of the data controls.

Revising EntityDataSource Control Markup to Improve Performance

In the markup for the `EntityDataSource` control, remove the `ConnectionString` and `DefaultContainerName` attributes and replace them with a `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. This is a change you should make every time you create an `EntityDataSource` control, unless you need to use a connection that is different from the one that's hard-coded in the object context class. Using the `ContextTypeName` attribute provides the following benefits:

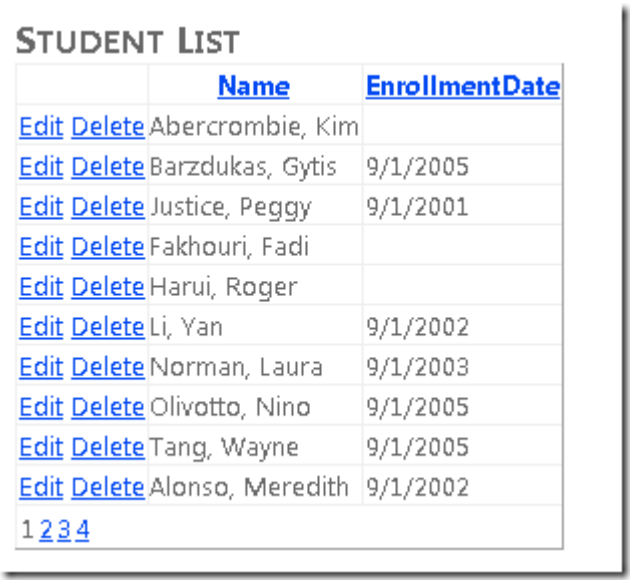
- Better performance. When the `EntityDataSource` control initializes the data model using the `ConnectionString` and `DefaultContainerName` attributes, it performs additional work to load metadata on every request. This isn't necessary if you specify the `ContextTypeName` attribute.
- Lazy loading is turned on by default in generated object context classes (such as `SchoolEntities` in this tutorial) in Entity Framework 4.0. This means that navigation properties are loaded with related data automatically right when you need it. Lazy loading is explained in more detail later in this tutorial.
- Any customizations that you've applied to the object context class (in this case, the `SchoolEntities` class) will be available to controls that use the `EntityDataSource` control. Customizing the object context class is an advanced topic that is not covered in this tutorial series. For more information, see Extending Entity Framework Generated Types.

The markup will now resemble the following example (the order of the properties might be different):

```
<asp:EntityDataSourceID="StudentsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"
EnableDelete="True"EnableUpdate="True">
</asp:EntityDataSource>
```

The `EnableFlattening` attribute refers to a feature that was needed in earlier versions of the Entity Framework because foreign key columns were not exposed as entity properties. The current version makes it possible to use *foreign key associations*, which means foreign key properties are exposed for all but many-to-many associations. If your entities have foreign key properties and no complex types, you can leave this attribute set to `False`. Don't remove the attribute from the markup, because the default value is `True`. For more information, see Flattening Objects (EntityDataSource).

Run the page and you see a list of students and employees (you'll filter for just students in the next tutorial). The first name and last name are displayed together.



To sort the display, click a column name.

Click **Edit** in any row. Text boxes are displayed where you can change the first and last name.

The **Delete** button also works. Click delete for a row that has an enrollment date and the row disappears. (Rows without an enrollment date represent instructors and you may get a referential integrity error. In the next tutorial you'll filter this list to include just students.)

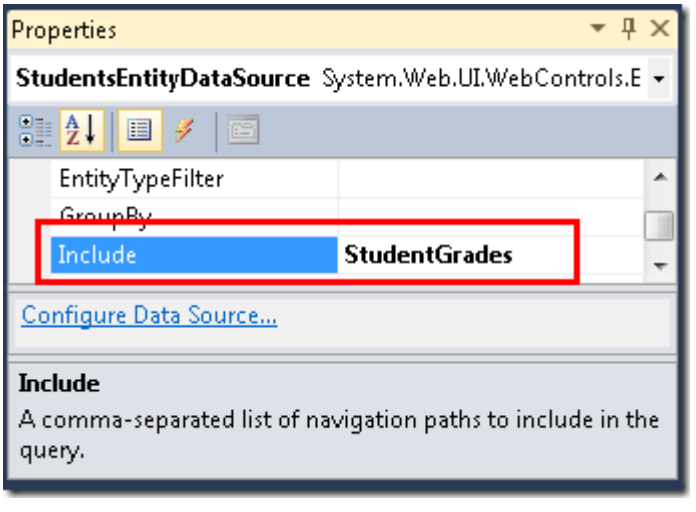
Displaying Data from a Navigation Property

Now suppose you want to know how many courses each student is enrolled in. The Entity Framework provides that information in the `StudentGrades` navigation property of the `Person` entity. Because the database design does not allow a student to be enrolled in a course without having a grade assigned, for this tutorial you can assume that having a row in the `StudentGrade` table row that is associated with a course is the same as being enrolled in the course. (The `Courses` navigation property is only for instructors.)

When you use the `ContextTypeName` attribute of the `EntityDataSource` control, the Entity Framework automatically retrieves information for a navigation property when you access that property. This is called *lazy loading*. However, this can be inefficient, because it results in a separate call to the database each time additional information is needed. If you need data from the navigation property for every entity returned by the `EntityDataSource` control, it's more efficient to retrieve the related data along with the entity itself in a single call to the database. This is called *eager loading*, and you specify eager loading for a navigation property by setting the `Include` property of the `EntityDataSource` control.

In *Students.aspx*, you want to show the number of courses for every student, so eager loading is the best choice. If you were displaying all students but showing the number of courses only for a few of them (which would require writing some code in addition to the markup), lazy loading might be a better choice.

Open or switch to *Students.aspx*, switch to **Design** view, select `StudentsEntityDataSource`, and in the **Properties** window set the **Include** property to **StudentGrades**. (If you wanted to get multiple navigation properties, you could specify their names separated by commas — for example, **StudentGrades, Courses**.)



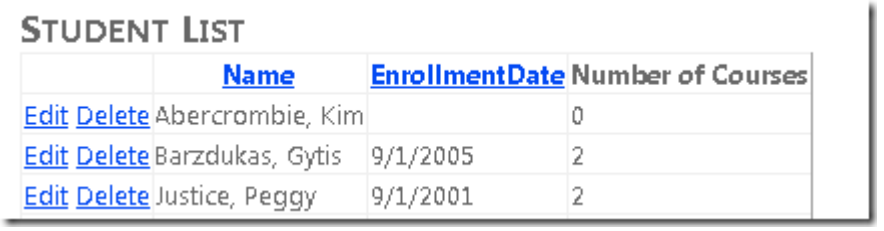
Switch to **Source** view. In the `StudentsGridView` control, after the last `asp:TemplateField` element, add the following new template field:

```
<asp:TemplateFieldHeaderText="Number of Courses">
<ItemTemplate>
```

```
<asp:Label ID="Label1" runat="server" Text='<%# Eval("StudentGrades.Count")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
```

In the `Eval` expression, you can reference the navigation property `StudentGrades`. Because this property contains a collection, it has a `Count` property that you can use to display the number of courses in which the student is enrolled. In a later tutorial you'll see how to display data from navigation properties that contain single entities instead of collections. (Note that you cannot use `BoundField` elements to display data from navigation properties.)

Run the page and you now see how many courses each student is enrolled in.



STUDENT LIST

	Name	EnrollmentDate	Number of Courses
Edit Delete	Abercrombie, Kim		0
Edit Delete	Barzdukas, Gytis	9/1/2005	2
Edit Delete	Justice, Peggy	9/1/2001	2

Using a DetailsView Control to Insert Entities

The next step is to create a page that has a `DetailsView` control that will let you add new students. Close the browser and then create a new web page using the *Site.Master* master page. Name the page *StudentsAdd.aspx*, and then switch to **Source** view.

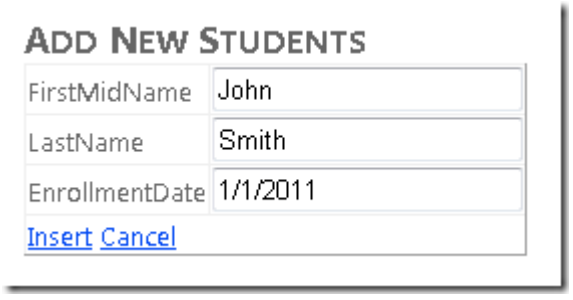
Add the following markup to replace the existing markup for the `Content` control named `Content2`:

```
<asp:ContentID="Content2"ContentPlaceHolderID="MainContent"runat="server">
<h2>Add New Students</h2>
<asp:EntityDataSourceID="StudentsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EnableInsert="True"EntitySetName="People">
</asp:EntityDataSource>
<asp:DetailsViewID="StudentsDetailsView"runat="server"
DataSourceID="StudentsEntityDataSource"AutoGenerateRows="False"
DefaultMode="Insert">
<Fields>
<asp:BoundFieldDataField="FirstMidName"HeaderText="First Name"
```

```
SortExpression="FirstMidName"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"
SortExpression="LastName"/>
<asp:BoundFieldDataField="EnrollmentDate"HeaderText="Enrollment Date"
SortExpression="EnrollmentDate"/>
<asp:CommandFieldShowInsertButton="True"/>
</Fields>
</asp:DetailsView>
</asp:Content>
```

This markup creates an `EntityDataSource` control that is similar to the one you created in *Students.aspx*, except it enables insertion. As with the `GridView` control, the bound fields of the `DetailsView` control are coded exactly as they would be for a data control that connects directly to a database, except that they reference entity properties. In this case, the `DetailsView` control is used only for inserting rows, so you have set the default mode to `Insert`.

Run the page and add a new student.



Nothing will happen after you insert a new student, but if you now run *Students.aspx*, you'll see the new student information.

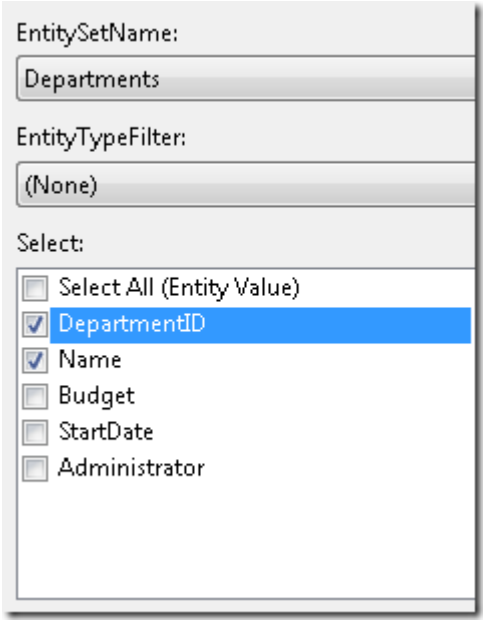
Displaying Data in a Drop-Down List

In the following steps you'll databind a `DropDownList` control to an entity set using an `EntityDataSource` control. In this part of the tutorial, you won't do much with this list. In subsequent parts, though, you'll use the list to let users select a department to display courses associated with the department.

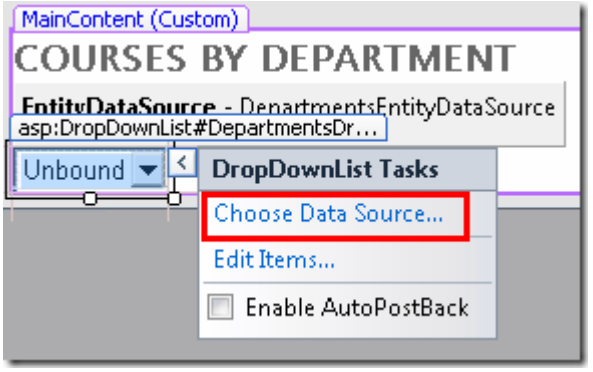
Create a new web page named *Courses.aspx*. In **Source** view, add a heading to the `Content` control that's named `Content2`:

```
<asp:ContentID="Content2"ContentPlaceHolderID="MainContent"runat="server">
<h2>Courses by Department</h2>
</asp:Content>
```

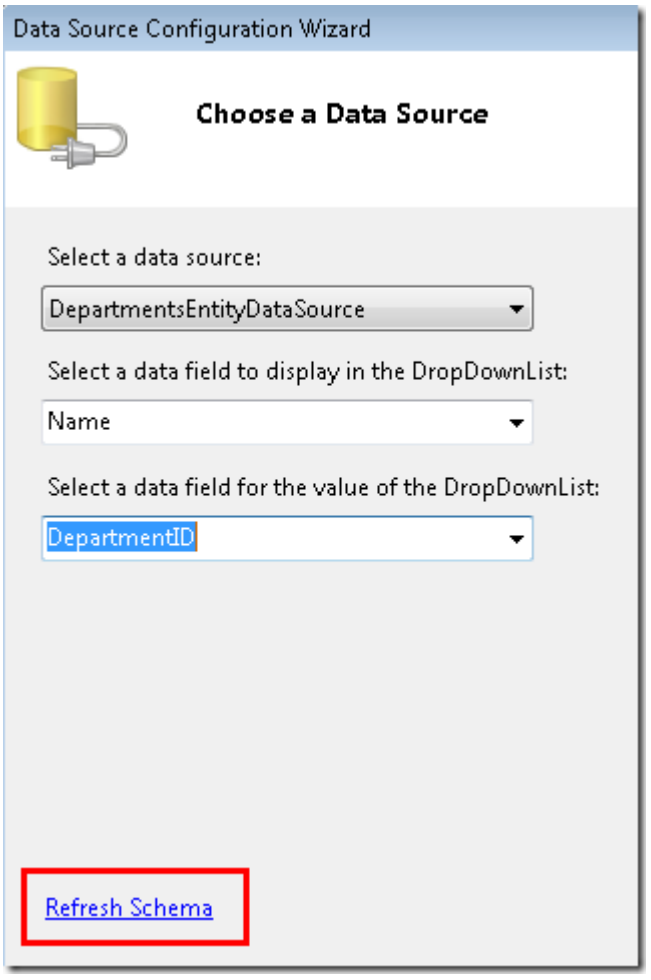
In **Design** view, add an `EntityDataSource` control to the page as you did before, except this time name it `DepartmentsEntityDataSource`. Select **Departments** as the **EntitySetName** value, and select only the **DepartmentID** and **Name** properties.



From the **Standard** tab of the **Toolbox**, drag a `DropDownList` control to the page, name it `DepartmentsDropDownList`, click the smart tag, and select **Choose Data Source** to start the **DataSource Configuration Wizard**.



In the **Choose a Data Source** step, select **DepartmentsEntityDataSource** as the data source, click **Refresh Schema**, and then select **Name** as the data field to display and **DepartmentID** as the value data field. Click **OK**.



The method you use to databind the control using the Entity Framework is the same as with other ASP.NET data source controls except you're specifying entities and entity properties.

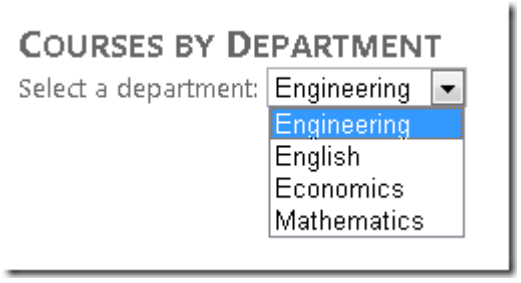
Switch to **Source** view and add "Select a department:" immediately before the `DropDownList` control.

```
Select a department:
<asp:DropDownList ID="DropDownList1" runat="server"
DataSourceID="EntityDataSource1"DataTextField="Name"
DataValueField="DepartmentID">
</asp:DropDownList>
```

As a reminder, change the markup for the `EntityDataSource` control at this point by replacing the `ConnectionString` and `DefaultContainerName` attributes with a `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. It's often best to wait until after you've created the data-bound control that is linked to the data source control before you change the

`EntityDataSource` control markup, because after you make the change, the designer will not provide you with a **Refresh Schema** option in the data-bound control.

Run the page and you can select a department from the drop-down list.

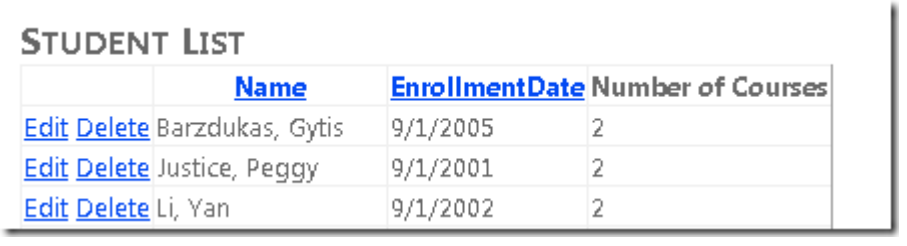


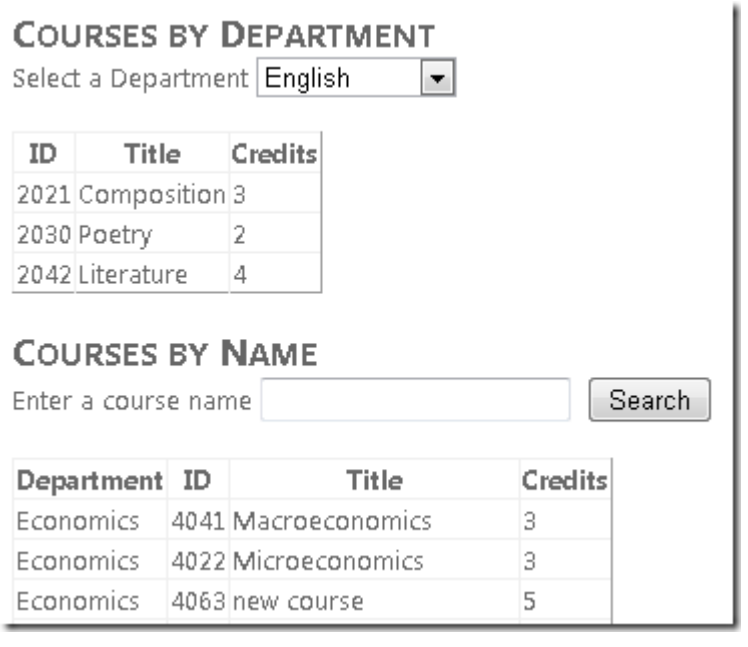
This completes the introduction to using the `EntityDataSource` control. Working with this control is generally no different from working with other ASP.NET data source controls, except that you reference entities and properties instead of tables and columns. The only exception is when you want to access navigation properties. In the next tutorial you'll see that the syntax you use with `EntityDataSource` control might also differ from other data source controls when you filter, group, and order data.

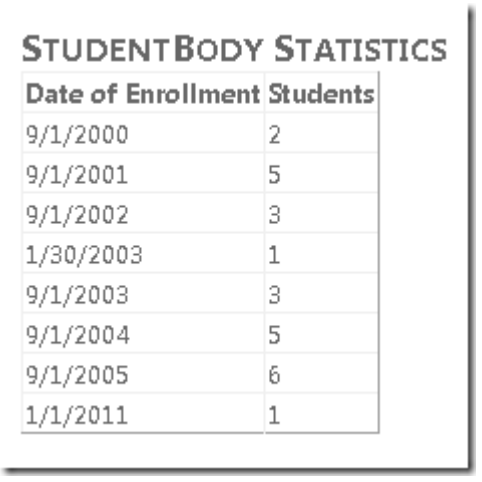
Filtering, Ordering, and Grouping Data

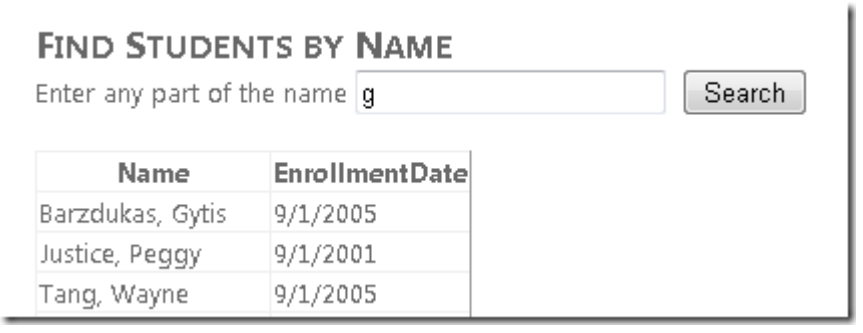
In the previous tutorial you used the `EntityDataSource` control to display and edit data. In this tutorial you'll filter, order, and group data. When you do this by setting properties of the `EntityDataSource` control, the syntax is different from other data source controls. As you'll see, however, you can use the `QueryExtender` control to minimize these differences.

You'll change the *Students.aspx* page to filter for students, sort by name, and search on name. You'll also change the *Courses.aspx* page to display courses for the selected department and search for courses by name. Finally, you'll add student statistics to the *About.aspx* page.





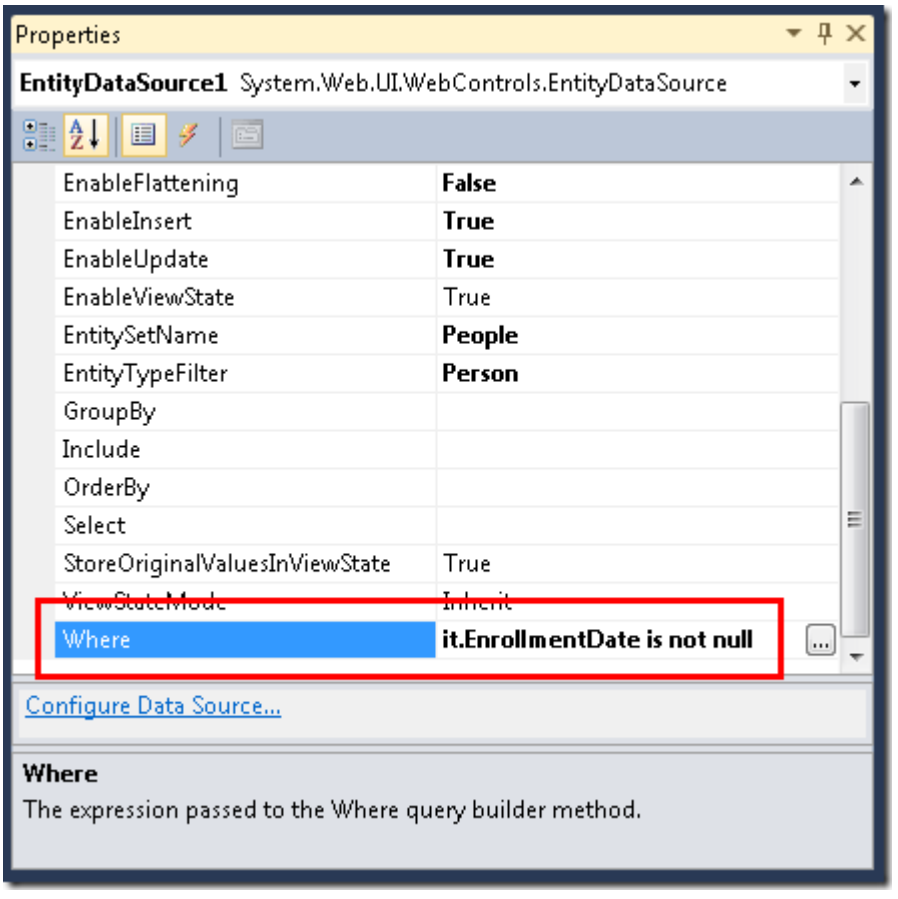




Using the EntityDataSource "Where" Property to Filter Data

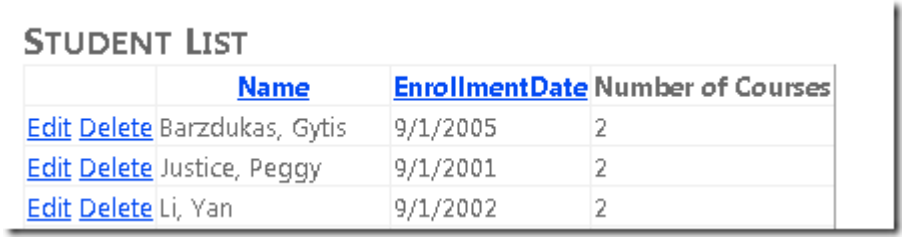
Open the *Students.aspx* page that you created in the previous tutorial. As currently configured, the `GridView` control in the page displays all the names from the `People` entity set. However, you want to show only students, which you can find by selecting `Person` entities that have non-null enrollment dates.

Switch to **Design** view and select the `EntityDataSource` control. In the **Properties** window, set the `Where` property to `it.EnrollmentDate is not null`.



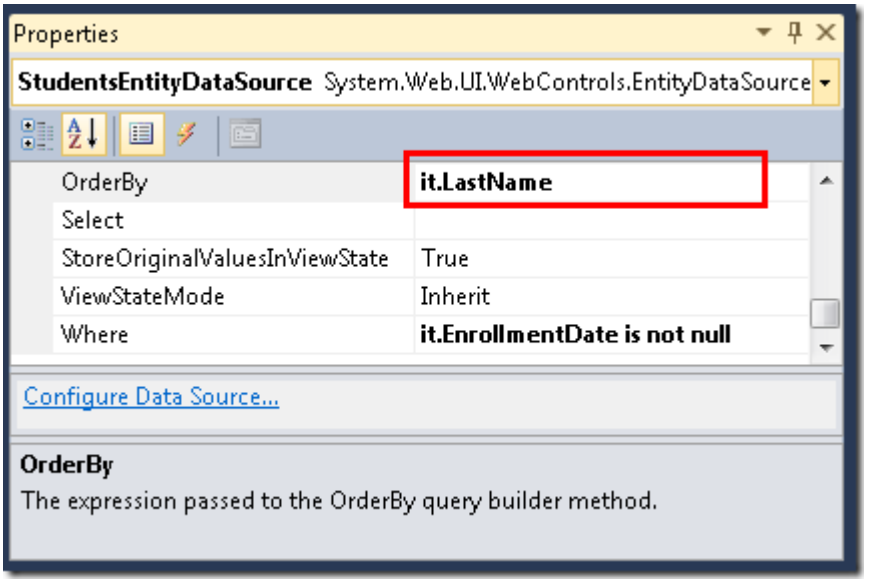
The syntax you use in the `Where` property of the `EntityDataSource` control is Entity SQL. Entity SQL is similar to Transact-SQL, but it's customized for use with entities rather than database objects. In the expression `it.EnrollmentDate is not null`, the word `it` represents a reference to the entity returned by the query. Therefore, `it.EnrollmentDate` refers to the `EnrollmentDate` property of the `Person` entity that the `EntityDataSource` control returns.

Run the page. The students list now contains only students. (There are no rows displayed where there's no enrollment date.)

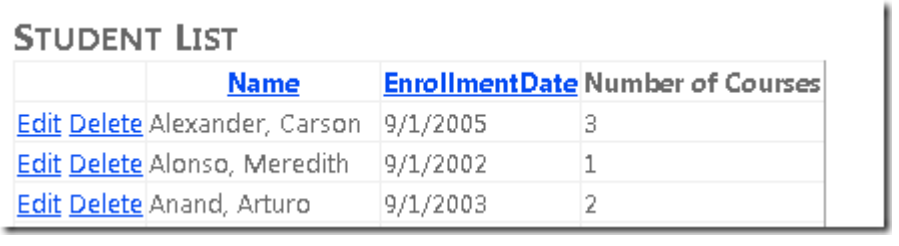


Using the EntityDataSource "OrderBy" Property to Order Data

You also want this list to be in name order when it's first displayed. With the *Students.aspx* page still open in **Design** view, and with the `EntityDataSource` control still selected, in the **Properties** window set the **OrderBy** property to `it.LastName`.



Run the page. The students list is now in order by last name.

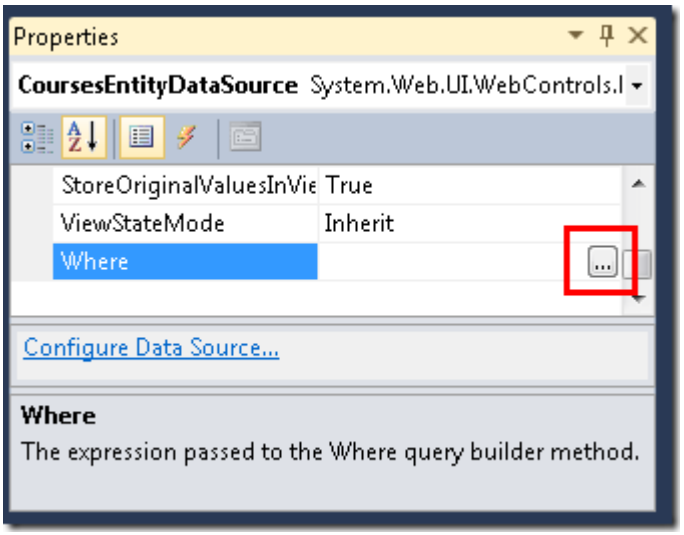


Using a Control Parameter to Set the "Where" Property

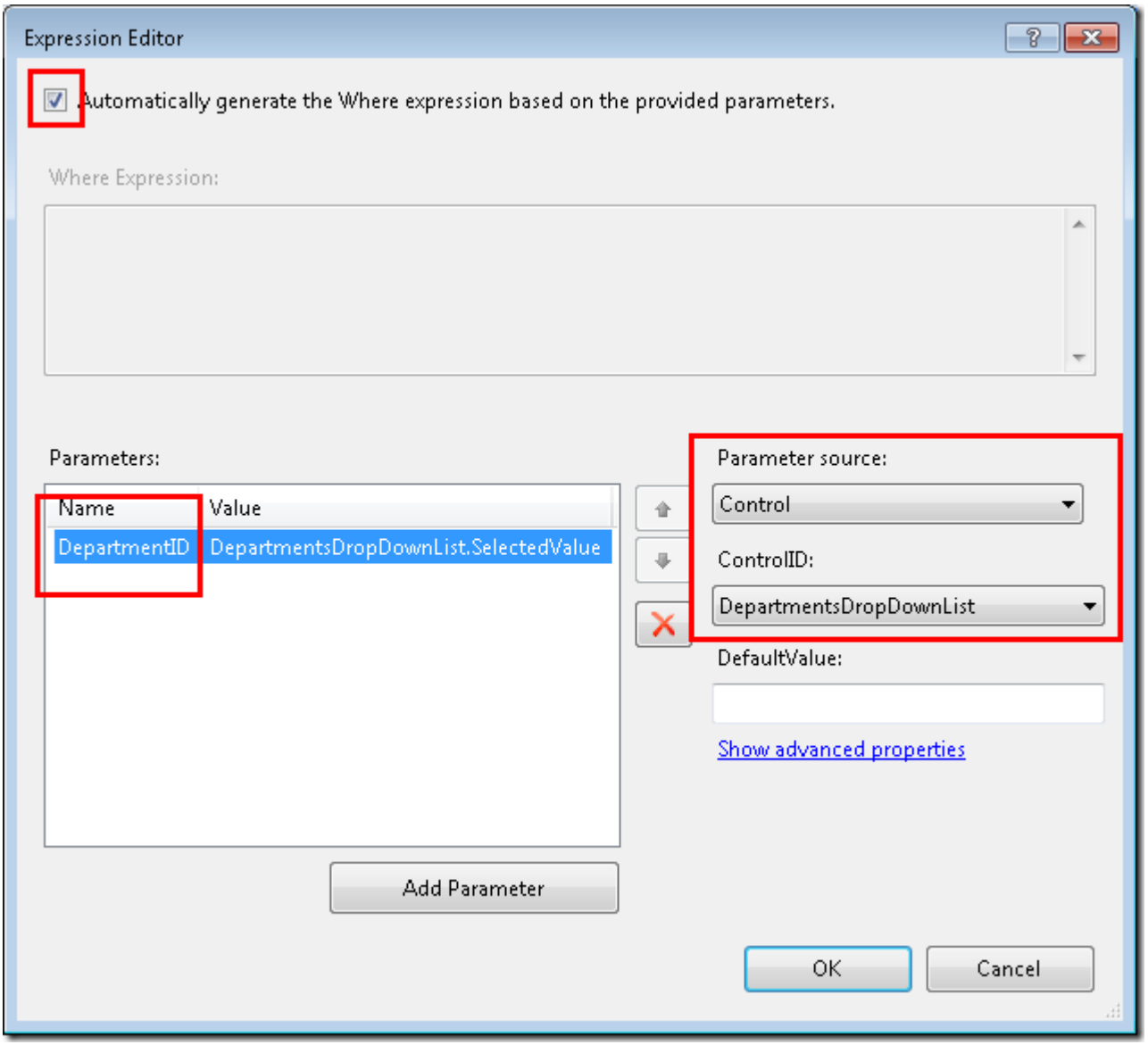
As with other data source controls, you can pass parameter values to the `Where` property. On the *Courses.aspx* page that you created in part 2 of the tutorial, you can use this method to display courses that are associated with the department that a user selects from the drop-down list.

Open *Courses.aspx* and switch to **Design** view. Add a second `EntityDataSource` control to the page, and name it `CoursesEntityDataSource`. Connect it to the `SchoolEntities` model, and select `Courses` as the **EntitySetName** value.

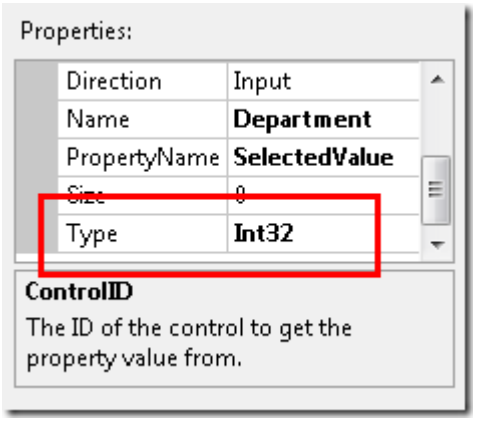
In the **Properties** window, click the ellipsis in the **Where** property box. (Make sure the `CoursesEntityDataSource` control is still selected before using the **Properties** window.)



The **Expression Editor** dialog box is displayed. In this dialog box, select **Automatically generate the Where expression based on the provided parameters**, and then click **Add Parameter**. Name the parameter `DepartmentID`, select **Control** as the **Parameter source** value, and select **DepartmentsDropDownList** as the **ControlID** value.



Click **Show advanced properties**, and in the **Properties** window of the **Expression Editor** dialog box, change the `Type` property to `Int32`.

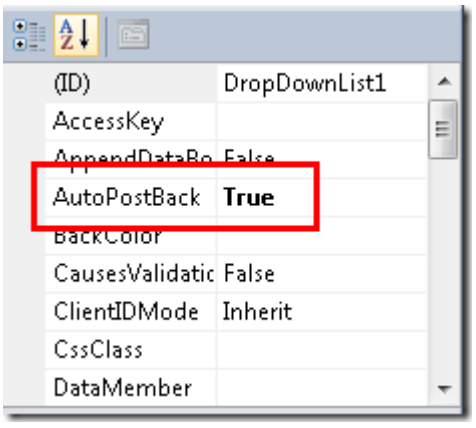


When you're done, click **OK**.

Below the drop-down list, add a `GridView` control to the page and name it `CoursesGridView`. Connect it to the `CoursesEntityDataSource` data source control, click **Refresh Schema**, click **Edit Columns**, and remove the `DepartmentID` column. The `GridView` control markup resembles the following example.

```
<asp:GridViewID="CoursesGridView"runat="server"AutoGenerateColumns="False"
DataKeyNames="CourseID"DataSourceID="CoursesEntityDataSource">
<Columns>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"ReadOnly="True"
SortExpression="CourseID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"SortExpression="Title"/>
<asp:BoundFieldDataField="Credits"HeaderText="Credits"
SortExpression="Credits"/>
</Columns>
</asp:GridView>
```

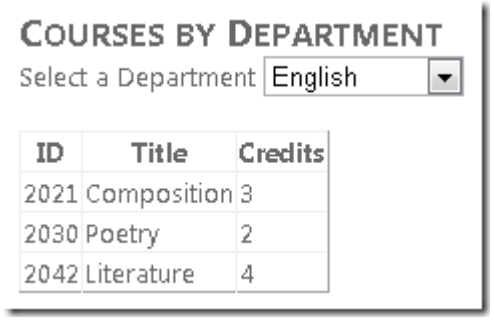
When the user changes the selected department in the drop-down list, you want the list of associated courses to change automatically. To make this happen, select the drop-down list, and in the **Properties** window set the `AutoPostBack` property to `True`.



Now that you're finished using the designer, switch to **Source** view and replace the `ConnectionString` and `DefaultContainer` name properties of the `CoursesEntityDataSource` control with the `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. When you're done, the markup for the control will look like the following example.

```
<asp:EntityDataSourceID="CoursesEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="false"
EntitySetName="Courses"
AutoGenerateWhereClause="true"Where="">
<WhereParameters>
<asp:ControlParameterControlID="DepartmentsDropDownList"Type="Int32"
Name="DepartmentID"PropertyName="SelectedValue"/>
</WhereParameters>
</asp:EntityDataSource>
```

Run the page and use the drop-down list to select different departments. Only courses that are offered by the selected department are displayed in the `GridView` control.



Using the EntityDataSource "GroupBy" Property to Group Data

Suppose Contoso University wants to put some student-body statistics on its About page. Specifically, it wants to show a breakdown of numbers of students by the date they enrolled.

Open *About.aspx*, and in **Source** view, replace the existing contents of the `BodyContent` control with "Student Body Statistics" between `h2` tags:

```
<asp:ContentID="BodyContent"runat="server"ContentPlaceHolderID="MainContent">
<h2>Student Body Statistics</h2>
</asp:Content>
```

After the heading, add an `EntityDataSource` control and name it `StudentStatisticsEntityDataSource`. Connect it to `SchoolEntities`, select the `People` entity set, and leave the **Select** box in the wizard unchanged. Set the following properties in the **Properties** window:

- To filter for students only, set the `Where` property to `it.EnrollmentDate is not null`.
- To group the results by the enrollment date, set the `GroupBy` property to `it.EnrollmentDate`.
- To select the enrollment date and the number of students, set the `Select` property to `it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents`.
- To order the results by the enrollment date, set the `OrderBy` property to `it.EnrollmentDate`.

In **Source** view, replace the `ConnectionString` and `DefaultContainer` name properties with a `ContextTypeName` property. The `EntityDataSource` control markup now resembles the following example.

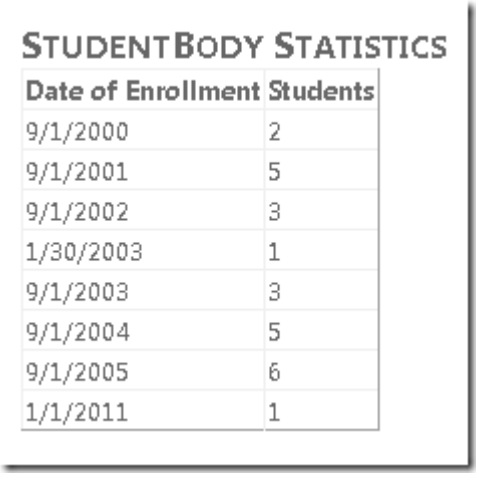
```
<asp:EntityDataSourceID="StudentStatisticsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"
Select="it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents"
OrderBy="it.EnrollmentDate"GroupBy="it.EnrollmentDate"
Where="it.EnrollmentDate is not null">
</asp:EntityDataSource>
```

The syntax of the `Select`, `GroupBy`, and `Where` properties resembles Transact-SQL except for the `it` keyword that specifies the current entity.

Add the following markup to create a `GridView` control to display the data.

```
<asp:GridViewID="StudentStatisticsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="StudentStatisticsEntityDataSource">
<Columns>
<asp:BoundFieldDataField="EnrollmentDate"DataFormatString="{0:d}"
HeaderText="Date of Enrollment"
ReadOnly="True"SortExpression="EnrollmentDate"/>
<asp:BoundFieldDataField="NumberOfStudents"HeaderText="Students"
ReadOnly="True"SortExpression="NumberOfStudents"/>
</Columns>
</asp:GridView>
```

Run the page to see a list showing the number of students by enrollment date.



STUDENT BODY STATISTICS

Date of Enrollment	Students
9/1/2000	2
9/1/2001	5
9/1/2002	3
1/30/2003	1
9/1/2003	3
9/1/2004	5
9/1/2005	6
1/1/2011	1

Using the QueryExtender Control for Filtering and Ordering

The `QueryExtender` control provides a way to specify filtering and sorting in markup. The syntax is independent of the database management system (DBMS) you're using. It's also generally independent of the Entity Framework, with the exception that syntax you use for navigation properties is unique to the Entity Framework.

In this part of the tutorial you'll use a `QueryExtender` control to filter and order data, and one of the order-by fields will be a navigation property.

(If you prefer to use code instead of markup to extend the queries that are automatically generated by the `EntityDataSource` control, you can do that by handling the `QueryCreated` event. This is how the `QueryExtender` control extends `EntityDataSource` control queries also.)

Open the *Courses.aspx* page, and below the markup you added previously, insert the following markup to create a heading, a text box for entering search strings, a search button, and an `EntityDataSource` control that's bound to the `Courses` entity set.

```
<h2>Courses by Name</h2>
    Enter a course name
<asp:TextBoxID="SearchTextBox"runat="server"></asp:TextBox>
<asp:ButtonID="SearchButton"runat="server"Text="Search"/>
<br/><br/>
<asp:EntityDataSourceID="SearchEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="Courses"
Include="Department">
</asp:EntityDataSource>
```

Notice that the `EntityDataSource` control's `Include` property is set to `Department`. In the database, the `Course` table does not contain the department name; it contains a `DepartmentID` foreign key column. If you were querying the database directly, to get the department name along with course data, you would have to join the `Course` and `Department` tables. By setting the `Include` property to `Department`, you specify that the Entity Framework should do the work of getting the related `Department` entity when it gets a `Course` entity. The `Department` entity is then stored in the `Department` navigation property of the `Course` entity. (By default, the `SchoolEntities` class that was generated by the data model designer retrieves related data when it's needed, and you've bound the data source control to that class, so setting the `Include` property is not necessary. However, setting it improves performance of the page, because otherwise the Entity Framework would make separate calls to the database to retrieve data for the `Course` entities and for the related `Department` entities.)

After the `EntityDataSource` control you just created, insert the following markup to create a `QueryExtender` control that's bound to that `EntityDataSource` control.

```
<asp:QueryExtenderID="SearchQueryExtender"runat="server"
TargetControlID="SearchEntityDataSource">
<asp:SearchExpressionSearchType="StartsWith"DataFields="Title">
<asp:ControlParameterControlID="SearchTextBox"/>
</asp:SearchExpression>
<asp:OrderByExpressionDataField="Department.Name"Direction="Ascending">
<asp:ThenByDataField="Title"Direction="Ascending"/>
```

```
</asp:OrderByExpression>
</asp:QueryExtender>
```

The `SearchExpression` element specifies that you want to select courses whose titles match the value entered in the text box. Only as many characters as are entered in the text box will be compared, because the `SearchType` property specifies `StartsWith`.

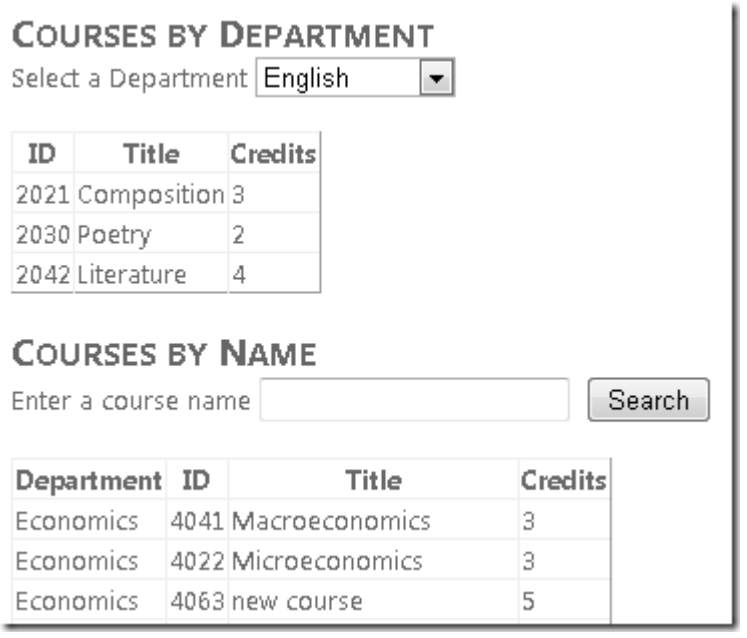
The `OrderByExpression` element specifies that the result set will be ordered by course title within department name. Notice how department name is specified: `Department.Name`. Because the association between the `Course` entity and the `Department` entity is one-to-one, the `Department` navigation property contains a `Department` entity. (If this were a one-to-many relationship, the property would contain a collection.) To get the department name, you must specify the `Name` property of the `Department` entity.

Finally, add a `GridView` control to display the list of courses:

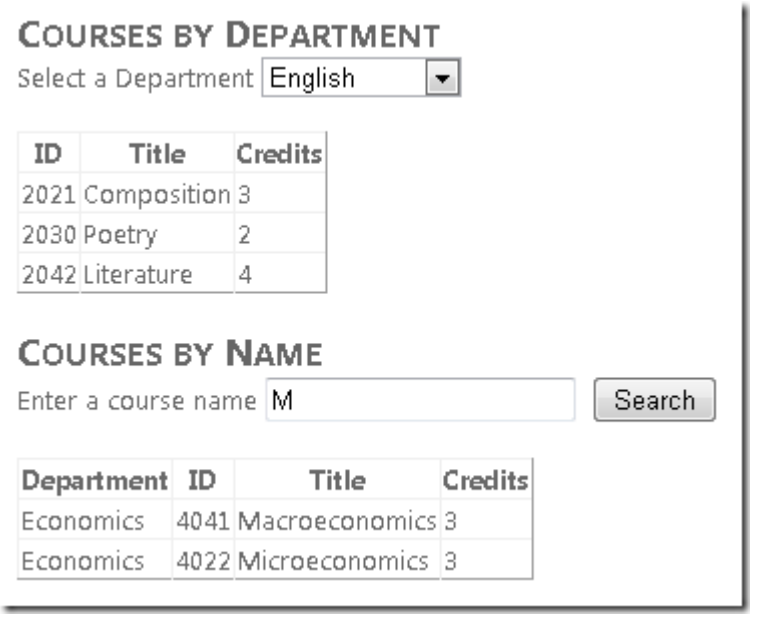
```
<asp:GridViewID="SearchGridView"runat="server"AutoGenerateColumns="False"
DataKeyNames="CourseID"DataSourceID="SearchEntityDataSource"AllowPaging="true">
<Columns>
<asp:TemplateFieldHeaderText="Department">
<ItemTemplate>
<asp:Label ID="Label2" runat="server" Text='<%# Eval("Department.Name")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"/>
<asp:BoundFieldDataField="Credits"HeaderText="Credits"/>
</Columns>
</asp:GridView>
```

The first column is a template field that displays the department name. The databinding expression specifies `Department.Name`, just as you saw in the `QueryExtender` control.

Run the page. The initial display shows a list of all courses in order by department and then by course title.



Enter an "m" and click **Search** to see all courses whose titles begin with "m" (the search is not case sensitive).



Using the "Like" Operator to Filter Data

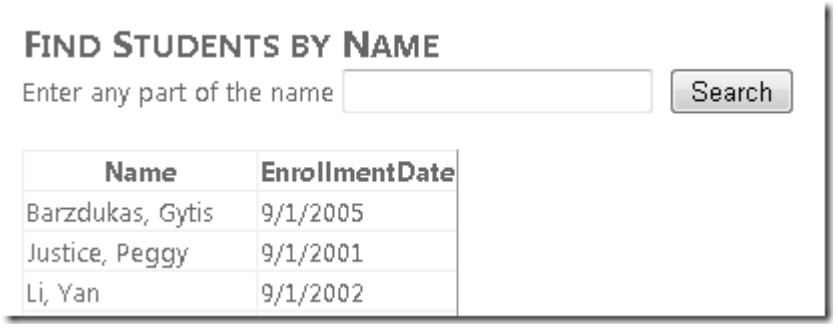
You can achieve an effect similar to the `QueryExtender` control's `StartsWith`, `Contains`, and `EndsWith` search types by using a `Like` operator in the `EntityDataSource` control's `Where` property. In this part of the tutorial, you'll see how to use the `Like` operator to search for a student by name.

Open *Students.aspx* in **Source** view. After the `GridView` control, add the following markup:

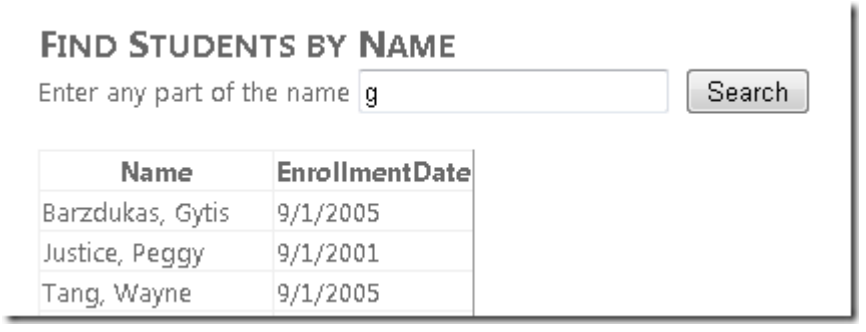
```
<h2>Find Students by Name</h2>
    Enter any part of the name
<asp:TextBoxID="SearchTextBox"runat="server"AutoPostBack="true"></asp:TextBox>
<asp:ButtonID="SearchButton"runat="server"Text="Search"/>
<br/>
<br/>
<asp:EntityDataSourceID="SearchEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"
Where="it.EnrollmentDate is not null and (it.FirstMidName Like '%' + @StudentName +
'%' or it.LastName Like '%' + @StudentName + '%')">
<WhereParameters>
<asp:ControlParameterControlID="SearchTextBox"Name="StudentName"PropertyName="Text"
Type="String"DefaultValue="%"/>
</WhereParameters>
</asp:EntityDataSource>
<asp:GridViewID="SearchGridView"runat="server"AutoGenerateColumns="False"DataKeyNames
="PersonID"
DataSourceID="SearchEntityDataSource"AllowPaging="true">
<Columns>
<asp:TemplateFieldHeaderText="Name"SortExpression="LastName, FirstMidName">
<ItemTemplate>
<asp:Label ID="LastNameFoundLabel" runat="server" Text='<%# Eval("LastName")
%>'></asp:Label>,
<asp:Label ID="FirstNameFoundLabel" runat="server" Text='<%# Eval("FirstMidName")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Enrollment Date"SortExpression="EnrollmentDate">
<ItemTemplate>
<asp:Label ID="EnrollmentDateFoundLabel" runat="server" Text='<%#
Eval("EnrollmentDate", "{0:d}") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```

This markup is similar to what you've seen earlier except for the `Where` property value. The second part of the `Where` expression defines a substring search (`LIKE %FirstMidName% or LIKE %LastName%`) that searches both the first and last names for whatever is entered in the text box.

Run the page. Initially you see all of the students because the default value for the `StudentName` parameter is "%".



Enter the letter "g" in the text box and click **Search**. You see a list of students that have a "g" in either the first or last name.



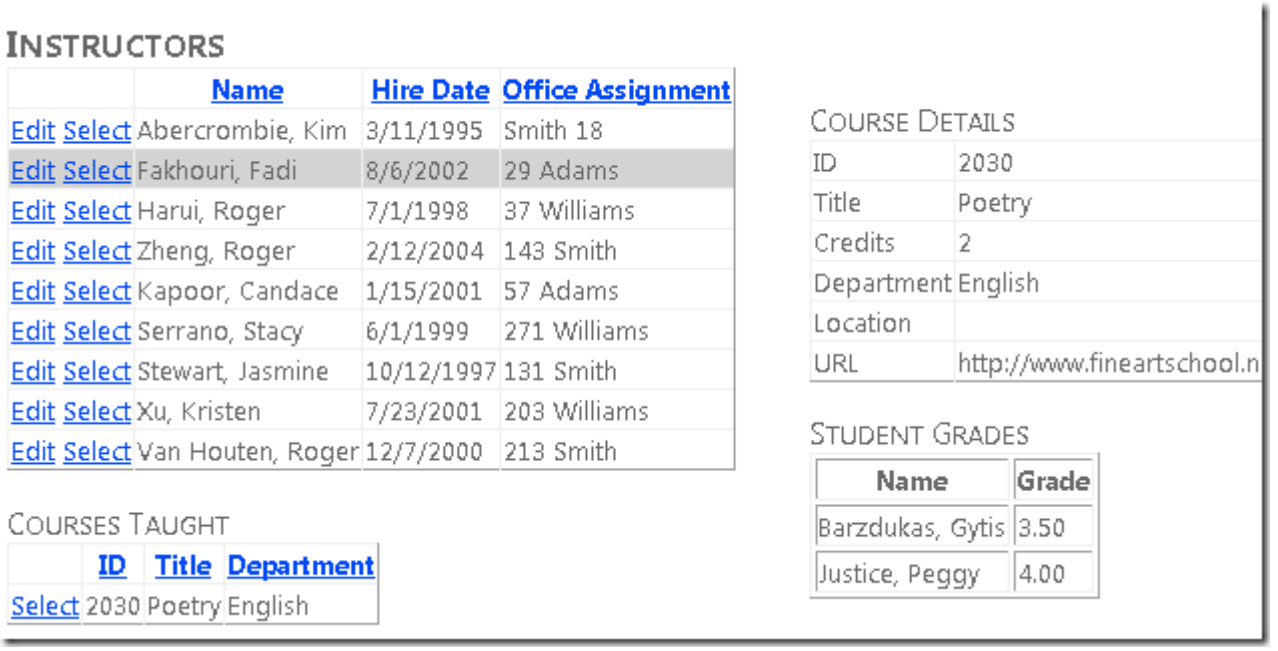
You've now displayed, updated, filtered, ordered, and grouped data from individual tables. In the next tutorial you'll begin to work with related data (master-detail scenarios).

Working with Related Data

In the previous tutorial you used the `EntityDataSource` control to filter, sort, and group data. In this tutorial you'll display and update related data.

You'll create the Instructors page that shows a list of instructors. When you select an instructor, you see a list of courses taught by that instructor. When you select a course, you see details for the course and a list of students enrolled in the course. You can edit the instructor name, hire date, and office assignment. The office assignment is a separate entity set that you access through a navigation property.

You can link master data to detail data in markup or in code. In this part of the tutorial, you'll use both methods.



Displaying and Updating Related Entities in a GridView Control

Create a new web page named *Instructors.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Instructors</h2>
<div>
<asp:EntityDataSourceID="InstructorsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"
Where="it.HireDate is not null"Include="OfficeAssignment"EnableUpdate="True">
</asp:EntityDataSource>
</div>
```

This markup creates an `EntityDataSource` control that selects instructors and enables updates. The `div` element configures markup to render on the left so that you can add a column on the right later.

Between the `EntityDataSource` markup and the closing `</div>` tag, add the following markup that creates a `GridView` control and a `Label` control that you'll use for error messages:

```
<asp:GridViewID="InstructorsGridView"runat="server"AllowPaging="True"AllowSorting="Tr
ue"
AutoGenerateColumns="False"DataKeyNames="PersonID"DataSourceID="InstructorsEntityData
Source"
OnSelectedIndexChanged="InstructorsGridView_SelectedIndexChanged"
SelectedRowStyle-BackColor="LightGray"
onrowupdating="InstructorsGridView_RowUpdating">
<Columns>
<asp:CommandFieldShowSelectButton="True"ShowEditButton="True"/>
<asp:TemplateFieldHeaderText="Name"SortExpression="LastName">
<ItemTemplate>
<asp:Label ID="InstructorLastNameLabel" runat="server" Text='<%# Eval("LastName")
%>'></asp:Label>,
<asp:Label ID="InstructorFirstNameLabel" runat="server" Text='<%#
Eval("FirstMidName") %>'></asp:Label>
</ItemTemplate>
<EditItemTemplate>
<asp:TextBox ID="InstructorLastNameTextBox" runat="server" Text='<%#
Bind("FirstMidName") %>' Width="7em"></asp:TextBox>
<asp:TextBox ID="InstructorFirstNameTextBox" runat="server" Text='<%#
Bind("LastName") %>' Width="7em"></asp:TextBox>
</EditItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Hire Date"SortExpression="HireDate">
<ItemTemplate>
<asp:Label ID="InstructorHireDateLabel" runat="server" Text='<%# Eval("HireDate",
"{0:d}") %>'></asp:Label>
</ItemTemplate>
<EditItemTemplate>
<asp:TextBox ID="InstructorHireDateTextBox" runat="server" Text='<%# Bind("HireDate",
"{0:d}") %>' Width="7em"></asp:TextBox>
</EditItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Office
Assignment"SortExpression="OfficeAssignment.Location">
<ItemTemplate>
<asp:Label ID="InstructorOfficeLabel" runat="server" Text='<%#
Eval("OfficeAssignment.Location") %>'></asp:Label>
```

```
</ItemTemplate>
<EditItemTemplate>
<asp:TextBox ID="InstructorOfficeTextBox" runat="server"
                        Text='<%# Eval("OfficeAssignment.Location") %>' Width="7em"
                        oninit="InstructorOfficeTextBox_Init"></asp:TextBox>
</EditItemTemplate>
</asp:TemplateField>
</Columns>
<SelectedRowStyleBackColor="LightGray"></SelectedRowStyle>
</asp:GridView>
<asp:LabelID="ErrorMessageLabel"runat="server"Text=""Visible="false"ViewStateMode="Di
sabled"></asp:Label>
```

This `GridView` control enables row selection, highlights the selected row with a light gray background color, and specifies handlers (which you'll create later) for the `SelectedIndexChanged` and `Updating` events. It also specifies `PersonID` for the `DataKeyNames` property, so that the key value of the selected row can be passed to another control that you'll add later.

The last column contains the instructor's office assignment, which is stored in a navigation property of the `Person` entity because it comes from an associated entity. Notice that the `EditItemTemplate` element specifies `Eval` instead of `Bind`, because the `GridView` control cannot directly bind to navigation properties in order to update them. You'll update the office assignment in code. To do that, you'll need a reference to the `TextBox` control, and you'll get and save that in the `TextBox` control's `Init` event.

Following the `GridView` control is a `Label` control that's used for error messages. The control's `Visible` property is `false`, and view state is turned off, so that the label will appear only when code makes it visible in response to an error.

Open the *Instructors.aspx.cs* file and add the following `using` statement:

```
usingContosoUniversity.DAL;
```

Add a private class field immediately after the partial-class name declaration to hold a reference to the office assignment text box.

```
privateTextBox instructorOfficeTextBox;
```

Add a stub for the `SelectedIndexChanged` event handler that you'll add code for later. Also add a handler for the office assignment `TextBox` control's `Init` event so that you can store a reference to the `TextBox` control. You'll use this reference to get the value the user entered in order to update the entity associated with the navigation property.

```
protectedvoidInstructorsGridView_SelectedIndexChanged(object sender,EventArgs e)
{
}

protectedvoidInstructorOfficeTextBox_Init(object sender,EventArgs e)
{
            instructorOfficeTextBox = sender asTextBox;
}
```

You'll use the `GridView` control's `Updating` event to update the `Location` property of the associated `OfficeAssignment` entity. Add the following handler for the `Updating` event:

```
protectedvoidInstructorsGridView_RowUpdating(object sender,GridViewUpdateEventArgs e)
{
using(var context =newSchoolEntities())
{
var instructorBeingUpdated =Convert.ToInt32(e.Keys[0]);
var officeAssignment =(from o in context.OfficeAssignments
where o.InstructorID== instructorBeingUpdated
select o).FirstOrDefault();

try
{
if(String.IsNullOrWhiteSpace(instructorOfficeTextBox.Text)==false)
{
if(officeAssignment ==null)
{

context.OfficeAssignments.AddObject(OfficeAssignment.CreateOfficeAssignment(instructo
rBeingUpdated, instructorOfficeTextBox.Text,null));
}
else
```

```
{
                            officeAssignment.Location= instructorOfficeTextBox.Text;
}
}
else
{
if(officeAssignment !=null)
{
                            context.DeleteObject(officeAssignment);
}
}
                    context.SaveChanges();
}
catch(Exception)
{
                    e.Cancel=true;
ErrorMessageLabel.Visible=true;
ErrorMessageLabel.Text="Update failed.";
//Add code to log the error.
}
}
}
```

This code is run when the user clicks **Update** in a `GridView` row. The code uses LINQ to Entities to retrieve the `OfficeAssignment` entity that's associated with the current `Person` entity, using the `PersonID` of the selected row from the event argument.

The code then takes one of the following actions depending on the value in the `InstructorOfficeTextBox` control:

- If the text box has a value and there's no `OfficeAssignment` entity to update, it creates one.
- If the text box has a value and there's an `OfficeAssignment` entity, it updates the `Location` property value.
- If the text box is empty and an `OfficeAssignment` entity exists, it deletes the entity.

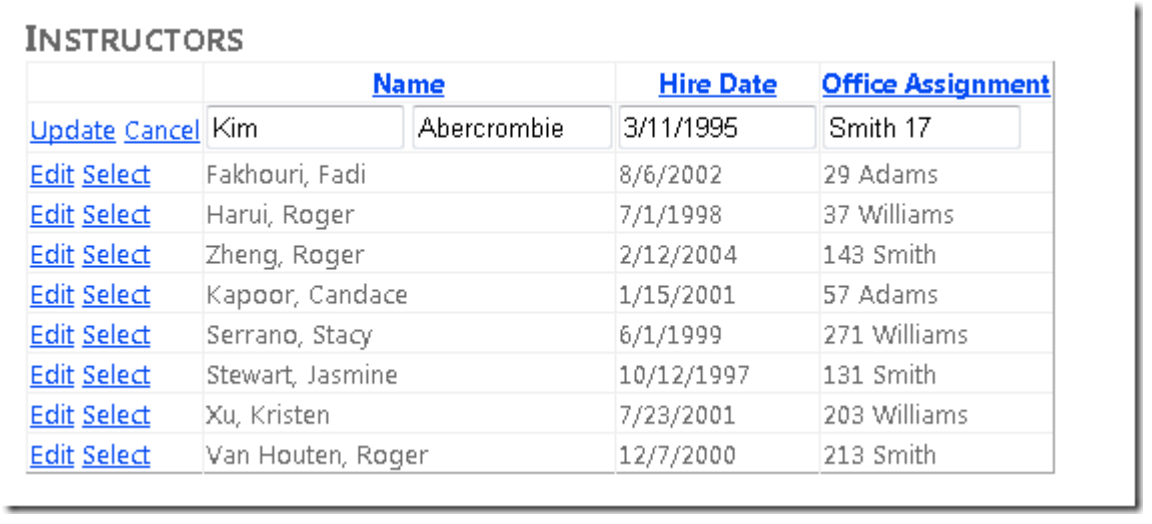
After this, it saves the changes to the database. If an exception occurs, it displays an error message.

Run the page.

INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

Click **Edit** and all fields change to text boxes.



INSTRUCTORS

	Name		Hire Date	Office Assignment
Update Cancel	Kim	Abercrombie	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi		8/6/2002	29 Adams
Edit Select	Harui, Roger		7/1/1998	37 Williams
Edit Select	Zheng, Roger		2/12/2004	143 Smith
Edit Select	Kapoor, Candace		1/15/2001	57 Adams
Edit Select	Serrano, Stacy		6/1/1999	271 Williams
Edit Select	Stewart, Jasmine		10/12/1997	131 Smith
Edit Select	Xu, Kristen		7/23/2001	203 Williams
Edit Select	Van Houten, Roger		12/7/2000	213 Smith

Change any of these values, including **Office Assignment**. Click **Update** and you'll see the changes reflected in the list.

Displaying Related Entities in a Separate Control

Each instructor can teach one or more courses, so you'll add an `EntityDataSource` control and a `GridView` control to list the courses associated with whichever instructor is selected in the instructors `GridView` control. To create a heading and the `EntityDataSource` control for courses entities, add the following markup between the error message `Label` control and the closing `</div>` tag:

```
<h3>Courses Taught</h3>
<asp:EntityDataSourceID="CoursesEntityDataSource"runat="server"
```

```
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="Courses"
Where="@PersonID IN (SELECT VALUE instructor.PersonID FROM it.People AS instructor)">
<WhereParameters>
<asp:ControlParameterControlID="InstructorsGridView"Type="Int32"Name="PersonID"Proper
tyName="SelectedValue"/>
</WhereParameters>
</asp:EntityDataSource>
```

The `Where` parameter contains the value of the `PersonID` of the instructor whose row is selected in the `InstructorsGridView` control. The `Where` property contains a subselect command that gets all associated `Person` entities from a `Course` entity's `People` navigation property and selects the `Course` entity only if one of the associated `Person` entities contains the selected `PersonID` value.

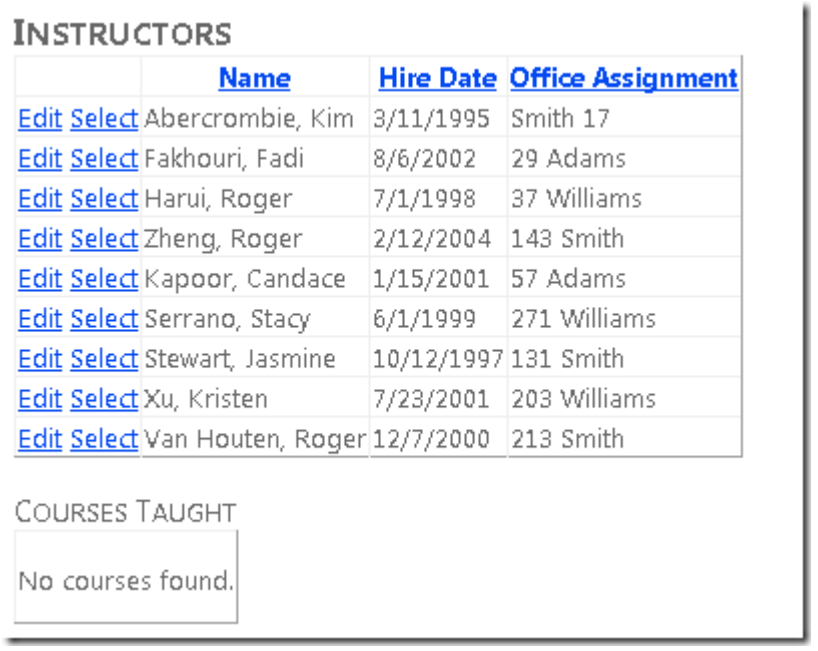
To create the `GridView` control., add the following markup immediately following the `CoursesEntityDataSource` control (before the closing `</div>` tag):

```
<asp:GridViewID="CoursesGridView"runat="server"
DataSourceID="CoursesEntityDataSource"
AllowSorting="True"AutoGenerateColumns="False"
SelectedRowStyle-BackColor="LightGray"
DataKeyNames="CourseID">
<EmptyDataTemplate>
<p>No courses found.</p>
</EmptyDataTemplate>
<Columns>
<asp:CommandFieldShowSelectButton="True"/>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"ReadOnly="True"SortExpression="Cour
seID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"SortExpression="Title"/>
<asp:TemplateFieldHeaderText="Department"SortExpression="DepartmentID">
<ItemTemplate>
<asp:Label ID="GridViewDepartmentLabel" runat="server" Text='<%#
Eval("Department.Name") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
```

```
</Columns>
</asp:GridView>
```

Because no courses will be displayed if no instructor is selected, an `EmptyDataTemplate` element is included.

Run the page.



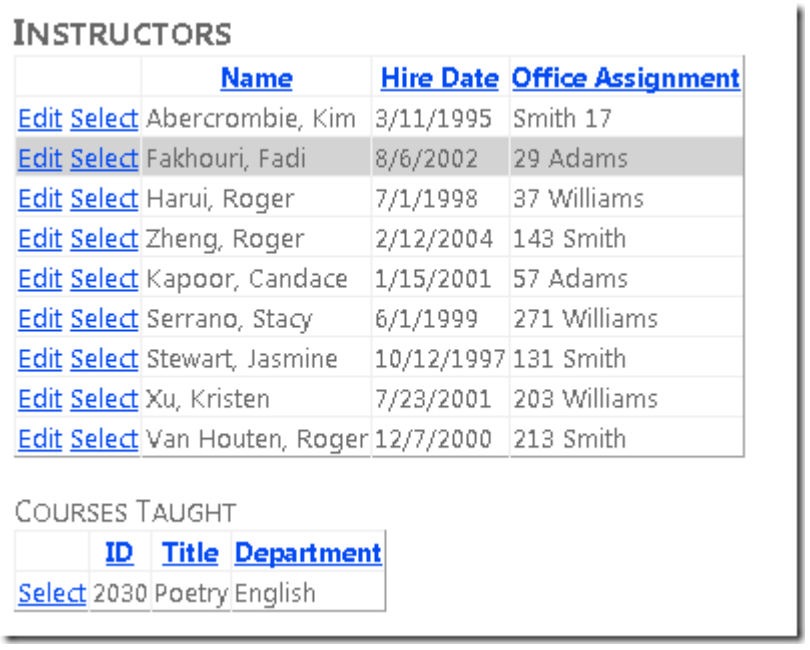
INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

COURSES TAUGHT

No courses found.

Select an instructor who has one or more courses assigned, and the course or courses appear in the list. (Note: although the database schema allows multiple courses, in the test data supplied with the database no instructor actually has more than one course. You can add courses to the database yourself using the **Server Explorer** window or the *CoursesAdd.aspx* page, which you'll add in a later tutorial.)



The `CoursesGridView` control shows only a few course fields. To display all the details for a course, you'll use a `DetailsView` control for the course that the user selects. In *Instructors.aspx*, add the following markup after the closing `</div>` tag (make sure you place this markup **after** the closing div tag, not before it):

```
<div>
<h3>Course Details</h3>
<asp:EntityDataSourceID="CourseDetailsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="Courses"
AutoGenerateWhereClause="False"Where="it.CourseID =
@CourseID"Include="Department,OnlineCourse,OnsiteCourse,StudentGrades.Person"
OnSelected="CourseDetailsEntityDataSource_Selected">
<WhereParameters>
<asp:ControlParameterControlID="CoursesGridView"Type="Int32"Name="CourseID"PropertyNa
me="SelectedValue"/>
</WhereParameters>
</asp:EntityDataSource>
<asp:DetailsViewID="CourseDetailsView"runat="server"AutoGenerateRows="False"
DataSourceID="CourseDetailsEntityDataSource">
<EmptyDataTemplate>
<p>
                    No course selected.</p>
</EmptyDataTemplate>
```

```
<Fields>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"ReadOnly="True"SortExpression="Cour
seID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"SortExpression="Title"/>
<asp:BoundFieldDataField="Credits"HeaderText="Credits"SortExpression="Credits"/>
<asp:TemplateFieldHeaderText="Department">
<ItemTemplate>
<asp:Label ID="DetailsViewDepartmentLabel" runat="server" Text='<%#
Eval("Department.Name") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Location">
<ItemTemplate>
<asp:Label ID="LocationLabel" runat="server" Text='<%# Eval("OnsiteCourse.Location")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="URL">
<ItemTemplate>
<asp:Label ID="URLLabel" runat="server" Text='<%# Eval("OnlineCourse.URL")
%>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
</Fields>
</asp:DetailsView>
</div>
```

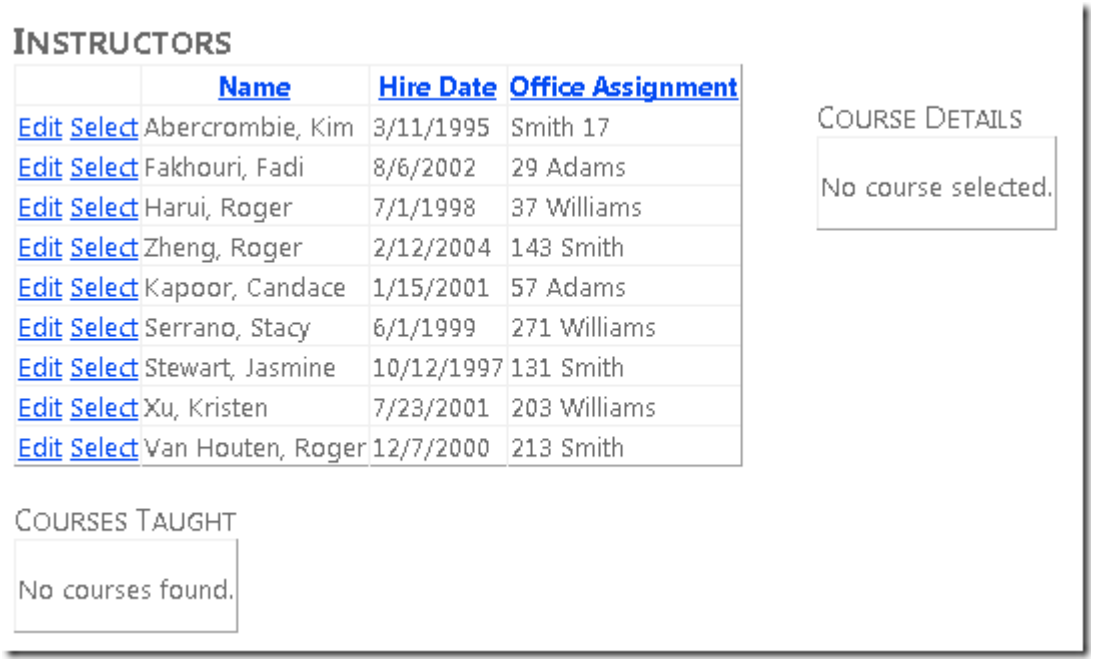
This markup creates an `EntityDataSource` control that's bound to the `Courses` entity set. The `Where` property selects a course using the `CourseID` value of the selected row in the courses `GridView` control. The markup specifies a handler for the `Selected` event, which you'll use later for displaying student grades, which is another level lower in the hierarchy.

In *Instructors.aspx.cs*, create the following stub for the `CourseDetailsEntityDataSource_Selected` method. (You'll fill this stub out later in the tutorial; for now, you need it so that the page will compile and run.)

```
protectedvoidCourseDetailsEntityDataSource_Selected(object
sender,EntityDataSourceSelectedEventArgs e)
```

```
{
}
```

Run the page.



INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

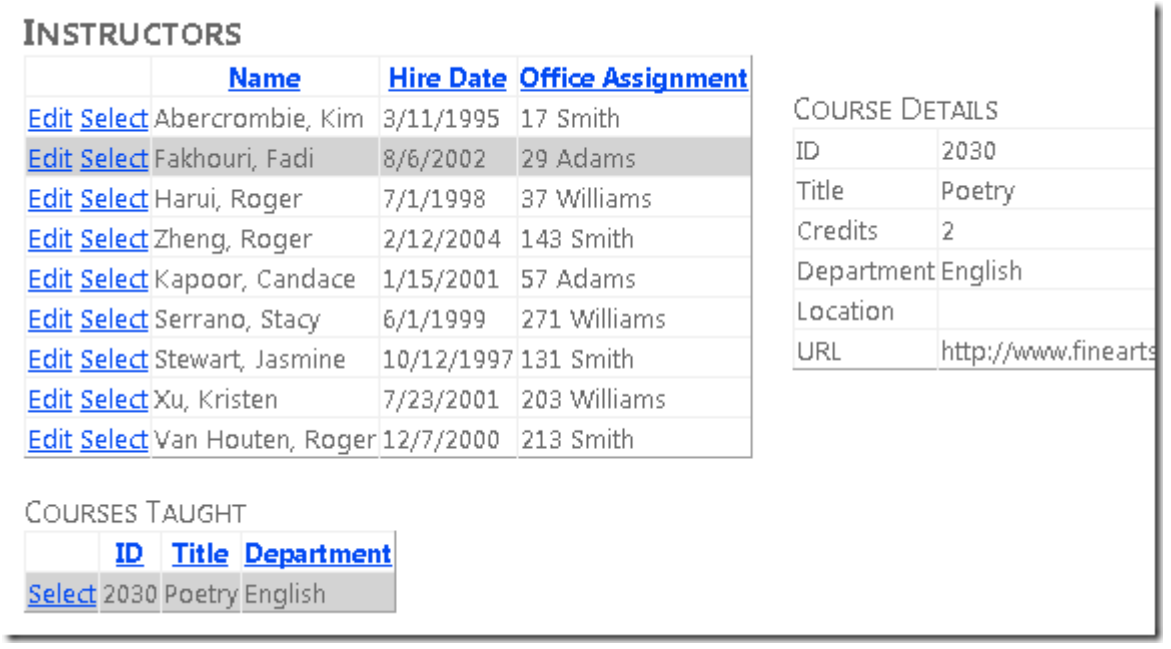
COURSE DETAILS

No course selected.

COURSES TAUGHT

No courses found.

Initially there are no course details because no course is selected. Select an instructor who has a course assigned, and then select a course to see the details.



INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	17 Smith
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

COURSE DETAILS

ID	2030
Title	Poetry
Credits	2
Department	English
Location	
URL	http://www.finearts

COURSES TAUGHT

	ID	Title	Department
Select	2030	Poetry	English

Using the EntityDataSource "Selected" Event to Display Related Data

Finally, you want to show all of the enrolled students and their grades for the selected course. To do this, you'll use the `Selected` event of the `EntityDataSource` control bound to the course `DetailsView`.

In *Instructors.aspx*, add the following markup after the `DetailsView` control:

```
<h3>Student Grades</h3>
<asp:ListViewID="GradesListView"runat="server">
<EmptyDataTemplate>
<p>No student grades found.</p>
</EmptyDataTemplate>
<LayoutTemplate>
<tableborder="1"runat="server"id="itemPlaceholderContainer">
<trrunat="server">
<thrunat="server">
                              Name
</th>
<thrunat="server">
                              Grade
</th>
</tr>
<trid="itemPlaceholder"runat="server">
</tr>
</table>
</LayoutTemplate>
<ItemTemplate>
<tr>
<td>
<asp:Label ID="StudentLastNameLabel" runat="server" Text='<%# Eval("Person.LastName")
%>' />,
<asp:Label ID="StudentFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>' />
</td>
<td>
<asp:Label ID="StudentGradeLabel" runat="server" Text='<%# Eval("Grade") %>' />
</td>
</tr>
```

```
</ItemTemplate>
</asp:ListView>
```

This markup creates a `ListView` control that displays a list of students and their grades for the selected course. No data source is specified because you'll databind the control in code. The `EmptyDataTemplate` element provides a message to display when no course is selected—in that case, there are no students to display. The `LayoutTemplate` element creates an HTML table to display the list, and the `ItemTemplate` specifies the columns to display. The student ID and the student grade are from the `StudentGrade` entity, and the student name is from the `Person` entity that the Entity Framework makes available in the `Person` navigation property of the `StudentGrade` entity.

In *Instructors.aspx.cs*, replace the stubbed-out `CourseDetailsEntityDataSource_Selected` method with the following code:

```
protectedvoidCourseDetailsEntityDataSource_Selected(object
sender,EntityDataSourceSelectedEventArgs e)
{
var course = e.Results.Cast<Course>().FirstOrDefault();
if(course !=null)
{
var studentGrades = course.StudentGrades.ToList();
GradesListView.DataSource= studentGrades;
GradesListView.DataBind();
}
}
```

The event argument for this event provides the selected data in the form of a collection, which will have zero items if nothing is selected or one item if a `Course` entity is selected. If a `Course` entity is selected, the code uses the `First` method to convert the collection to a single object. It then gets `StudentGrade` entities from the navigation property, converts them to a collection, and binds the `GradesListView` control to the collection.

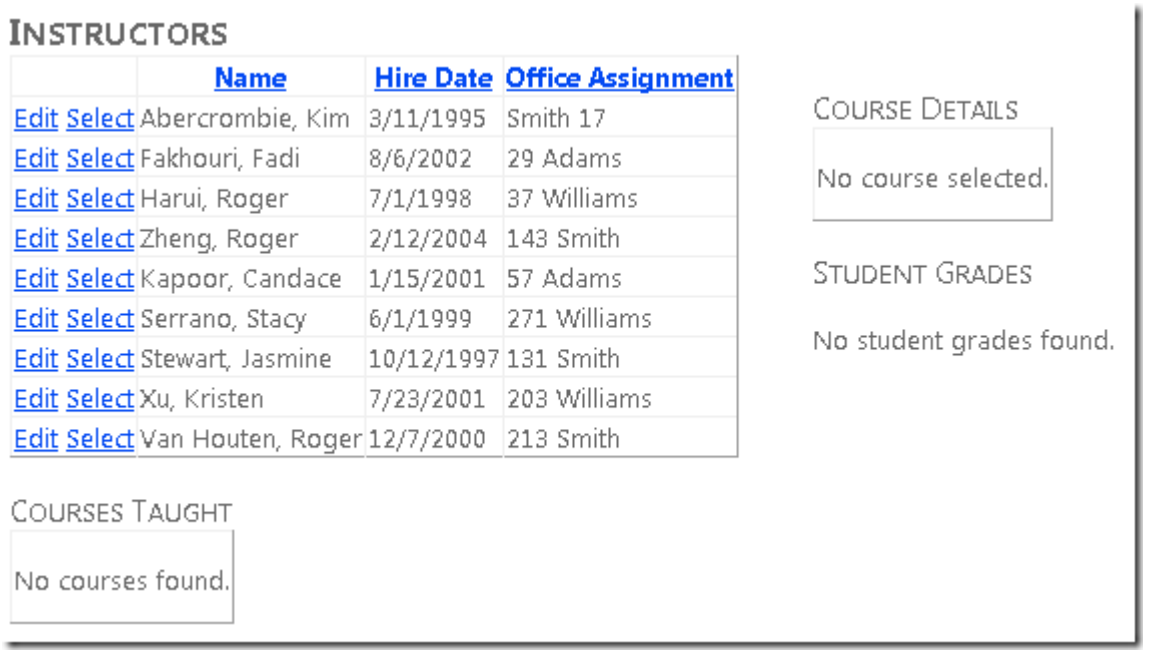
This is sufficient to display grades, but you want to make sure that the message in the empty data template is displayed the first time the page is displayed and whenever a course is not selected. To do that, create the following method, which you'll call from two places:

```
privatevoidClearStudentGradesDataSource()
{
var emptyStudentGradesList =newList<StudentGrade>();
GradesListView.DataSource= emptyStudentGradesList;
GradesListView.DataBind();
}
```

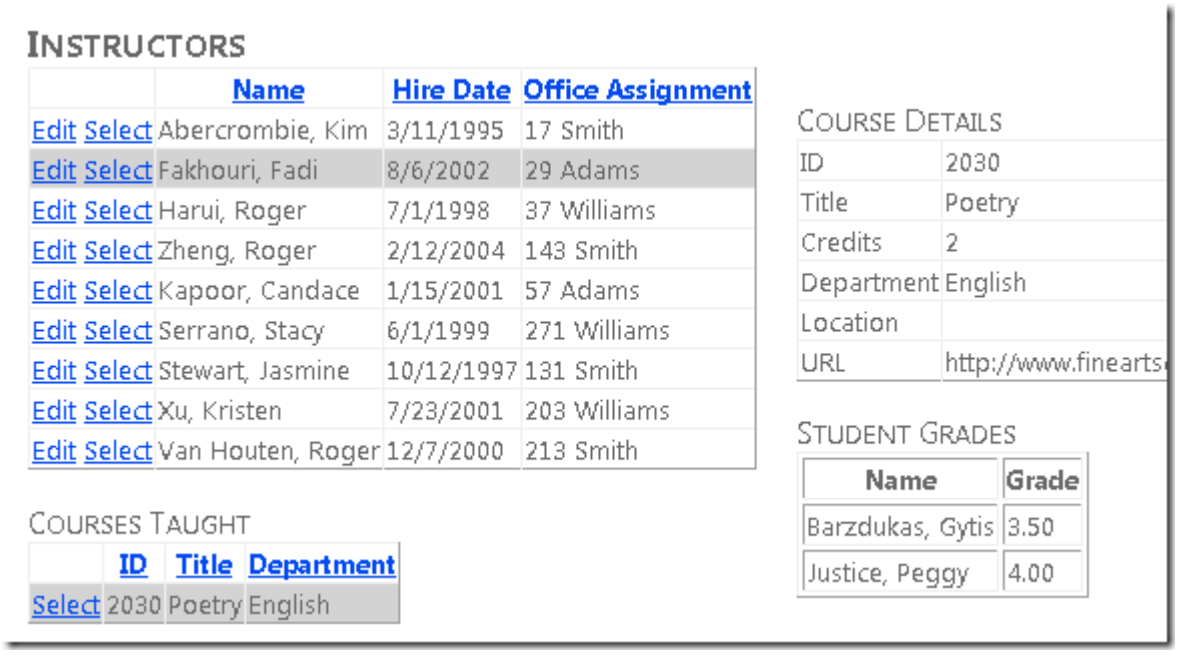
Call this new method from the `Page_Load` method to display the empty data template the first time the page is displayed. And call it from the `InstructorsGridView_SelectedIndexChanged` method because that event is raised when an instructor is selected, which means new courses are loaded into the courses `GridView` control and none is selected yet. Here are the two calls:

```
protectedvoidPage_Load(object sender,EventArgs e)
{
if(!IsPostBack)
{
ClearStudentGradesDataSource();
}
}

protectedvoidInstructorsGridView_SelectedIndexChanged(object sender,EventArgs e)
{
ClearStudentGradesDataSource();
}
```

Run the page.



Select an instructor that has a course assigned, and then select the course.

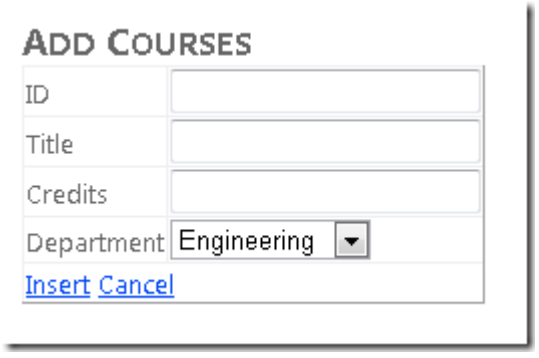


You have now seen a few ways to work with related data. In the following tutorial, you'll learn how to add relationships between existing entities, how to remove relationships, and how to add a new entity that has a relationship to an existing entity.

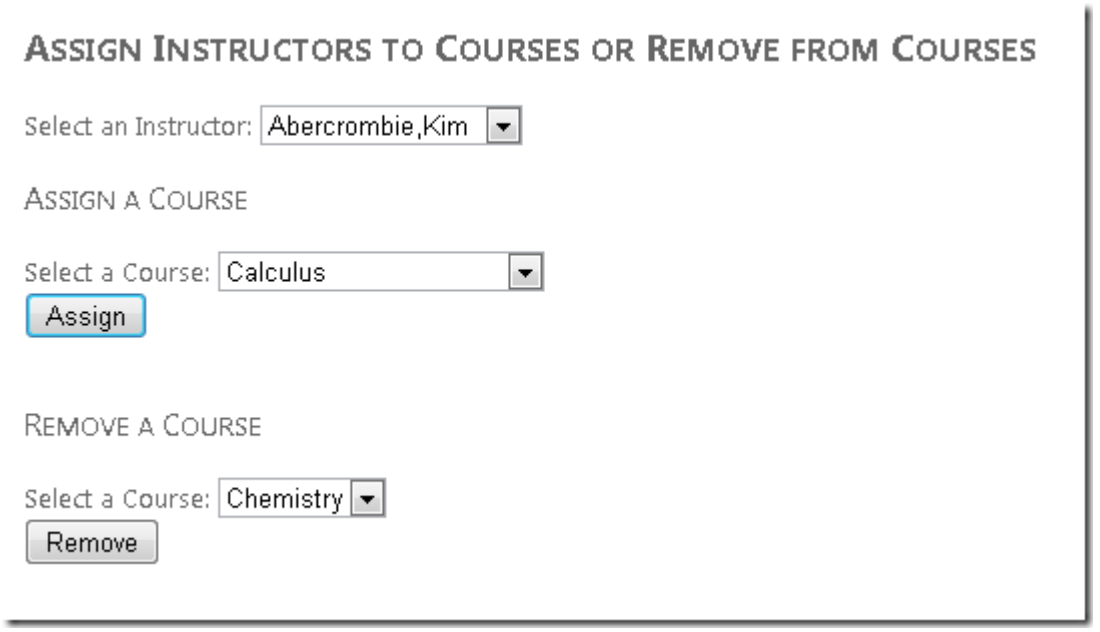
Working with Related Data, Continued

In the previous tutorial you began to use the `EntityDataSource` control to work with related data. You displayed multiple levels of hierarchy and edited data in navigation properties. In this tutorial you'll continue to work with related data by adding and deleting relationships and by adding a new entity that has a relationship to an existing entity.

You'll create a page that adds courses that are assigned to departments. The departments already exist, and when you create a new course, at the same time you'll establish a relationship between it and an existing department.



You'll also create a page that works with a many-to-many relationship by assigning an instructor to a course (adding a relationship between two entities that you select) or removing an instructor from a course (removing a relationship between two entities that you select). In the database, adding a relationship between an instructor and a course results in a new row being added to the `CourseInstructor` association table; removing a relationship involves deleting a row from the `CourseInstructor` association table. However, you do this in the Entity Framework by setting navigation properties, without referring to the `CourseInstructor` table explicitly.



Adding an Entity with a Relationship to an Existing Entity

Create a new web page named *CoursesAdd.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Add Courses</h2>
<asp:EntityDataSourceID="CoursesEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="Courses"
EnableInsert="True"EnableDelete="True">
</asp:EntityDataSource>
<asp:DetailsViewID="CoursesDetailsView"runat="server"AutoGenerateRows="False"
DataSourceID="CoursesEntityDataSource"DataKeyNames="CourseID"
DefaultMode="Insert"oniteminserting="CoursesDetailsView_ItemInserting">
<Fields>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"/>
<asp:BoundFieldDataField="Credits"HeaderText="Credits"/>
<asp:TemplateFieldHeaderText="Department">
<InsertItemTemplate>
<asp:EntityDataSourceID="DepartmentsEntityDataSource"runat="server"ConnectionString="
name=SchoolEntities"
DefaultContainerName="SchoolEntities"EnableDelete="True"EnableFlattening="False"
```

```
EntitySetName="Departments"EntityTypeFilter="Department">
</asp:EntityDataSource>
<asp:DropDownListID="DepartmentsDropDownList"runat="server"DataSourceID="DepartmentsE
ntityDataSource"
DataTextField="Name"DataValueField="DepartmentID"
oninit="DepartmentsDropDownList_Init">
</asp:DropDownList>
</InsertItemTemplate>
</asp:TemplateField>
<asp:CommandFieldShowInsertButton="True"/>
</Fields>
</asp:DetailsView>
```

This markup creates an `EntityDataSource` control that selects courses, that enables inserting, and that specifies a handler for the `Inserting` event. You'll use the handler to update the `Department` navigation property when a new `Course` entity is created.

The markup also creates a `DetailsView` control to use for adding new `Course` entities. The markup uses bound fields for `Course` entity properties. You have to enter the `CourseID` value because this is not a system-generated ID field. Instead, it's a course number that must be specified manually when the course is created.

You use a template field for the `Department` navigation property because navigation properties cannot be used with `BoundField` controls. The template field provides a drop-down list to select the department. The drop-down list is bound to the `Departments` entity set by using `Eval` rather than `Bind`, again because you cannot directly bind navigation properties in order to update them. You specify a handler for the `DropDownList` control's `Init` event so that you can store a reference to the control for use by the code that updates the `DepartmentID` foreign key.

In *CoursesAdd.aspx.cs* just after the partial-class declaration, add a class field to hold a reference to the `DepartmentsDropDownList` control:

```
privateDropDownList departmentDropDownList;
```

Add a handler for the `DepartmentsDropDownList` control's `Init` event so that you can store a reference to the control. This lets you get the value the user has entered and use it to update the `DepartmentID` value of the `Course` entity.

```
protectedvoidDepartmentsDropDownList_Init(object sender,EventArgs e)
{
            departmentDropDownList = sender asDropDownList;
}
```

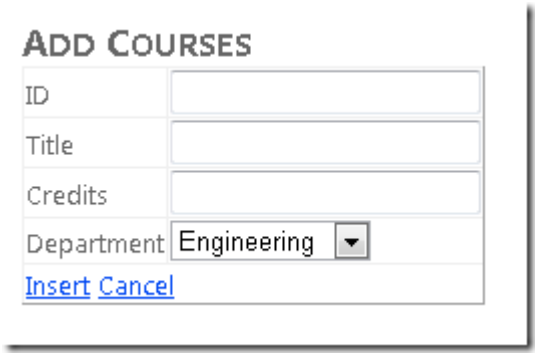
Add a handler for the `DetailsView` control's `Inserting` event:

```
protectedvoidCoursesDetailsView_ItemInserting(object
sender,DetailsViewInsertEventArgs e)
{
var departmentID =Convert.ToInt32(departmentDropDownList.SelectedValue);
            e.Values["DepartmentID"]= departmentID;
}
```

When the user clicks `Insert`, the `Inserting` event is raised before the new record is inserted. The code in the handler gets the `DepartmentID` from the `DropDownList` control and uses it to set the value that will be used for the `DepartmentID` property of the `Course` entity.

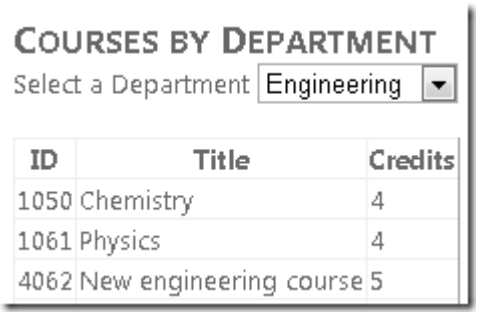
The Entity Framework will take care of adding this course to the `Courses` navigation property of the associated `Department` entity. It also adds the department to the `Department` navigation property of the `Course` entity.

Run the page.



Enter an ID, a title, a number of credits, and select a department, then click **Insert**.

Run the *Courses.aspx* page, and select the same department to see the new course.



Working with Many-to-Many Relationships

The relationship between the `Courses` entity set and the `People` entity set is a many-to-many relationship. A `Course` entity has a navigation property named `People` that can contain zero, one, or more related `Person` entities (representing instructors assigned to teach that course). And a `Person` entity has a navigation property named `Courses` that can contain zero, one, or more related `Course` entities (representing courses that that instructor is assigned to teach). One instructor might teach multiple courses, and one course might be taught by multiple instructors. In this section of the walkthrough, you'll add and remove relationships between `Person` and `Course` entities by updating the navigation properties of the related entities.

Create a new web page named *InstructorsCourses.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Assign Instructors to Courses or Remove from Courses</h2>
<br/>
<asp:EntityDataSourceID="InstructorsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"
Where="it.HireDate is not null"Select="it.LastName + ', ' + it.FirstMidName AS Name,
it.PersonID">
</asp:EntityDataSource>
    Select an Instructor:
<asp:DropDownListID="InstructorsDropDownList"runat="server"DataSourceID="InstructorsE
ntityDataSource"
AutoPostBack="true"DataTextField="Name"DataValueField="PersonID"
OnSelectedIndexChanged="InstructorsDropDownList_SelectedIndexChanged"
OnDataBound="InstructorsDropDownList_DataBound">
</asp:DropDownList>
<h3>
        Assign a Course</h3>
```

```
<br/>
    Select a Course:
<asp:DropDownListID="UnassignedCoursesDropDownList"runat="server"
DataTextField="Title"DataValueField="CourseID">
</asp:DropDownList>
<br/>
<asp:ButtonID="AssignCourseButton"runat="server"Text="Assign"OnClick="AssignCourseBut
ton_Click"/>
<br/>
<asp:LabelID="CourseAssignedLabel"runat="server"Visible="false"Text="Assignment
successful"></asp:Label>
<br/>
<h3>
        Remove a Course</h3>
<br/>
    Select a Course:
<asp:DropDownListID="AssignedCoursesDropDownList"runat="server"
DataTextField="title"DataValueField="courseiD">
</asp:DropDownList>
<br/>
<asp:ButtonID="RemoveCourseButton"runat="server"Text="Remove"OnClick="RemoveCourseBut
ton_Click"/>
<br/>
<asp:LabelID="CourseRemovedLabel"runat="server"Visible="false"Text="Removal
successful"></asp:Label>
```

This markup creates an `EntityDataSource` control that retrieves the name and `PersonID` of `Person` entities for instructors. A `DropDownList` control is bound to the `EntityDataSource` control. The `DropDownList` control specifies a handler for the `DataBound` event. You'll use this handler to databind the two drop-down lists that display courses.

The markup also creates the following group of controls to use for assigning a course to the selected instructor:

- A `DropDownList` control for selecting a course to assign. This control will be populated with courses that are currently not assigned to the selected instructor.
- A `Button` control to initiate the assignment.
- A `Label` control to display an error message if the assignment fails.

Finally, the markup also creates a group of controls to use for removing a course from the selected instructor.

In *InstructorsCourses.aspx.cs*, add a using statement:

```
usingContosoUniversity.DAL;
```

Add a method for populating the two drop-down lists that display courses:

```
privatevoidPopulateDropDownLists()
{
using(var context =newSchoolEntities())
{
var allCourses =(from c in context.Courses
select c).ToList();

var instructorID =Convert.ToInt32(InstructorsDropDownList.SelectedValue);
var instructor =(from p in context.People.Include("Courses")
where p.PersonID== instructorID
select p).First();

var assignedCourses = instructor.Courses.ToList();
var unassignedCourses = allCourses.Except(assignedCourses.AsEnumerable()).ToList();

UnassignedCoursesDropDownList.DataSource= unassignedCourses;
UnassignedCoursesDropDownList.DataBind();
UnassignedCoursesDropDownList.Visible=true;

AssignedCoursesDropDownList.DataSource= assignedCourses;
AssignedCoursesDropDownList.DataBind();
AssignedCoursesDropDownList.Visible=true;
}
}
```

This code gets all courses from the `Courses` entity set and gets the courses from the `Courses` navigation property of the `Person` entity for the selected instructor. It then determines which courses are assigned to that instructor and populates the drop-down lists accordingly.

Add a handler for the `Assign` button's `Click` event:

```
protectedvoidAssignCourseButton_Click(object sender,EventArgs e)
{
using(var context =newSchoolEntities())
{
var instructorID =Convert.ToInt32(InstructorsDropDownList.SelectedValue);
var instructor =(from p in context.People
where p.PersonID== instructorID
select p).First();
var courseID =Convert.ToInt32(UnassignedCoursesDropDownList.SelectedValue);
var course =(from c in context.Courses
where c.CourseID== courseID
select c).First();
                instructor.Courses.Add(course);
try
{
                    context.SaveChanges();
PopulateDropDownLists();
CourseAssignedLabel.Text="Assignment successful.";
}
catch(Exception)
{
CourseAssignedLabel.Text="Assignment unsuccessful.";
//Add code to log the error.
}
CourseAssignedLabel.Visible=true;
}
}
```

This code gets the `Person` entity for the selected instructor, gets the `Course` entity for the selected course, and adds the selected course to the `Courses` navigation property of the instructor's `Person` entity. It then saves the changes to the database and repopulates the drop-down lists so the results can be seen immediately.

Add a handler for the `Remove` button's `Click` event:

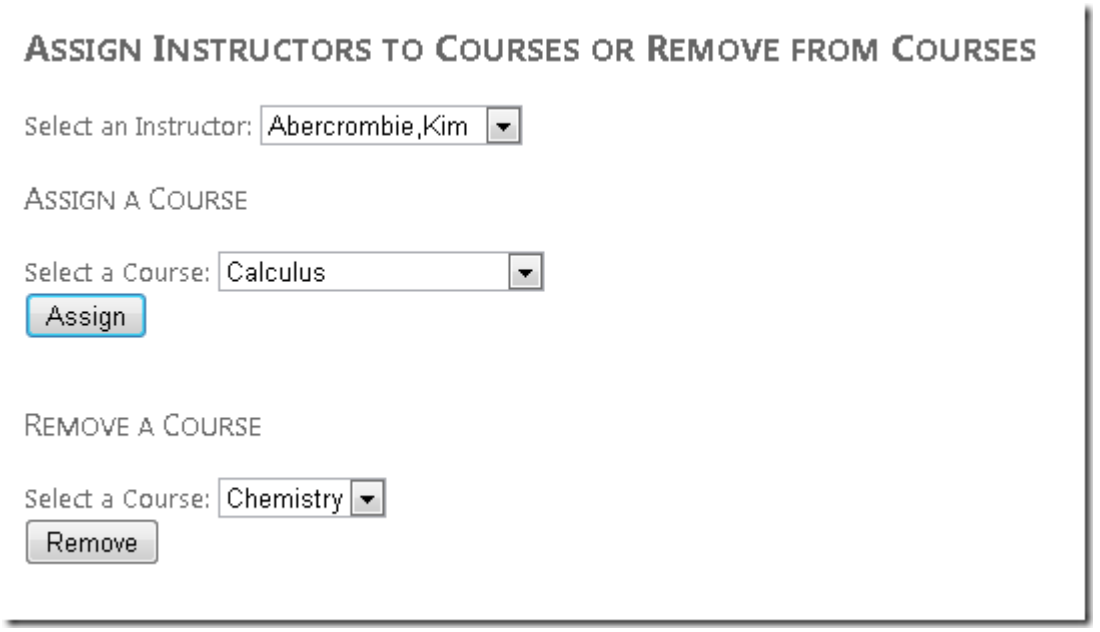
```
protectedvoidRemoveCourseButton_Click(object sender,EventArgs e)
{
using(var context =newSchoolEntities())
{
var instructorID =Convert.ToInt32(InstructorsDropDownList.SelectedValue);
var instructor =(from p in context.People
where p.PersonID== instructorID
select p).First();
var courseID =Convert.ToInt32(AssignedCoursesDropDownList.SelectedValue);
var courses = instructor.Courses;
var courseToRemove =newCourse();
foreach(Course c in courses)
{
if(c.CourseID== courseID)
{
                        courseToRemove = c;
break;
}
}
try
{
                    courses.Remove(courseToRemove);
                    context.SaveChanges();
PopulateDropDownLists();
CourseRemovedLabel.Text="Removal successful.";
}
catch(Exception)
{
CourseRemovedLabel.Text="Removal unsuccessful.";
//Add code to log the error.
}
CourseRemovedLabel.Visible=true;
}
}
```

This code gets the `Person` entity for the selected instructor, gets the `Course` entity for the selected course, and removes the selected course from the `Person` entity's `Courses` navigation property. It then saves the changes to the database and repopulates the drop-down lists so the results can be seen immediately.

Add code to the `Page_Load` method that makes sure the error messages are not visible when there's no error to report, and add handlers for the `DataBound` and `SelectedIndexChanged` events of the instructors drop-down list to populate the courses drop-down lists:

```
protectedvoidPage_Load(object sender,EventArgs e)
{
CourseAssignedLabel.Visible=false;
CourseRemovedLabel.Visible=false;
}

protectedvoidInstructorsDropDownList_DataBound(object sender,EventArgs e)
{
PopulateDropDownLists();
}

protectedvoidInstructorsDropDownList_SelectedIndexChanged(object sender,EventArgs e)
{
PopulateDropDownLists();
}
```

Run the page.



Select an instructor. The **Assign a Course** drop-down list displays the courses that the instructor doesn't teach, and the **Remove a Course** drop-down list displays the courses that the instructor is already assigned to. In the **Assign a Course** section, select a course and then click **Assign**. The course moves to the **Remove a Course** drop-down list. Select a course in the **Remove a Course** section and click **Remove**. The course moves to the **Assign a Course** drop-down list.

You have now seen some more ways to work with related data. In the following tutorial, you'll learn how to use inheritance in the data model to improve the maintainability of your application.

Implementing Table-per-Hierarchy Inheritance

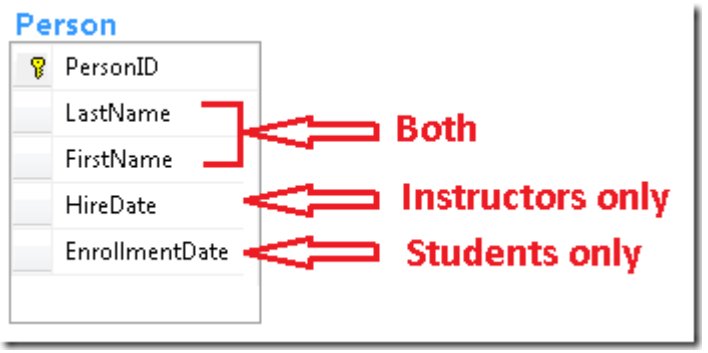
In the previous tutorial you worked with related data by adding and deleting relationships and by adding a new entity that had a relationship to an existing entity. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to make it easier to work with related classes. For example, you could create `Instructor` and `Student` classes that derive from a `Person` base class. You can create the same kinds of inheritance structures among entities in the Entity Framework.

In this part of the tutorial, you won't create any new web pages. Instead, you'll add derived entities to the data model and modify existing pages to use the new entities.

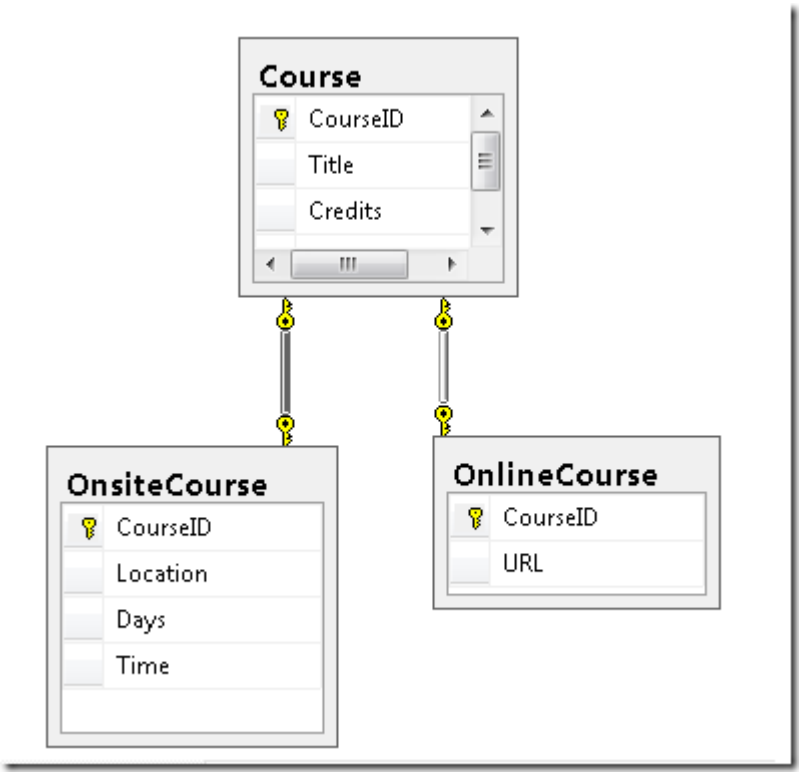
Table-per-Hierarchy versus Table-per-Type Inheritance

A database can store information about related objects in one table or in multiple tables. For example, in the `School` database, the `Person` table includes information about both students and instructors in a single table. Some of the columns apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), and some to both (`LastName`, `FirstName`).



You can configure the Entity Framework to create `Instructor` and `Student` entities that inherit from the `Person` entity. This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy* (TPH) inheritance.

For courses, the `School` database uses a different pattern. Online courses and onsite courses are stored in separate tables, each of which has a foreign key that points to the `Course` table. Information common to both course types is stored only in the `Course` table.



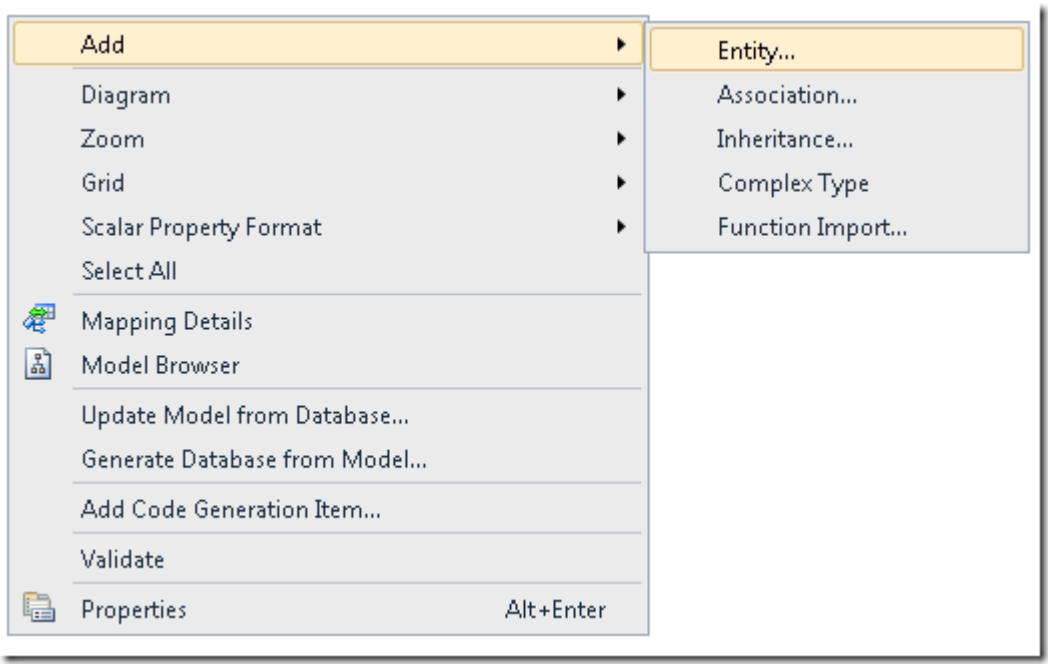
You can configure the Entity Framework data model so that `OnlineCourse` and `OnsiteCourse` entities inherit from the `Course` entity. This pattern of generating an entity inheritance structure from separate tables for each type, with each separate table referring back to a table that stores data common to all types, is called *table per type* (TPT) inheritance.

TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance patterns, because TPT patterns can result in complex join queries. This walkthrough demonstrates how to implement TPH inheritance. You'll do that by performing the following steps:

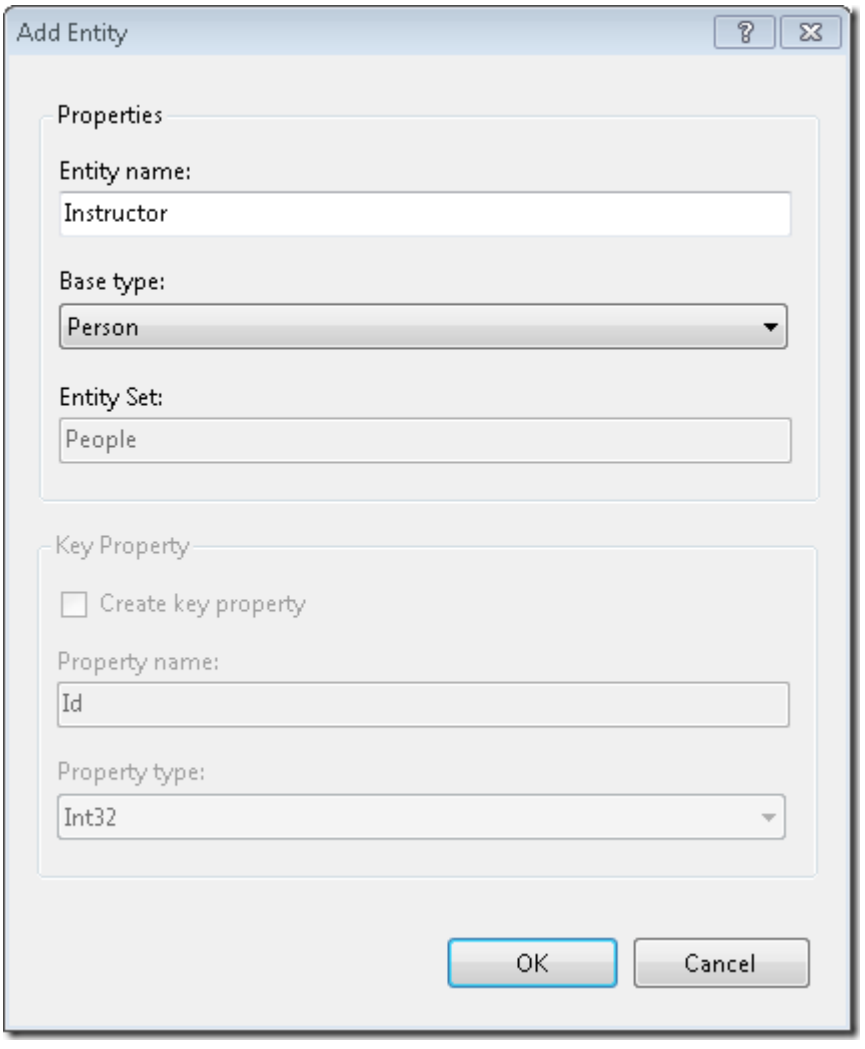
- Create `Instructor` and `Student` entity types that derive from `Person`.
- Move properties that pertain to the derived entities from the `Person` entity to the derived entities.
- Set constraints on properties in the derived types.
- Make the `Person` entity an abstract entity.
- Map each derived entity to the `Person` table with a condition that specifies how to determine whether a `Person` row represents that derived type.

Adding Instructor and Student Entities

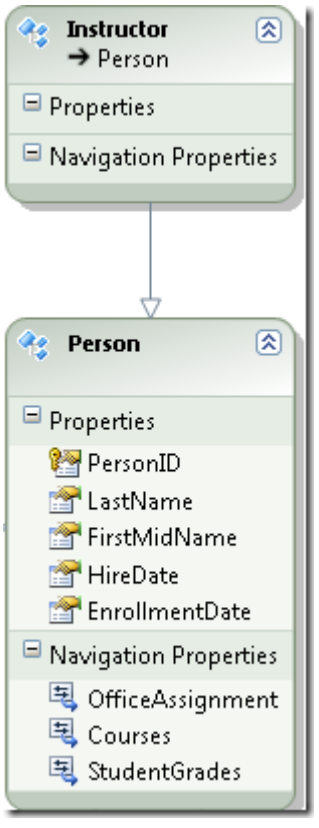
Open the *SchoolModel.edmx* file, right-click an unoccupied area in the designer, select **Add**, then select **Entity**.



In the **Add Entity** dialog box, name the entity `Instructor` and set its **Base type** option to `Person`.

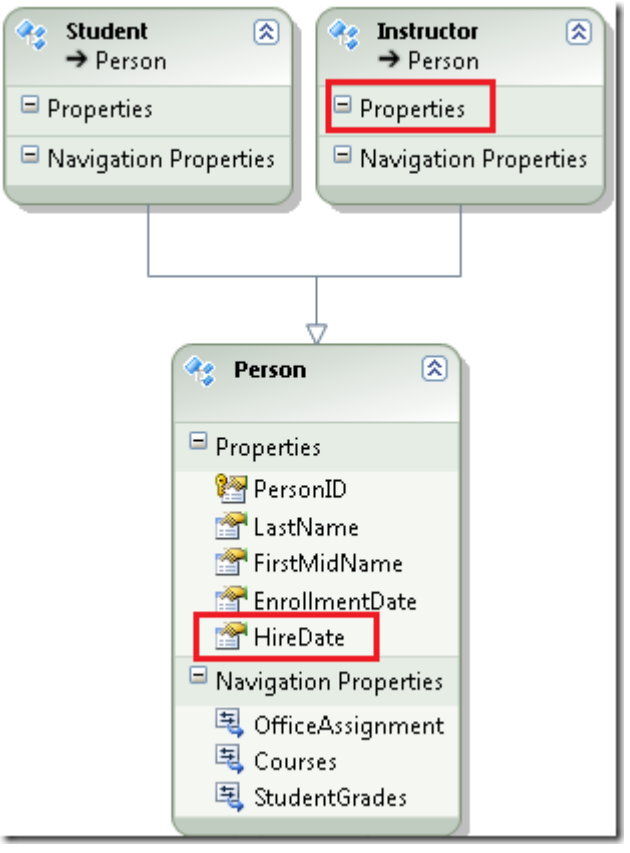


Click **OK**. The designer creates an `Instructor` entity that derives from the `Person` entity. The new entity does not yet have any properties.

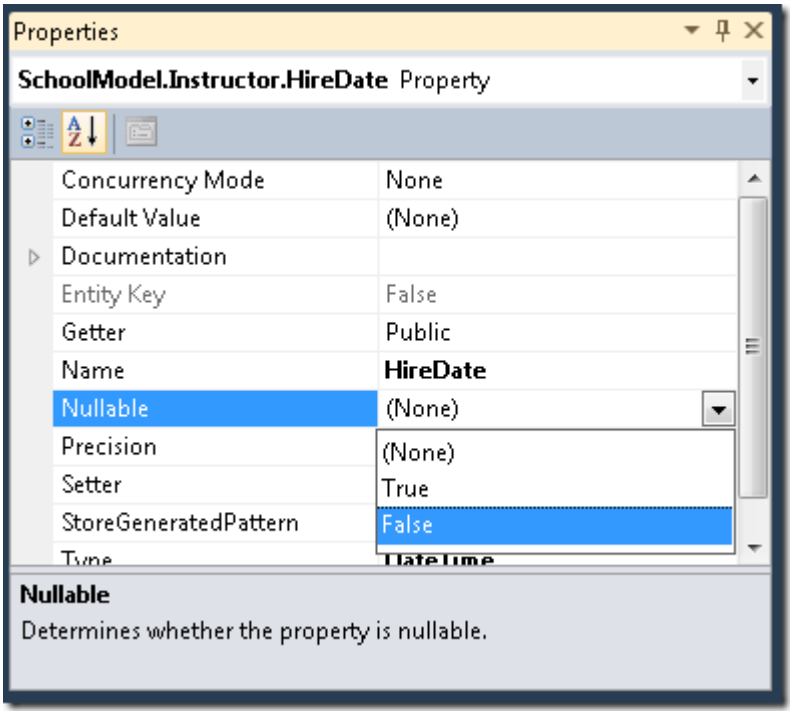


Repeat the procedure to create a `Student` entity that also derives from `Person`.

Only instructors have hire dates, so you need to move that property from the `Person` entity to the `Instructor` entity. In the `Person` entity, right-click the `HireDate` property and click **Cut**. Then right-click **Properties** in the `Instructor` entity and click **Paste**.

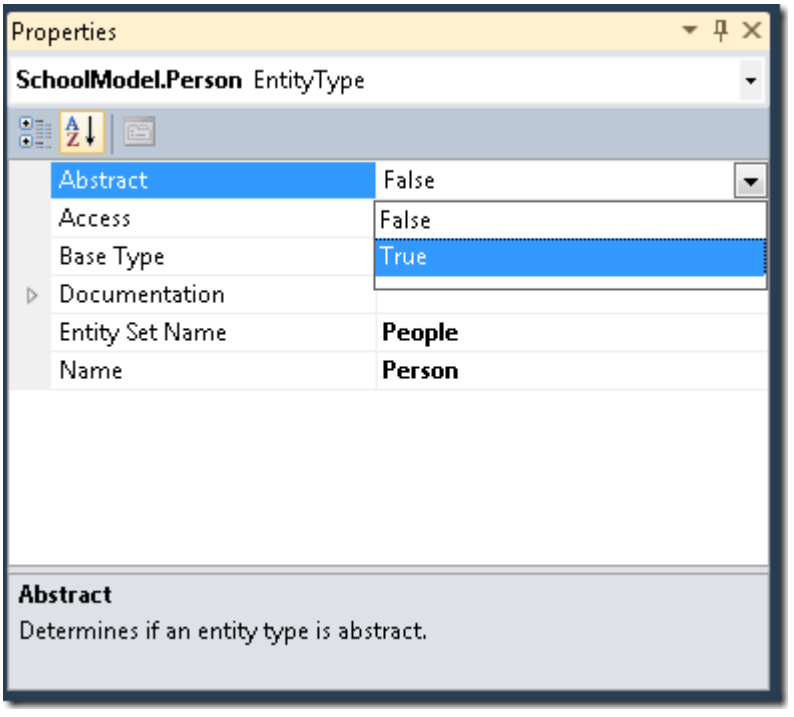


The hire date of an `Instructor` entity cannot be null. Right-click the `HireDate` property, click **Properties**, and then in the **Properties** window change `Nullable` to `False`.



Repeat the procedure to move the `EnrollmentDate` property from the `Person` entity to the `Student` entity. Make sure that you also set `Nullable` to `False` for the `EnrollmentDate` property.

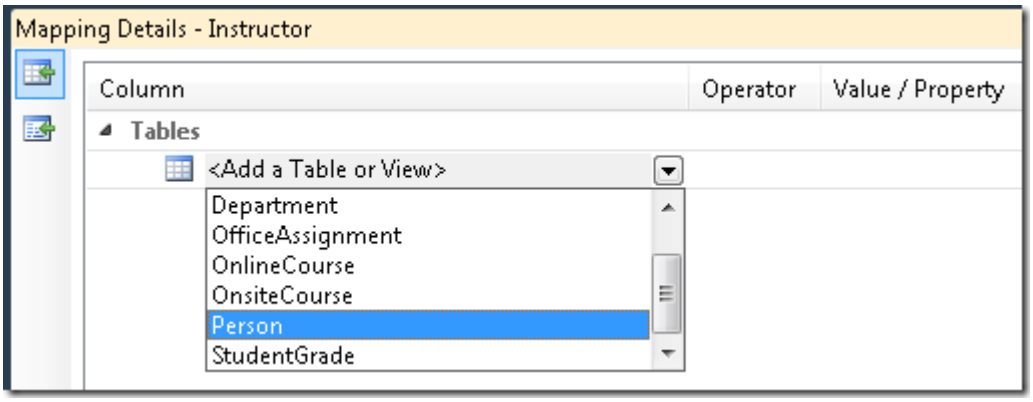
Now that a `Person` entity has only the properties that are common to `Instructor` and `Student` entities (aside from navigation properties, which you're not moving), the entity can only be used as a base entity in the inheritance structure. Therefore, you need to ensure that it's never treated as an independent entity. Right-click the `Person` entity, select **Properties**, and then in the **Properties** window change the value of the **Abstract** property to **True**.



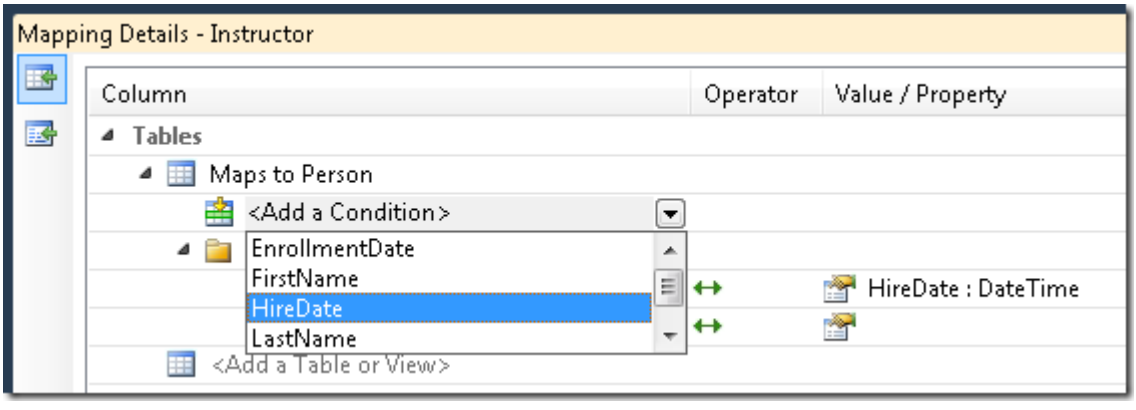
Mapping Instructor and Student Entities to the Person Table

Now you need to tell the Entity Framework how to differentiate between `Instructor` and `Student` entities in the database.

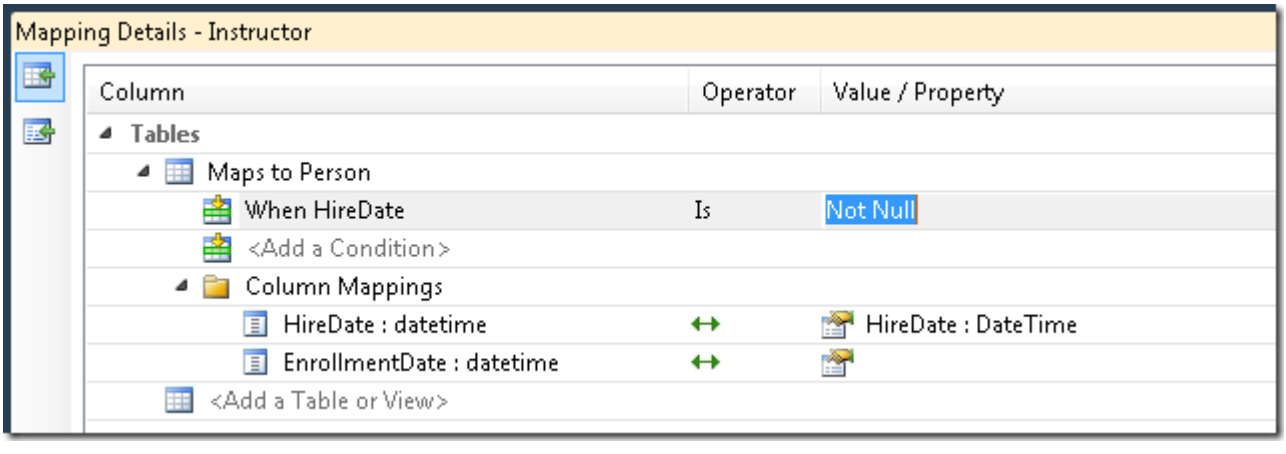
Right-click the `Instructor` entity and select **Table Mapping**. In the **Mapping Details** window, click **Add a Table or View** and select **Person**.



Click **Add a Condition**, and then select **HireDate**.



Change **Operator** to **Is** and **Value / Property** to **Not Null**.



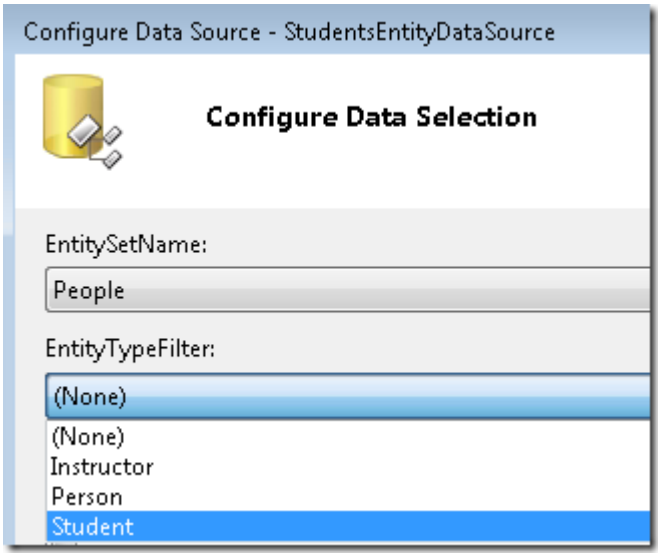
Repeat the procedure for the `Students` entity, specifying that this entity maps to the `Person` table when the `EnrollmentDate` column is not null. Then save and close the data model.

Build the project in order to create the new entities as classes and make them available in the designer.

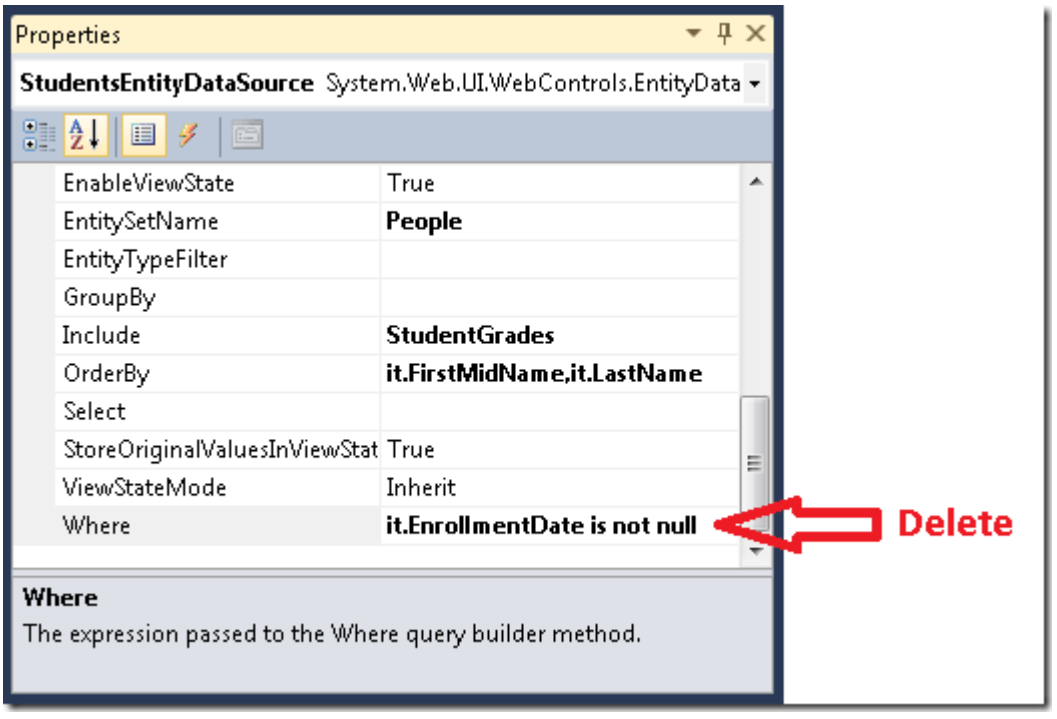
Using the Instructor and Student Entities

When you created the web pages that work with student and instructor data, you databound them to the `Person` entity set, and you filtered on the `HireDate` or `EnrollmentDate` property to restrict the returned data to students or instructors. However, now when you bind each data source control to the `Person` entity set, you can specify that only `Student` or `Instructor` entity types should be selected. Because the Entity Framework knows how to differentiate students and instructors in the `Person` entity set, you can remove the `Where` property settings you entered manually to do that.

In the Visual Studio Designer, you can specify the entity type that an `EntityDataSource` control should select in the **EntityTypeFilter** drop-down box of the `Configure Data Source` wizard, as shown in the following example.



And in the **Properties** window you can remove `Where` clause values that are no longer needed, as shown in the following example.



However, because you've changed the markup for `EntityDataSource` controls to use the `ContextTypeName` attribute, you cannot run the **Configure Data Source** wizard on `EntityDataSource` controls that you've already created. Therefore, you'll make the required changes by changing markup instead.

Open the *Students.aspx* page. In the `StudentsEntityDataSource` control, remove the `Where` attribute and add an `EntityTypeFilter="Student"` attribute. The markup will now resemble the following example:

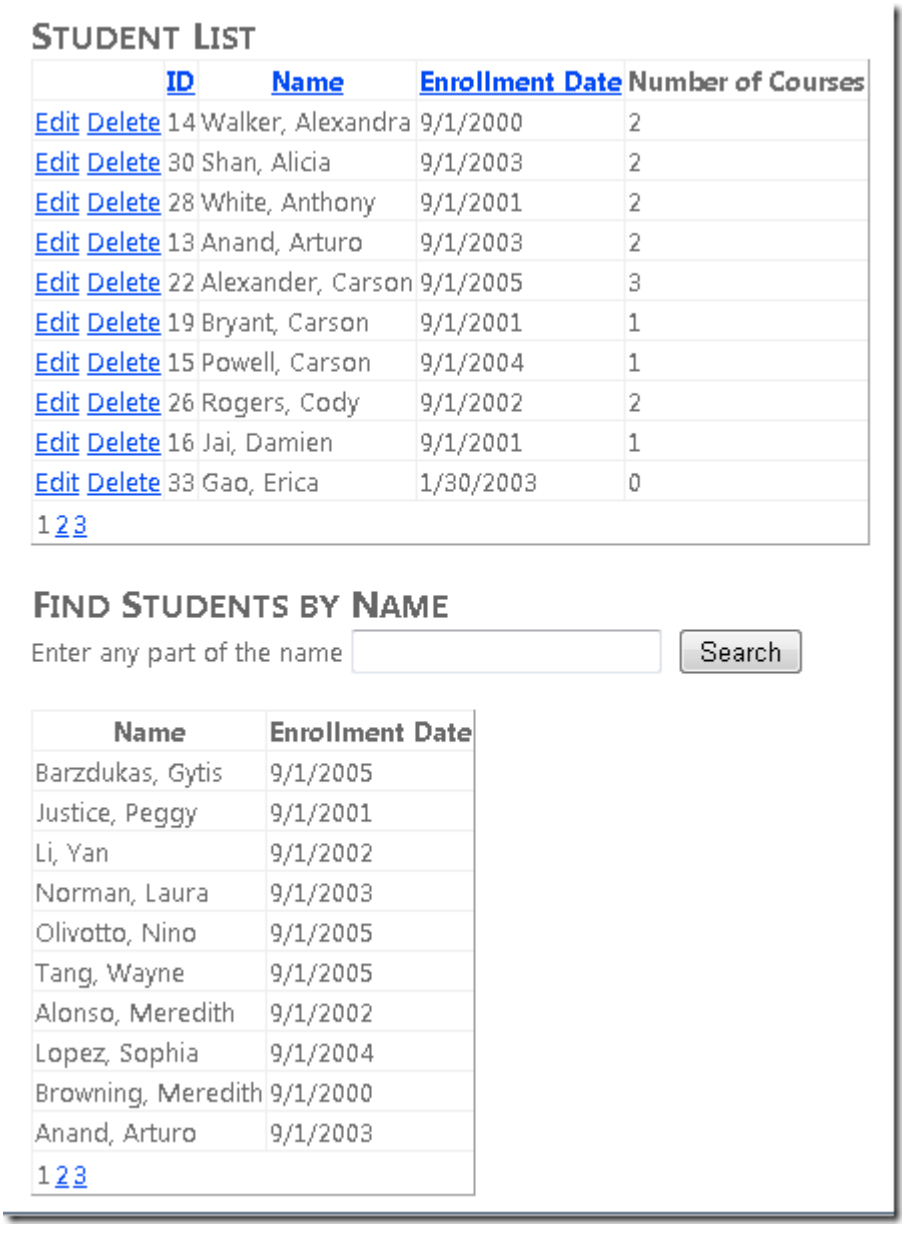
```
<asp:EntityDataSourceID="StudentsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"EntityTypeFilter="Student"
Include="StudentGrades"
EnableDelete="True"EnableUpdate="True"
OrderBy="it.LastName">
</asp:EntityDataSource>
```

Setting the `EntityTypeFilter` attribute ensures that the `EntityDataSource` control will select only the specified entity type. If you wanted to retrieve both `Student` and `Instructor` entity types, you would not set this attribute. (You have the option of retrieving multiple entity types with one `EntityDataSource` control only if you're using the control for read-only data access. If you're using an `EntityDataSource` control to insert, update, or delete entities, and if the entity set it's bound to can contain multiple types, you can only work with one entity type, and you have to set this attribute.)

Repeat the procedure for the `SearchEntityDataSource` control, except remove only the part of the `Where` attribute that selects `Student` entities instead of removing the property altogether. The opening tag of the control will now resemble the following example:

```
<asp:EntityDataSourceID="SearchEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"EntityTypeFilter="Student"
Where="it.FirstMidName Like '%' + @StudentName + '%' or it.LastName Like '%' +
@StudentName + '%'">
```

Run the page to verify that it still works as it did before.

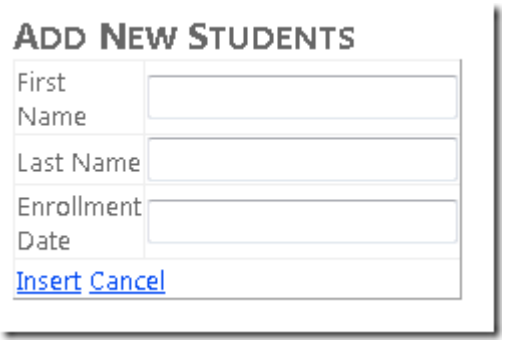


Update the following pages that you created in earlier tutorials so that they use the new `Student` and `Instructor` entities instead of `Person` entities, then run them to verify that they work as they did before:

- In *StudentsAdd.aspx*, add `EntityTypeFilter="Student"` to the `StudentsEntityDataSource` control. The markup will now resemble the following example:

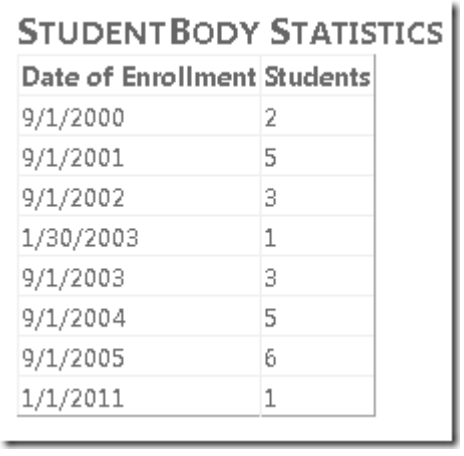
```
<asp:EntityDataSource ID="StudentsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities"
EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Student"
```

```
        EnableInsert="True"
</asp:EntityDataSource>
```



- In *About.aspx*, add `EntityTypeFilter="Student"` to the `StudentStatisticsEntityDataSource` control and remove `Where="it.EnrollmentDate is not null"`. The markup will now resemble the following example:

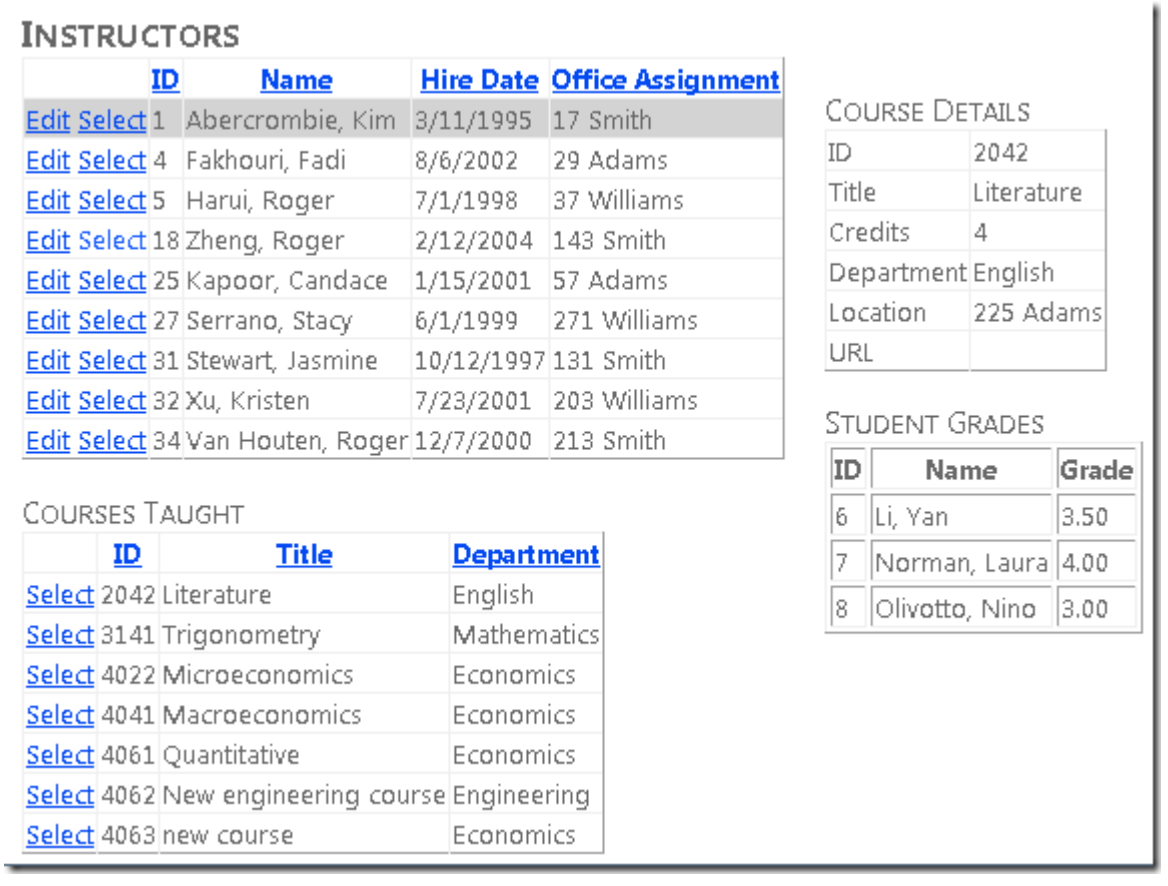
```
<asp:EntityDataSourceID="StudentStatisticsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"EntityTypeFilter="Student"
Select="it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents"
OrderBy="it.EnrollmentDate"GroupBy="it.EnrollmentDate">
</asp:EntityDataSource>
```


STUDENT BODY STATISTICS

Date of Enrollment	Students
9/1/2000	2
9/1/2001	5
9/1/2002	3
1/30/2003	1
9/1/2003	3
9/1/2004	5
9/1/2005	6
1/1/2011	1

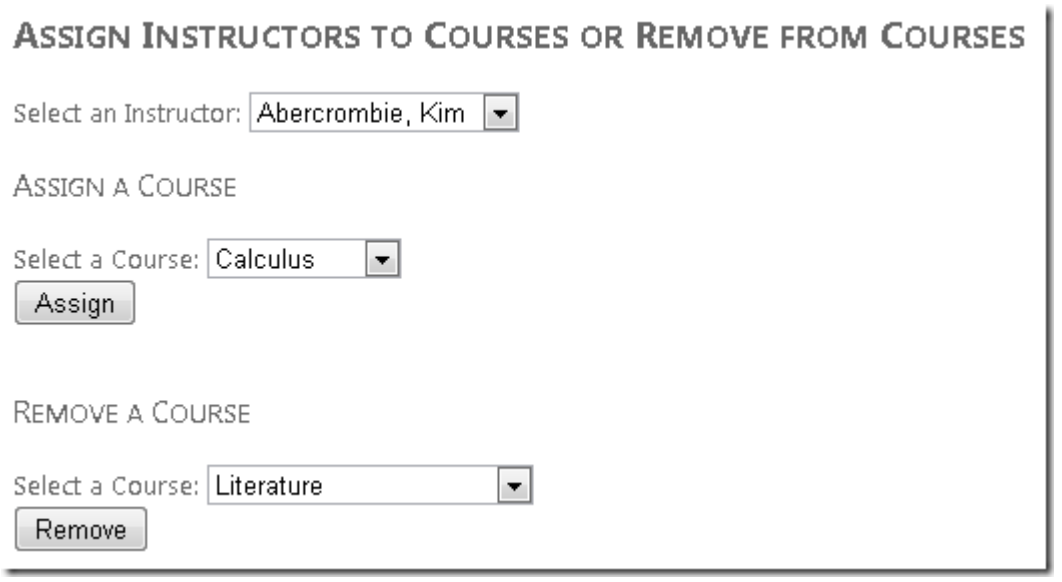
- In *Instructors.aspx* and *InstructorsCourses.aspx*, add `EntityTypeFilter="Instructor"` to the `InstructorsEntityDataSource` control and remove `Where="it.HireDate is not null"`. The markup in *Instructors.aspx* now resembles the following example:

```
<asp:EntityDataSourceID="InstructorsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="false"
EntitySetName="People"EntityTypeFilter="Instructor"
Include="OfficeAssignment"
EnableUpdate="True">
</asp:EntityDataSource>
```



The markup in *InstructorsCourses.aspx* will now resemble the following example:

```
<asp:EntityDataSourceID="InstructorsEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="False"
EntitySetName="People"EntityTypeFilter="Instructor"
Select="it.LastName + ',' + it.FirstMidName AS Name, it.PersonID">
</asp:EntityDataSource>
```



As a result of these changes, you've improved the Contoso University application's maintainability in several ways. You've moved selection and validation logic out of the UI layer (*.aspx* markup) and made it an integral part of the data access layer. This helps to isolate your application code from changes that you might make in the future to the database schema or the data model. For example, you could decide that students might be hired as teachers' aids and therefore would get a hire date. You could then add a new property to differentiate students from instructors and update the data model. No code in the web application would need to change except where you wanted to show a hire date for students. Another benefit of adding `Instructor` and `Student` entities is that your code is more readily understandable than when it referred to `Person` objects that were actually students or instructors.

You've now seen one way to implement an inheritance pattern in the Entity Framework. In the following tutorial, you'll learn how to use stored procedures in order to have more control over how the Entity Framework accesses the database.

Using Stored Procedures

In the previous tutorial you implemented a table-per-hierarchy inheritance pattern. This tutorial will show you how to use stored procedures to gain more control over database access.

The Entity Framework lets you specify that it should use stored procedures for database access. For any entity type, you can specify a stored procedure to use for creating, updating, or deleting entities of that type. Then in the data model you can add references to stored procedures that you can use to perform tasks such as retrieving sets of entities.

Using stored procedures is a common requirement for database access. In some cases a database administrator may require that all database access go through stored procedures for security reasons. In other cases you may want to build business logic into some of the processes that the Entity Framework uses when it updates the database. For example, whenever an entity is deleted you might want to copy it to an archive database. Or whenever a row is updated you might want to write a row to a logging table that records who made the change. You can perform these kinds of tasks in a stored procedure that's called whenever the Entity Framework deletes an entity or updates an entity.

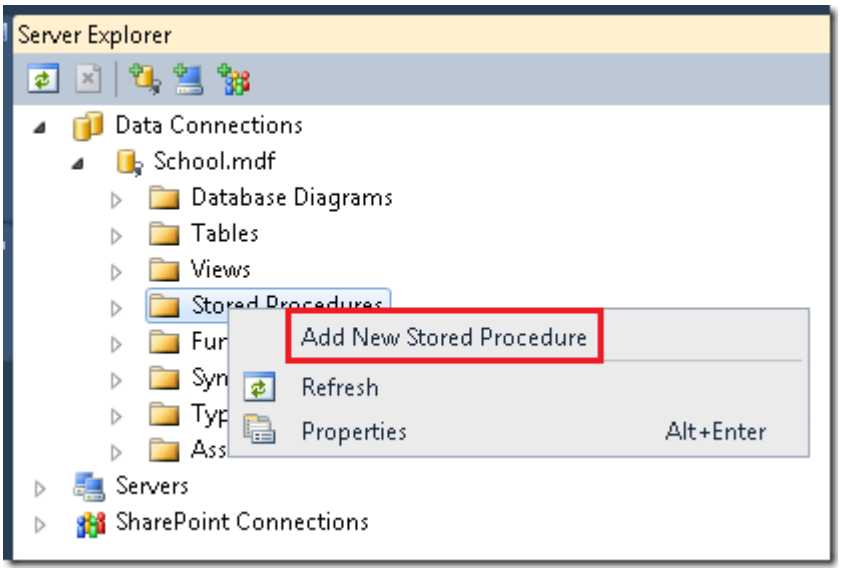
As in the previous tutorial, you'll not create any new pages. Instead, you'll change the way the Entity Framework accesses the database for some of the pages you already created.

In this tutorial you'll create stored procedures in the database for inserting `Student` and `Instructor` entities. You'll add them to the data model, and you'll specify that the Entity Framework should use them for adding `Student` and `Instructor` entities to the database. You'll also create a stored procedure that you can use to retrieve `Course` entities.

Creating Stored Procedures in the Database

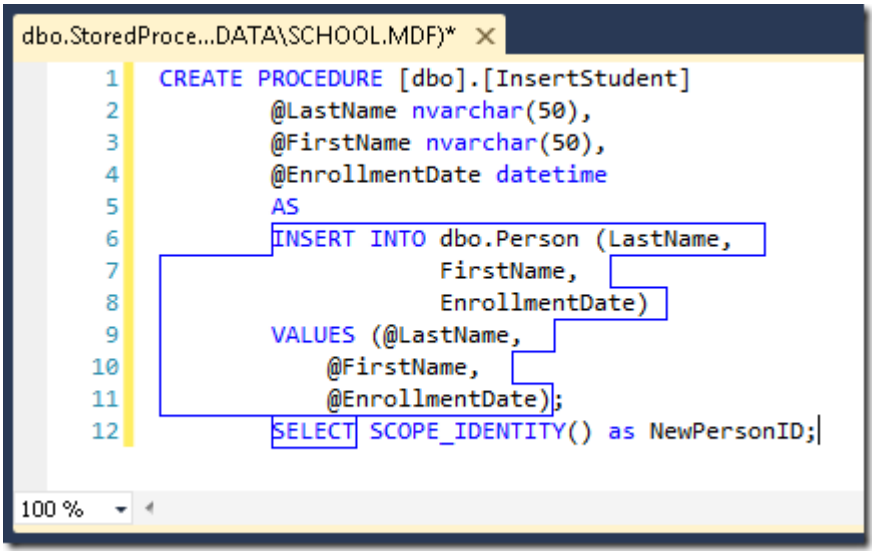
(If you're using the *School.mdf* file from the project available for download with this tutorial, you can skip this section because the stored procedures already exist.)

In **Server Explorer**, expand *School.mdf*, right-click **Stored Procedures**, and select **Add New Stored Procedure**.



Copy the following SQL statements and paste them into the stored procedure window, replacing the skeleton stored procedure.

```
CREATE PROCEDURE [dbo].[InsertStudent]
@LastName nvarchar(50),
@FirstName nvarchar(50),
@EnrollmentDate datetime
                AS
                INSERT INTO dbo.Person(LastName,
FirstName,
EnrollmentDate)
                VALUES (@LastName,
@FirstName,
@EnrollmentDate);
                SELECT SCOPE_IDENTITY()asNewPersonID;
```



`Student` entities have four properties: `PersonID`, `LastName`, `FirstName`, and `EnrollmentDate`. The database generates the ID value automatically, and the stored procedure accepts parameters for the other three. The stored procedure returns the value of the new row's record key so that the Entity Framework can keep track of that in the version of the entity it keeps in memory.

Save and close the stored procedure window.

Create an `InsertInstructor` stored procedure in the same manner, using the following SQL statements:

```
CREATE PROCEDURE [dbo].[InsertInstructor]
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime
                AS
                INSERT INTO dbo.Person(LastName,
FirstName,
HireDate)
                VALUES (@LastName,
@FirstName,
@HireDate);
                SELECT SCOPE_IDENTITY()asNewPersonID;
```

Create `Update` stored procedures for `Student` and `Instructor` entities also. (The database already has a `DeletePerson` stored procedure which will work for both `Instructor` and `Student` entities.)

```
CREATE PROCEDURE [dbo].[UpdateStudent]
@PersonIDint,
@LastName nvarchar(50),
@FirstName nvarchar(50),
@EnrollmentDate datetime
                AS
                UPDATE Person SET LastName=@LastName,
FirstName=@FirstName,
EnrollmentDate=@EnrollmentDate
                WHERE PersonID=@PersonID;


CREATE PROCEDURE [dbo].[UpdateInstructor]
@PersonIDint,
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime
                AS
                UPDATE Person SET LastName=@LastName,
FirstName=@FirstName,
HireDate=@HireDate
                WHERE PersonID=@PersonID;
```

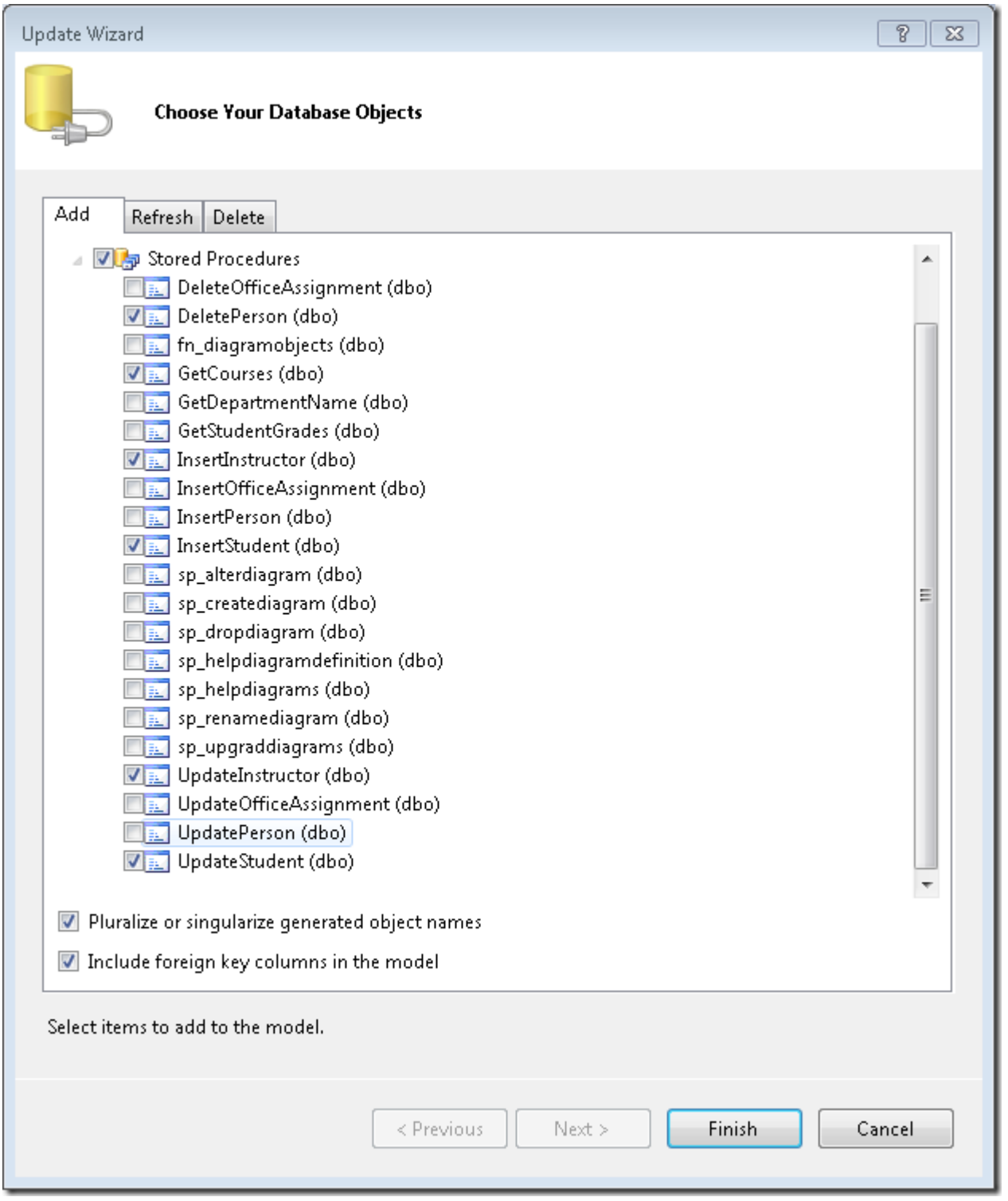
In this tutorial you'll map all three functions -- insert, update, and delete -- for each entity type. The Entity Framework version 4 allows you to map just one or two of these functions to stored procedures without mapping the others, with one exception: if you map the update function but not the delete function, the Entity Framework will throw an exception when you attempt to delete an entity. In the Entity Framework version 3.5, you did not have this much flexibility in mapping stored procedures: if you mapped one function you were required to map all three.

To create a stored procedure that reads rather than updates data, create one that selects all `Course` entities, using the following SQL statements:

```
CREATE PROCEDURE [dbo].[GetCourses]
            AS
            SELECT CourseID,Title,Credits,DepartmentID FROM dbo.Course
```

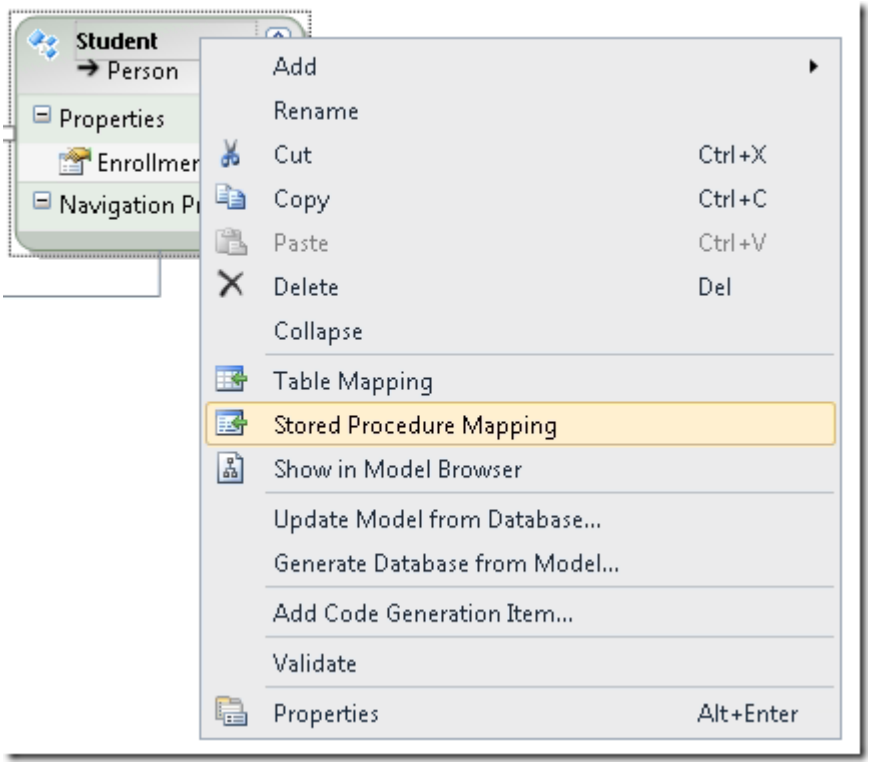
Adding the Stored Procedures to the Data Model

The stored procedures are now defined in the database, but they must be added to the data model to make them available to the Entity Framework. Open *SchoolModel.edmx*, right-click the design surface, and select **Update Model from Database**. In the **Add** tab of the **Choose Your Database Objects** dialog box, expand **Stored Procedures**, select the newly created stored procedures and the `DeletePerson` stored procedure, and then click **Finish**.

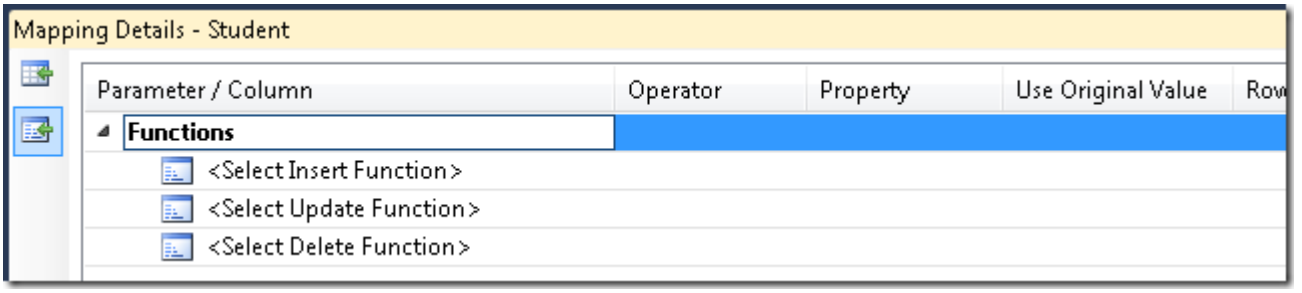


Mapping the Stored Procedures

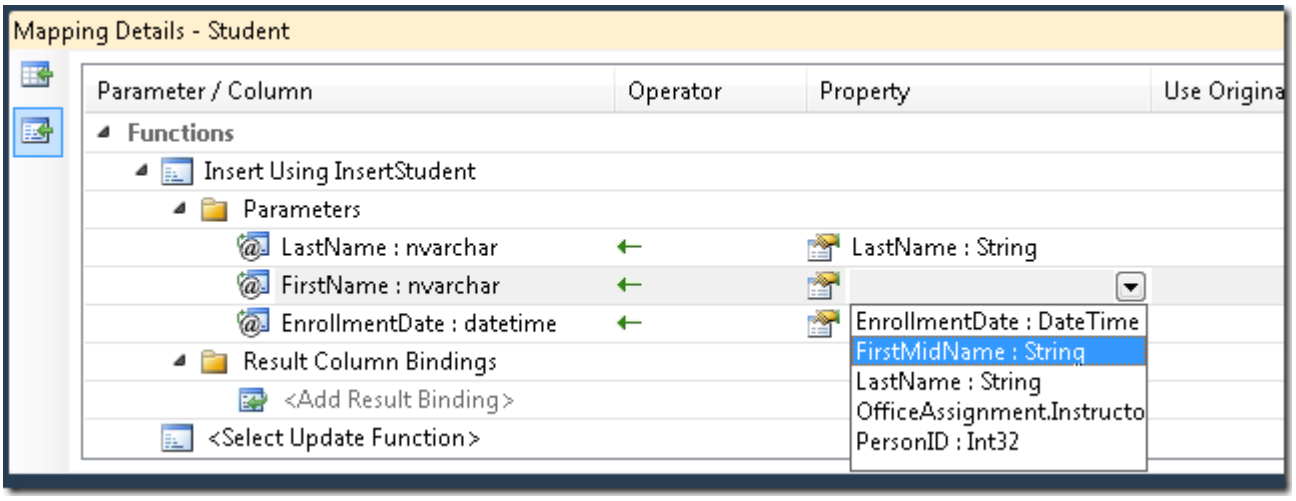
In the data model designer, right-click the `Student` entity and select **Stored Procedure Mapping**.



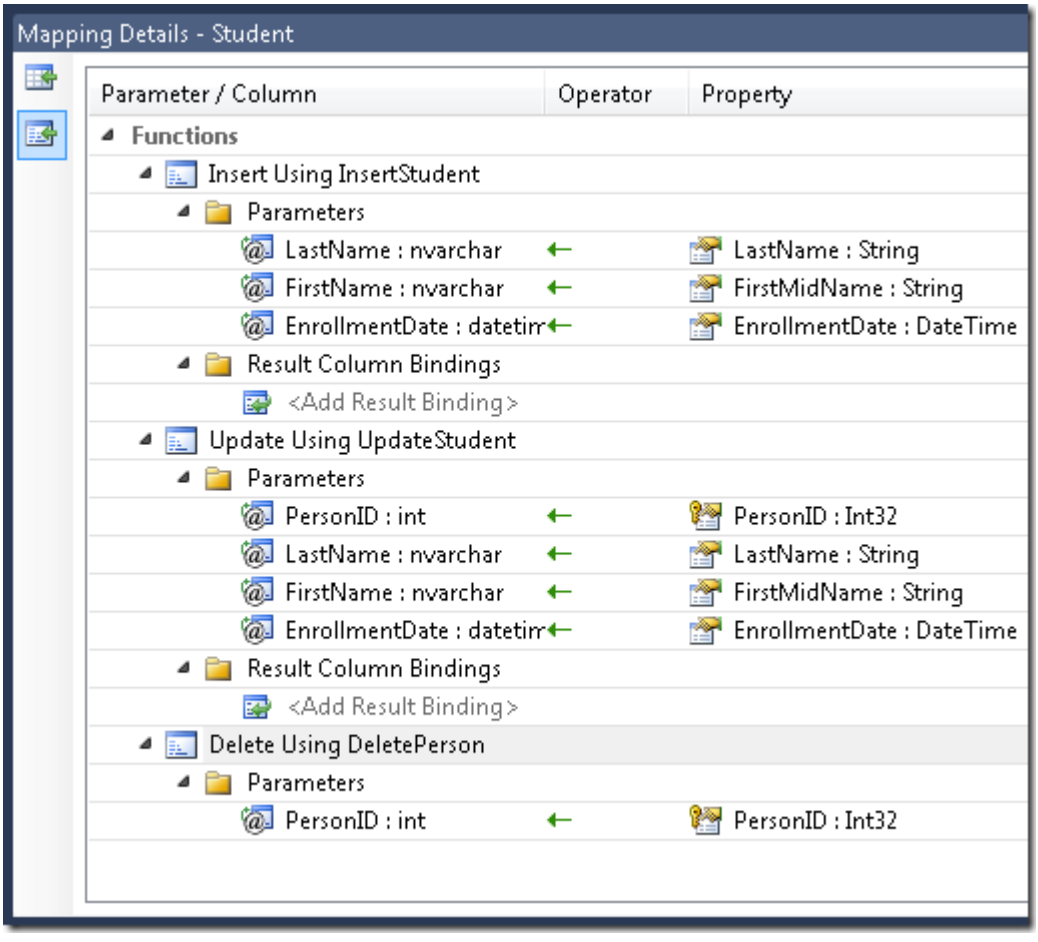
The **Mapping Details** window appears, in which you can specify stored procedures that the Entity Framework should use for inserting, updating, and deleting entities of this type.



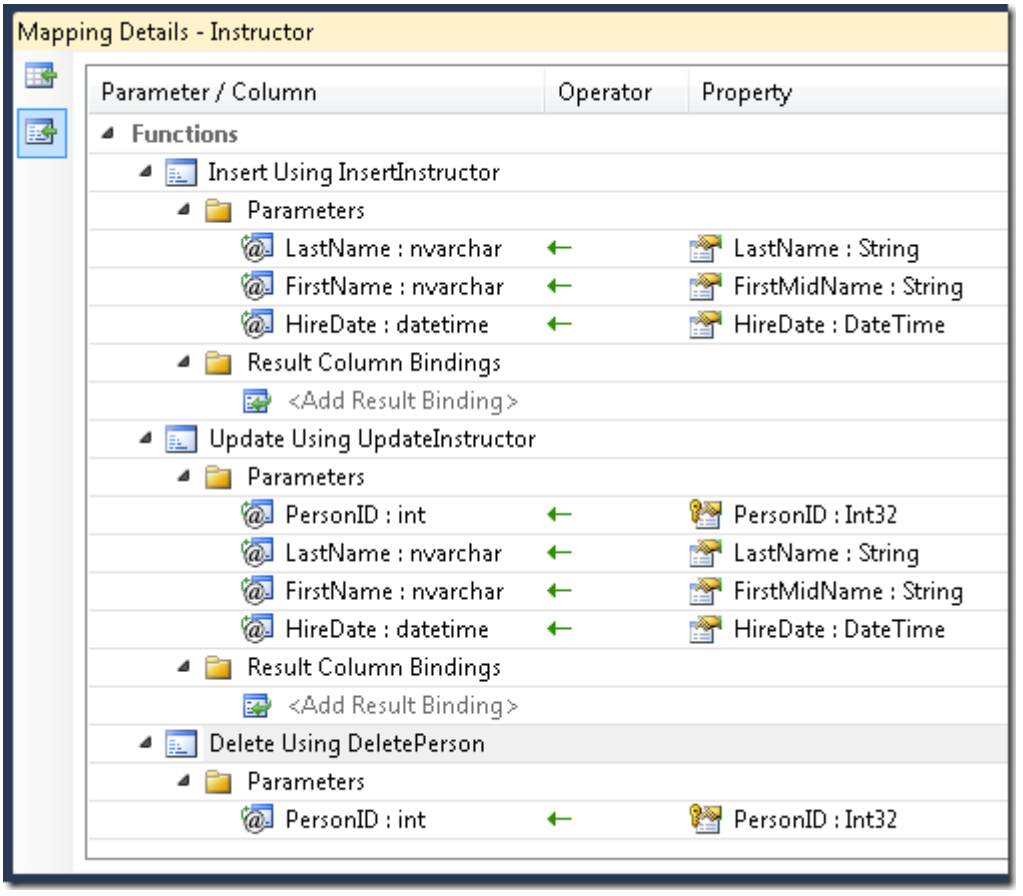
Set the **Insert** function to **InsertStudent**. The window shows a list of stored procedure parameters, each of which must be mapped to an entity property. Two of these are mapped automatically because the names are the same. There's no entity property named `FirstName`, so you must manually select `FirstMidName` from a drop-down list that shows available entity properties. (This is because you changed the name of the `FirstName` property to `FirstMidName` in the first tutorial.)



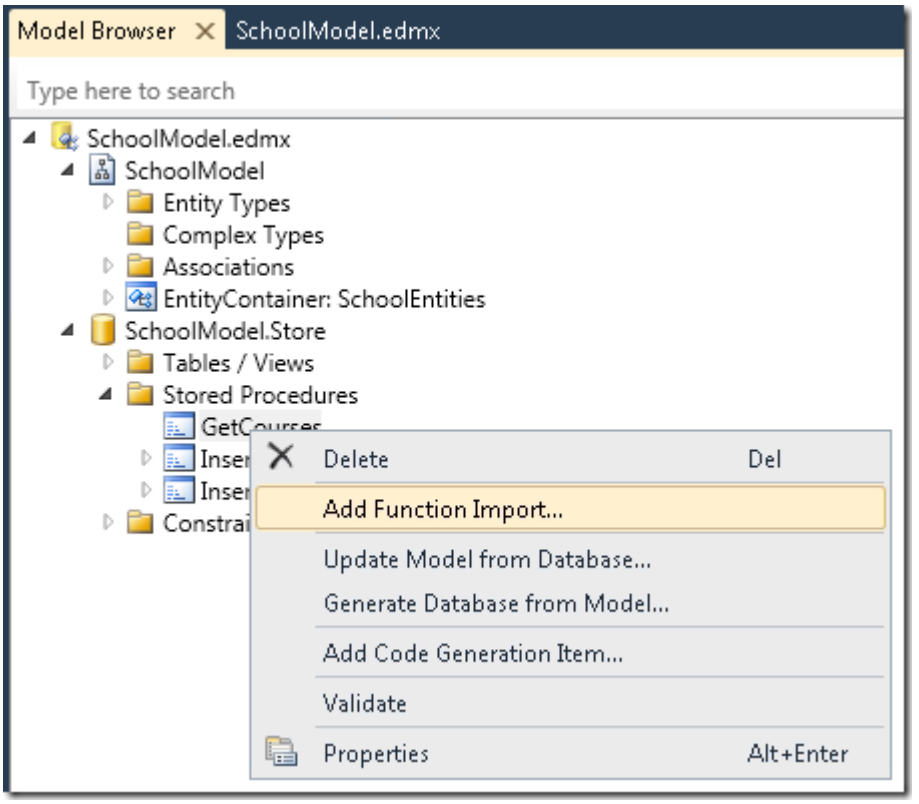
In the same **Mapping Details** window, map the `Update` function to the `UpdateStudent` stored procedure (make sure you specify `FirstMidName` as the parameter value for `FirstName`, as you did for the `Insert` stored procedure) and the `Delete` function to the `DeletePerson` stored procedure.



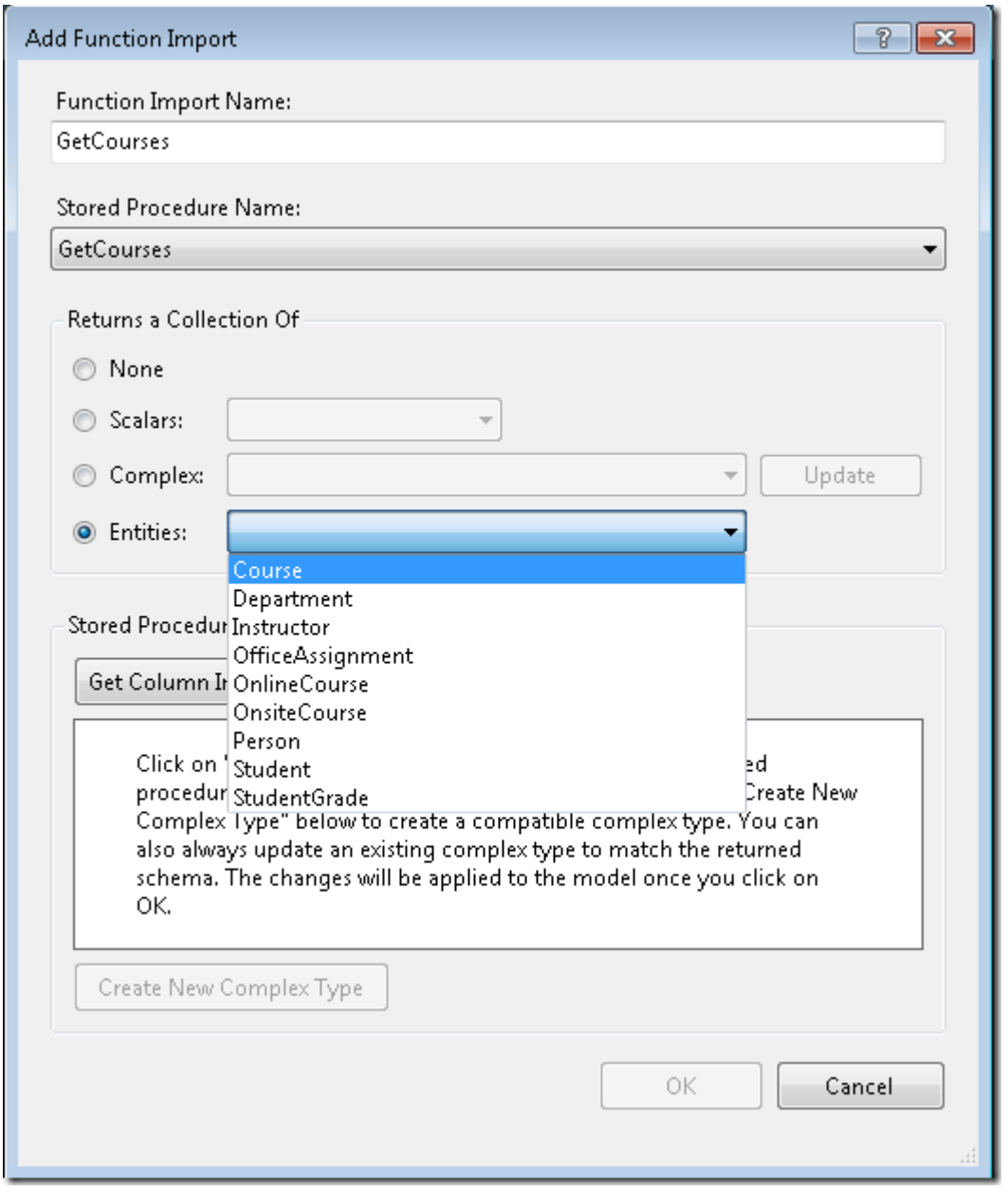
Follow the same procedure to map the insert, update, and delete stored procedures for instructors to the `Instructor` entity.



For stored procedures that read rather than update data, you use the **Model Browser** window to map the stored procedure to the entity type it returns. In the data model designer, right-click the design surface and select **Model Browser**. Open the **SchoolModel.Store** node and then open the **Stored Procedures** node. Then right-click the `GetCourses` stored procedure and select **Add Function Import**.

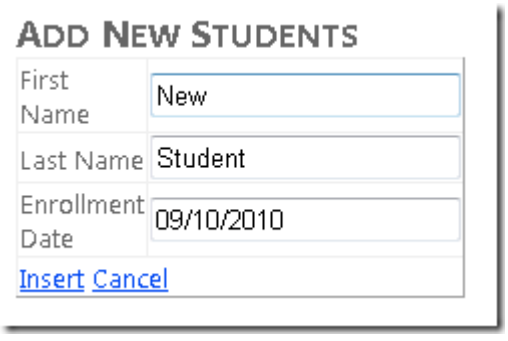


In the **Add Function Import** dialog box, under **Returns a Collection Of** select **Entities**, and then select `Course` as the entity type returned. When you're done, click **OK**. Save and close the *.edmx* file.

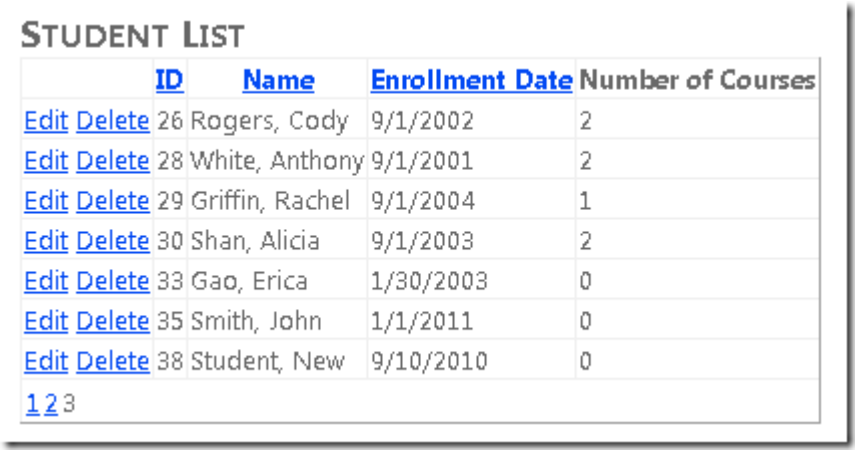


Using Insert, Update, and Delete Stored Procedures

Stored procedures to insert, update, and delete data are used by the Entity Framework automatically after you've added them to the data model and mapped them to the appropriate entities. You can now run the *StudentsAdd.aspx* page, and every time you create a new student, the Entity Framework will use the `InsertStudent` stored procedure to add the new row to the `Student` table.



Run the *Students.aspx* page and the new student appears in the list.



Change the name to verify that the update function works, and then delete the student to verify that the delete function works.

Using Select Stored Procedures

The Entity Framework does not automatically run stored procedures such as `GetCourses`, and you cannot use them with the `EntityDataSource` control. To use them, you call them from code.

Open the *InstructorsCourses.aspx.cs* file. The `PopulateDropDownLists` method uses a LINQ-to-Entities query to retrieve all course entities so that it can loop through the list and determine which ones an instructor is assigned to and which ones are unassigned:

```
var allCourses =(from c in context.Courses
select c).ToList();
```

Replace this with the following code:

```
var allCourses = context.GetCourses();
```

The page now uses the `GetCourses` stored procedure to retrieve the list of all courses. Run the page to verify that it works as it did before.

(Navigation properties of entities retrieved by a stored procedure might not be automatically populated with the data related to those entities, depending on `ObjectContext` default settings. For more information, see Loading Related Objects in the MSDN Library.)

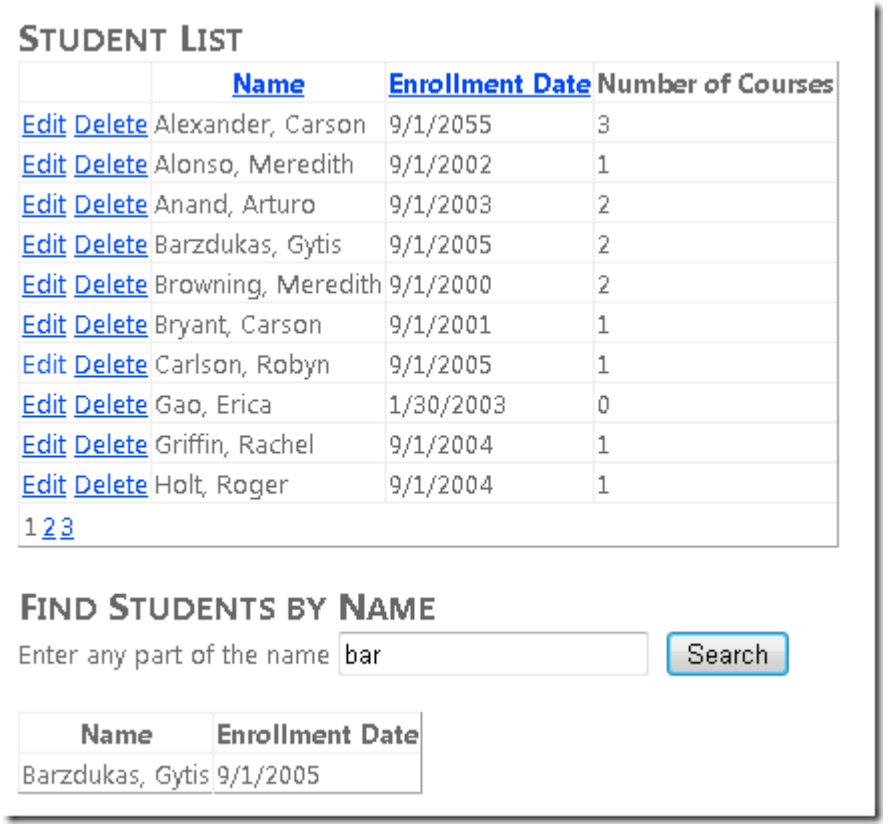
In the next tutorial, you'll learn how to use Dynamic Data functionality to make it easier to program and test data formatting and validation rules. Instead of specifying on each web page rules such as data format strings and whether or not a field is required, you can specify such rules in data model metadata and they're automatically applied on every page.

Using Dynamic Data Functionality to Format and Validate Data

In the previous tutorial you implemented stored procedures. This tutorial will show you how Dynamic Data functionality can provide the following benefits:

- Fields are automatically formatted for display based on their data type.
- Fields are automatically validated based on their data type.
- You can add metadata to the data model to customize formatting and validation behavior. When you do this, you can add the formatting and validation rules in just one place, and they're automatically applied everywhere you access the fields using Dynamic Data controls.

To see how this works, you'll change the controls you use to display and edit fields in the existing *Students.aspx* page, and you'll add formatting and validation metadata to the name and date fields of the `Student` entity type.



Using DynamicField and DynamicControl Controls

Open the *Students.aspx* page and in the `StudentsGridView` control replace the **Name** and **Enrollment Date**`TemplateField` elements with the following markup:

```
<asp:TemplateFieldHeaderText="Name"SortExpression="LastName">
<EditItemTemplate>
<asp:DynamicControlID="LastNameTextBox"runat="server"DataField="LastName"Mode="Edit"/
>
<asp:DynamicControlID="FirstNameTextBox"runat="server"DataField="FirstMidName"Mode="E
dit"/>
</EditItemTemplate>
<ItemTemplate>
<asp:DynamicControlID="LastNameLabel"runat="server"DataField="LastName"Mode="ReadOnly
"/>,
<asp:DynamicControlID="FirstNameLabel"runat="server"DataField="FirstMidName"Mode="Rea
dOnly"/>
</ItemTemplate>
</asp:TemplateField>
<asp:DynamicFieldDataField="EnrollmentDate"HeaderText="Enrollment
Date"SortExpression="EnrollmentDate"/>
```

This markup uses `DynamicControl` controls in place of `TextBox` and `Label` controls in the student name template field, and it uses a `DynamicField` control for the enrollment date. No format strings are specified.

Add a `ValidationSummary` control after the `StudentsGridView` control.

```
<asp:ValidationSummaryID="StudentsValidationSummary"runat="server"ShowSummary="true"
DisplayMode="BulletList"Style="color:Red"/>
```

In the `SearchGridView` control replace the markup for the **Name** and **Enrollment Date** columns as you did in the `StudentsGridView` control, except omit the `EditItemTemplate` element. The `Columns` element of the `SearchGridView` control now contains the following markup:

```
<asp:TemplateFieldHeaderText="Name"SortExpression="LastName">
<ItemTemplate>
<asp:DynamicControlID="LastNameLabel"runat="server"DataField="LastName"Mode="ReadOnly
"/>,
```

```
<asp:DynamicControlID="FirstNameLabel"runat="server"DataField="FirstMidName"Mode="Rea
dOnly"/>
</ItemTemplate>
</asp:TemplateField>
<asp:DynamicFieldDataField="EnrollmentDate"HeaderText="Enrollment
Date"SortExpression="EnrollmentDate"/>
```

Open *Students.aspx.cs* and add the following `using` statement:

```
usingContosoUniversity.DAL;
```

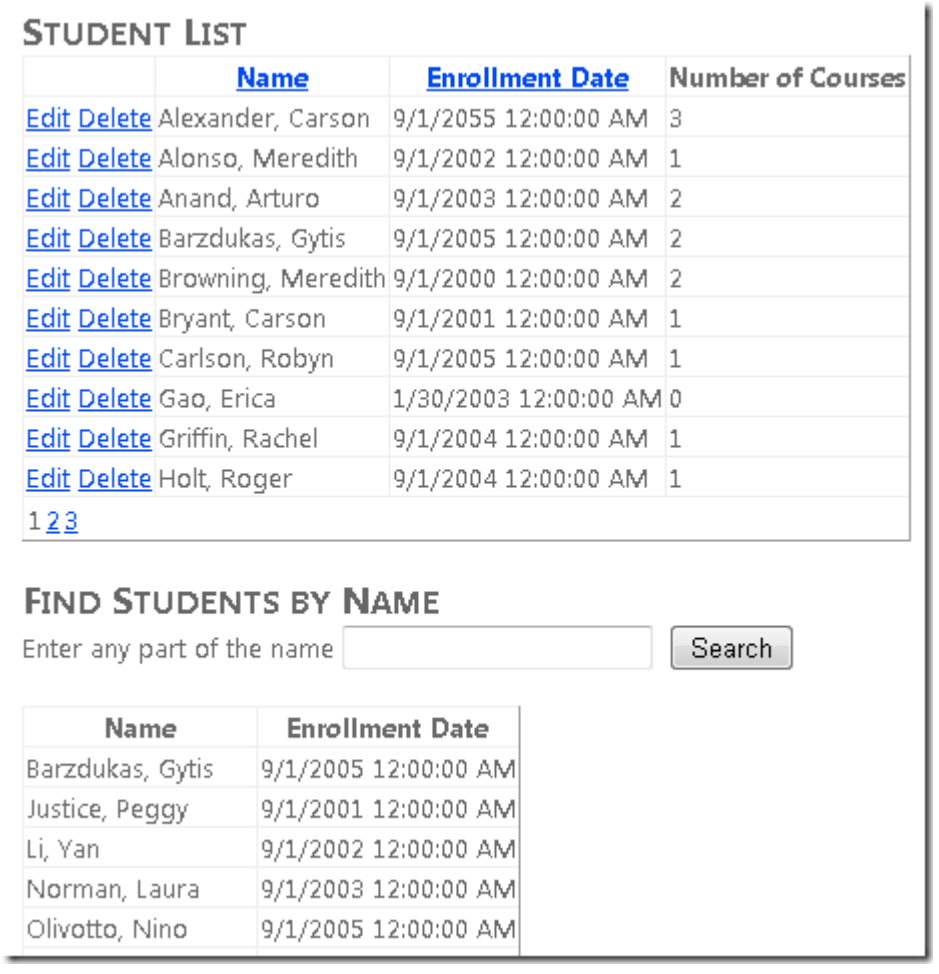
Add a handler for the page's `Init` event:

```
protectedvoidPage_Init(object sender,EventArgs e)
{
StudentsGridView.EnableDynamicData(typeof(Student));
SearchGridView.EnableDynamicData(typeof(Student));
}
```

This code specifies that Dynamic Data will provide formatting and validation in these data-bound controls for fields of the `Student` entity. If you get an error message like the following example when you run the page, it typically means you've forgotten to call the `EnableDynamicData` method in `Page_Init`:

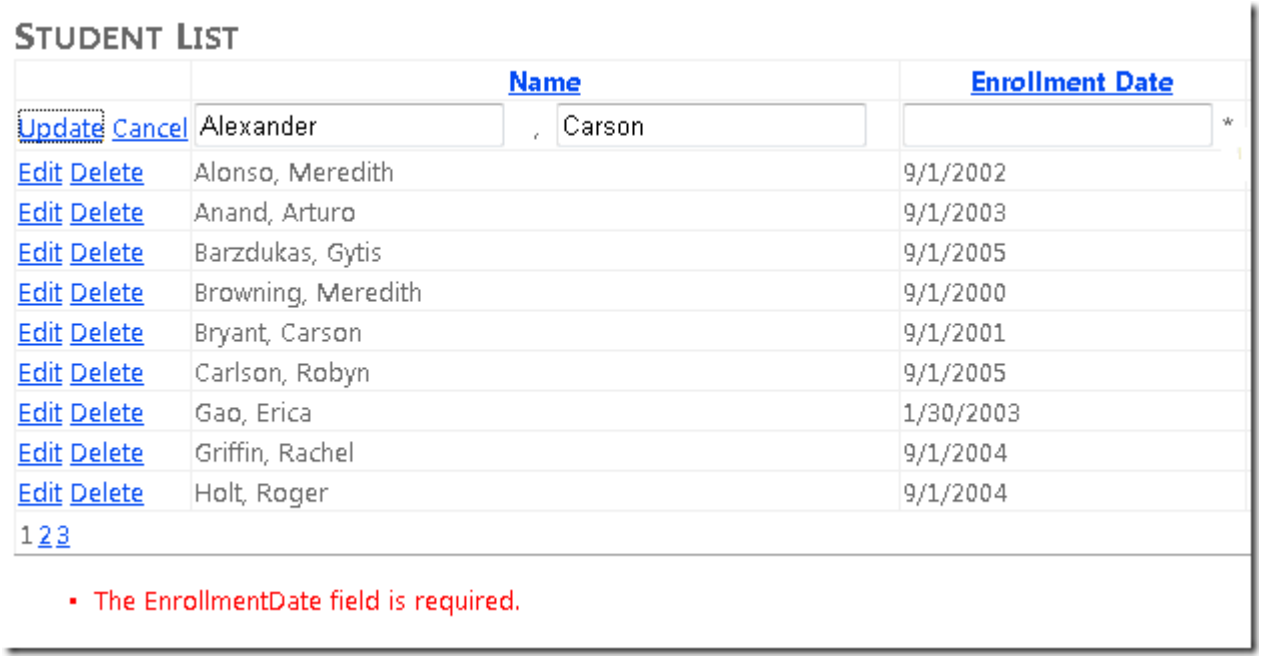
```
Could not determine a MetaTable. A MetaTable could not be determined for the data
source 'StudentsEntityDataSource' and one could not be inferred from the request URL.
```

Run the page.

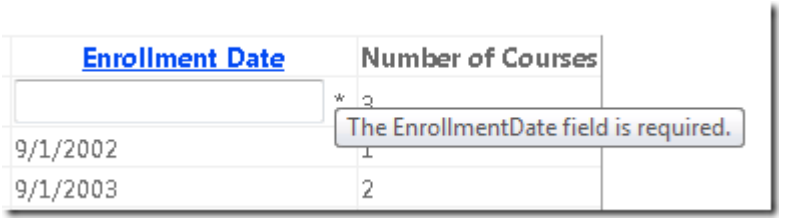


In the **Enrollment Date** column, the time is displayed along with the date because the property type is `DateTime`. You'll fix that later.

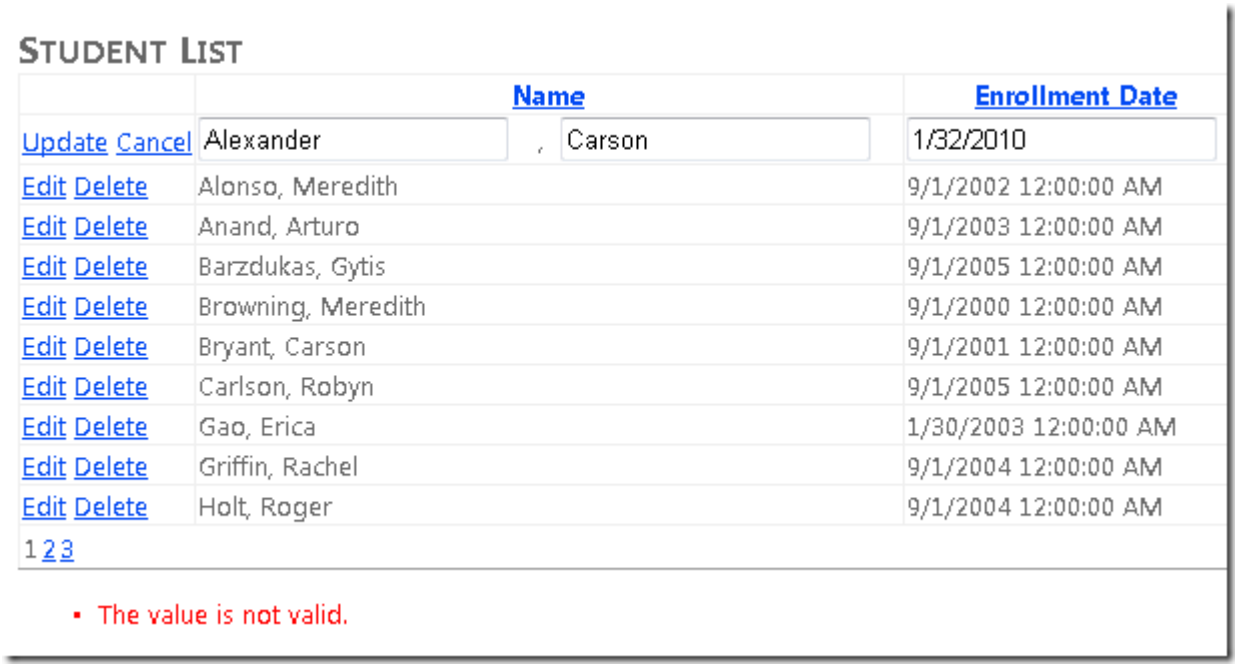
For now, notice that Dynamic Data automatically provides basic data validation. For example, click **Edit**, clear the date field, click **Update**, and you see that Dynamic Data automatically makes this a required field because the value is not nullable in the data model. The page displays an asterisk after the field and an error message in the `ValidationSummary` control:



You could omit the `ValidationSummary` control, because you can also hold the mouse pointer over the asterisk to see the error message:



Dynamic Data will also validate that data entered in the **Enrollment Date** field is a valid date:

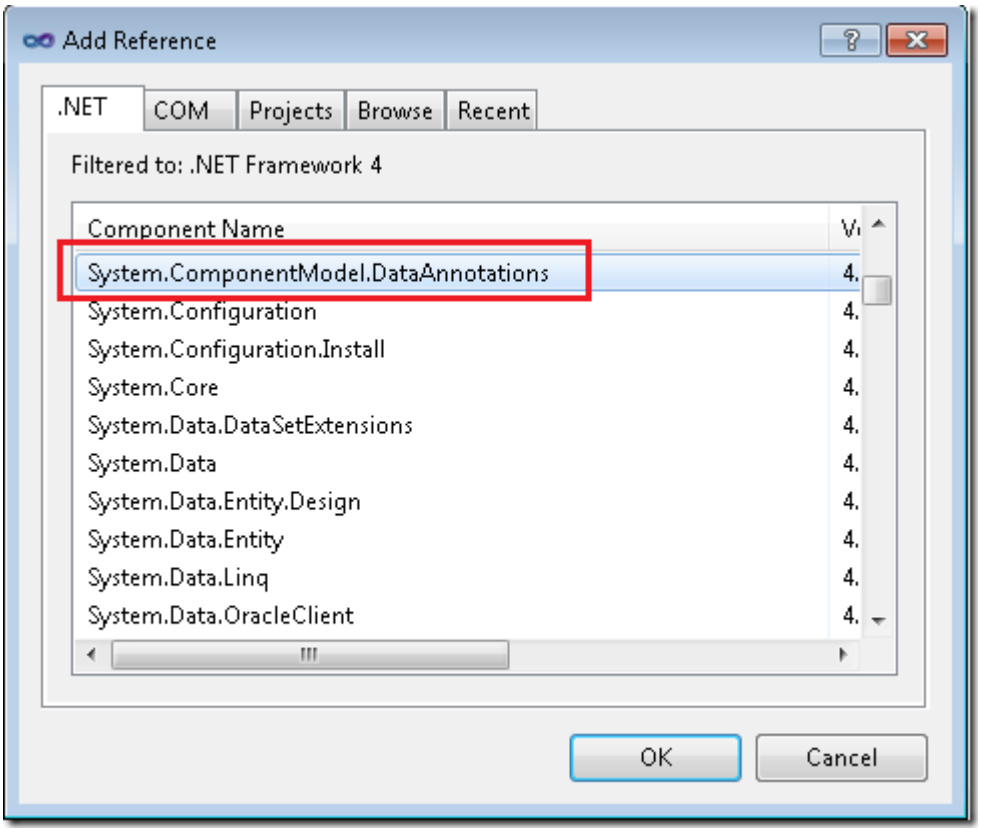


As you can see, this is a generic error message. In the next section you'll see how to customize messages as well as validation and formatting rules.

Adding Metadata to the Data Model

Typically, you want to customize the functionality provided by Dynamic Data. For example, you might change how data is displayed and the content of error messages. You typically also customize data validation rules to provide more functionality than what Dynamic Data provides automatically based on data types. To do this, you create partial classes that correspond to entity types.

In **Solution Explorer**, right-click the **ContosoUniversity** project, select **Add Reference**, and add a reference to `System.ComponentModel.DataAnnotations`.



In the *DAL* folder, create a new class file, name it *Student.cs*, and replace the template code in it with the following code.

```
usingSystem;
usingSystem.ComponentModel;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.DAL
{
[MetadataType(typeof(StudentMetadata))]
publicpartialclassStudent
{
}

publicclassStudentMetadata
{
[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
publicDateTimeEnrollmentDate{get;set;}
```

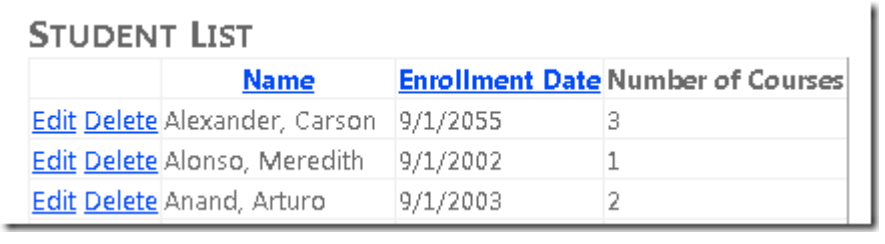
```
[StringLength(25,ErrorMessage="First name must be 25 characters or less in length.")]
[Required(ErrorMessage="First name is required.")]
publicStringFirstMidName{get;set;}

[StringLength(25,ErrorMessage="Last name must be 25 characters or less in length.")]
[Required(ErrorMessage="Last name is required.")]
publicStringLastName{get;set;}
}
}
```

This code creates a partial class for the `Student` entity. The `MetadataType` attribute applied to this partial class identifies the class that you're using to specify metadata. The metadata class can have any name, but using the entity name plus "Metadata" is a common practice.

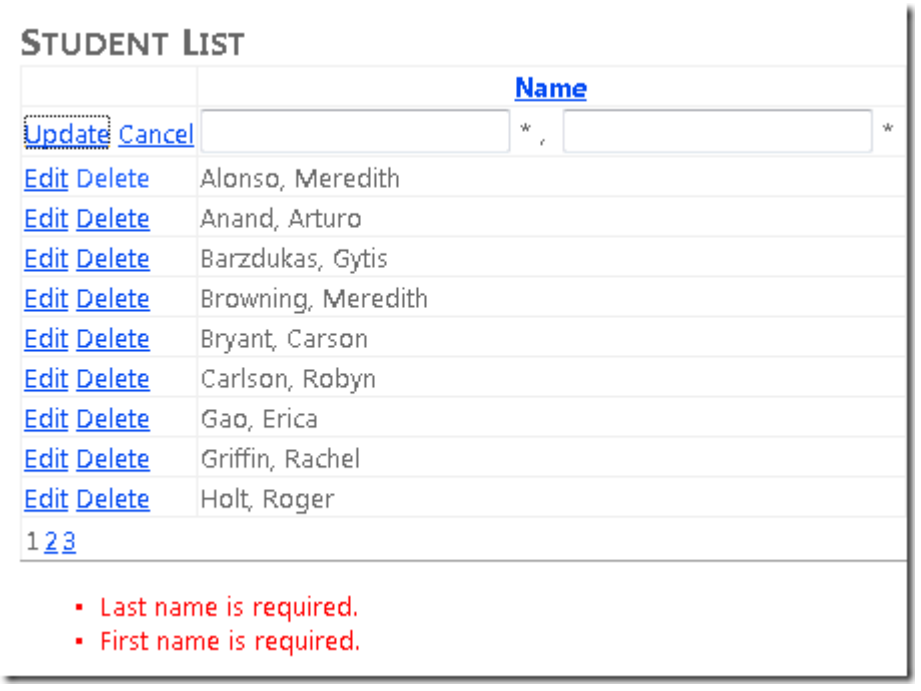
The attributes applied to properties in the metadata class specify formatting, validation, rules, and error messages. The attributes shown here will have the following results:

- `EnrollmentDate` will display as a date (without a time).
- Both name fields must be 25 characters or less in length, and a custom error message is provided.
- Both name fields are required, and a custom error message is provided.

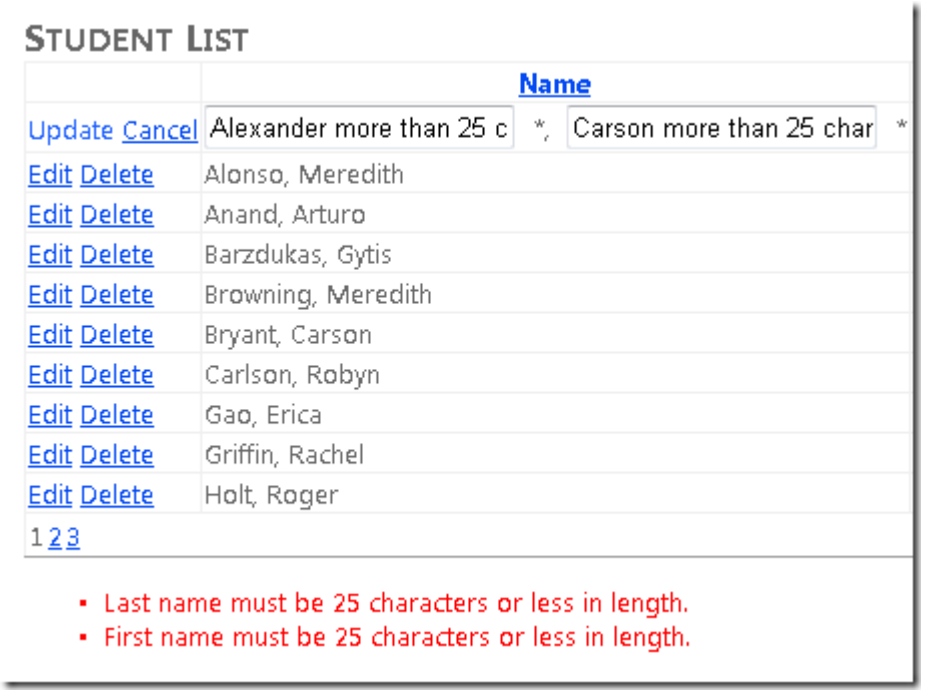
Run the *Students.aspx* page again, and you see that the dates are now displayed without times:



Edit a row and try to clear the values in the name fields. The asterisks indicating field errors appear as soon as you leave a field, before you click **Update**. When you click **Update**, the page displays the error message text you specified.



Try to enter names that are longer than 25 characters, click **Update**, and the page displays the error message text you specified.



Now that you've set up these formatting and validation rules in the data model metadata, the rules will automatically be applied on every page that displays or allows changes to these fields, so long as you use `DynamicControl` or `DynamicField` controls. This reduces the amount of redundant code you have to write,

which makes programming and testing easier, and it ensures that data formatting and validation are consistent throughout an application.

The ObjectDataSource Control

The `EntityDataSource` control enables you to create an application very quickly, but it typically requires you to keep a significant amount of business logic and data-access logic in your *.aspx* pages. If you expect your application to grow in complexity and to require ongoing maintenance, you can invest more development time up front in order to create an *n-tier* or *layered* application structure that's more maintainable. To implement this architecture, you separate the presentation layer from the business logic layer (BLL) and the data access layer (DAL). One way to implement this structure is to use the `ObjectDataSource` control instead of the `EntityDataSource` control. When you use the `ObjectDataSource` control, you implement your own data-access code and then invoke it in *.aspx* pages using a control that has many of the same features as other data-source controls. This lets you combine the advantages of an n-tier approach with the benefits of using a Web Forms control for data access.

The `ObjectDataSource` control gives you more flexibility in other ways as well. Because you write your own data-access code, it's easier to do more than just read, insert, update, or delete a specific entity type, which are the tasks that the `EntityDataSource` control is designed to perform. For example, you can perform logging every time an entity is updated, archive data whenever an entity is deleted, or automatically check and update related data as needed when inserting a row with a foreign key value.

Business Logic and Repository Classes

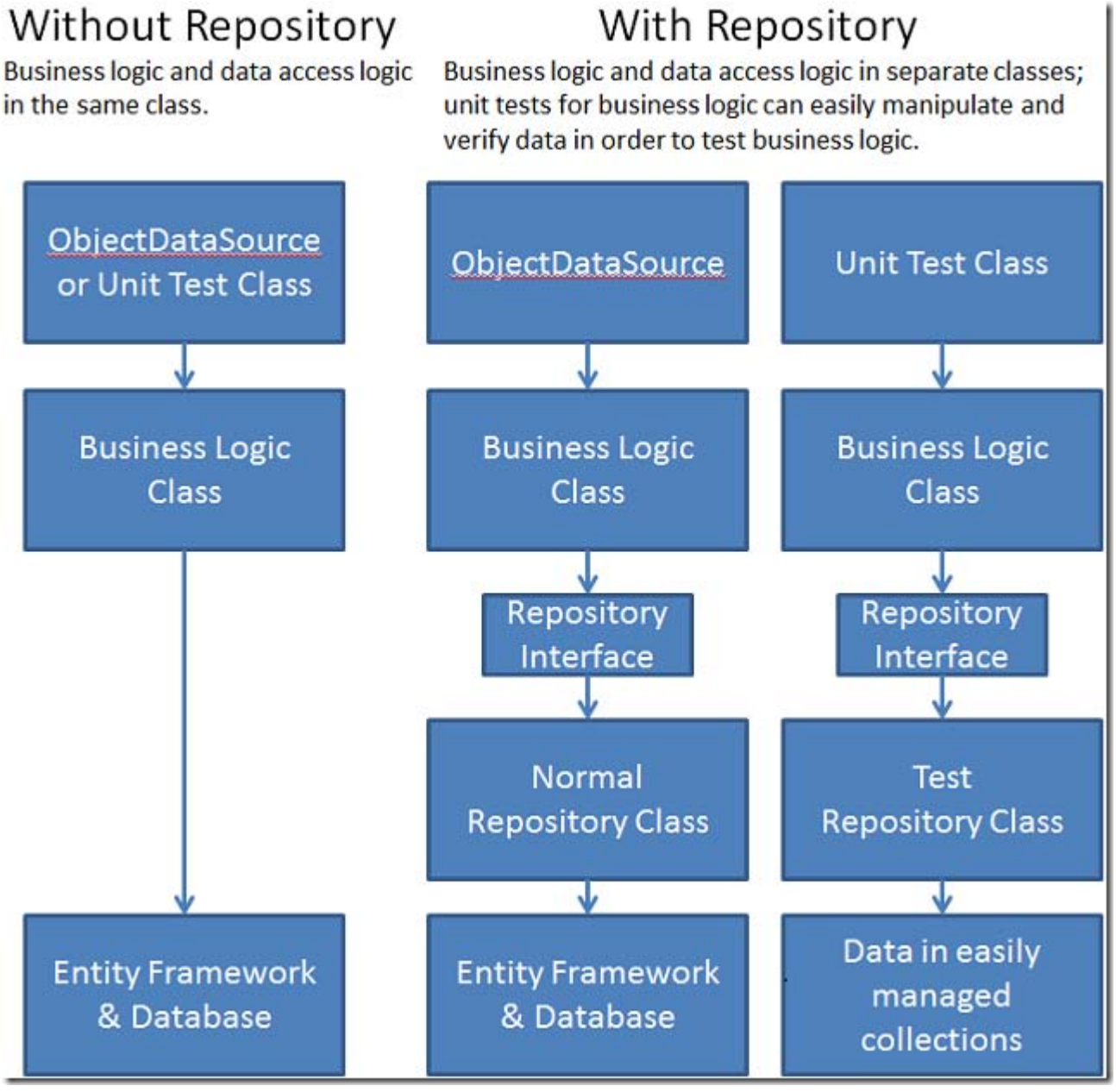
An `ObjectDataSource` control works by invoking a class that you create. The class includes methods that retrieve and update data, and you provide the names of those methods to the `ObjectDataSource` control in markup. During rendering or postback processing, the `ObjectDataSource` calls the methods that you've specified.

Besides basic CRUD operations, the class that you create to use with the `ObjectDataSource` control might need to execute business logic when the `ObjectDataSource` reads or updates data. For example, when you update a department, you might need to validate that no other departments have the same administrator because one person cannot be administrator of more than one department.

In some `ObjectDataSource` documentation, such as the ObjectDataSource Class overview, the control calls a class referred to as a *business object* that includes both business logic and data-access logic. In this tutorial you will create separate classes for business logic and for data-access logic. The class that encapsulates data-access logic is called a *repository*. The business logic class includes both business-logic methods and data-access methods, but the data-access methods call the repository to perform data-access tasks.

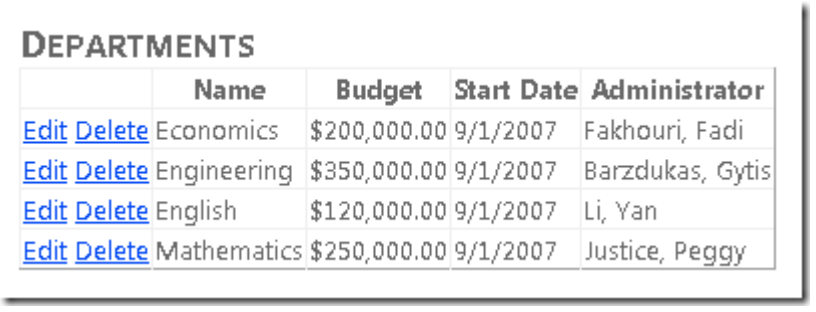
You will also create an abstraction layer between your BLL and DAL that facilitates automated unit testing of the BLL. This abstraction layer is implemented by creating an interface and using the interface when you instantiate the repository in the business-logic class. This makes it possible for you to provide the business-logic class with a reference to any object that implements the repository interface. For normal operation, you provide a repository object that works with the Entity Framework. For testing, you provide a repository object that works with data stored in a way that you can easily manipulate, such as class variables defined as collections.

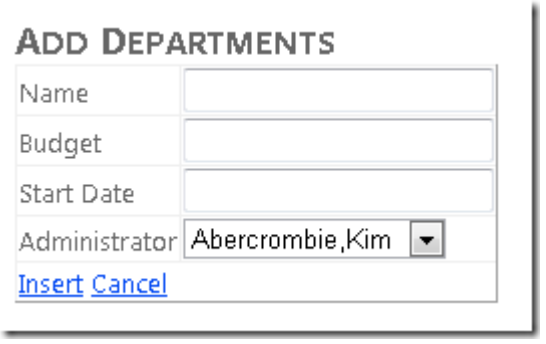
The following illustration shows the difference between a business-logic class that includes data-access logic without a repository and one that uses a repository.



You will begin by creating web pages in which the `ObjectDataSource` control is bound directly to a repository because it only performs basic data-access tasks. In the next tutorial you will create a business logic class with validation logic and bind the `ObjectDataSource` control to that class instead of to the repository class. You will also create unit tests for the validation logic. In the third tutorial in this series you will add sorting and filtering functionality to the application.

The pages you create in this tutorial work with the `Departments` entity set of the data model that you created in previous tutorials in this series.





Updating the Database and the Data Model

You will begin this tutorial by making two changes to the database, both of which require corresponding changes to the data model that you created earlier. In one of the earlier tutorials, you made changes in the designer manually to synchronize the data model with the database after a database change. In this tutorial, you will use the designer's **Update Model From Database** tool to update the data model automatically.

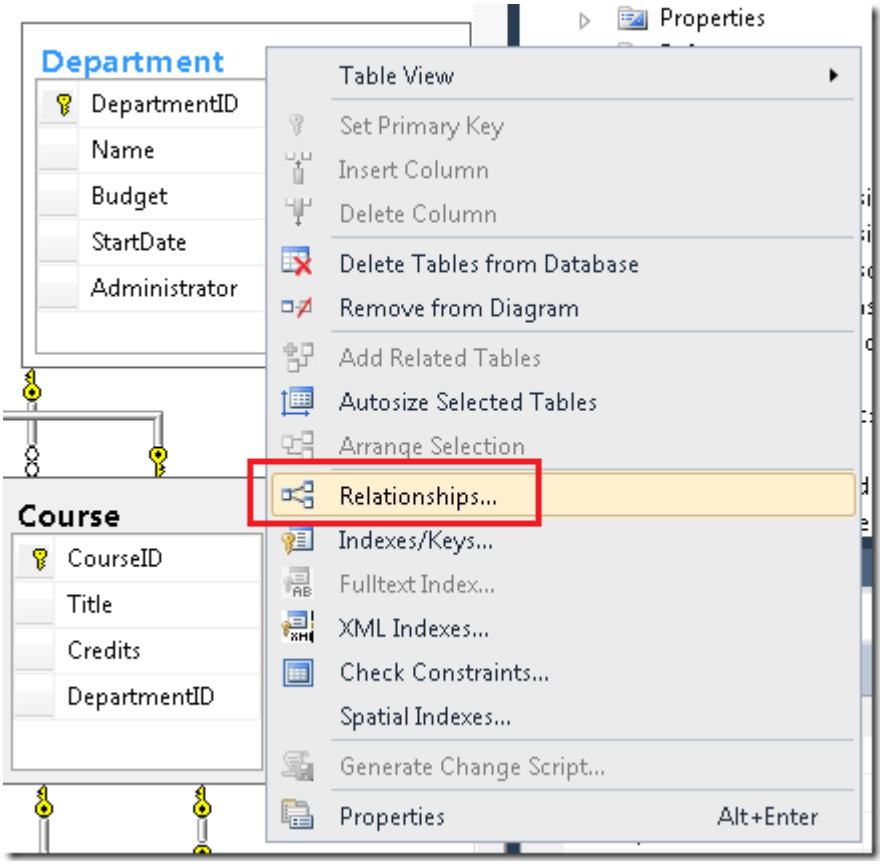
Adding a Relationship to the Database

In Visual Studio, open the Contoso University web application you created in the previous tutorials, and then open the `SchoolDiagram` database diagram.

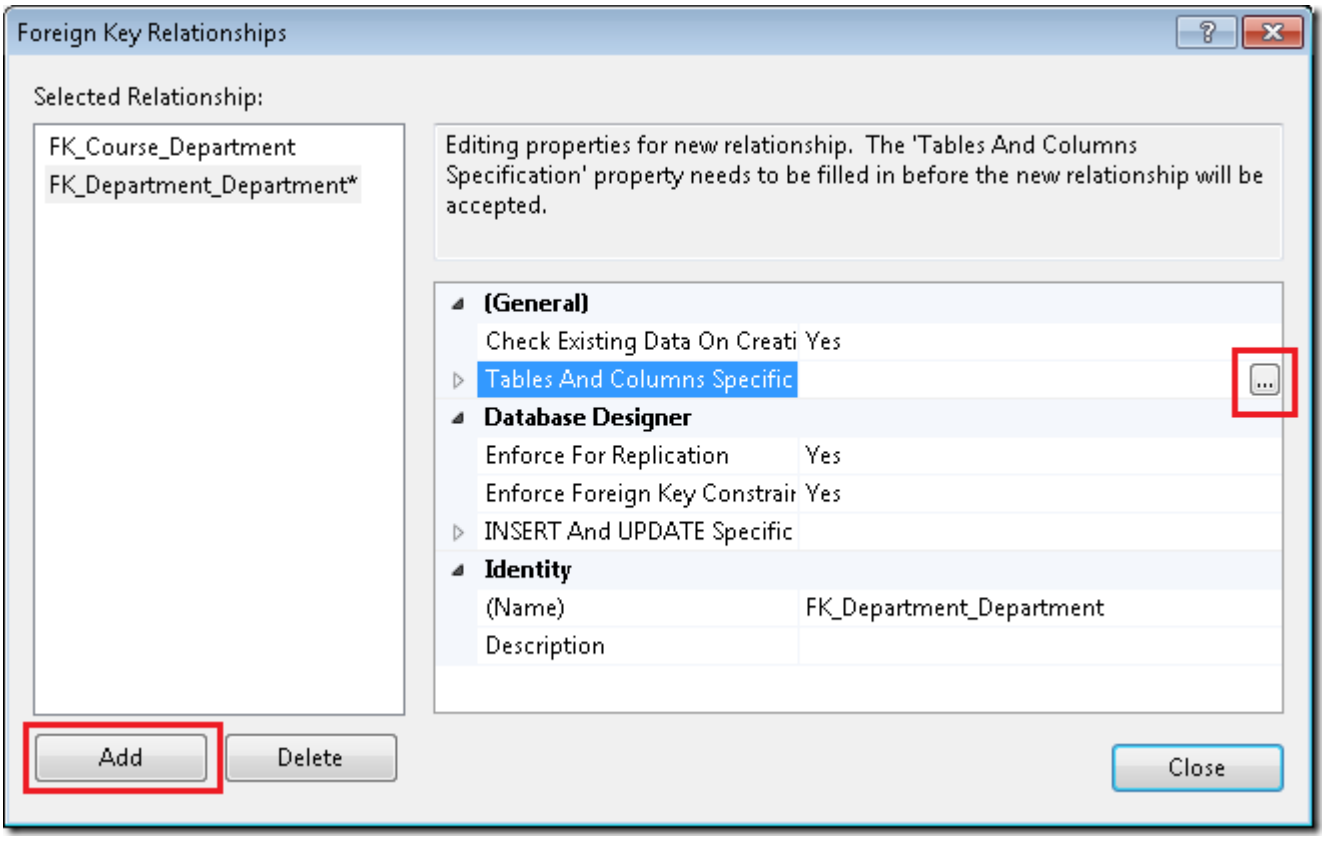
If you look at the `Department` table in the database diagram, you will see that it has an `Administrator` column. This column is a foreign key to the `Person` table, but no foreign key relationship is defined in the

database. You need to create the relationship and update the data model so that the Entity Framework can automatically handle this relationship.

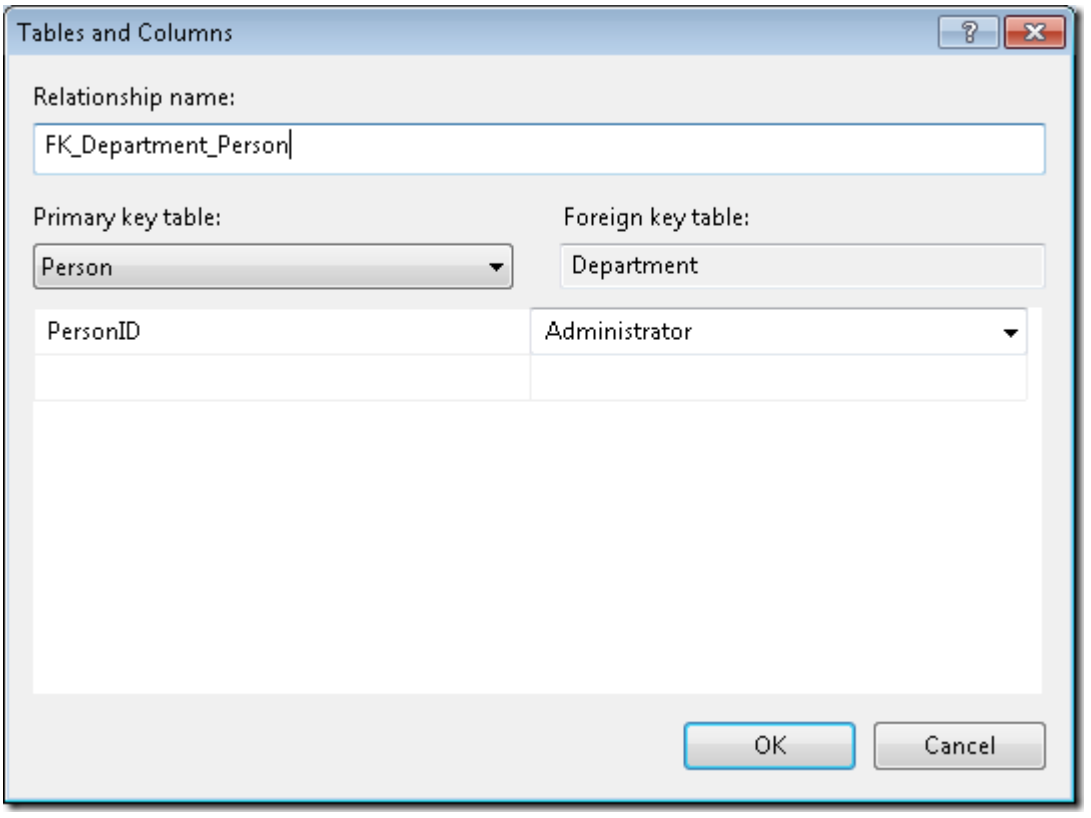
In the database diagram, right-click the `Department` table, and select **Relationships**.



In the **Foreign Key Relationships** box click **Add**, then click the ellipsis for **Tables and Columns Specification**.



In the **Tables and Columns** dialog box, set the primary key table and field to `Person` and `PersonID`, and set the foreign key table and field to `Department` and `Administrator`. (When you do this, the relationship name will change from `FK_Department_Department` to `FK_Department_Person`.)



Click **OK** in the **Tables and Columns** box, click **Close** in the **Foreign Key Relationships** box, and save the changes. If you're asked if you want to save the `Person` and `Department` tables, click **Yes**.

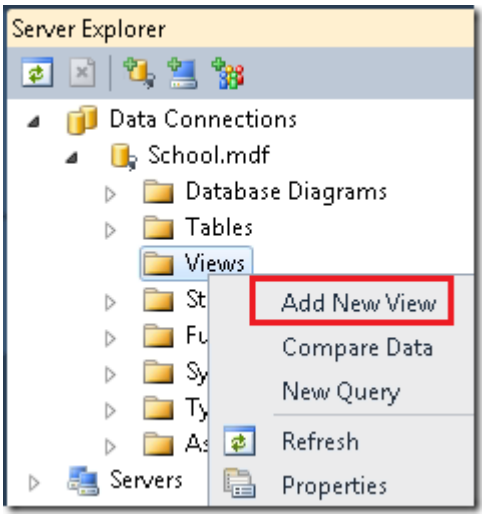
Note If you've deleted `Person` rows that correspond to data that's already in the `Administrator` column, you will not be able to save this change. In that case, use the table editor in **Server Explorer** to make sure that the `Administrator` value in every `Department` row contains the ID of a record that actually exists in the `Person` table.

After you save the change, you will not be able to delete a row from the `Person` table if that person is a department administrator. In a production application, you would provide a specific error message when a database constraint prevents a deletion, or you would specify a cascading delete. For an example of how to specify a cascading delete, see The Entity Framework and ASP.NET – Getting Started Part 2.

Adding a View to the Database

In the new *Departments.aspx* page that you will be creating, you want to provide a drop-down list of instructors, with names in "last, first" format so that users can select department administrators. To make it easier to do that, you will create a view in the database. The view will consist of just the data needed by the drop-down list: the full name (properly formatted) and the record key.

In **Server Explorer**, expand *School.mdf*, right-click the **Views** folder, and select **Add New View**.



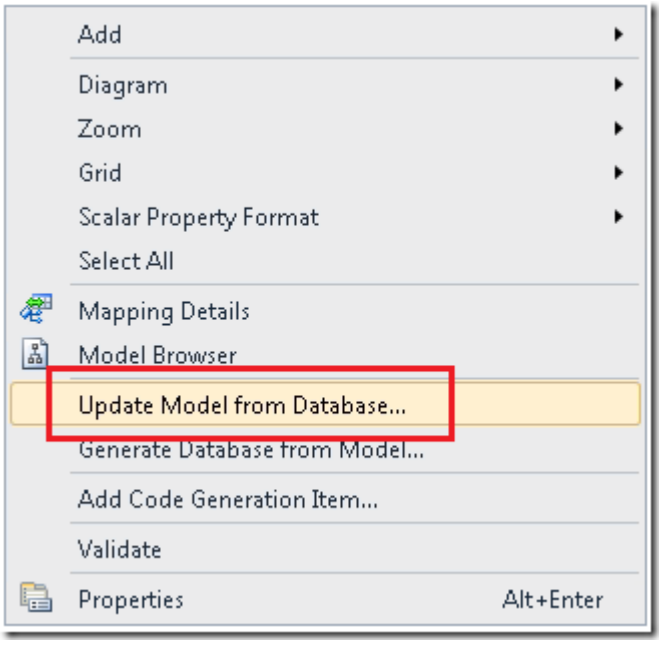
Click **Close** when the **Add Table** dialog box appears, and paste the following SQL statement into the SQL pane:

```
SELECT        LastName+','+FirstName AS FullName,PersonID
FROM          dbo.Person
WHERE        (HireDate IS NOT NULL)
```

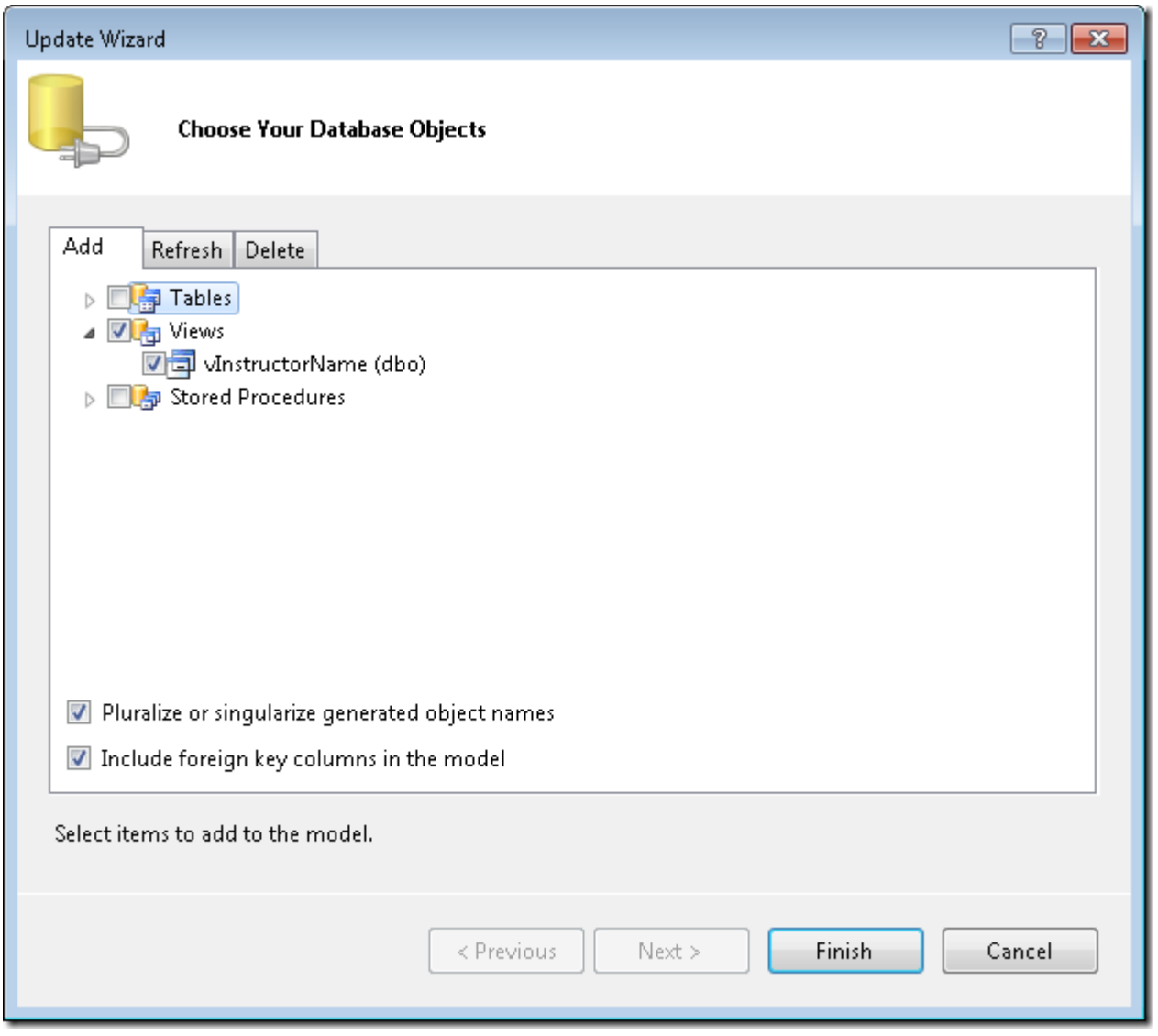
Save the view as `vInstructorName`.

Updating the Data Model

In the *DAL* folder, open the *SchoolModel.edmx* file, right-click the design surface, and select **Update Model from Database**.

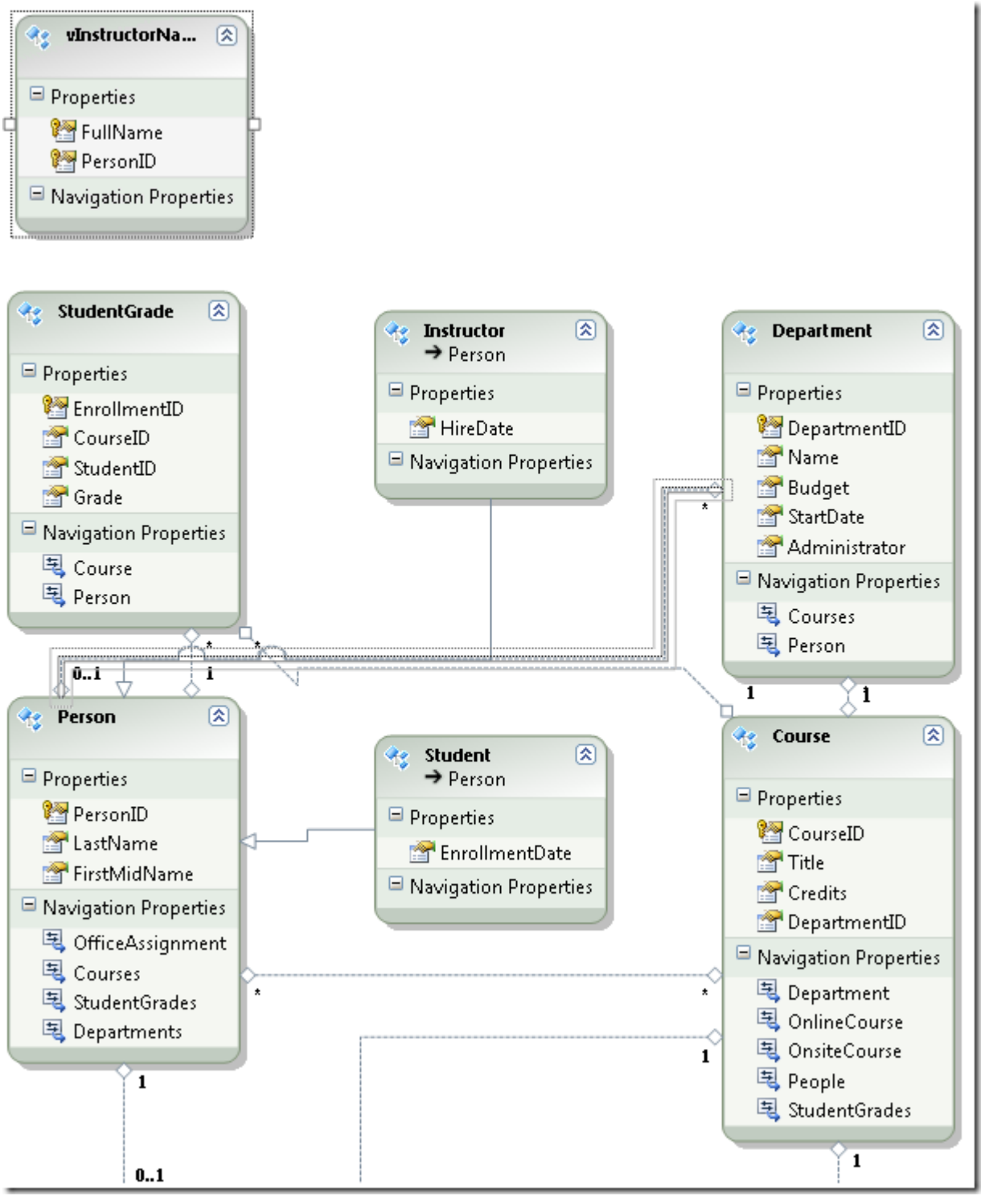


In the **Choose Your Database Objects** dialog box, select the **Add** tab and select the view you just created.



Click **Finish**.

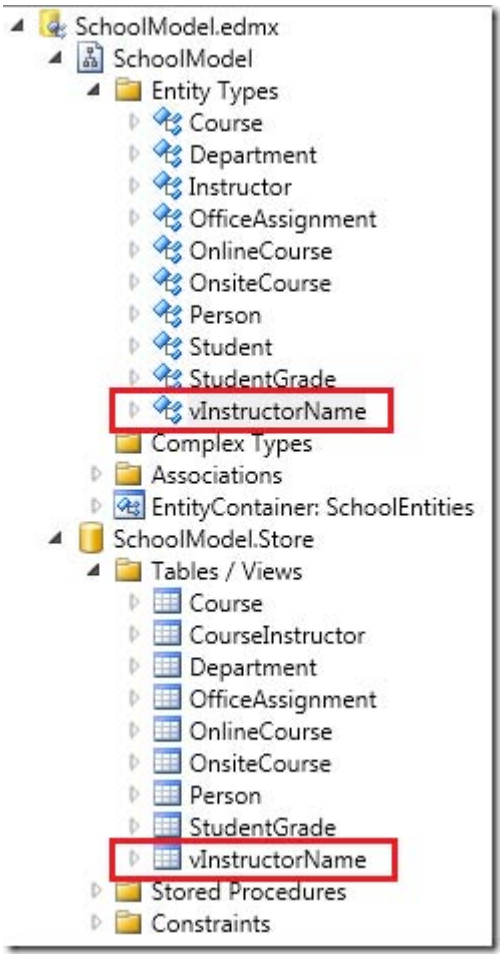
In the designer, you see that the tool created a `vInstructorName` entity and a new association between the `Department` and `Person` entities.



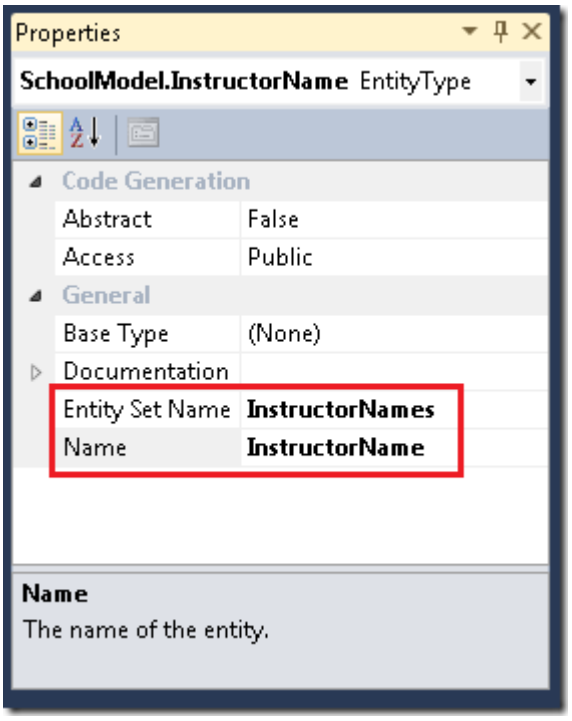
Note In the **Output** and **Error List** windows you might see a warning message informing you that the tool automatically created a primary key for the new `vInstructorName` view. This is expected behavior.

When you refer to the new `vInstructorName` entity in code, you don't want to use the database convention of prefixing a lower-case "v" to it. Therefore, you will rename the entity and entity set in the model.

Open the **Model Browser**. You see `vInstructorName` listed as an entity type and a view.



Under **SchoolModel** (not **SchoolModel.Store**), right-click **vInstructorName** and select **Properties**. In the **Properties** window, change the **Name** property to "InstructorName" and change the **Entity Set Name** property to "InstructorNames".



Save and close the data model, and then rebuild the project.

Using a Repository Class and an ObjectDataSource Control

Create a new class file in the *DAL* folder, name it *SchoolRepository.cs*, and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingContosoUniversity.DAL;

namespaceContosoUniversity.DAL
{
publicclassSchoolRepository:IDisposable
{
privateSchoolEntities context =newSchoolEntities();

publicIEnumerable<Department>GetDepartments()
{
return context.Departments.Include("Person").ToList();
```

```
}

privatebool disposedValue =false;

protectedvirtualvoidDispose(bool disposing)
{
if(!this.disposedValue)
{
if(disposing)
{
                    context.Dispose();
}
}
this.disposedValue =true;
}

publicvoidDispose()
{
Dispose(true);
            GC.SuppressFinalize(this);
}

}
}
```

This code provides a single `GetDepartments` method that returns all of the entities in the `Departments` entity set. Because you know that you will be accessing the `Person` navigation property for every row returned, you specify eager loading for that property by using the `Include` method. The class also implements the `IDisposable` interface to ensure that the database connection is released when the object is disposed.

Note A common practice is to create a repository class for each entity type. In this tutorial, one repository class for multiple entity types is used. For more information about the repository pattern, see the posts in the Entity Framework team's blog and Julie Lerman's blog.

The `GetDepartments` method returns an `IEnumerable` object rather than an `IQueryable` object in order to ensure that the returned collection is usable even after the repository object itself is disposed. An `IQueryable` object can cause database access whenever it's accessed, but the repository object might be disposed by the time a databound control attempts to render the data. You could return another collection type, such as an

`IList` object instead of an `IEnumerable` object. However, returning an `IEnumerable` object ensures that you can perform typical read-only list processing tasks such as `foreach` loops and LINQ queries, but you cannot add to or remove items in the collection, which might imply that such changes would be persisted to the database.

Create a *Departments.aspx* page that uses the *Site.Master* master page, and add the following markup in the `Content` control named `Content2`:

```
<h2>Departments</h2>
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.DAL.SchoolRepository"
DataObjectTypeName="ContosoUniversity.DAL.Department"
SelectMethod="GetDepartments">
</asp:ObjectDataSource>
<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource">
<Columns>
<asp:CommandFieldShowEditButton="True"ShowDeleteButton="True"
ItemStyle-VerticalAlign="Top">
</asp:CommandField>
<asp:DynamicFieldDataField="Name"HeaderText="Name"SortExpression="Name"ItemStyle-
VerticalAlign="Top"/>
<asp:DynamicFieldDataField="Budget"HeaderText="Budget"SortExpression="Budget"ItemStyl
e-VerticalAlign="Top"/>
<asp:DynamicFieldDataField="StartDate"HeaderText="Start Date"ItemStyle-
VerticalAlign="Top"/>
<asp:TemplateFieldHeaderText="Administrator"SortExpression="Person.LastName"ItemStyle
-VerticalAlign="Top">
<ItemTemplate>
<asp:Label ID="AdministratorLastNameLabel" runat="server" Text='<%#
Eval("Person.LastName") %>'></asp:Label>,
<asp:Label ID="AdministratorFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```

This markup creates an `ObjectDataSource` control that uses the repository class you just created, and a `GridView` control to display the data. The `GridView` control specifies **Edit** and **Delete** commands, but you haven't added code to support them yet.

Several columns use `DynamicField` controls so that you can take advantage of automatic data formatting and validation functionality. For these to work, you will have to call the `EnableDynamicData` method in the `Page_Init` event handler. (`DynamicControl` controls are not used in the `Administrator` field because they don't work with navigation properties.)

The `Vertical-Align="Top"` attributes will become important later when you add a column that has a nested `GridView` control to the grid.

Open the *Departments.aspx.cs* file and add the following `using` statement:

```
usingContosoUniversity.DAL;
```

Then add the following handler for the page's `Init` event:

```
protectedvoidPage_Init(object sender,EventArgs e)
{
DepartmentsGridView.EnableDynamicData(typeof(Department));
}
```

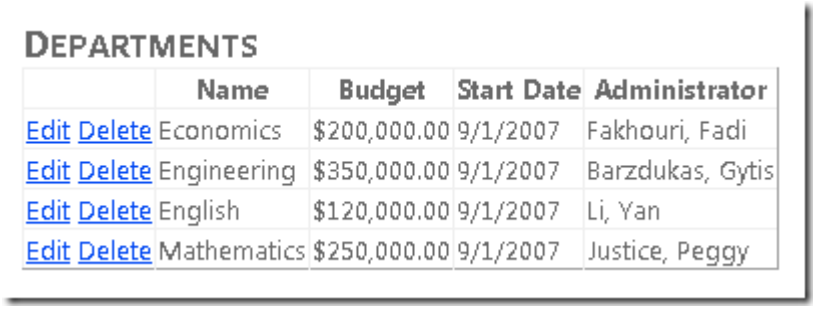
In the *DAL* folder, create a new class file named *Department.cs* and replace the existing code with the following code:

```
usingSystem;
usingSystem.ComponentModel;
usingSystem.ComponentModel.DataAnnotations;

namespaceContosoUniversity.DAL
{
[MetadataType(typeof(DepartmentMetaData))]
publicpartialclassDepartment
{
}
```

```
publicclassDepartmentMetaData
{
[DataType(DataType.Currency)]
[Range(0,1000000,ErrorMessage="Budget must be less than $1,000,000.00")]
publicDecimalBudget{get;set;}

[DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
publicDateTimeStartDate{get;set;}

}
}
```

This code adds metadata to the data model. It specifies that the `Budget` property of the `Department` entity actually represents currency although its data type is `Decimal`, and it specifies that the value must be between 0 and $1,000,000.00. It also specifies that the `StartDate` property should be formatted as a date in the format mm/dd/yyyy.

Run the *Departments.aspx* page.



DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy

Notice that although you did not specify a format string in the *Departments.aspx* page markup for the **Budget** or **Start Date** columns, default currency and date formatting has been applied to them by the `DynamicField` controls, using the metadata that you supplied in the *Department.cs* file.

Adding Insert and Delete Functionality

Open *SchoolRepository.cs*, add the following code in order to create an `Insert` method and a `Delete` method. The code also includes a method named `GenerateDepartmentID` that calculates the next available record key value for use by the `Insert` method. This is required because the database is not configured to calculate this automatically for the `Department` table.

```
publicvoidInsertDepartment(Department department)
{
try
{
                department.DepartmentID=GenerateDepartmentID();
                context.Departments.AddObject(department);
                context.SaveChanges();
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

publicvoidDeleteDepartment(Department department)
{
try
{
                context.Departments.Attach(department);
                context.Departments.DeleteObject(department);
                context.SaveChanges();
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

privateInt32GenerateDepartmentID()
{
Int32 maxDepartmentID =0;
var department =(from d inGetDepartments()
```

```
orderby d.DepartmentIDdescending
select d).FirstOrDefault();
if(department !=null)
{
                maxDepartmentID = department.DepartmentID+1;
}
return maxDepartmentID;
}
```

The Attach Method

The `DeleteDepartment` method calls the `Attach` method in order to re-establish the link that's maintained in the object context's object state manager between the entity in memory and the database row it represents. This must occur before the method calls the `SaveChanges` method.

The term *object context* refers to the Entity Framework class that derives from the `ObjectContext` class that you use to access your entity sets and entities. In the code for this project, the class is named `SchoolEntities`, and an instance of it is always named `context`. The object context's *object state manager* is a class that derives from the `ObjectStateManager` class. The object contact uses the object state manager to store entity objects and to keep track of whether each one is in sync with its corresponding table row or rows in the database.

When you read an entity, the object context stores it in the object state manager and keeps track of whether that representation of the object is in sync with the database. For example, if you change a property value, a flag is set to indicate that the property you changed is no longer in sync with the database. Then when you call the `SaveChanges` method, the object context knows what to do in the database because the object state manager knows exactly what's different between the current state of the entity and the state of the database.

However, this process typically does not work in a web application, because the object context instance that reads an entity, along with everything in its object state manager, is disposed after a page is rendered. The object context instance that must apply changes is a new one that's instantiated for postback processing. In the case of the `DeleteDepartment` method, the `ObjectDataSource` control re-creates the original version of the entity for you from values in view state, but this re-created `Department` entity does not exist in the object state manager. If you called the `DeleteObject` method on this re-created entity, the call would fail because the object context does not know whether the entity is in sync with the database. However, calling the `Attach` method re-establishes the same tracking between the re-created entity and the values in the database that was originally done automatically when the entity was read in an earlier instance of the object context.

There are times when you don't want the object context to track entities in the object state manager, and you can set flags to prevent it from doing that. Examples of this are shown in later tutorials in this series.

The SaveChanges Method

This simple repository class illustrates basic principles of how to perform CRUD operations. In this example, the `SaveChanges` method is called immediately after each update. In a production application you might want to call the `SaveChanges` method from a separate method to give you more control over when the database is updated. (At the end of the next tutorial you will find a link to a white paper that discusses the unit of work pattern which is one approach to coordinating related updates.) Notice also that in the example, the `DeleteDepartment` method does not include code for handling concurrency conflicts; code to do that will be added in a later tutorial in this series.

Retrieving Instructor Names to Select When Inserting

Users must be able to select an administrator from a list of instructors in a drop-down list when creating new departments. Therefore, add the following code to *SchoolRepository.cs* to create a method to retrieve the list of instructors using the view that you created earlier:

```
publicIEnumerable<InstructorName>GetInstructorNames()
{
return context.InstructorNames.OrderBy("it.FullName").ToList();
}
```

Creating a Page for Inserting Departments

Create a *DepartmentsAdd.aspx* page that uses the *Site.Master* page, and add the following markup in the `Content` control named `Content2`:

```
<h2>Departments</h2>
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.DAL.SchoolRepository"DataObjectTypeName="ContosoUniversit
y.DAL.Department"
InsertMethod="InsertDepartment">
</asp:ObjectDataSource>
<asp:DetailsViewID="DepartmentsDetailsView"runat="server"
DataSourceID="DepartmentsObjectDataSource"AutoGenerateRows="False"
DefaultMode="Insert"OnItemInserting="DepartmentsDetailsView_ItemInserting">
```

```
<Fields>
<asp:DynamicFieldDataField="Name"HeaderText="Name"/>
<asp:DynamicFieldDataField="Budget"HeaderText="Budget"/>
<asp:DynamicFieldDataField="StartDate"HeaderText="Start Date"/>
<asp:TemplateFieldHeaderText="Administrator">
<InsertItemTemplate>
<asp:ObjectDataSourceID="InstructorsObjectDataSource"runat="server"
TypeName="ContosoUniversity.DAL.SchoolRepository"
DataObjectTypeName="ContosoUniversity.DAL.InstructorName"
SelectMethod="GetInstructorNames">
</asp:ObjectDataSource>
<asp:DropDownListID="InstructorsDropDownList"runat="server"
DataSourceID="InstructorsObjectDataSource"
DataTextField="FullName"DataValueField="PersonID"OnInit="DepartmentsDropDownList_Init
">
</asp:DropDownList>
</InsertItemTemplate>
</asp:TemplateField>
<asp:CommandFieldShowInsertButton="True"/>
</Fields>
</asp:DetailsView>
<asp:ValidationSummaryID="DepartmentsValidationSummary"runat="server"
ShowSummary="true"DisplayMode="BulletList"/>
```

This markup creates two `ObjectDataSource` controls, one for inserting new `Department` entities and one for retrieving instructor names for the `DropDownList` control that's used for selecting department administrators. The markup creates a `DetailsView` control for entering new departments, and it specifies a handler for the control's `ItemInserting` event so that you can set the `Administrator` foreign key value. At the end is a `ValidationSummary` control to display error messages.

Open *DepartmentsAdd.aspx.cs* and add the following `using` statement:

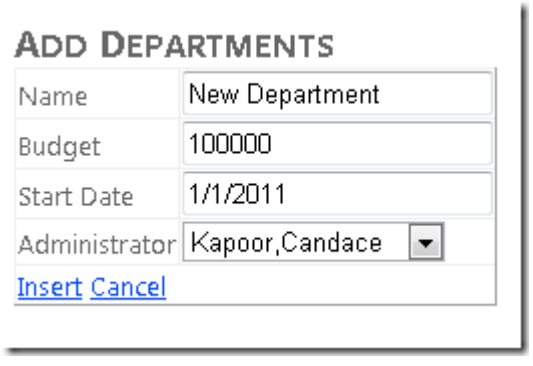
```
usingContosoUniversity.DAL;
```

Add the following class variable and methods:

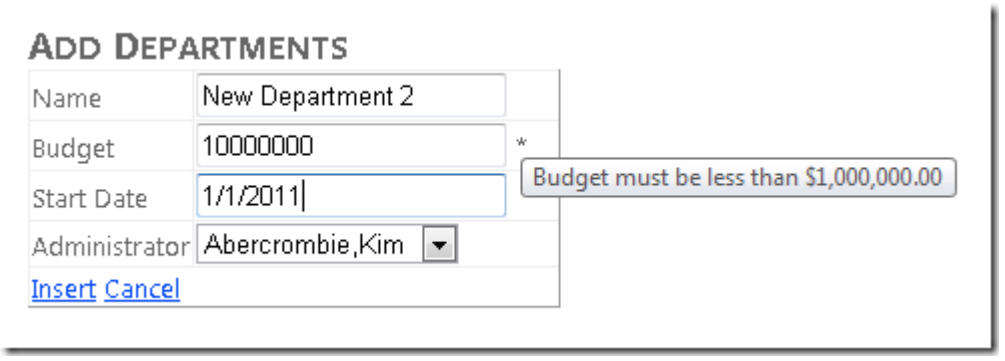
```
privateDropDownList administratorsDropDownList;

protectedvoidPage_Init(object sender,EventArgs e)
{
DepartmentsDetailsView.EnableDynamicData(typeof(Department));
}

protectedvoidDepartmentsDropDownList_Init(object sender,EventArgs e)
{
            administratorsDropDownList = sender asDropDownList;
}

protectedvoidDepartmentsDetailsView_ItemInserting(object
sender,DetailsViewInsertEventArgs e)
{
            e.Values["Administrator"]= administratorsDropDownList.SelectedValue;
}
```

The `Page_Init` method enables Dynamic Data functionality. The handler for the `DropDownList` control's `Init` event saves a reference to the control, and the handler for the `DetailsView` control's `Inserting` event uses that reference to get the `PersonID` value of the selected instructor and update the `Administrator` foreign key property of the `Department` entity.

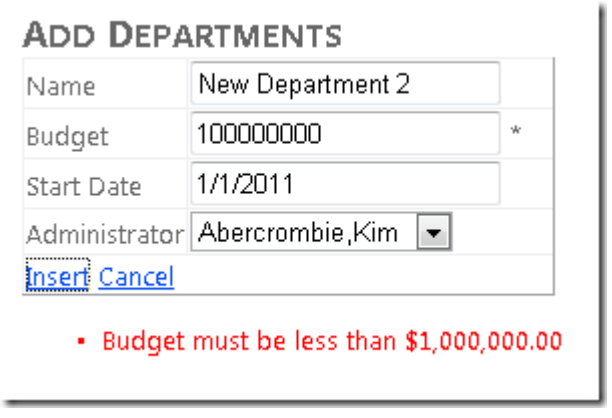
Run the page, add information for a new department, and then click the **Insert** link.



Enter values for another new department. Enter a number greater than 1,000,000.00 in the **Budget** field and tab to the next field. An asterisk appears in the field, and if you hold the mouse pointer over it, you can see the error message that you entered in the metadata for that field.



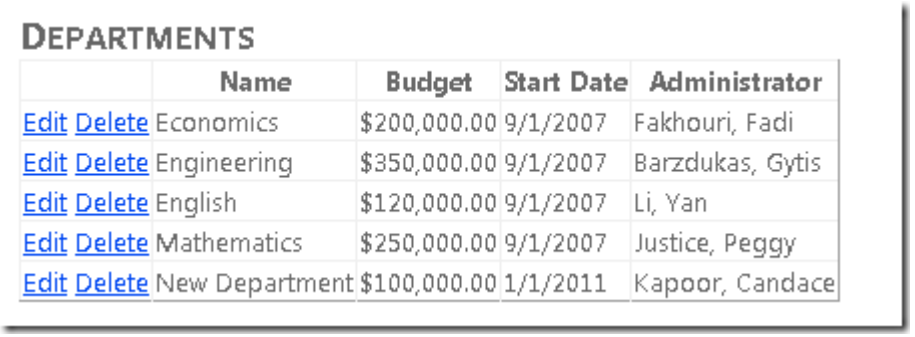
Click **Insert**, and you see the error message displayed by the `ValidationSummary` control at the bottom of the page.



Next, close the browser and open the *Departments.aspx* page. Add delete capability to the *Departments.aspx* page by adding a `DeleteMethod` attribute to the `ObjectDataSource` control, and a `DataKeyNames` attribute to the `GridView` control. The opening tags for these controls will now resemble the following example:

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.DAL.SchoolRepository"
DataObjectTypeName="ContosoUniversity.DAL.Department"
SelectMethod="GetDepartments"
DeleteMethod="DeleteDepartment">

<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource"DataKeyNames="DepartmentID">
```

Run the page.



Delete the department you added when you ran the *DepartmentsAdd.aspx* page.

Adding Update Functionality

Open *SchoolRepository.cs* and add the following `Update` method:

```
publicvoidUpdateDepartment(Department department,Department origDepartment)
{
try
{
                context.Departments.Attach(origDepartment);
                context.ApplyCurrentValues("Departments", department);
                context.SaveChanges();
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}
```

When you click **Update** in the *Departments.aspx* page, the `ObjectDataSource` control creates two `Department` entities to pass to the `UpdateDepartment` method. One contains the original values that have been stored in view state, and the other contains the new values that were entered in the `GridView` control. The code in the `UpdateDepartment` method passes the `Department` entity that has the original values to the `Attach` method in order to establish the tracking between the entity and what's in the database. Then the code passes the `Department` entity that has the new values to the `ApplyCurrentValues` method. The object

context compares the old and new values. If a new value is different from an old value, the object context changes the property value. The `SaveChanges` method then updates only the changed columns in the database. (However, if the update function for this entity were mapped to a stored procedure, the entire row would be updated regardless of which columns were changed.)

Open the *Departments.aspx* file and add the following attributes to the `DepartmentsObjectDataSource` control:

- `UpdateMethod="UpdateDepartment"`
- `ConflictDetection="CompareAllValues"`
  This causes old values to be stored in view state so that they can be compared with the new values in the `Update` method.
- `OldValuesParameterFormatString="orig{0}"`
  This informs the control that the name of the original values parameter is `origDepartment`.

The markup for the opening tag of the `ObjectDataSource` control now resembles the following example:

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.DAL.SchoolRepository"
DataObjectTypeName="ContosoUniversity.DAL.Department"
SelectMethod="GetDepartments"DeleteMethod="DeleteDepartment"
UpdateMethod="UpdateDepartment"
ConflictDetection="CompareAllValues"
OldValuesParameterFormatString="orig{0}">
```

Add an `OnRowUpdating="DepartmentsGridView_RowUpdating"` attribute to the `GridView` control. You will use this to set the `Administrator` property value based on the row the user selects in a drop-down list. The `GridView` opening tag now resembles the following example:

```
<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource"DataKeyNames="DepartmentID"
OnRowUpdating="DepartmentsGridView_RowUpdating">
```

Add an `EditItemTemplate` control for the `Administrator` column to the `GridView` control, immediately after the `ItemTemplate` control for that column:

```
<EditItemTemplate>
<asp:ObjectDataSourceID="InstructorsObjectDataSource"runat="server"DataObjectTypeName
="ContosoUniversity.DAL.InstructorName"
SelectMethod="GetInstructorNames"TypeName="ContosoUniversity.DAL.SchoolRepository">
</asp:ObjectDataSource>
<asp:DropDownList ID="InstructorsDropDownList" runat="server"
DataSourceID="InstructorsObjectDataSource"
                        SelectedValue='<%# Eval("Administrator")  %>'
                        DataTextField="FullName" DataValueField="PersonID"
OnInit="DepartmentsDropDownList_Init" >
</asp:DropDownList>
</EditItemTemplate>
```

This `EditItemTemplate` control is similar to the `InsertItemTemplate` control in the *DepartmentsAdd.aspx* page. The difference is that the initial value of the control is set using the `SelectedValue` attribute.

Before the `GridView` control, add a `ValidationSummary` control as you did in the *DepartmentsAdd.aspx* page.

```
<asp:ValidationSummaryID="DepartmentsValidationSummary"runat="server"
ShowSummary="true"DisplayMode="BulletList"/>
```

Open *Departments.aspx.cs* and immediately after the partial-class declaration, add the following code to create a private field to reference the `DropDownList` control:

```
privateDropDownList administratorsDropDownList;
```

Then add handlers for the `DropDownList` control's `Init` event and the `GridView` control's `RowUpdating` event:

```
protectedvoidDepartmentsDropDownList_Init(object sender,EventArgs e)
{
            administratorsDropDownList = sender asDropDownList;
}

protectedvoidDepartmentsGridView_RowUpdating(object sender,GridViewUpdateEventArgs e)
```

```
{
            e.NewValues["Administrator"]= administratorsDropDownList.SelectedValue;
}
```

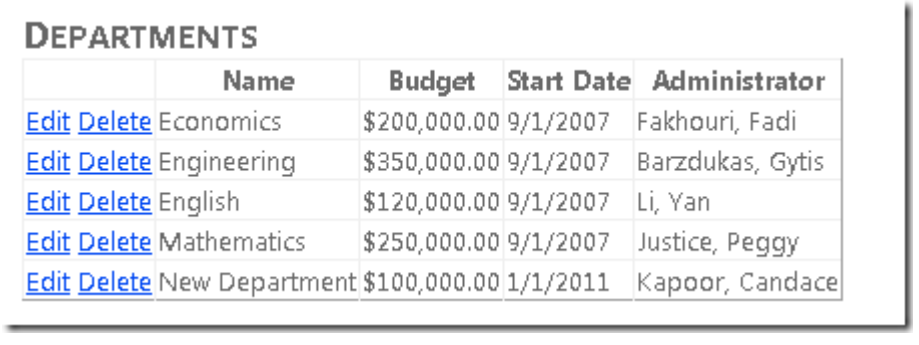
The handler for the `Init` event saves a reference to the `DropDownList` control in the class field. The handler for the `RowUpdating` event uses the reference to get the value the user entered and apply it to the `Administrator` property of the `Department` entity.

Use the *DepartmentsAdd.aspx* page to add a new department, then run the *Departments.aspx* page and click **Edit** on the row that you added.

Note You will not be able to edit rows that you did not add (that is, that were already in the database), because of invalid data in the database; the administrators for the rows that were created with the database are students. If you try to edit one of them, you will get an error page that reports an error like `'InstructorsDropDownList' has a SelectedValue which is invalid because it does not exist in the list of items.`

If you enter an invalid **Budget** amount and then click **Update**, you see the same asterisk and error message that you saw in the *Departments.aspx* page.

Change a field value or select a different administrator and click **Update**. The change is displayed.



This completes the introduction to using the `ObjectDataSource` control for basic CRUD (create, read, update, delete) operations with the Entity Framework. You've built a simple n-tier application, but the business-logic layer is still tightly coupled to the data-access layer, which complicates automated unit testing. In the following tutorial you'll see how to implement the repository pattern to facilitate unit testing.

Adding a Business Logic Layer and Unit Tests

In the previous tutorial you created an n-tier web application using the Entity Framework and the `ObjectDataSource` control. This tutorial shows how to add business logic while keeping the business-logic layer (BLL) and the data-access layer (DAL) separate, and it shows how to create automated unit tests for the BLL.

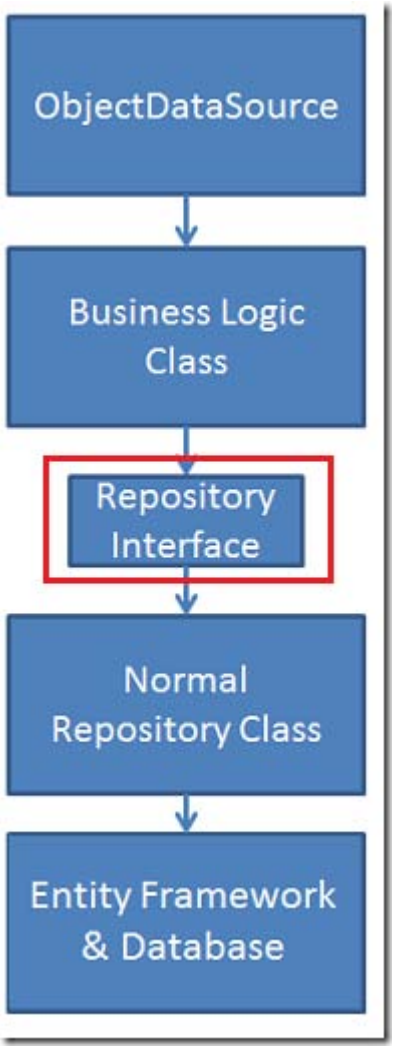
In this tutorial you'll complete the following tasks:

- Create a repository interface that declares the data-access methods you need.
- Implement the repository interface in the repository class.
- Create a business-logic class that calls the repository class to perform data-access functions.
- Connect the `ObjectDataSource` control to the business-logic class instead of to the repository class.
- Create a unit-test project and a repository class that uses in-memory collections for its data store.
- Create a unit test for business logic that you want to add to the business-logic class, then run the test and see it fail.
- Implement the business logic in the business-logic class, then re-run the unit test and see it pass.

You'll work with the *Departments.aspx* and *DepartmentsAdd.aspx* pages that you created in the previous tutorial.

Creating a Repository Interface

You'll begin by creating the repository interface.



In the *DAL* folder, create a new class file, name it *ISchoolRepository.cs*, and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;

namespaceContosoUniversity.DAL
{
publicinterfaceISchoolRepository:IDisposable
{
IEnumerable<Department>GetDepartments();
voidInsertDepartment(Department department);
voidDeleteDepartment(Department department);
voidUpdateDepartment(Department department,Department origDepartment);
```

```
IEnumerable<InstructorName>GetInstructorNames();
}
}
```

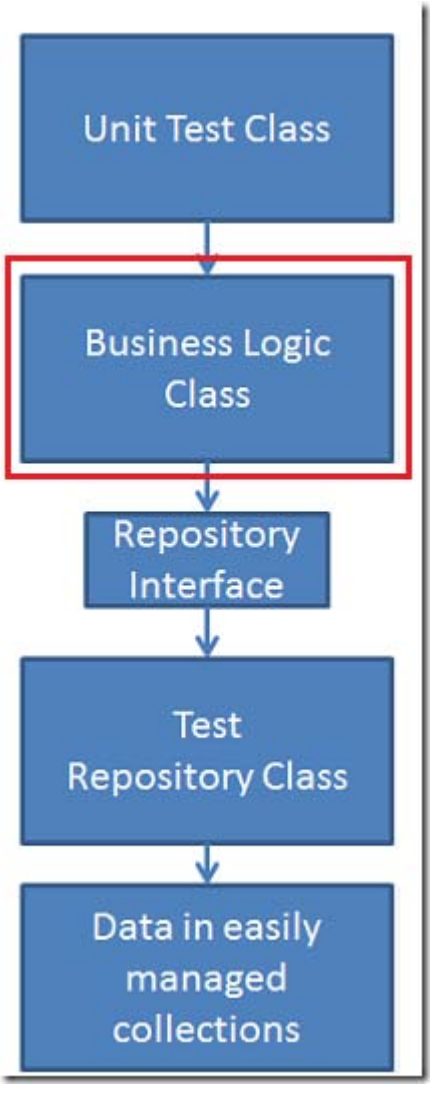
The interface defines one method for each of the CRUD (create, read, update, delete) methods that you created in the repository class.

In the `SchoolRepository` class in *SchoolRepository.cs*, indicate that this class implements the `ISchoolRepository` interface:

```
publicclassSchoolRepository:IDisposable,ISchoolRepository
```

Creating a Business-Logic Class

Next, you'll create the business-logic class. You do this so that you can add business logic that will be executed by the `ObjectDataSource` control, although you will not do that yet. For now, the new business-logic class will only perform the same CRUD operations that the repository does.



Create a new folder and name it *BLL*. (In a real-world application, the business-logic layer would typically be implemented as a class library — a separate project — but to keep this tutorial simple, BLL classes will be kept in a project folder.)

In the *BLL* folder, create a new class file, name it *SchoolBL.cs*, and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Web;
usingContosoUniversity.DAL;
```

```
namespaceContosoUniversity.BLL
{
publicclassSchoolBL:IDisposable
{
privateISchoolRepository schoolRepository;

publicSchoolBL()
{
this.schoolRepository =newSchoolRepository();
}

publicSchoolBL(ISchoolRepository schoolRepository)
{
this.schoolRepository = schoolRepository;
}

publicIEnumerable<Department>GetDepartments()
{
return schoolRepository.GetDepartments();
}

publicvoidInsertDepartment(Department department)
{
try
{
                schoolRepository.InsertDepartment(department);
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

publicvoidDeleteDepartment(Department department)
{
```

```
try
{
                schoolRepository.DeleteDepartment(department);
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

publicvoidUpdateDepartment(Department department,Department origDepartment)
{
try
{
                schoolRepository.UpdateDepartment(department, origDepartment);
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,

//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}

}

publicIEnumerable<InstructorName>GetInstructorNames()
{
return schoolRepository.GetInstructorNames();
}

privatebool disposedValue =false;

protectedvirtualvoidDispose(bool disposing)
```

```
{
if(!this.disposedValue)
{
if(disposing)
{
                    schoolRepository.Dispose();
}
}
this.disposedValue =true;
}

publicvoidDispose()
{
Dispose(true);
            GC.SuppressFinalize(this);
}

}
}
```

This code creates the same CRUD methods you saw earlier in the repository class, but instead of accessing the Entity Framework methods directly, it calls the repository class methods.

The class variable that holds a reference to the repository class is defined as an interface type, and the code that instantiates the repository class is contained in two constructors. The parameterless constructor will be used by the `ObjectDataSource` control. It creates an instance of the `SchoolRepository` class that you created earlier. The other constructor allows whatever code that instantiates the business-logic class to pass in any object that implements the repository interface.

The CRUD methods that call the repository class and the two constructors make it possible to use the business-logic class with whatever back-end data store you choose. The business-logic class does not need to be aware of how the class that it's calling persists the data. (This is often called *persistence ignorance*.) This facilitates unit testing, because you can connect the business-logic class to a repository implementation that uses something as simple as in-memory `List` collections to store data.

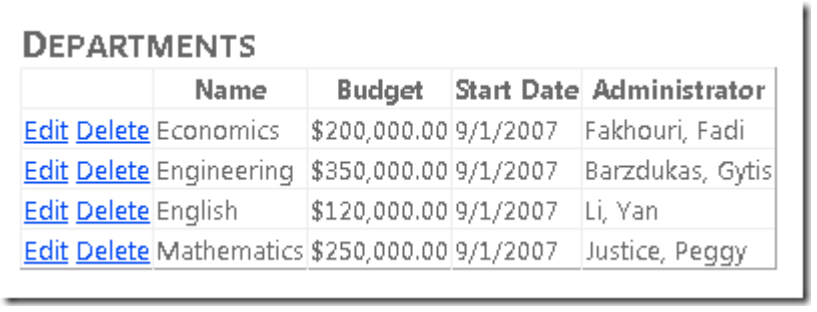
Note Technically, the entity objects are still not persistence-ignorant, because they're instantiated from classes that inherit from the Entity Framework's `EntityObject` class. For complete persistence ignorance, you can use *plain old CLR objects*, or *POCOs*, in place of objects that inherit from the `EntityObject` class. Using POCOs is

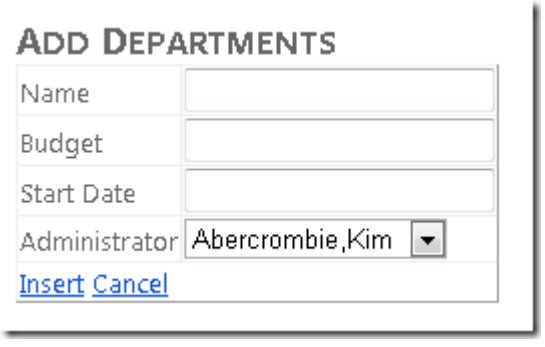
beyond the scope of this tutorial. For more information, see Testability and Entity Framework 4.0 on the MSDN website.)

Now you can connect the `ObjectDataSource` controls to the business-logic class instead of to the repository and verify that everything works as it did before.

In *Departments.aspx* and *DepartmentsAdd.aspx*, change each occurrence of `TypeName="ContosoUniversity.DAL.SchoolRepository"` to `TypeName="ContosoUniversity.BLL.SchoolBL"`. (There are four instances in all.)

Run the *Departments.aspx* and *DepartmentsAdd.aspx* pages to verify that they still work as they did before.



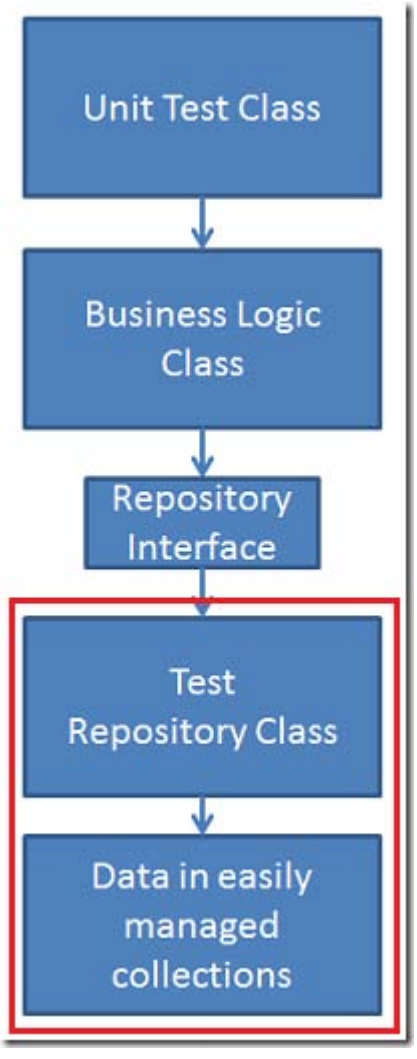


Creating a Unit-Test Project and Repository Implementation

Add a new project to the solution using the **Test Project** template, and name it `ContosoUniversity.Tests`.

In the test project, add a reference to `System.Data.Entity` and add a project reference to the `ContosoUniversity` project.

You can now create the repository class that you'll use with unit tests. The data store for this repository will be within the class.



In the test project, create a new class file, name it *MockSchoolRepository.cs*, and replace the existing code with the following code:

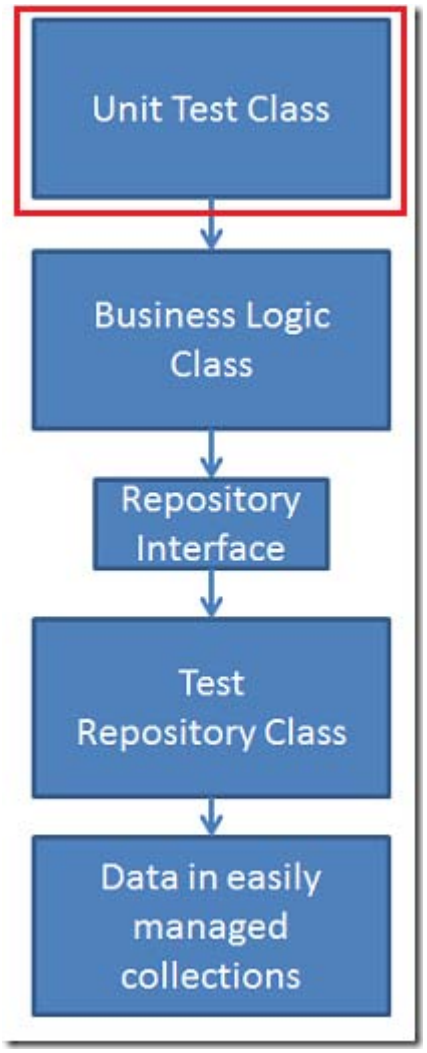
```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Text;
usingContosoUniversity.DAL;
usingContosoUniversity.BLL;

namespaceContosoUniversity.Tests
{
classMockSchoolRepository:ISchoolRepository,IDisposable
```

```
{
List<Department> departments =newList<Department>();
List<InstructorName> instructors =newList<InstructorName>();

publicIEnumerable<Department>GetDepartments()
{
return departments;
}

publicvoidInsertDepartment(Department department)
{
            departments.Add(department);
}

publicvoidDeleteDepartment(Department department)
{
            departments.Remove(department);
}

publicvoidUpdateDepartment(Department department,Department origDepartment)
{
            departments.Remove(origDepartment);
            departments.Add(department);
}

publicIEnumerable<InstructorName>GetInstructorNames()
{
return instructors;
}

publicvoidDispose()
{

}
}
}
```

This repository class has the same CRUD methods as the one that accesses the Entity Framework directly, but they work with `List` collections in memory instead of with a database. This makes it easier for a test class to set up and validate unit tests for the business-logic class.

Creating Unit Tests

The **Test** project template created a stub unit test class for you, and your next task is to modify this class by adding unit test methods to it for business logic that you want to add to the business-logic class.



At Contoso University, any individual instructor can only be the administrator of a single department, and you need to add business logic to enforce this rule. You will start by adding tests and running the tests to see them fail. You'll then add the code and rerun the tests, expecting to see them pass.

Open the *UnitTest1.cs* file and add `using` statements for the business logic and data-access layers that you created in the ContosoUniversity project:

```
usingContosoUniversity.BLL;
usingContosoUniversity.DAL;
```

Replace the `TestMethod1` method with the following methods:

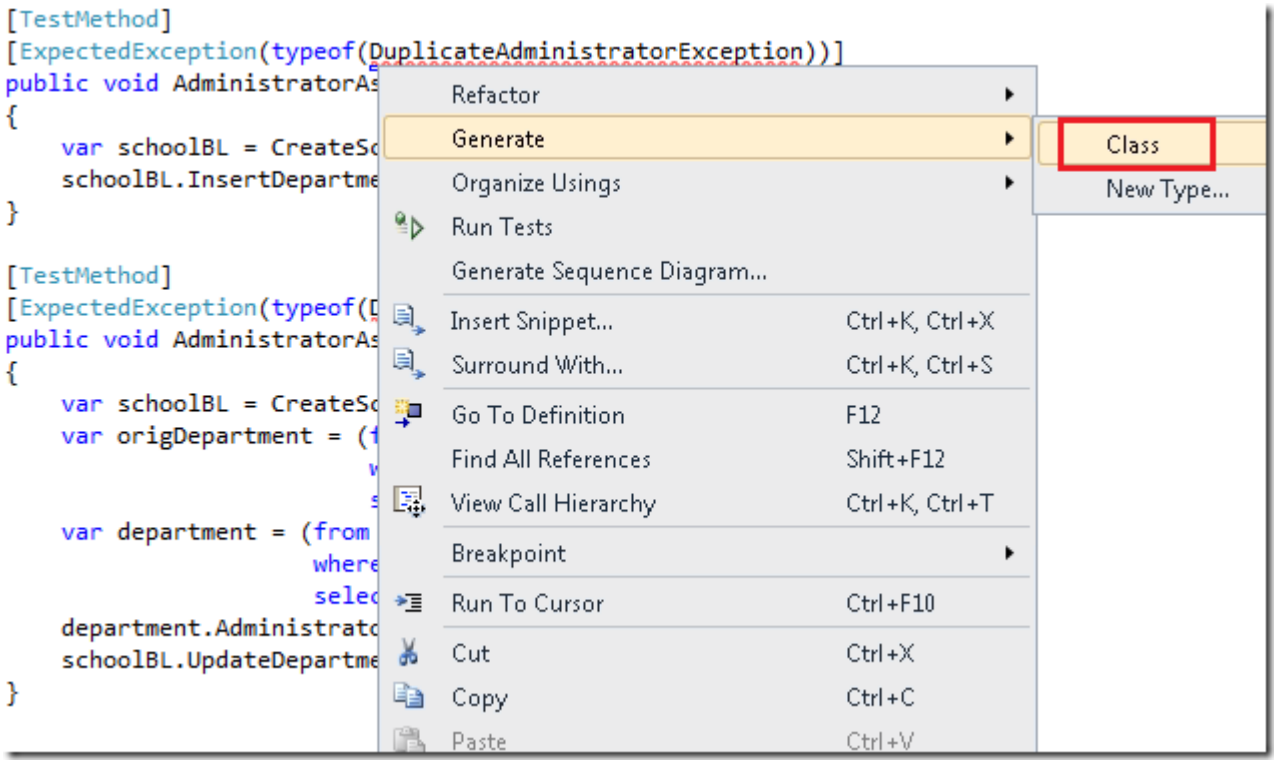
```
privateSchoolBLCreateSchoolBL()
{
var schoolRepository =newMockSchoolRepository();
var schoolBL =newSchoolBL(schoolRepository);
            schoolBL.InsertDepartment(newDepartment(){Name="First
Department",DepartmentID=0,Administrator=1,Person=newInstructor(){FirstMidName="Admin
",LastName="One"}});
            schoolBL.InsertDepartment(newDepartment(){Name="Second
Department",DepartmentID=1,Administrator=2,Person=newInstructor(){FirstMidName="Admin
",LastName="Two"}});
            schoolBL.InsertDepartment(newDepartment(){Name="Third
Department",DepartmentID=2,Administrator=3,Person=newInstructor(){FirstMidName="Admin
",LastName="Three"}});
return schoolBL;
}

[TestMethod]
[ExpectedException(typeof(DuplicateAdministratorException))]
publicvoidAdministratorAssignmentRestrictionOnInsert()
{
var schoolBL =CreateSchoolBL();
            schoolBL.InsertDepartment(newDepartment(){Name="Fourth
Department",DepartmentID=3,Administrator=2,Person=newInstructor(){FirstMidName="Admin
",LastName="Two"}});
}

[TestMethod]
[ExpectedException(typeof(DuplicateAdministratorException))]
publicvoidAdministratorAssignmentRestrictionOnUpdate()
```

```
{
var schoolBL =CreateSchoolBL();
var origDepartment =(from d in schoolBL.GetDepartments()
where d.Name=="Second Department"
select d).First();
var department =(from d in schoolBL.GetDepartments()
where d.Name=="Second Department"
select d).First();
            department.Administrator=1;
            schoolBL.UpdateDepartment(department, origDepartment);
}
```

The `CreateSchoolBL` method creates an instance of the repository class that you created for the unit test project, which it then passes to a new instance of the business-logic class. The method then uses the business-logic class to insert three departments that you can use in test methods.

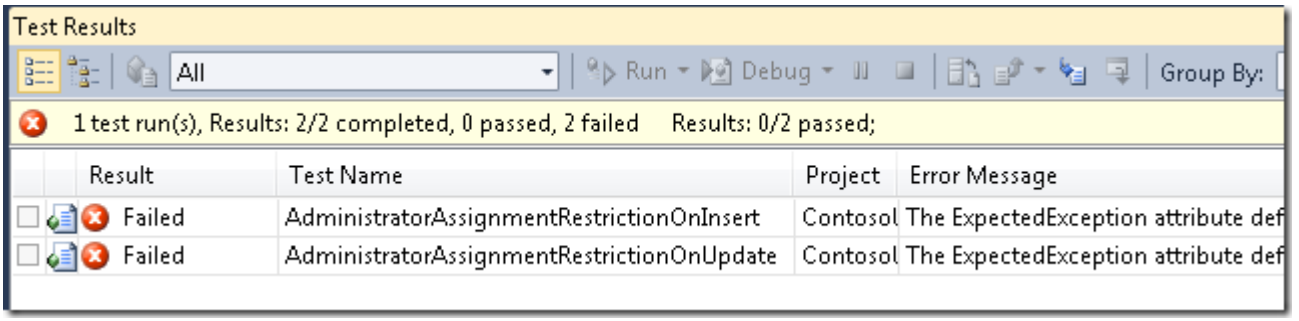
The test methods verify that the business-logic class throws an exception if someone tries to insert a new department with the same administrator as an existing department, or if someone tries to update a department's administrator by setting it to the ID of a person who is already the administrator of another department.

You haven't created the exception class yet, so this code will not compile. To get it to compile, right-click `DuplicateAdministratorException` and select **Generate**, and then **Class**.



This creates a class in the test project which you can delete after you've created the exception class in the main project. and implemented the business logic.

Run the test project. As expected, the tests fail.



Adding Business Logic to Make a Test Pass

Next, you'll implement the business logic that makes it impossible to set as the administrator of a department someone who is already administrator of another department. You'll throw an exception from the business-logic layer, and then catch it in the presentation layer if a user edits a department and clicks **Update** after selecting someone who is already an administrator. (You could also remove instructors from the drop-down list who are already administrators before you render the page, but the purpose here is to work with the business-logic layer.)

Start by creating the exception class that you'll throw when a user tries to make an instructor the administrator of more than one department. In the main project, create a new class file in the *BLL* folder, name it *DuplicateAdministratorException.cs*, and replace the existing code with the following code:

```
usingSystem;

namespaceContosoUniversity.BLL
{
publicclassDuplicateAdministratorException:Exception
{
publicDuplicateAdministratorException(string message)
:base(message)
{
}
}
}
```

Now delete the temporary *DuplicateAdministratorException.cs* file that you created in the test project earlier in order to be able to compile.

In the main project, open the *SchoolBL.cs* file and add the following method that contains the validation logic. (The code refers to a method that you'll create later.)

```
privatevoidValidateOneAdministratorAssignmentPerInstructor(Department department)
{
if(department.Administrator!=null)
{
var duplicateDepartment =
schoolRepository.GetDepartmentsByAdministrator(department.Administrator.GetValueOrDef
ault()).FirstOrDefault();
if(duplicateDepartment !=null&& duplicateDepartment.DepartmentID!=
department.DepartmentID)
{
thrownewDuplicateAdministratorException(String.Format(
"Instructor {0} {1} is already administrator of the {2} department.",
                        duplicateDepartment.Person.FirstMidName,
                        duplicateDepartment.Person.LastName,
```

```
                        duplicateDepartment.Name));
}
}
}
```

You'll call this method when you're inserting or updating `Department` entities in order to check whether another department already has the same administrator.

The code calls a method to search the database for a `Department` entity that has the same `Administrator` property value as the entity being inserted or updated. If one is found, the code throws an exception. No validation check is required if the entity being inserted or updated has no `Administrator` value, and no exception is thrown if the method is called during an update and the `Department` entity found matches the `Department` entity being updated.

Call the new method from the `Insert` and `Update` methods:

```
publicvoidInsertDepartment(Department department)
{
ValidateOneAdministratorAssignmentPerInstructor(department);
try
...

publicvoidUpdateDepartment(Department department,Department origDepartment)
{
ValidateOneAdministratorAssignmentPerInstructor(department);
try
...
```

In *ISchoolRepository.cs*, add the following declaration for the new data-access method:

```
IEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator);
```

In *SchoolRepository.cs*, add the following `using` statement:

```
usingSystem.Data.Objects;
```

In *SchoolRepository.cs*, add the following new data-access method:

```
publicIEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator)
{
returnnewObjectQuery<Department>("SELECT VALUE d FROM Departments as d",
context,MergeOption.NoTracking).Include("Person").Where(d => d.Administrator==
administrator).ToList();
}
```

This code retrieves `Department` entities that have a specified administrator. Only one department should be found (if any). However, because no constraint is built into the database, the return type is a collection in case multiple departments are found.

By default, when the object context retrieves entities from the database, it keeps track of them in its object state manager. The `MergeOption.NoTracking` parameter specifies that this tracking will not be done for this query. This is necessary because the query might return the exact entity that you're trying to update, and then you would not be able to attach that entity. For example, if you edit the History department in the *Departments.aspx* page and leave the administrator unchanged, this query will return the History department. If `NoTracking` is not set, the object context would already have the History department entity in its object state manager. Then when you attach the History department entity that's re-created from view state, the object context would throw an exception that says `"An object with the same key already exists in the ObjectStateManager. The ObjectStateManager cannot track multiple objects with the same key"`.

(As an alternative to specifying `MergeOption.NoTracking`, you could create a new object context just for this query. Because the new object context would have its own object state manager, there would be no conflict when you call the `Attach` method. The new object context would share metadata and database connection with the original object context, so the performance penalty of this alternate approach would be minimal. The approach shown here, however, introduces the `NoTracking` option, which you'll find useful in other contexts. The `NoTracking` option is discussed further in a later tutorial in this series.)

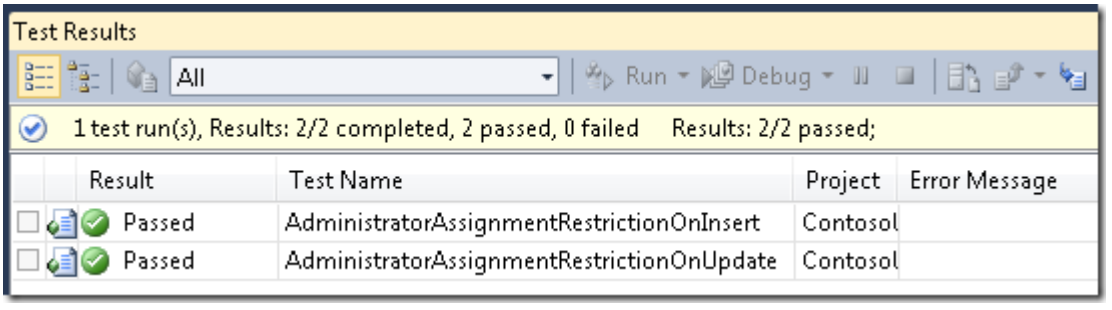
In the test project, add the new data-access method to *MockSchoolRepository.cs*:

```
publicIEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator)
{
return(from d in departments
where d.Administrator== administrator
```

```
select d);
}
```

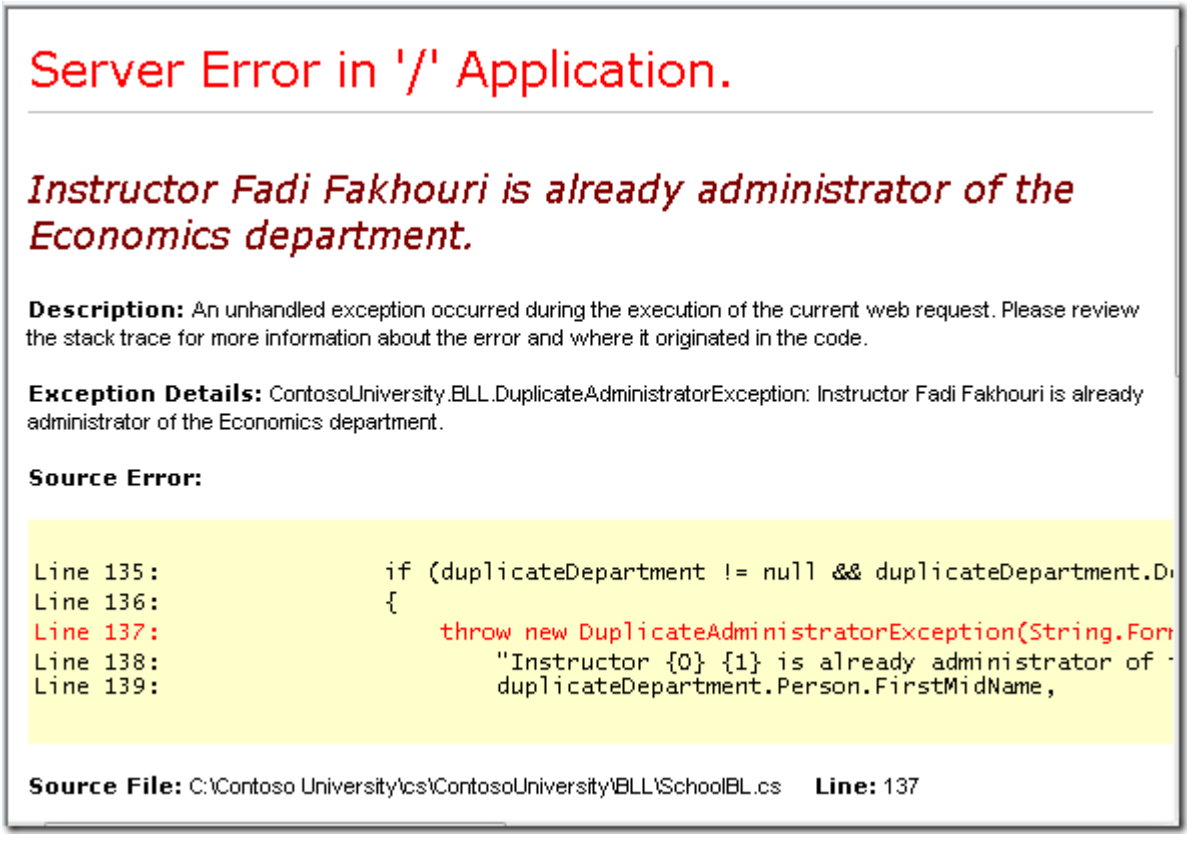
This code uses LINQ to perform the same data selection that the `ContosoUniversity` project repository uses LINQ to Entities for.

Run the test project again. This time the tests pass.



Handling ObjectDataSource Exceptions

In the `ContosoUniversity` project, run the *Departments.aspx* page and try to change the administrator for a department to someone who is already administrator for another department. (Remember that you can only edit departments that you added during this tutorial, because the database comes preloaded with invalid data.) You get the following server error page:



You don't want users to see this kind of error page, so you need to add error-handling code. Open *Departments.aspx* and specify a handler for the `OnUpdated` event of the `DepartmentsObjectDataSource`. The `ObjectDataSource` opening tag now resembles the following example.

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"
DataObjectTypeName="ContosoUniversity.DAL.Department"
SelectMethod="GetDepartments"
DeleteMethod="DeleteDepartment"
UpdateMethod="UpdateDepartment"
ConflictDetection="CompareAllValues"
OldValuesParameterFormatString="orig{0}"
OnUpdated="DepartmentsObjectDataSource_Updated">
```

In *Departments.aspx.cs*, add the following `using` statement:

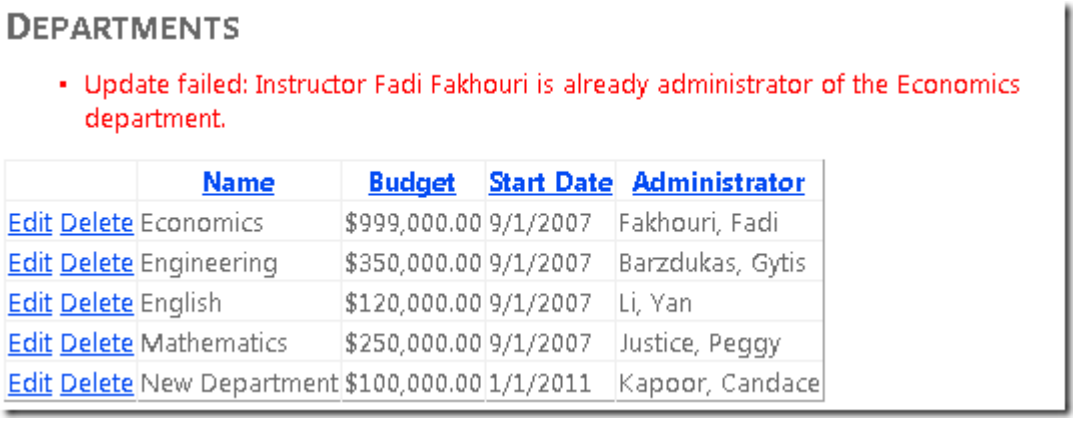
```
usingContosoUniversity.BLL;
```

Add the following handler for the `Updated` event:

```
protectedvoidDepartmentsObjectDataSource_Updated(object
sender,ObjectDataSourceStatusEventArgs e)
{
if(e.Exception!=null)
{
if(e.Exception.InnerExceptionisDuplicateAdministratorException)
{
var duplicateAdministratorValidator =newCustomValidator();
                    duplicateAdministratorValidator.IsValid=false;
                    duplicateAdministratorValidator.ErrorMessage="Update failed: "+
e.Exception.InnerException.Message;
Page.Validators.Add(duplicateAdministratorValidator);
                    e.ExceptionHandled=true;
}
}
}
```

If the `ObjectDataSource` control catches an exception when it tries to perform the update, it passes the exception in the event argument (`e`) to this handler. The code in the handler checks to see if the exception is the duplicate administrator exception. If it is, the code creates a validator control that contains an error message for the `ValidationSummary` control to display.

Run the page and attempt to make someone the administrator of two departments again. This time the `ValidationSummary` control displays an error message.



Make similar changes to the *DepartmentsAdd.aspx* page. In *DepartmentsAdd.aspx*, specify a handler for the `OnInserted` event of the `DepartmentsObjectDataSource`. The resulting markup will resemble the following example.

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"DataObjectTypeName="ContosoUniversity.DAL.De
partment"
InsertMethod="InsertDepartment"
OnInserted="DepartmentsObjectDataSource_Inserted">
```

In *DepartmentsAdd.aspx.cs*, add the same `using` statement:

```
usingContosoUniversity.BLL;
```

Add the following event handler:

```
protectedvoidDepartmentsObjectDataSource_Inserted(object
sender,ObjectDataSourceStatusEventArgs e)
{
if(e.Exception!=null)
{
if(e.Exception.InnerExceptionisDuplicateAdministratorException)
{
var duplicateAdministratorValidator =newCustomValidator();
                    duplicateAdministratorValidator.IsValid=false;
                    duplicateAdministratorValidator.ErrorMessage="Insert failed: "+
e.Exception.InnerException.Message;
Page.Validators.Add(duplicateAdministratorValidator);
                    e.ExceptionHandled=true;
}
}
}
```

You can now test the *DepartmentsAdd.aspx.cs* page to verify that it also correctly handles attempts to make one person the administrator of more than one department.

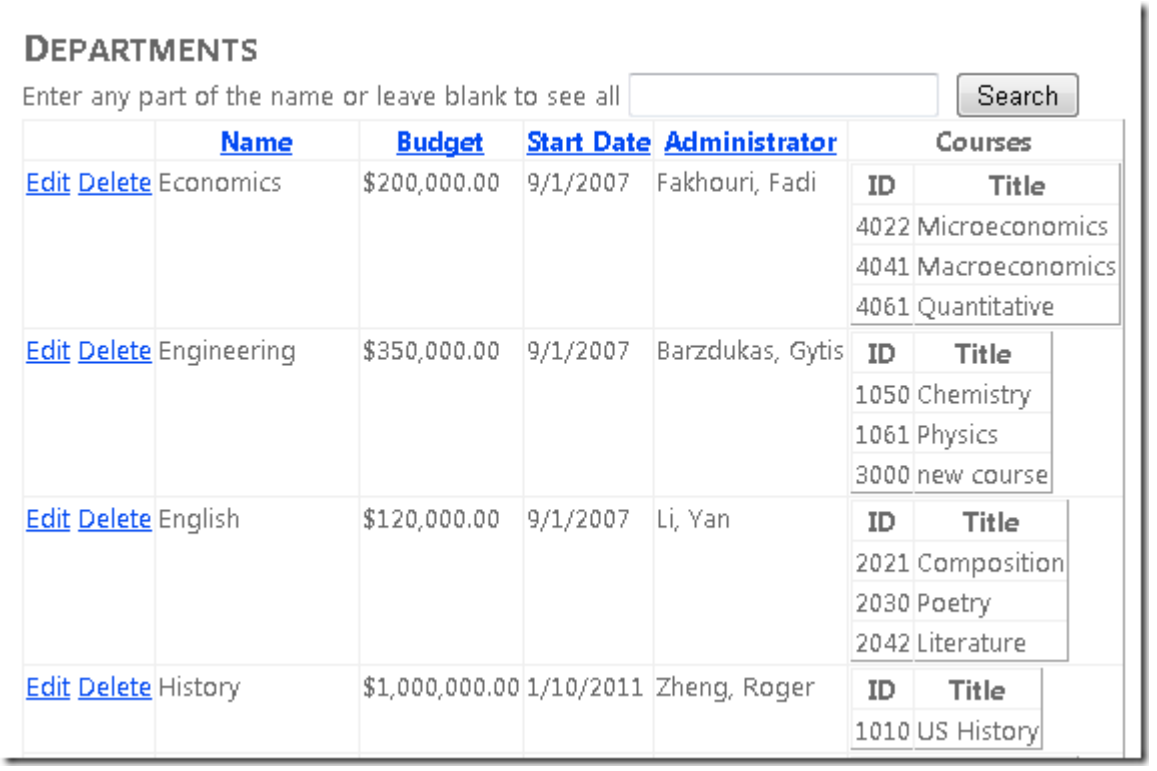
This completes the introduction to implementing the repository pattern for using the `ObjectDataSource` control with the Entity Framework. For more information about the repository pattern and testability, see the MSDN whitepaper Testability and Entity Framework 4.0.

In the following tutorial you'll see how to add sorting and filtering functionality to the application.

Sorting and Filtering

In the previous tutorial you implemented the repository pattern in an n-tier web application that uses the Entity Framework and the `ObjectDataSource` control. This tutorial shows how to do sorting and filtering and handle master-detail scenarios. You'll add the following enhancements to the *Departments.aspx* page:

- A text box to allow users to select departments by name.
- A list of courses for each department that's shown in the grid.
- The ability to sort by clicking column headings.



Adding the Ability to Sort GridView Columns

Open the *Departments.aspx* page and add a `SortParameterName="sortExpression"` attribute to the `ObjectDataSource` control named `DepartmentsObjectDataSource`. (Later you'll create a `GetDepartments` method that takes a parameter named `sortExpression`.) The markup for the opening tag of the control now resembles the following example.

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"DataObjectTypeName="ContosoUniversity.DAL.De
partment"
SelectMethod="GetDepartments"DeleteMethod="DeleteDepartment"UpdateMethod="UpdateDepar
tment"
ConflictDetection="CompareAllValues"OldValuesParameterFormatString="orig{0}"
OnUpdated="DepartmentsObjectDataSource_Updated"SortParameterName="sortExpression">
```

Add the `AllowSorting="true"` attribute to the opening tag of the `GridView` control. The markup for the opening tag of the control now resembles the following example.

```
<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource"DataKeyNames="DepartmentID"
OnRowUpdating="DepartmentsGridView_RowUpdating"
AllowSorting="true">
```

In *Departments.aspx.cs*, set the default sort order by calling the `GridView` control's `Sort` method from the `Page_Load` method:

```
protectedvoidPage_Load(object sender,EventArgs e)
{
if(!IsPostBack)
{
DepartmentsGridView.Sort("Name",SortDirection.Ascending);
}
}
```

You can add code that sorts or filters in either the business logic class or the repository class. If you do it in the business logic class, the sorting or filtering work will be done after the data is retrieved from the database, because the business logic class is working with an `IEnumerable` object returned by the repository. If you add sorting and filtering code in the repository class and you do it before a LINQ expression or object query has been converted to an `IEnumerable` object, your commands will be passed through to the database for processing, which is typically more efficient. In this tutorial you'll implement sorting and filtering in a way that causes the processing to be done by the database — that is, in the repository.

To add sorting capability, you must add a new method to the repository interface and repository classes as well as to the business logic class. In the *ISchoolRepository.cs* file, add a new `GetDepartments` method that takes a `sortExpression` parameter that will be used to sort the list of departments that's returned:

```
IEnumerable<Department>GetDepartments(string sortExpression);
```

The `sortExpression` parameter will specify the column to sort on and the sort direction.

Add code for the new method to the *SchoolRepository.cs* file:

```
publicIEnumerable<Department>GetDepartments(string sortExpression)
{
if(String.IsNullOrWhiteSpace(sortExpression))
{
                sortExpression ="Name";
}
return context.Departments.Include("Person").OrderBy("it."+ sortExpression).ToList();
}
```

Change the existing parameterless `GetDepartments` method to call the new method:

```
publicIEnumerable<Department>GetDepartments()
{
returnGetDepartments("");
}
```

In the test project, add the following new method to *MockSchoolRepository.cs*:

```
publicIEnumerable<Department>GetDepartments(string sortExpression)
{
return departments;
}
```

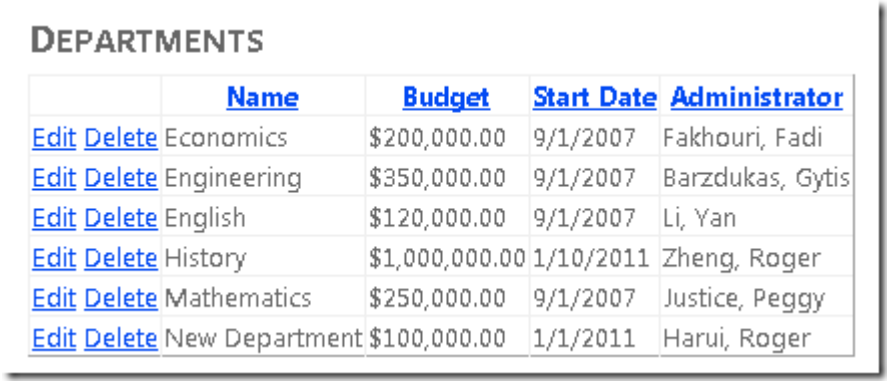
If you were going to create any unit tests that depended on this method returning a sorted list, you would need to sort the list before returning it. You won't be creating tests like that in this tutorial, so the method can just return the unsorted list of departments.

In the *SchoolBL.cs* file, add the following new method to the business logic class:

```
publicIEnumerable<Department>GetDepartments(string sortExpression)
{
return schoolRepository.GetDepartments(sortExpression);
}
```

This code passes the sort parameter to the repository method.

Run the *Departments.aspx* page.



DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	History	$1,000,000.00	1/10/2011	Zheng, Roger
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy
Edit Delete	New Department	$100,000.00	1/1/2011	Harui, Roger

You can now click any column heading to sort by that column. If the column is already sorted, clicking the heading reverses the sort direction.

Adding a Search Box

In this section you'll add a search text box, link it to the `ObjectDataSource` control using a control parameter, and add a method to the business logic class to support filtering.

Open the *Departments.aspx* page and add the following markup between the heading and the first `ObjectDataSource` control:

```
Enter any part of the name or leave the box blank to see all names:
<asp:TextBox ID="SearchTextBox" runat="server"AutoPostBack="true"></asp:TextBox>
<asp:Button ID="SearchButton" runat="server"Text="Search"/>
```

In the `ObjectDataSource` control named `DepartmentsObjectDataSource`, do the following:

- Add a `SelectParameters` element for a parameter named `nameSearchString` that gets the value entered in the `SearchTextBox` control.
- Change the `SelectMethod` attribute value to `GetDepartmentsByName`. (You'll create this method later.)

The markup for the `ObjectDataSource` control now resembles the following example:

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"TypeName="ContosoU
niversity.BLL.SchoolBL"
SelectMethod="GetDepartmentsByName"DeleteMethod="DeleteDepartment"UpdateMethod="Updat
eDepartment"
DataObjectTypeName="ContosoUniversity.DAL.Department"ConflictDetection="CompareAllVal
ues"
SortParameterName="sortExpression"OldValuesParameterFormatString="orig{0}"
OnUpdated="DepartmentsObjectDataSource_Updated">
<SelectParameters>
<asp:ControlParameterControlID="SearchTextBox"Name="nameSearchString"PropertyName="Te
xt"
Type="String"/>
</SelectParameters>
</asp:ObjectDataSource>
```

In *ISchoolRepository.cs*, add a `GetDepartmentsByName` method that takes both `sortExpression` and `nameSearchString` parameters:

```
IEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString);
```

In *SchoolRepository.cs*, add the following new method:

```
publicIEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString)
{
if(String.IsNullOrWhiteSpace(sortExpression))
{
                sortExpression ="Name";
}
if(String.IsNullOrWhiteSpace(nameSearchString))
{
                nameSearchString ="";
}
return context.Departments.Include("Person").OrderBy("it."+ sortExpression).Where(d
=> d.Name.Contains(nameSearchString)).ToList();
}
```

This code uses a `Where` method to select items that contain the search string. If the search string is empty, all records will be selected. Note that when you specify method calls together in one statement like this (`Include`, then `OrderBy`, then `Where`), the `Where` method must always be last.

Change the existing `GetDepartments` method that takes a `sortExpression` parameter to call the new method:

```
publicIEnumerable<Department>GetDepartments(string sortExpression)
{
returnGetDepartmentsByName(sortExpression,"");
}
```

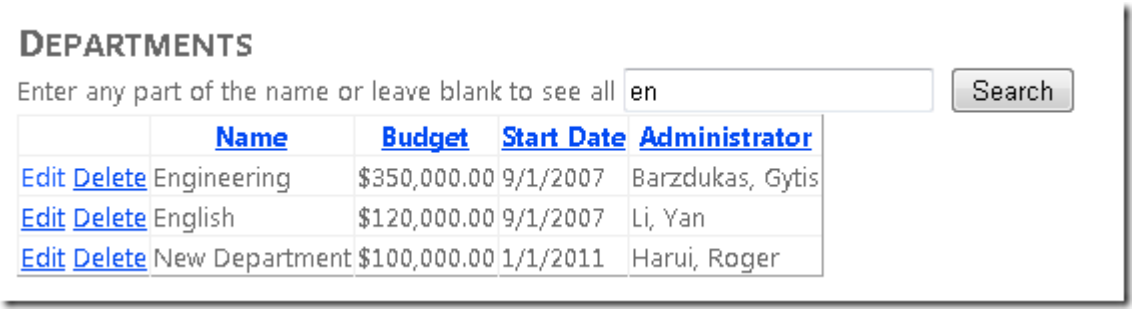
In *MockSchoolRepository.cs* in the test project, add the following new method:

```
publicIEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString)
{
return departments;
}
```

In *SchoolBL.cs*, add the following new method:

```
publicIEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString)
{
return schoolRepository.GetDepartmentsByName(sortExpression, nameSearchString);
}
```

Run the *Departments.aspx* page and enter a search string to make sure that the selection logic works. Leave the text box empty and try a search to make sure that all records are returned.



Adding a Details Column for Each Grid Row

Next, you want to see all of the courses for each department displayed in the right-hand cell of the grid. To do this, you'll use a nested `GridView` control and databind it to data from the `Courses` navigation property of the `Department` entity.

Open *Departments.aspx* and in the markup for the `GridView` control, specify a handler for the `RowDataBound` event. The markup for the opening tag of the control now resembles the following example.

```
<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource"DataKeyNames="DepartmentID"
OnRowUpdating="DepartmentsGridView_RowUpdating"
OnRowDataBound="DepartmentsGridView_RowDataBound"
AllowSorting="True">
```

Add a new `TemplateField` element after the `Administrator` template field:

```
<asp:TemplateFieldHeaderText="Courses">
<ItemTemplate>
<asp:GridViewID="CoursesGridView"runat="server"AutoGenerateColumns="False">
```

```
<Columns>
<asp:BoundFieldDataField="CourseID"HeaderText="ID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"/>
</Columns>
</asp:GridView>
</ItemTemplate>
</asp:TemplateField>
```

This markup creates a nested `GridView` control that shows the course number and title of a list of courses. It does not specify a data source because you'll databind it in code in the `RowDataBound` handler.

Open *Departments.aspx.cs* and add the following handler for the `RowDataBound` event:

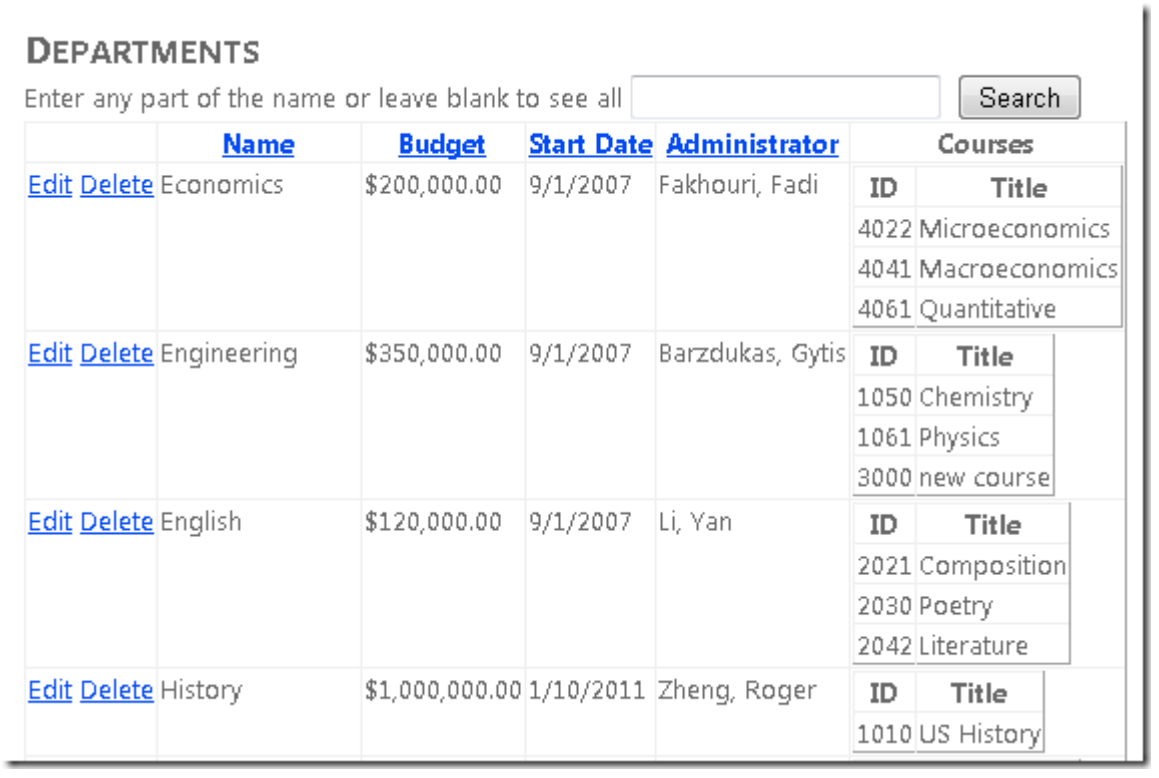
```
protectedvoidDepartmentsGridView_RowDataBound(object sender,GridViewRowEventArgs e)
{
if(e.Row.RowType==DataControlRowType.DataRow)
{
var department = e.Row.DataItemasDepartment;
var coursesGridView =(GridView)e.Row.FindControl("CoursesGridView");
                coursesGridView.DataSource= department.Courses.ToList();
                coursesGridView.DataBind();
}
}
```

This code gets the `Department` entity from the event arguments, converts the `Courses` navigation property to a `List` collection, and databinds the nested `GridView` to the collection.

Open the *SchoolRepository.cs* file and specify eager loading for the `Courses` navigation property by calling the `Include` method in the object query that you create in the `GetDepartmentsByName` method. The `return` statement in the `GetDepartmentsByName` method now resembles the following example.

```
return context.Departments.Include("Person").Include("Courses").
OrderBy("it."+ sortExpression).Where(d =>
d.Name.Contains(nameSearchString)).ToList();
```

Run the page. In addition to the sorting and filtering capability that you added earlier, the GridView control now shows nested course details for each department.



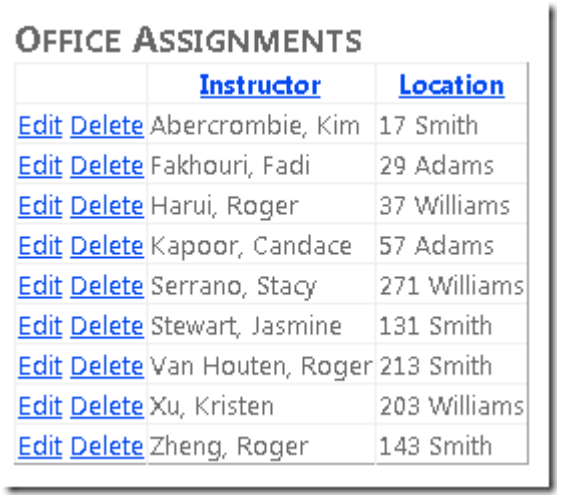
This completes the introduction to sorting, filtering, and master-detail scenarios. In the next tutorial, you'll see how to handle concurrency.

Handling Concurrency

By Tom Dykstra|January 26, 2011

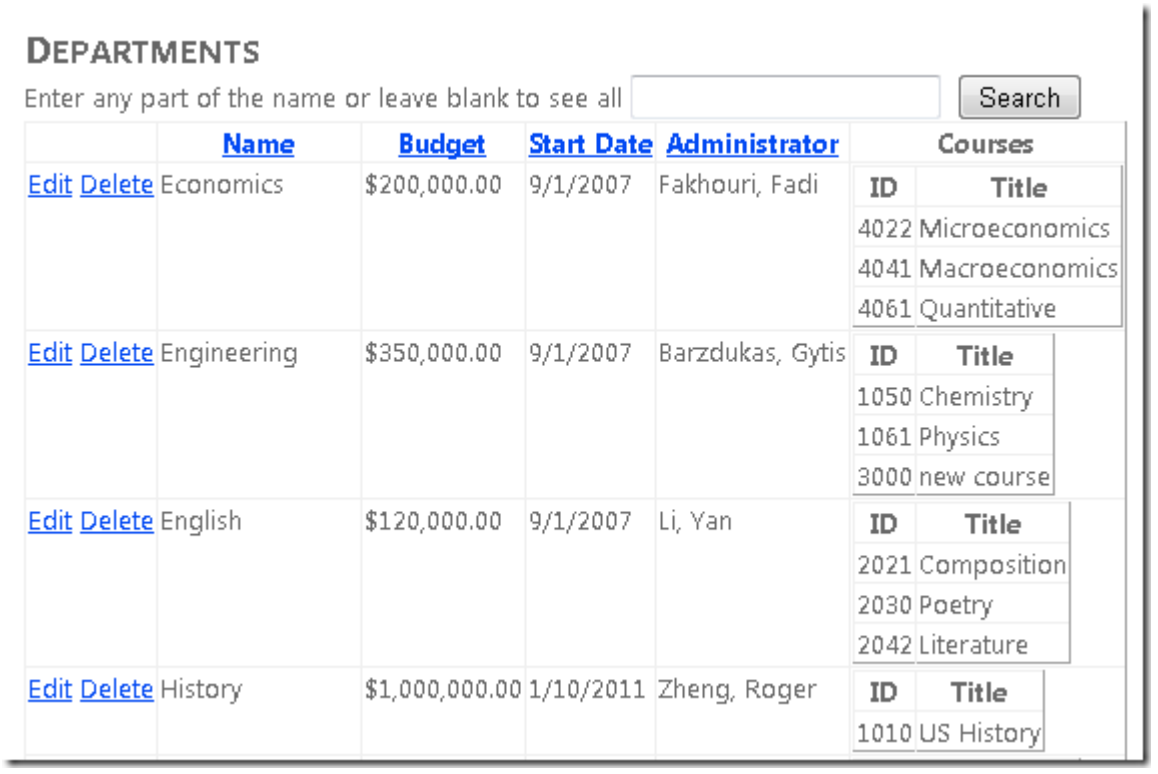
This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you learned how to sort and filter data using the `ObjectDataSource` control and the Entity Framework. This tutorial shows options for handling concurrency in an ASP.NET web application that uses the Entity Framework. You will create a new web page that's dedicated to updating instructor office assignments. You'll handle concurrency issues in that page and in the Departments page that you created earlier.



OFFICE ASSIGNMENTS

	Instructor	Location
Edit Delete	Abercrombie, Kim	17 Smith
Edit Delete	Fakhouri, Fadi	29 Adams
Edit Delete	Harui, Roger	37 Williams
Edit Delete	Kapoor, Candace	57 Adams
Edit Delete	Serrano, Stacy	271 Williams
Edit Delete	Stewart, Jasmine	131 Smith
Edit Delete	Van Houten, Roger	213 Smith
Edit Delete	Xu, Kristen	203 Williams
Edit Delete	Zheng, Roger	143 Smith



Concurrency Conflicts

A concurrency conflict occurs when one user edits a record and another user edits the same record before the first user's change is written to the database. If you don't set up the Entity Framework to detect such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable, and you don't have to configure the application to handle possible concurrency conflicts. (If there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit.) If you don't need to worry about concurrency conflicts, you can skip this tutorial; the remaining two tutorials in the series don't depend on anything you build in this one.

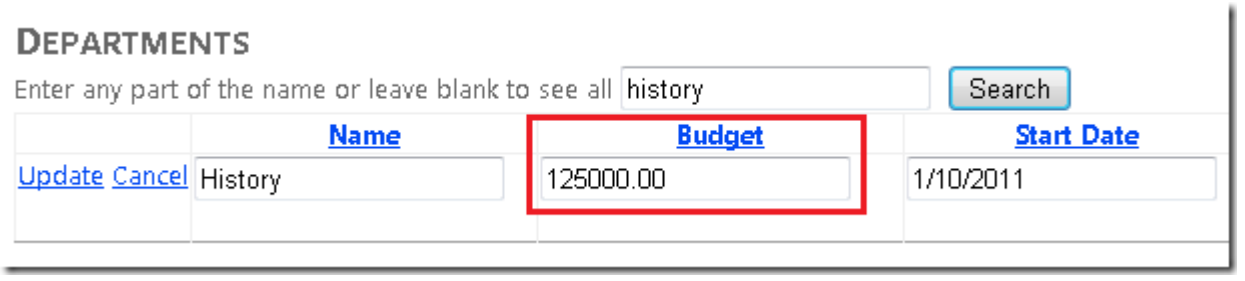
Pessimistic Concurrency (Locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

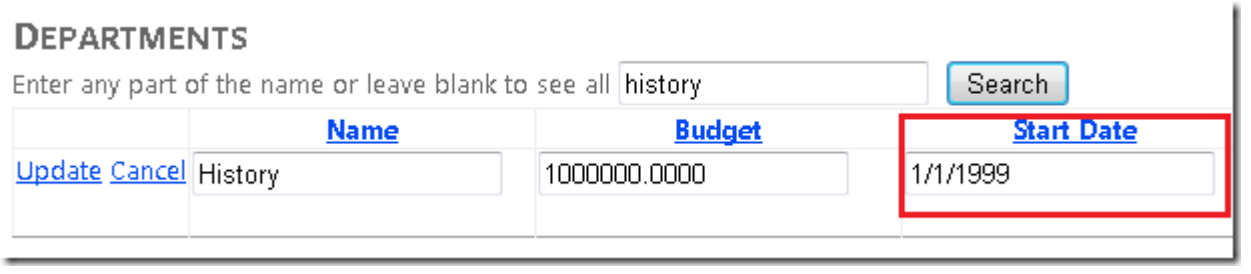
Managing locks has some disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases (that is, it doesn't scale well). For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the *Department.aspx* page, clicks the **Edit** link for the History department, and reduces the **Budget** amount from $1,000,000.00 to $125,000.00. (John administers a competing department and wants to free up money for his own department.)



Before John clicks **Update**, Jane runs the same page, clicks the **Edit** link for the History department, and then changes the **Start Date** field from 1/10/2011 to 1/1/1999. (Jane administers the History department and wants to give it more seniority.)



John clicks **Update** first, then Jane clicks **Update**. Jane's browser now lists the **Budget** amount as $1,000,000.00, but this is incorrect because the amount has been changed by John to $125,000.00.

Some of the actions you can take in this scenario include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated

by the two users. The next time someone browses the History department, they will see 1/1/999 and $125,000.00.

This is the default behavior in the Entity Framework, and it can substantially reduce the number of conflicts that could result in data loss. However, this behavior doesn't avoid data loss if competing changes are made to the same property of an entity. In addition, this behavior isn't always possible; when you map stored procedures to an entity type, all of an entity's properties are updated when any changes to the entity are made in the database.

- You can let Jane's change overwrite John's change. After Jane clicks **Update**, the **Budget** amount goes back to $1,000,000.00. This is called a *Client Wins* or *Last in Wins* scenario. (The client's values take precedence over what's in the data store.)

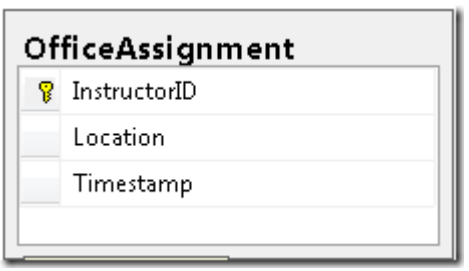
- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reenter her changes if she still wants to make them. You could further automate the process by saving her input and giving her an opportunity to reapply it without having to reenter it. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.)

Detecting Concurrency Conflicts

In the Entity Framework, you can resolve conflicts by handling `OptimisticConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database, include a table column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of SQL `Update` or `Delete` commands.

  That's the purpose of the `Timestamp` column in the `OfficeAssignment` table.



The data type of the `Timestamp` column is also called `Timestamp`. However, the column doesn't actually contain a date or time value. Instead, the value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original `Timestamp` value. If the row being updated has been changed by another user, the value in `Timestamp` is different than the original value, so the `Where` clause returns no row to update. When the Entity Framework finds that no rows have been updated by the current `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

  As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. This method is as effective as using a `Timestamp` field, but can be inefficient. For database tables that have many columns, it can result in very large `Where` clauses, and in a web application it can require that you maintain large amounts of state. Maintaining large amounts of state can affect application performance because it either requires server resources (for example, session state) or must be included in the web page itself (for example, view state).

In this tutorial you will add error handling for optimistic concurrency conflicts for an entity that doesn't have a tracking property (the `Department` entity) and for an entity that does have a tracking property (the `OfficeAssignment` entity).

Handling Optimistic Concurrency Without a Tracking Property

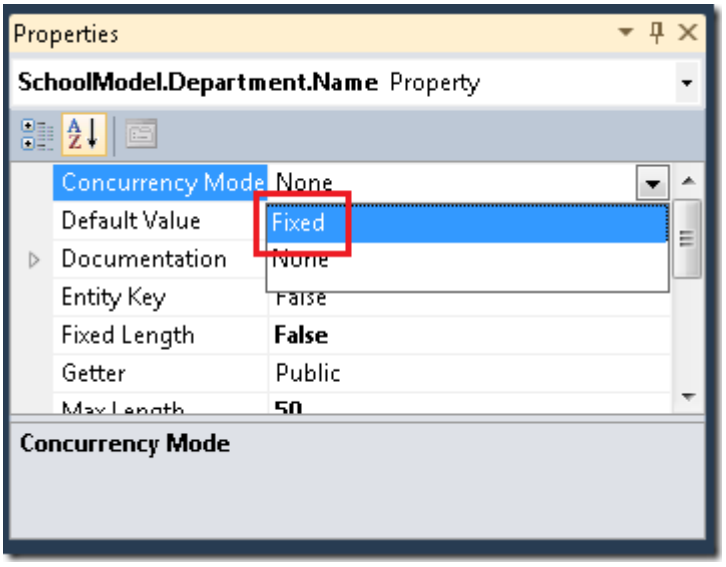
To implement optimistic concurrency for the `Department` entity, which doesn't have a tracking (`Timestamp`) property, you will complete the following tasks:

- Change the data model to enable concurrency tracking for `Department` entities.
- In the `SchoolRepository` class, handle concurrency exceptions in the `SaveChanges` method.
- In the *Departments.aspx* page, handle concurrency exceptions by displaying a message to the user warning that the attempted changes were unsuccessful. The user can then see the current values and retry the changes if they are still needed.

Enabling Concurrency Tracking in the Data Model

In Visual Studio, open the Contoso University web application that you were working with in the previous tutorial in this series.

Open *SchoolModel.edmx*, and in the data model designer, right-click the `Name` property in the `Department` entity and then click **Properties**. In the **Properties** window, change the `ConcurrencyMode` property to `Fixed`.



Do the same for the other non-primary-key scalar properties (`Budget`, `StartDate`, and `Administrator`.) (You can't do this for navigation properties.) This specifies that whenever the Entity Framework generates a `Update` or `Delete` SQL command to update the `Department` entity in the database, these columns (with original values) must be included in the `Where` clause. If no row is found when the `Update` or `Delete` command executes, the Entity Framework will throw an optimistic-concurrency exception.

Save and close the data model.

Handling Concurrency Exceptions in the DAL

Open *SchoolRepository.cs* and add the following `using` statement for the `System.Data` namespace:

```
usingSystem.Data;
```

Add the following new `SaveChanges` method, which handles optimistic concurrency exceptions:

```
publicvoidSaveChanges()
{
try
{
                context.SaveChanges();
```

```
}
catch(OptimisticConcurrencyException ocex)
{
                context.Refresh(RefreshMode.StoreWins, ocex.StateEntries[0].Entity);
throw ocex;
}
}
```

If a concurrency error occurs when this method is called, the property values of the entity in memory are replaced with the values currently in the database. The concurrency exception is rethrown so that the web page can handle it.

In the `DeleteDepartment` and `UpdateDepartment` methods, replace the existing call to `context.SaveChanges()` with a call to `SaveChanges()` in order to invoke the new method.

Handling Concurrency Exceptions in the Presentation Layer

Open *Departments.aspx* and add an `OnDeleted="DepartmentsObjectDataSource_Deleted"` attribute to the `DepartmentsObjectDataSource` control. The opening tag for the control will now resemble the following example.

```
<asp:ObjectDataSourceID="DepartmentsObjectDataSource"runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"DataObjectTypeName="ContosoUniversity.DAL.De
partment"
SelectMethod="GetDepartmentsByName"DeleteMethod="DeleteDepartment"UpdateMethod="Updat
eDepartment"
ConflictDetection="CompareAllValues"OldValuesParameterFormatString="orig{0}"
OnUpdated="DepartmentsObjectDataSource_Updated"SortParameterName="sortExpression"
OnDeleted="DepartmentsObjectDataSource_Deleted">
```

In the `DepartmentsGridView` control, specify all of the table columns in the `DataKeyNames` attribute, as shown in the following example. Note that this will create very large view state fields, which is one reason why using a tracking field is generally the preferred way to track concurrency conflicts.

```
<asp:GridViewID="DepartmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="DepartmentsObjectDataSource"
DataKeyNames="DepartmentID,Name,Budget,StartDate,Administrator"
```

```
OnRowUpdating="DepartmentsGridView_RowUpdating"
OnRowDataBound="DepartmentsGridView_RowDataBound"
AllowSorting="True">
```

Open *Departments.aspx.cs* and add the following `using` statement for the `System.Data` namespace:

```
usingSystem.Data;
```

Add the following new method, which you will call from the data source control's `Updated` and `Deleted` event handlers for handling concurrency exceptions:

```
privatevoidCheckForOptimisticConcurrencyException(ObjectDataSourceStatusEventArgs
e,stringfunction)
{
if(e.Exception.InnerExceptionisOptimisticConcurrencyException)
{
var concurrencyExceptionValidator =newCustomValidator();
                concurrencyExceptionValidator.IsValid=false;
                concurrencyExceptionValidator.ErrorMessage=
"The record you attempted to edit or delete was modified by another "+
"user after you got the original value. The edit or delete operation was canceled "+
"and the other user's values have been displayed so you can "+
"determine whether you still want to edit or delete this record.";
Page.Validators.Add(concurrencyExceptionValidator);
                e.ExceptionHandled=true;
}
}
```

This code checks the exception type, and if it's a concurrency exception, the code dynamically creates a `CustomValidator` control that in turn displays a message in the `ValidationSummary` control.

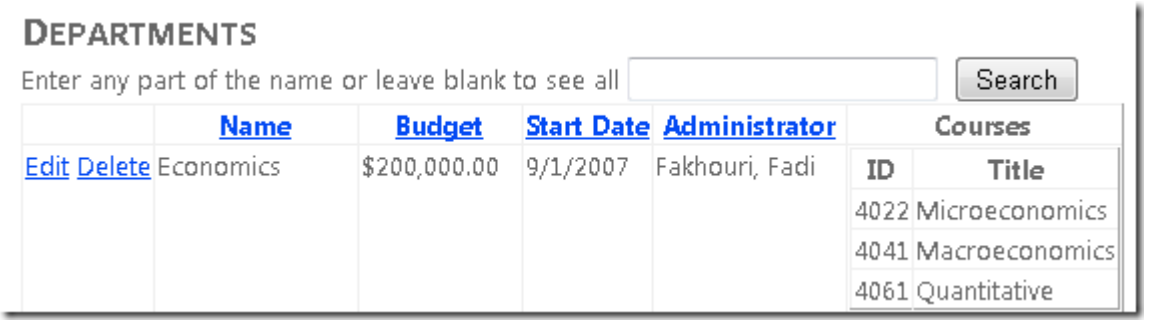
Call the new method from the `Updated` event handler that you added earlier. In addition, create a new `Deleted` event handler that calls the same method (but doesn't do anything else):

```
protectedvoidDepartmentsObjectDataSource_Updated(object
sender,ObjectDataSourceStatusEventArgs e)
```

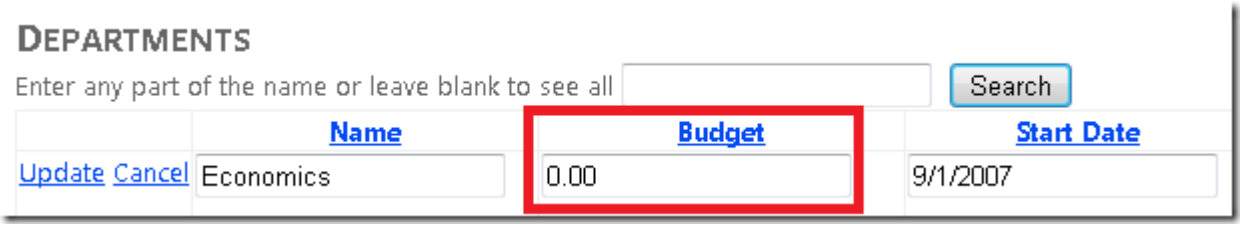
```
{
if(e.Exception!=null)
{
CheckForOptimisticConcurrencyException(e,"update");
// ...
}
}

protectedvoidDepartmentsObjectDataSource_Deleted(object
sender,ObjectDataSourceStatusEventArgs e)
{
if(e.Exception!=null)
{
CheckForOptimisticConcurrencyException(e,"delete");
}
}
```

Testing Optimistic Concurrency in the Departments Page

Run the *Departments.aspx* page.

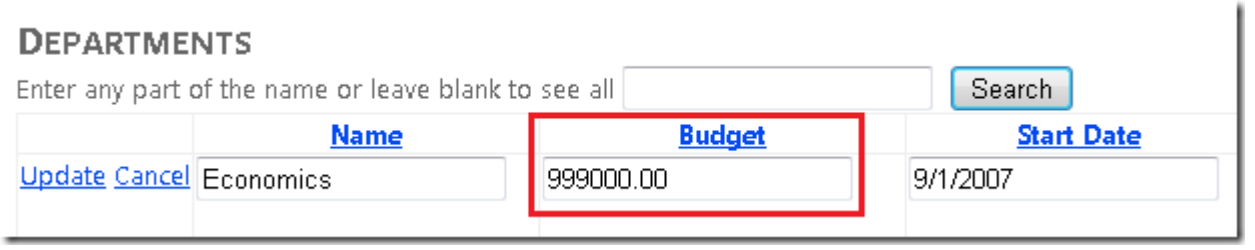


Click **Edit** in a row and change the value in the **Budget** column. (Remember that you can only edit records that you've created for this tutorial, because the existing `School` database records contain some invalid data. The record for the Economics department is a safe one to experiment with.)



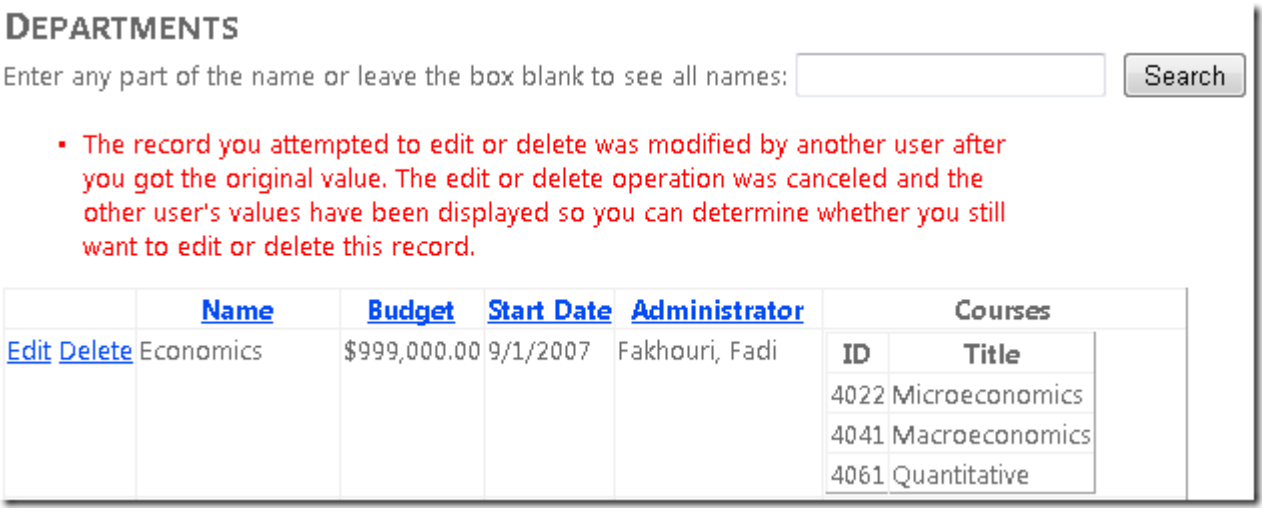
Open a new browser window and run the page again (copy the URL from the first browser window's address box to the second browser window).

Click **Edit** in the same row you edited earlier and change the **Budget** value to something different.



In the second browser window, click **Update**. The **Budget** amount is successfully changed to this new value.

In the first browser window, click **Update**. The update fails. The **Budget** amount is redisplayed using the value you set in the second browser window, and you see an error message.



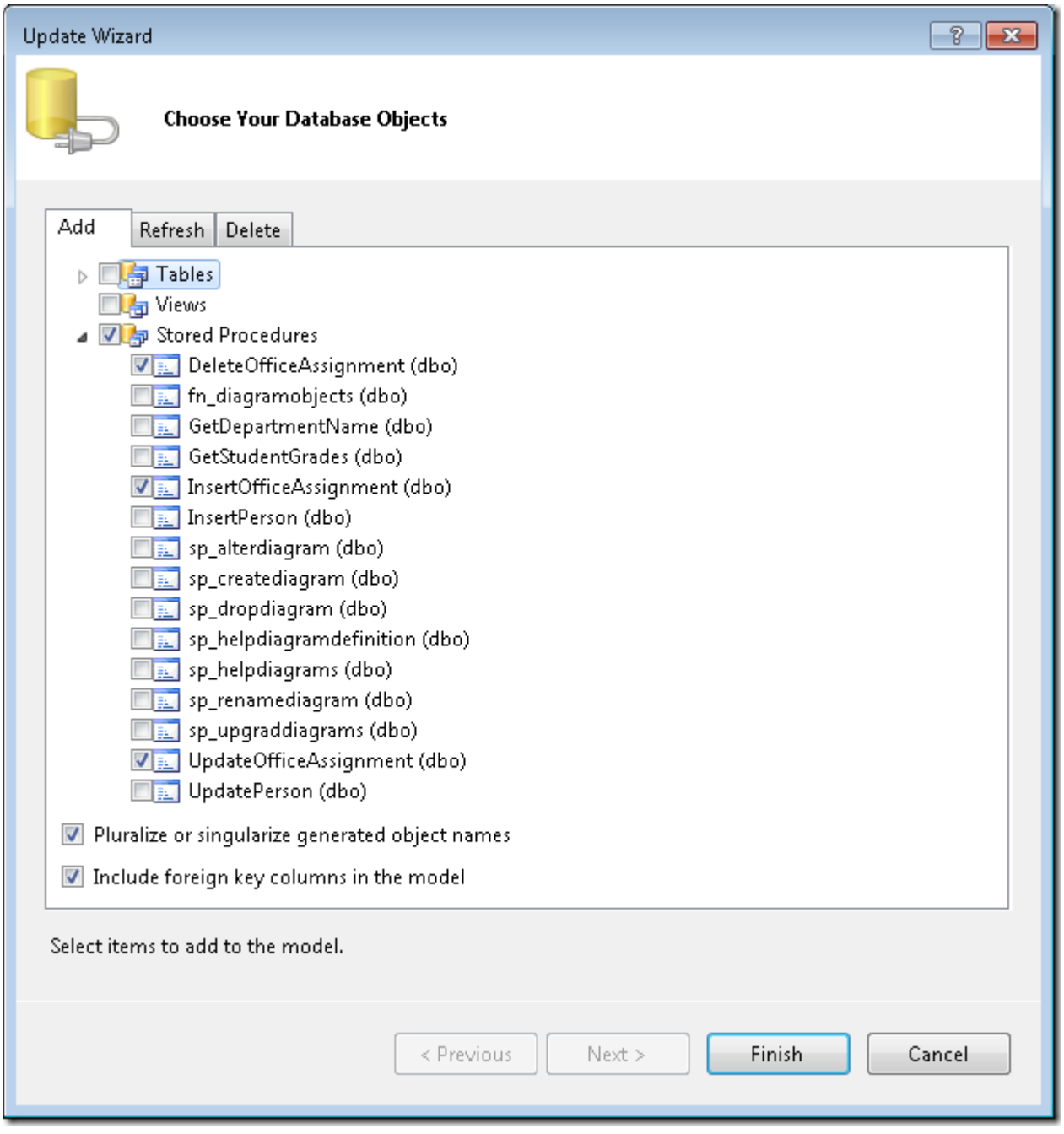
Handling Optimistic Concurrency Using a Tracking Property

To handle optimistic concurrency for an entity that has a tracking property, you will complete the following tasks:

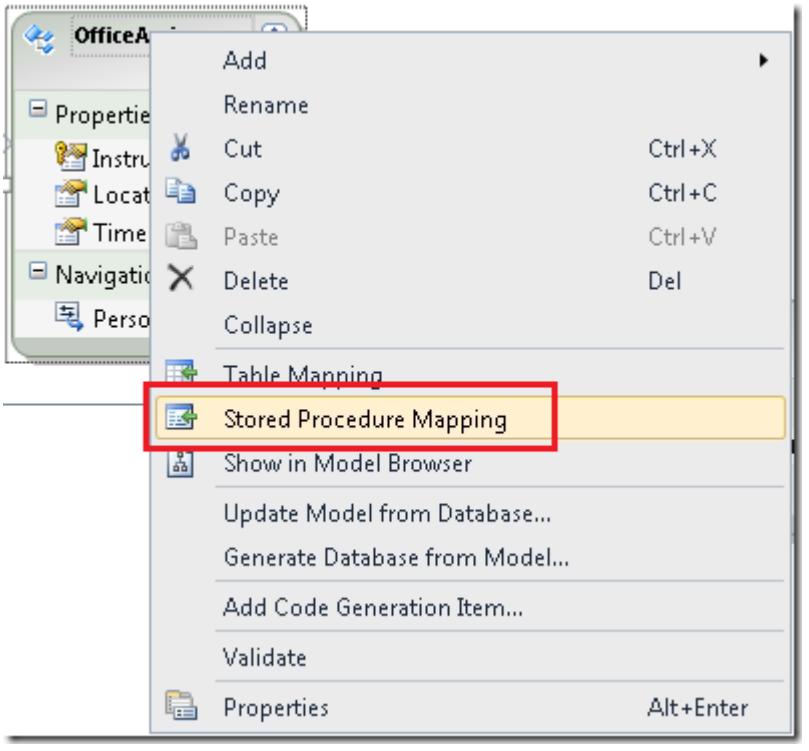
- Add stored procedures to the data model to manage `OfficeAssignment` entities. (Tracking properties and stored procedures don't have to be used together; they're just grouped together here for illustration.)
- Add CRUD methods to the DAL and the BLL for `OfficeAssignment` entities, including code to handle optimistic concurrency exceptions in the DAL.
- Create an office-assignments web page.
- Test optimistic concurrency in the new web page.

Adding OfficeAssignment Stored Procedures to the Data Model

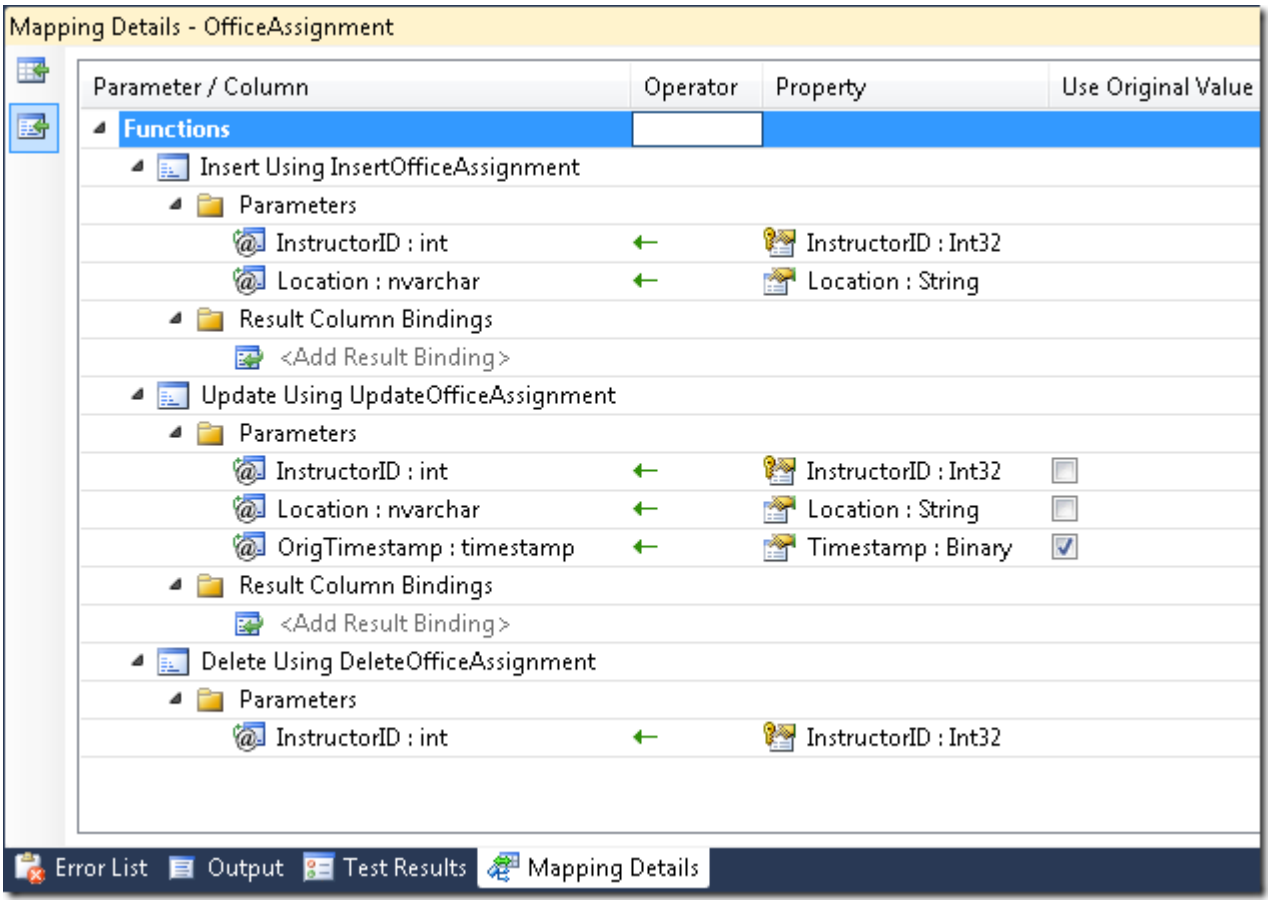
Open the *SchoolModel.edmx* file in the model designer, right-click the design surface, and click **Update Model from Database**. In the **Add** tab of the **Choose Your Database Objects** dialog box, expand **Stored Procedures** and select the three `OfficeAssignment` stored procedures (see the following screenshot), and then click **Finish**. (These stored procedures were already in the database when you downloaded or created it using a script.)



Right-click the `OfficeAssignment` entity and select **Stored Procedure Mapping**.



Set the **Insert**, **Update**, and **Delete** functions to use their corresponding stored procedures. For the `OrigTimestamp` parameter of the `Update` function, set the **Property** to `Timestamp` and select the **Use Original Value** option.



When the Entity Framework calls the `UpdateOfficeAssignment` stored procedure, it will pass the original value of the `Timestamp` column in the `OrigTimestamp` parameter. The stored procedure uses this parameter in its `Where` clause:

```
ALTER PROCEDURE [dbo].[UpdateOfficeAssignment]
@InstructorIDint,
@Location nvarchar(50),
@OrigTimestamp timestamp
                AS
                UPDATE OfficeAssignment SET Location=@Location
                WHERE InstructorID=@InstructorID AND [Timestamp]=@OrigTimestamp;
                IF @@ROWCOUNT>0
BEGIN
                        SELECT [Timestamp] FROM OfficeAssignment
                                WHERE InstructorID=@InstructorID;
END
```

The stored procedure also selects the new value of the `Timestamp` column after the update so that the Entity Framework can keep the `OfficeAssignment` entity that's in memory in sync with the corresponding database row.

(Note that the stored procedure for deleting an office assignment doesn't have an `OrigTimestamp` parameter. Because of this, the Entity Framework can't verify that an entity is unchanged before deleting it.)

Save and close the data model.

Adding OfficeAssignment Methods to the DAL

Open *ISchoolRepository.cs* and add the following CRUD methods for the `OfficeAssignment` entity set:

```
IEnumerable<OfficeAssignment>GetOfficeAssignments(string sortExpression);
voidInsertOfficeAssignment(OfficeAssignmentOfficeAssignment);
voidDeleteOfficeAssignment(OfficeAssignmentOfficeAssignment);
voidUpdateOfficeAssignment(OfficeAssignmentOfficeAssignment,OfficeAssignment
origOfficeAssignment);
```

Add the following new methods to *SchoolRepository.cs*. In the `UpdateOfficeAssignment` method, you call the local `SaveChanges` method instead of `context.SaveChanges`.

```
publicIEnumerable<OfficeAssignment>GetOfficeAssignments(string sortExpression)
{
returnnewObjectQuery<OfficeAssignment>("SELECT VALUE o FROM OfficeAssignments AS o",
context).Include("Person").OrderBy("it."+ sortExpression).ToList();
}

publicvoidInsertOfficeAssignment(OfficeAssignment officeAssignment)
{
            context.OfficeAssignments.AddObject(officeAssignment);
            context.SaveChanges();
}

publicvoidDeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
            context.OfficeAssignments.Attach(officeAssignment);
            context.OfficeAssignments.DeleteObject(officeAssignment);
```

```
            context.SaveChanges();
}

publicvoidUpdateOfficeAssignment(OfficeAssignment officeAssignment,OfficeAssignment
origOfficeAssignment)
{
            context.OfficeAssignments.Attach(origOfficeAssignment);
            context.ApplyCurrentValues("OfficeAssignments", officeAssignment);
SaveChanges();
}
```

In the test project, open *MockSchoolRepository.cs* and add the following `OfficeAssignment` collection and CRUD methods to it. (The mock repository must implement the repository interface, or the solution won't compile.)

```
List<OfficeAssignment> officeAssignments =newList<OfficeAssignment>();

publicIEnumerable<OfficeAssignment>GetOfficeAssignments(string sortExpression)
{
return officeAssignments;
}

publicvoidInsertOfficeAssignment(OfficeAssignment officeAssignment)
{
            officeAssignments.Add(officeAssignment);
}

publicvoidDeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
            officeAssignments.Remove(officeAssignment);
}

publicvoidUpdateOfficeAssignment(OfficeAssignment officeAssignment,OfficeAssignment
origOfficeAssignment)
{
            officeAssignments.Remove(origOfficeAssignment);
```

```
            officeAssignments.Add(officeAssignment);
}
```

Adding OfficeAssignment Methods to the BLL

In the main project, open *SchoolBL.cs* and add the following CRUD methods for the `OfficeAssignment` entity set to it:

```
publicIEnumerable<OfficeAssignment>GetOfficeAssignments(string sortExpression)
{
if(string.IsNullOrEmpty(sortExpression)) sortExpression ="Person.LastName";
return schoolRepository.GetOfficeAssignments(sortExpression);
}

publicvoidInsertOfficeAssignment(OfficeAssignment officeAssignment)
{
try
{
                schoolRepository.InsertOfficeAssignment(officeAssignment);
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

publicvoidDeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
try
{
                schoolRepository.DeleteOfficeAssignment(officeAssignment);
}
catch(Exception ex)
{
```

```
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}

publicvoidUpdateOfficeAssignment(OfficeAssignment officeAssignment,OfficeAssignment
origOfficeAssignment)
{
try
{
                schoolRepository.UpdateOfficeAssignment(officeAssignment,
origOfficeAssignment);
}
catch(Exception ex)
{
//Include catch blocks for specific exceptions first,
//and handle or log the error as appropriate in each.
//Include a generic catch block like this one last.
throw ex;
}
}
```

Creating an OfficeAssignments Web Page

Create a new web page that uses the *Site.Master* master page and name it *OfficeAssignments.aspx*. Add the following markup to the `Content` control named `Content2`:

```
<h2>Office Assignments</h2>
<asp:ObjectDataSourceID="OfficeAssignmentsObjectDataSource"runat="server"TypeName="Co
ntosoUniversity.BLL.SchoolBL"
DataObjectTypeName="ContosoUniversity.DAL.OfficeAssignment"SelectMethod="GetOfficeAss
ignments"
DeleteMethod="DeleteOfficeAssignment"UpdateMethod="UpdateOfficeAssignment"ConflictDet
ection="CompareAllValues"
OldValuesParameterFormatString="orig{0}"
```

```
SortParameterName="sortExpression"OnUpdated="OfficeAssignmentsObjectDataSource_Update
d">
</asp:ObjectDataSource>
<asp:ValidationSummaryID="OfficeAssignmentsValidationSummary"runat="server"ShowSummar
y="true"
DisplayMode="BulletList"Style="color:Red; width:40em;"/>
<asp:GridViewID="OfficeAssignmentsGridView"runat="server"AutoGenerateColumns="False"
DataSourceID="OfficeAssignmentsObjectDataSource"DataKeyNames="InstructorID,Timestamp"
AllowSorting="True">
<Columns>
<asp:CommandFieldShowEditButton="True"ShowDeleteButton="True"ItemStyle-
VerticalAlign="Top">
<ItemStyleVerticalAlign="Top"></ItemStyle>
</asp:CommandField>
<asp:TemplateFieldHeaderText="Instructor"SortExpression="Person.LastName">
<ItemTemplate>
<asp:Label ID="InstructorLastNameLabel" runat="server" Text='<%#
Eval("Person.LastName") %>'></asp:Label>,
<asp:Label ID="InstructorFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>'></asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:DynamicFieldDataField="Location"HeaderText="Location"SortExpression="Location"/>
</Columns>
<SelectedRowStyleBackColor="LightGray"></SelectedRowStyle>
</asp:GridView>
```

Notice that in the `DataKeyNames` attribute, the markup specifies the `Timestamp` property as well as the record key (`InstructorID`). Specifying properties in the `DataKeyNames` attribute causes the control to save them in control state (which is similar to view state) so that the original values are available during postback processing.

If you didn't save the `Timestamp` value, the Entity Framework would not have it for the `Where` clause of the SQL `Update` command. Consequently nothing would be found to update. As a result, the Entity Framework would throw an optimistic concurrency exception every time an `OfficeAssignment` entity is updated.

Open *OfficeAssignments.aspx.cs* and add the following `using` statement for the data access layer:

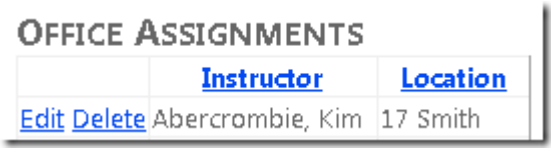
```
usingContosoUniversity.DAL;
```

Add the following `Page_Init` method, which enables Dynamic Data functionality. Also add the following handler for the `ObjectDataSource` control's `Updated` event in order to check for concurrency errors:

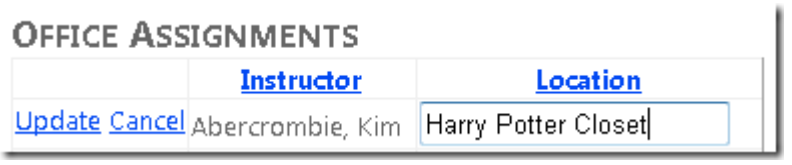
```
protectedvoidPage_Init(object sender,EventArgs e)
{
OfficeAssignmentsGridView.EnableDynamicData(typeof(OfficeAssignment));
}

protectedvoidOfficeAssignmentsObjectDataSource_Updated(object
sender,ObjectDataSourceStatusEventArgs e)
{
if(e.Exception!=null)
{
var concurrencyExceptionValidator =newCustomValidator();
                concurrencyExceptionValidator.IsValid=false;
                concurrencyExceptionValidator.ErrorMessage="The record you attempted
to "+
"update has been modified by another user since you last visited this page. "+
"Your update was canceled to allow you to review the other user's "+
"changes and determine if you still want to update this record.";
Page.Validators.Add(concurrencyExceptionValidator);
                e.ExceptionHandled=true;
}
}
```

Testing Optimistic Concurrency in the OfficeAssignments Page

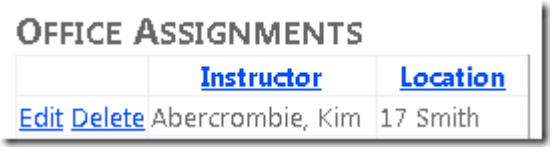
Run the *OfficeAssignments.aspx* page.



Click **Edit** in a row and change the value in the **Location** column.



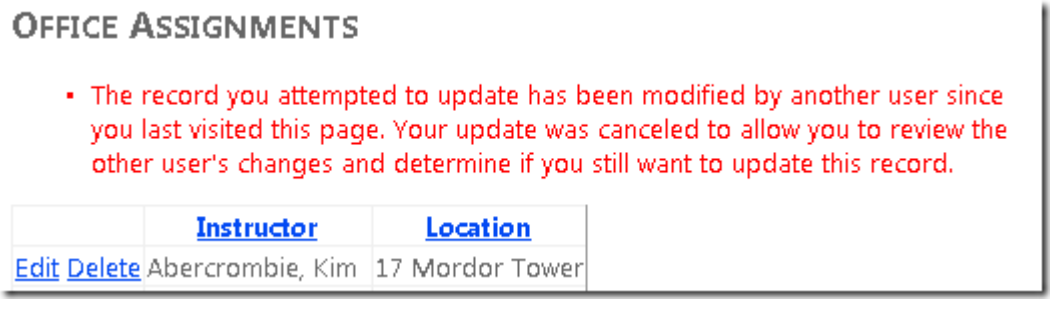
Open a new browser window and run the page again (copy the URL from the first browser window to the second browser window).



Click **Edit** in the same row you edited earlier and change the **Location** value to something different.

In the second browser window, click **Update**.

Switch to the first browser window and click **Update**.



You see an error message and the **Location** value has been updated to show the value you changed it to in the second browser window.

Handling Concurrency with the EntityDataSource Control

The `EntityDataSource` control includes built-in logic that recognizes the concurrency settings in the data model and handles update and delete operations accordingly. However, as with all exceptions, you must handle `OptimisticConcurrencyException` exceptions yourself in order to provide a user-friendly error message.

Next, you will configure the *Courses.aspx* page (which uses an `EntityDataSource` control) to allow update and delete operations and to display an error message if a concurrency conflict occurs. The `Course` entity doesn't have a concurrency tracking column, so you will use the same method that you did with the `Department` entity: track the values of all non-key properties.

Open the *SchoolModel.edmx* file. For the non-key properties of the `Course` entity (`Title`, `Credits`, and `DepartmentID`), set the **Concurrency Mode** property to `Fixed`. Then save and close the data model.

Open the *Courses.aspx* page and make the following changes:

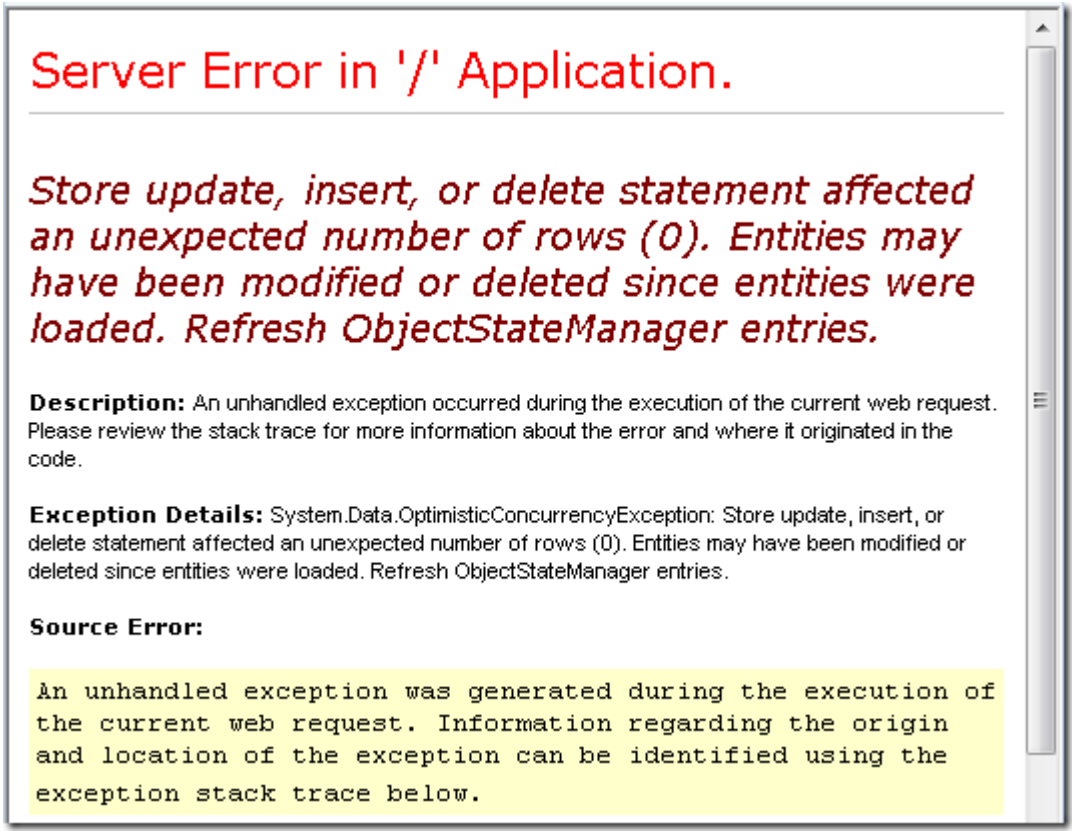
- In the `CoursesEntityDataSource` control, add `EnableUpdate="true"` and `EnableDelete="true"` attributes. The opening tag for that control now resembles the following example:

```
<asp:EntityDataSourceID="CoursesEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="false"
AutoGenerateWhereClause="True"EntitySetName="Courses"
EnableUpdate="true"EnableDelete="true">
```

- In the `CoursesGridView` control, change the `DataKeyNames` attribute value to `"CourseID,Title,Credits,DepartmentID"`. Then add a `CommandField` element to the `Columns`element that shows **Edit** and **Delete** buttons (`<asp:CommandField ShowEditButton="True" ShowDeleteButton="True" />`). The `GridView` control now resembles the following example:

```
<asp:GridViewID="CoursesGridView"runat="server"AutoGenerateColumns="False"
DataKeyNames="CourseID,Title,Credits,DepartmentID"
DataSourceID="CoursesEntityDataSource">
<Columns>
<asp:CommandFieldShowEditButton="True"ShowDeleteButton="True"/>
<asp:BoundFieldDataField="CourseID"HeaderText="CourseID"ReadOnly="True"SortExpres
sion="CourseID"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"SortExpression="Title"/>
<asp:BoundFieldDataField="Credits"HeaderText="Credits"SortExpression="Credits"/>
</Columns>
</asp:GridView>
```

Run the page and create a conflict situation as you did before in the Departments page. Run the page in two browser windows, click **Edit** in the same line in each window, and make a different change in each one. Click **Update** in one window and then click **Update** in the other window. When you click **Update** the second time, you see the error page that results from an unhandled concurrency exception.



You handle this error in a manner very similar to how you handled it for the `ObjectDataSource` control. Open the *Courses.aspx* page, and in the `CoursesEntityDataSource` control, specify handlers for the `Deleted` and `Updated` events. The opening tag of the control now resembles the following example:

```
<asp:EntityDataSourceID="CoursesEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.SchoolEntities"EnableFlattening="false"
AutoGenerateWhereClause="true"EntitySetName="Courses"
EnableUpdate="true"EnableDelete="true"
OnDeleted="CoursesEntityDataSource_Deleted"
OnUpdated="CoursesEntityDataSource_Updated">
```

Before the `CoursesGridView` control, add the following `ValidationSummary` control:

```
<asp:ValidationSummaryID="CoursesValidationSummary"runat="server"
ShowSummary="true"DisplayMode="BulletList"/>
```

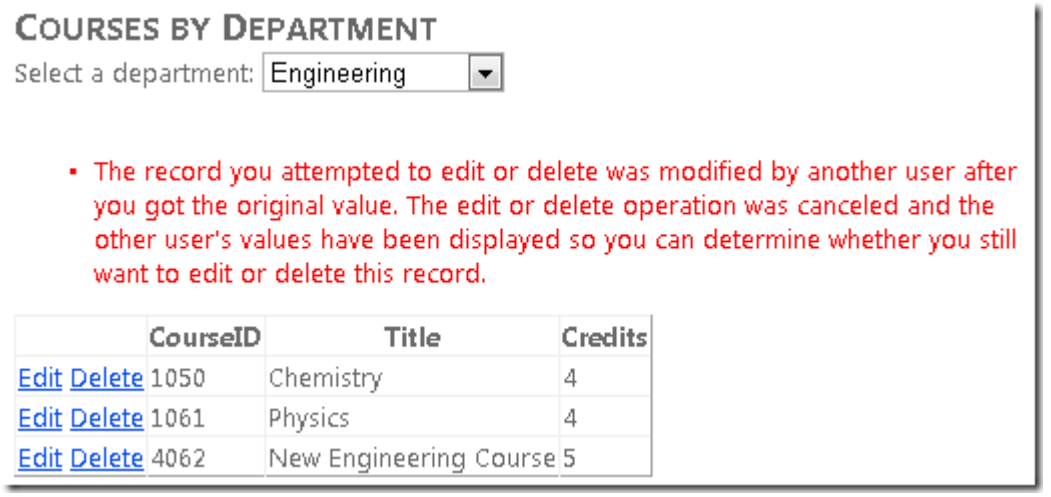
In *Courses.aspx.cs*, add a `using` statement for the `System.Data` namespace, add a method that checks for concurrency exceptions, and add handlers for the `EntityDataSource` control's `Updated` and `Deleted` handlers. The code will look like the following:

```
usingSystem.Data;

protectedvoidCoursesEntityDataSource_Updated(object
sender,EntityDataSourceChangedEventArgs e)
{
CheckForOptimisticConcurrencyException(e,"update");
}

protectedvoidCoursesEntityDataSource_Deleted(object
sender,EntityDataSourceChangedEventArgs e)
{
CheckForOptimisticConcurrencyException(e,"delete");
}

privatevoidCheckForOptimisticConcurrencyException(EntityDataSourceChangedEventArgs
e,stringfunction)
{
if(e.Exception!=null&& e.ExceptionisOptimisticConcurrencyException)
{
var concurrencyExceptionValidator =newCustomValidator();
                concurrencyExceptionValidator.IsValid=false;
                concurrencyExceptionValidator.ErrorMessage=
"The record you attempted to edit or delete was modified by another "+
"user after you got the original value. The edit or delete operation was canceled "+
"and the other user's values have been displayed so you can "+
"determine whether you still want to edit or delete this record.";
Page.Validators.Add(concurrencyExceptionValidator);
                e.ExceptionHandled=true;
```

```
}
}
```

The only difference between this code and what you did for the `ObjectDataSource` control is that in this case the concurrency exception is in the `Exception` property of the event arguments object rather than in that exception's `InnerException` property.

Run the page and create a concurrency conflict again. This time you see an error message:



This completes the introduction to handling concurrency conflicts. The next tutorial will provide guidance on how to improve performance in a web application that uses the Entity Framework.

Maximizing Performance

In the previous tutorial, you saw how to handle concurrency conflicts. This tutorial shows options for improving the performance of an ASP.NET web application that uses the Entity Framework. You'll learn several methods for maximizing performance or for diagnosing performance problems.

Information presented in the following sections is likely to be useful in a broad variety of scenarios:

- Efficiently load related data.
- Manage view state.

Information presented in the following sections might be useful if you have individual queries that present performance problems:

- Use the `NoTracking` merge option.
- Pre-compile LINQ queries.
- Examine query commands sent to the database.

Information presented in the following section is potentially useful for applications that have extremely large data models:

- Pre-generate views.

Note Web application performance is affected by many factors, including things like the size of request and response data, the speed of database queries, how many requests the server can queue and how quickly it can service them, and even the efficiency of any client-script libraries you might be using. If performance is critical in your application, or if testing or experience shows that application performance isn't satisfactory, you should follow normal protocol for performance tuning. Measure to determine where performance bottlenecks are occurring, and then address the areas that will have the greatest impact on overall application performance.

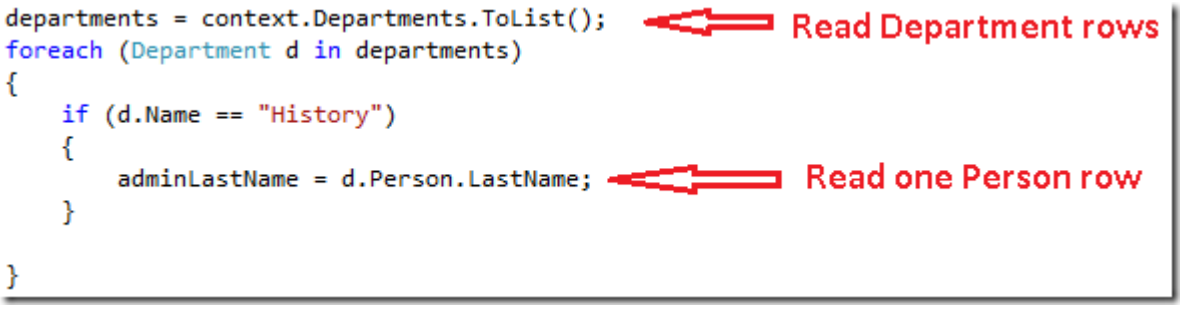
This topic focuses mainly on ways in which you can potentially improve the performance specifically of the Entity Framework in ASP.NET. The suggestions here are useful if you determine that data access is one of the performance bottlenecks in your application. Except as noted, the methods explained here shouldn't be considered "best practices" in general — many of them are appropriate only in exceptional situations or to address very specific kinds of performance bottlenecks.

To start the tutorial, start Visual Studio and open the Contoso University web application that you were working with in the previous tutorial.

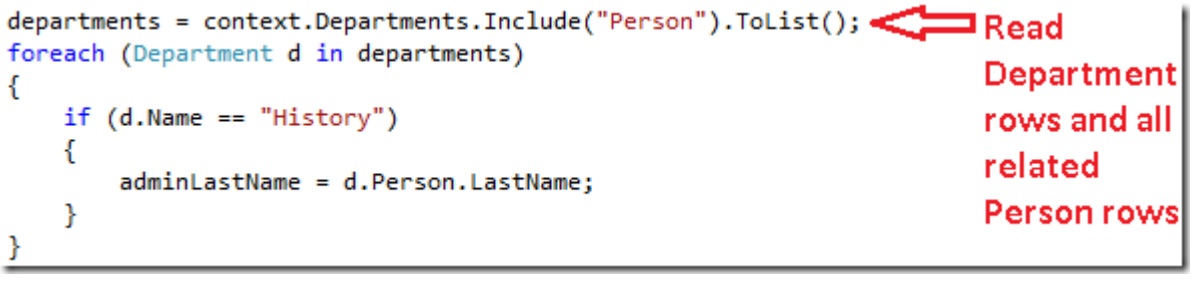
Efficiently Loading Related Data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved.



Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the `Include` method, as you've seen already in these tutorials.



- *Explicit loading*. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually using the `Load` method of the navigation property for collections, or you use the `Load` method of the reference property for properties that hold a single object. (For example, you call the `PersonReference.Load` method to load the `Person` navigation property of a `Department` entity.)

```
departments = context.Departments.ToList();
foreach (Department d in departments)
{
    if (d.Name == "History")
    {
        d.PersonReference.Load();
        adminLastName = d.Person.LastName;
    }
}
```

Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as *deferred loading*.

Lazy loading is the default behavior for an object context that has been generated by the designer. If you open the *SchoolModel.Designer.cs* file that defines the object context class, you'll find three constructor methods, and each of them includes the following statement:

```
this.ContextOptions.LazyLoadingEnabled=true;
```

In general, if you know you need related data for every entity retrieved, eager loading offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. On the other hand, if you need to access an entity's navigation properties only infrequently or only for a small set of the entities, lazy loading or explicit loading may be more efficient, because eager loading would retrieve more data than you need.

In a web application, lazy loading may be of relatively little value anyway, because user actions that affect the need for related data take place in the browser, which has no connection to the object context that rendered the page. On the other hand, when you databind a control, you typically know what data you need, and so it's generally best to choose eager loading or deferred loading based on what's appropriate in each scenario.

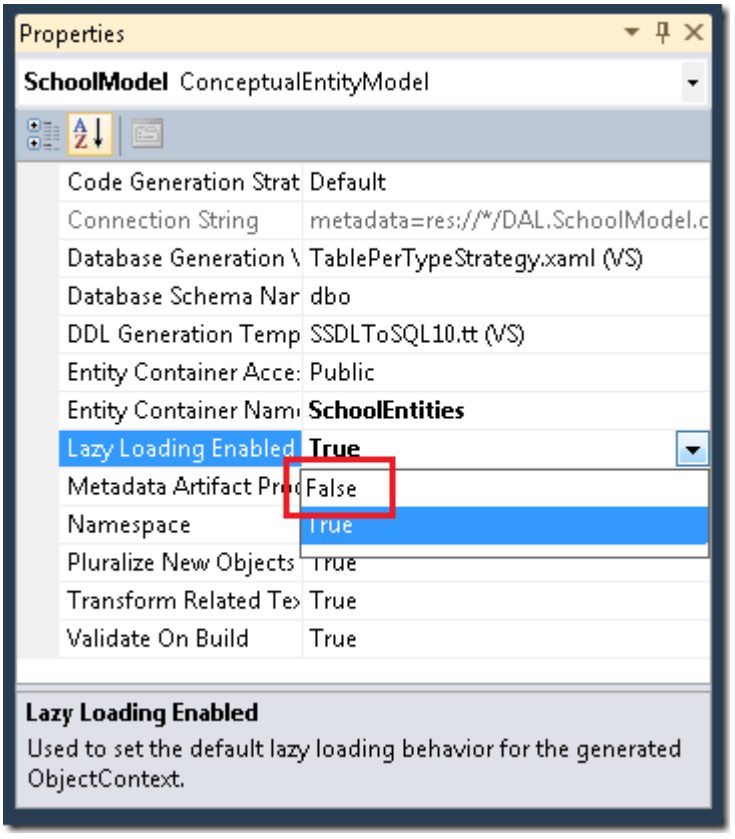
In addition, a databound control might use an entity object after the object context is disposed. In that case, an attempt to lazy-load a navigation property would fail. The error message you receive is clear: "`The ObjectContext instance has been disposed and can no longer be used for operations that require a connection.`"

The `EntityDataSource` control disables lazy loading by default. For the `ObjectDataSource` control that you're using for the current tutorial (or if you access the object context from page code), there are several ways you can make lazy loading disabled by default. You can disable it when you instantiate an object context. For example, you can add the following line to the constructor method of the `SchoolRepository` class:

```
context.ContextOptions.LazyLoadingEnabled=false;
```

For the Contoso University application, you'll make the object context automatically disable lazy loading so that this property doesn't have to be set whenever a context is instantiated.

Open the *SchoolModel.edmx* data model, click the design surface, and then in the properties pane set the **Lazy Loading Enabled** property to `False`. Save and close the data model.



Managing View State

In order to provide update functionality, an ASP.NET web page must store the original property values of an entity when a page is rendered. During postback processing the control can re-create the original state of the entity and call the entity's `Attach` method before applying changes and calling the `SaveChanges` method. By default, ASP.NET Web Forms data controls use view state to store the original values. However, view state can affect performance, because it's stored in hidden fields that can substantially increase the size of the page that's sent to and from the browser.

Techniques for managing view state, or alternatives such as session state, aren't unique to the Entity Framework, so this tutorial doesn't go into this topic in detail. For more information see the links at the end of the tutorial.

However, version 4 of ASP.NET provides a new way of working with view state that every ASP.NET developer of Web Forms applications should be aware of: the `ViewStateMode` property. This new property can be set at the page or control level, and it enables you to disable view state by default for a page and enable it only for controls that need it.

For applications where performance is critical, a good practice is to always disable view state at the page level and enable it only for controls that require it. The size of view state in the Contoso University pages wouldn't be substantially decreased by this method, but to see how it works, you'll do it for the *Instructors.aspx* page. That page contains many controls, including a `Label` control that has view state disabled. None of the controls on this page actually need to have view state enabled. (The `DataKeyNames` property of the `GridView` control specifies state that must be maintained between postbacks, but these values are kept in control state, which isn't affected by the `ViewStateMode` property.)

The `Page` directive and `Label` control markup currently resembles the following example:

```
<%@PageTitle=""Language="C#"MasterPageFile="~/Site.Master"AutoEventWireup="true"
CodeBehind="Instructors.aspx.cs"Inherits="ContosoUniversity.Instructors" %>
    ...
<asp:LabelID="ErrorMessageLabel"runat="server"Text=""Visible="false"ViewStateMode="Di
sabled"></asp:Label>
    ...
```

Make the following changes:

- Add `ViewStateMode="Disabled"` to the `Page` directive.
- Remove `ViewStateMode="Disabled"` from the `Label` control.

The markup now resembles the following example:

```
<%@PageTitle=""Language="C#"MasterPageFile="~/Site.Master"AutoEventWireup="true"
CodeBehind="Instructors.aspx.cs"Inherits="ContosoUniversity.Instructors"
ViewStateMode="Disabled" %>
    ...
```

```
<asp:LabelID="ErrorMessageLabel"runat="server"Text=""Visible="false"></asp:Label>
    ...
```

View state is now disabled for all controls. If you later add a control that does need to use view state, all you need to do is include the `ViewStateMode="Enabled"` attribute for that control.

Using The NoTracking Merge Option

When an object context retrieves database rows and creates entity objects that represent them, by default it also tracks those entity objects using its object state manager. This tracking data acts as a cache and is used when you update an entity. Because a web application typically has short-lived object context instances, queries often return data that doesn't need to be tracked, because the object context that reads them will be disposed before any of the entities it reads are used again or updated.

In the Entity Framework, you can specify whether the object context tracks entity objects by setting a *merge option*. You can set the merge option for individual queries or for entity sets. If you set it for an entity set, that means that you're setting the default merge option for all queries that are created for that entity set.

For the Contoso University application, tracking isn't needed for any of the entity sets that you access from the repository, so you can set the merge option to `NoTracking` for those entity sets when you instantiate the object context in the repository class. (Note that in this tutorial, setting the merge option won't have a noticeable effect on the application's performance. The `NoTracking` option is likely to make an observable performance improvement only in certain high-data-volume scenarios.)

In the DAL folder, open the *SchoolRepository.cs* file and add a constructor method that sets the merge option for the entity sets that the repository accesses:

```
publicSchoolRepository()
{
            context.Departments.MergeOption=MergeOption.NoTracking;
            context.InstructorNames.MergeOption=MergeOption.NoTracking;
            context.OfficeAssignments.MergeOption=MergeOption.NoTracking;
}
```

Pre-Compiling LINQ Queries

The first time that the Entity Framework executes an Entity SQL query within the life of a given `ObjectContext` instance, it takes some time to compile the query. The result of compilation is cached, which means that subsequent executions of the query are much quicker. LINQ queries follow a similar pattern, except that some of the work required to compile the query is done every time the query is executed. In other words, for LINQ queries, by default not all of the results of compilation are cached.

If you have a LINQ query that you expect to run repeatedly in the life of an object context, you can write code that causes all of the results of compilation to be cached the first time the LINQ query is run.

As an illustration, you'll do this for two `Get` methods in the `SchoolRepository` class, one of which doesn't take any parameters (the `GetInstructorNames` method), and one that does require a parameter (the `GetDepartmentsByAdministrator` method). These methods as they stand now actually don't need to be compiled because they aren't LINQ queries:

```
publicIEnumerable<InstructorName>GetInstructorNames()
{
return context.InstructorNames.OrderBy("it.FullName").ToList();
}


publicIEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator)
{
returnnewObjectQuery<Department>("SELECT VALUE d FROM Departments as d",
context,MergeOption.NoTracking).Include("Person").Where(d => d.Administrator==
administrator).ToList();
}
```

However, so that you can try out compiled queries, you'll proceed as if these had been written as the following LINQ queries:

```
publicIEnumerable<InstructorName>GetInstructorNames()
{
return(from i in context.InstructorNamesorderby i.FullNameselect i).ToList();
}


publicIEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator)
{
            context.Departments.MergeOption=MergeOption.NoTracking;
```

```
return(from d in context.Departmentswhere d.Administrator== administrator select
d).ToList();
}
```

You could change the code in these methods to what's shown above and run the application to verify that it works before continuing. But the following instructions jump right into creating pre-compiled versions of them.

Create a class file in the *DAL* folder, name it *SchoolEntities.cs*, and replace the existing code with the following code:

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Data.Objects;

namespaceContosoUniversity.DAL
{
publicpartialclassSchoolEntities
{
privatestaticreadonlyFunc<SchoolEntities,IQueryable<InstructorName>>
compiledInstructorNamesQuery =
CompiledQuery.Compile((SchoolEntities context)=>from i in
context.InstructorNamesorderby i.FullNameselect i);

publicIEnumerable<InstructorName>CompiledInstructorNamesQuery()
{
return compiledInstructorNamesQuery(this).ToList();
}

privatestaticreadonlyFunc<SchoolEntities,Int32,IQueryable<Department>>
compiledDepartmentsByAdministratorQuery =
CompiledQuery.Compile((SchoolEntities context,Int32 administrator)=>from d in
context.Departments.Include("Person")where d.Administrator==administrator select d);

publicIEnumerable<Department>CompiledDepartmentsByAdministratorQuery(Int32
administrator)
{
```

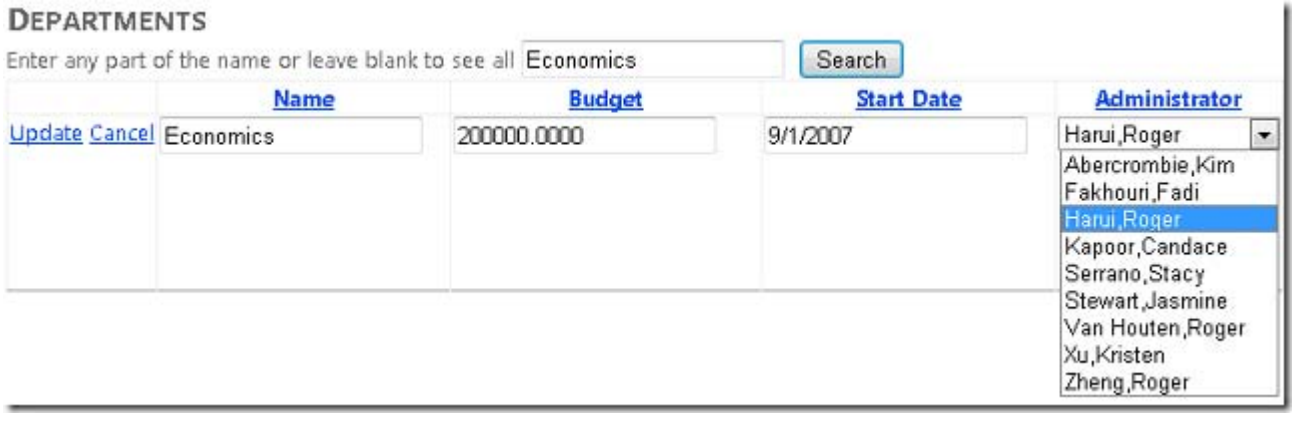
```
return compiledDepartmentsByAdministratorQuery(this, administrator).ToList();
}
}
}
```

This code creates a partial class that extends the automatically generated object context class. The partial class includes two compiled LINQ queries using the `Compile` method of the `CompiledQuery` class. It also creates methods that you can use to call the queries. Save and close this file.

Next, in *SchoolRepository.cs*, change the existing `GetInstructorNames` and `GetDepartmentsByAdministrator` methods in the repository class so that they call the compiled queries:

```
publicIEnumerable<InstructorName>GetInstructorNames()
{
return context.CompiledInstructorNamesQuery();
}

publicIEnumerable<Department>GetDepartmentsByAdministrator(Int32 administrator)
{
return context.CompiledDepartmentsByAdministratorQuery(administrator);
}
```

Run the *Departments.aspx* page to verify that it works as it did before. The `GetInstructorNames` method is called in order to populate the administrator drop-down list, and the `GetDepartmentsByAdministrator` method is called when you click **Update** in order to verify that no instructor is an administrator of more than one department.



You've pre-compiled queries in the Contoso University application only to see how to do it, not because it would measurably improve performance. Pre-compiling LINQ queries does add a level of complexity to your code, so make sure you do it only for queries that actually represent performance bottlenecks in your application.

Examining Queries Sent to the Database

When you're investigating performance issues, sometimes it's helpful to know the exact SQL commands that the Entity Framework is sending to the database. If you're working with an `IQueryable` object, one way to do this is to use the `ToTraceString` method.

In *SchoolRepository.cs*, change the code in the `GetDepartmentsByName` method to match the following example:

```
publicIEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString)
{
...
var departments =newObjectQuery<Department>("SELECT VALUE d FROM Departments AS d",
context).OrderBy("it."+ sortExpression).Include("Person").Include("Courses").Where(d
=> d.Name.Contains(nameSearchString));
string commandText =((ObjectQuery)departments).ToTraceString();
return departments.ToList();
}
```

The `departments` variable must be cast to an `ObjectQuery` type only because the `Where` method at the end of the preceding line creates an `IQueryable` object; without the `Where` method, the cast would not be necessary.

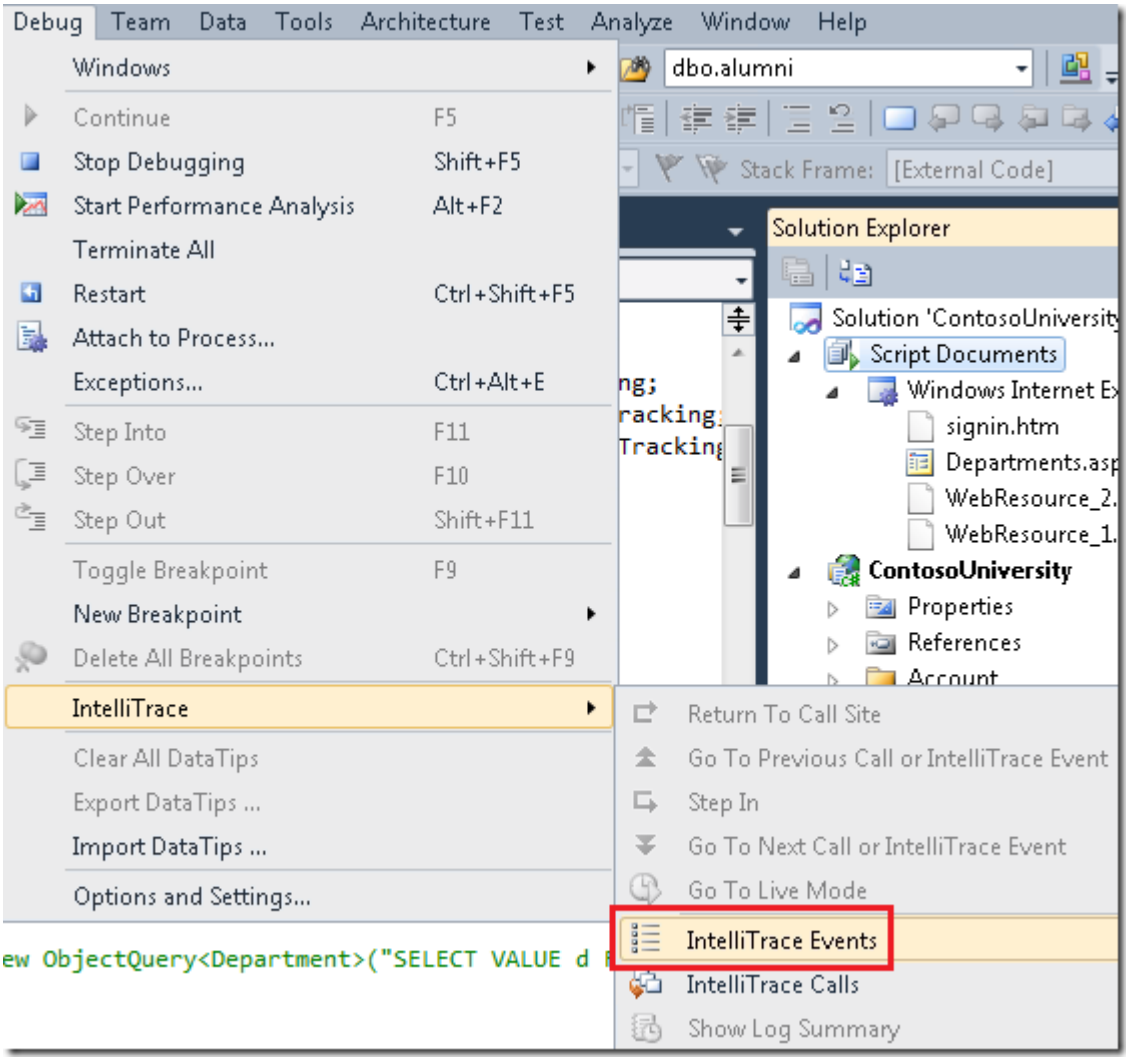
Set a breakpoint on the `return` line, and then run the *Departments.aspx* page in the debugger. When you hit the breakpoint, examine the `commandText` variable in the **Locals** window and use the text visualizer (the magnifying glass in the **Value** column) to display its value in the **Text Visualizer** window. You can see the entire SQL command that results from this code:

As an alternative, the IntelliTrace feature in Visual Studio Ultimate provides a way to view SQL commands generated by the Entity Framework that doesn't require you to change your code or even set a breakpoint.

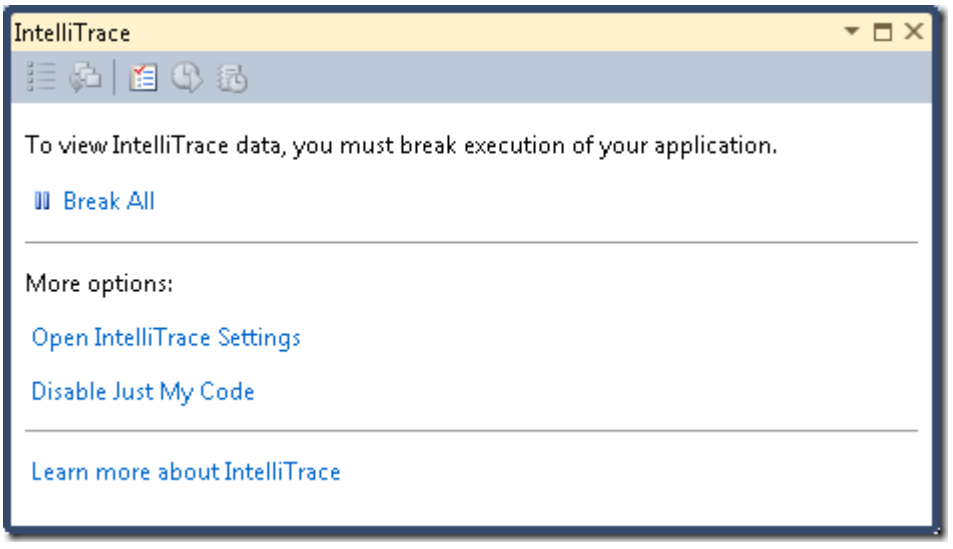
Note You can perform the following procedures only if you have Visual Studio Ultimate.

Restore the original code in the `GetDepartmentsByName` method, and then run the *Departments.aspx* page in the debugger.

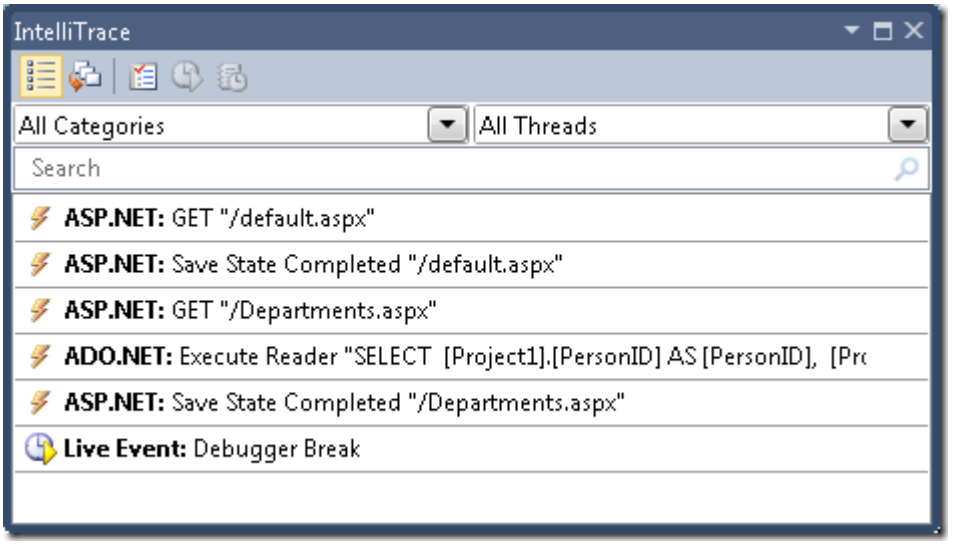
In Visual Studio, select the **Debug** menu, then **IntelliTrace**, and then **IntelliTrace Events**.



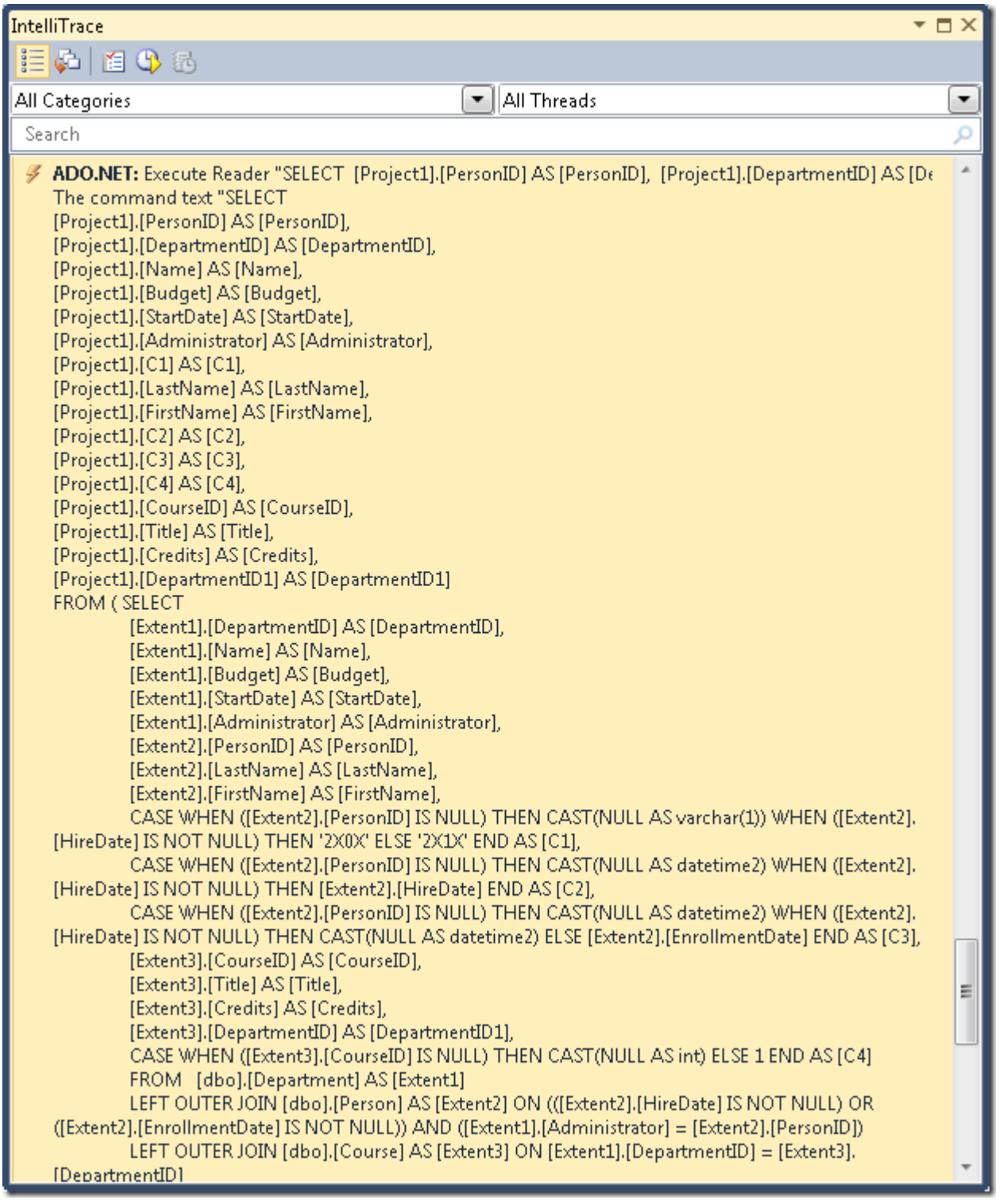
In the **IntelliTrace** window, click **Break All**.



The **IntelliTrace** window displays a list of recent events:



Click the **ADO.NET** line. It expands to show you the command text:



You can copy the entire command text string to the clipboard from the **Locals** window.

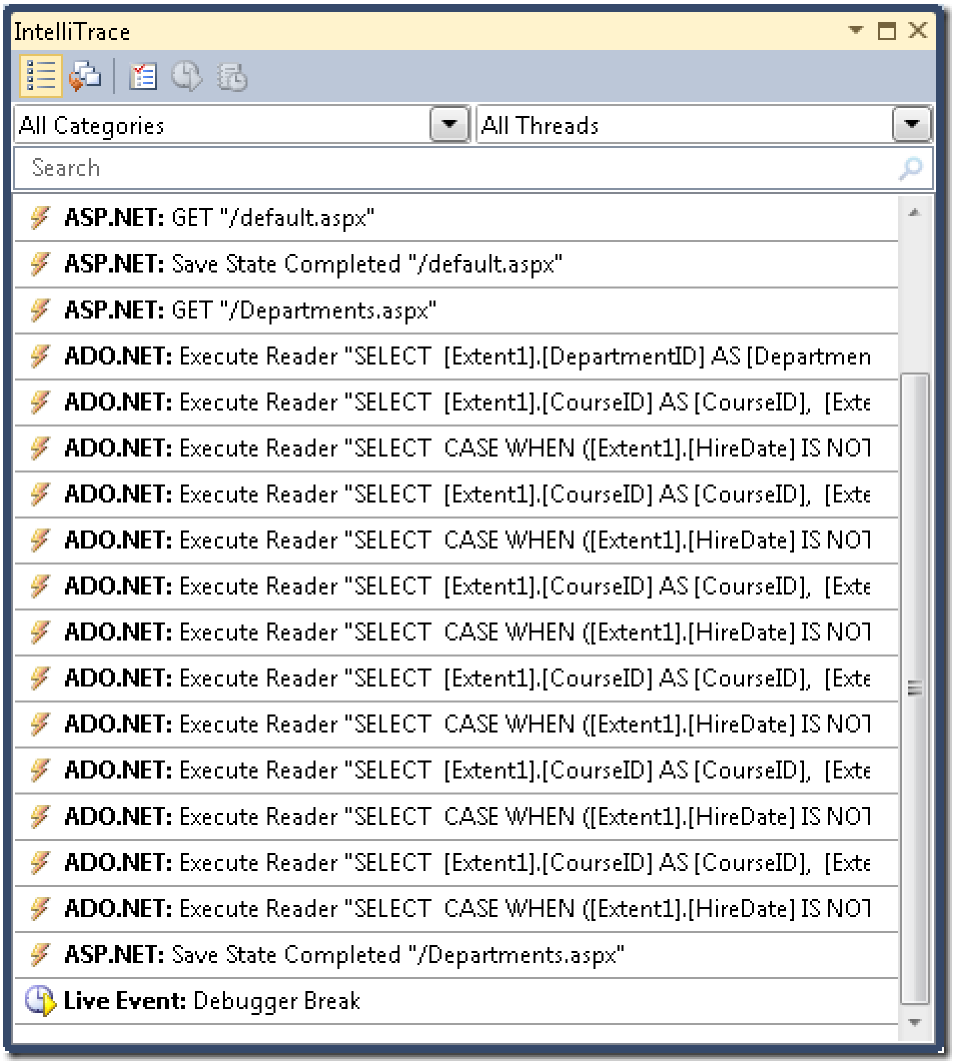
Suppose you were working with a database with more tables, relationships, and columns than the simple `School` database. You might find that a query that gathers all the information you need in a single

`Select`statement containing multiple `Join` clauses becomes too complex to work efficiently. In that case you can switch from eager loading to explicit loading to simplify the query.

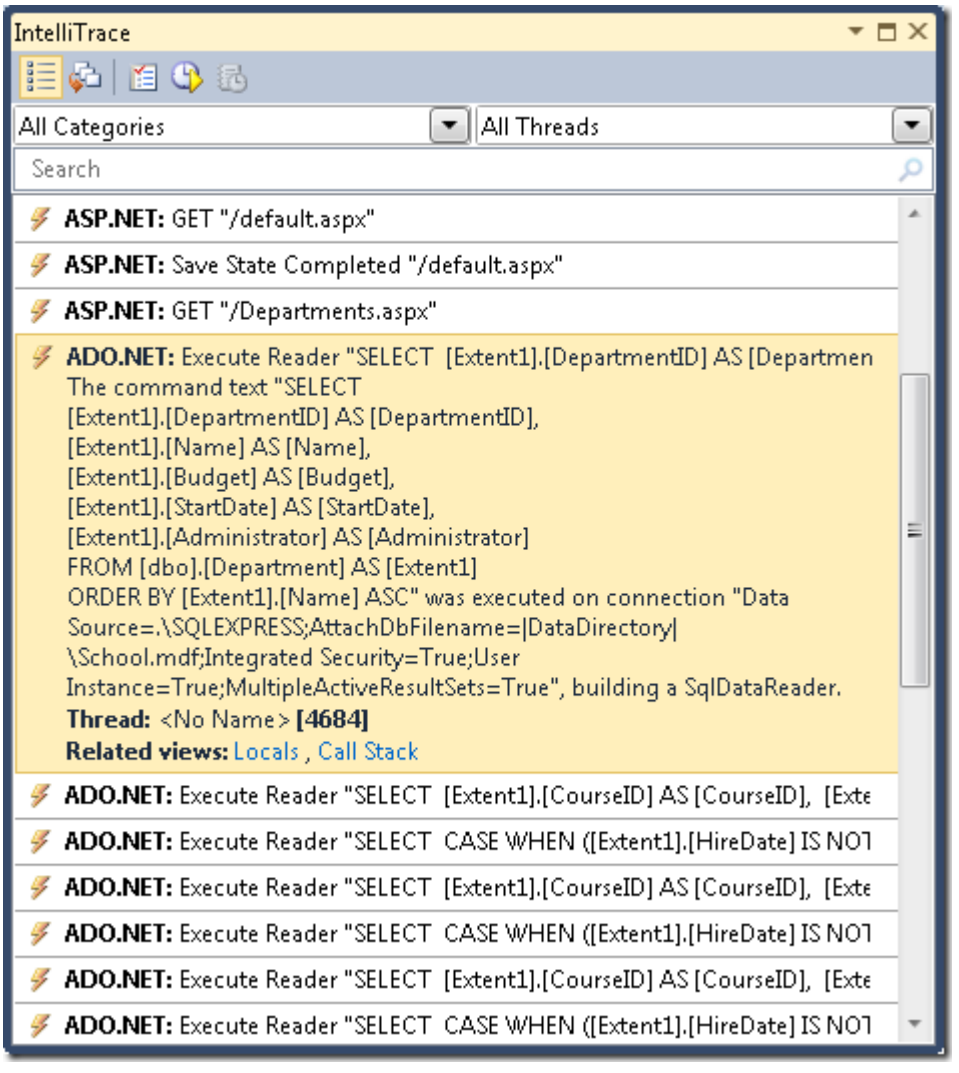
For example, try changing the code in the `GetDepartmentsByName` method in *SchoolRepository.cs*. Currently in that method you have an object query that has `Include` methods for the `Person` and `Courses` navigation properties. Replace the `return` statement with code that performs explicit loading, as shown in the following example:

```
publicIEnumerable<Department>GetDepartmentsByName(string sortExpression,string
nameSearchString)
{
...
var departments =newObjectQuery<Department>("SELECT VALUE d FROM Departments AS d",
context).OrderBy("it."+ sortExpression).Where(d =>
d.Name.Contains(nameSearchString)).ToList();
foreach(Department d in departments)
{
                d.Courses.Load();
                d.PersonReference.Load();
}
return departments;
}
```

Run the *Departments.aspx* page in the debugger and check the **IntelliTrace** window again as you did before. Now, where there was a single query before, you see a long sequence of them.



Click the first **ADO.NET** line to see what has happened to the complex query you viewed earlier.



The query from Departments has become a simple `Select` query with no `Join` clause, but it's followed by separate queries that retrieve related courses and an administrator, using a set of two queries for each department returned by the original query.

Note If you leave lazy loading enabled, the pattern you see here, with the same query repeated many times, might result from lazy loading. A pattern that you typically want to avoid is lazy-loading related data for every row of the primary table. Unless you've verified that a single join query is too complex to be efficient, you'd typically be able to improve performance in such cases by changing the primary query to use eager loading.

Pre-Generating Views

When an `ObjectContext` object is first created in a new application domain, the Entity Framework generates a set of classes that it uses to access the database. These classes are called *views*, and if you have a very large data model, generating these views can delay the web site's response to the first request for a page after a new

application domain is initialized. You can reduce this first-request delay by creating the views at compile time rather than at run time.

Note If your application doesn't have an extremely large data model, or if it does have a large data model but you aren't concerned about a performance problem that affects only the very first page request after IIS is recycled, you can skip this section. View creation doesn't happen every time you instantiate an `ObjectContext` object, because the views are cached in the application domain. Therefore, unless you're frequently recycling your application in IIS, very few page requests would benefit from pre-generated views.

You can pre-generate views using the *EdmGen.exe* command-line tool or by using a *Text Template Transformation Toolkit* (T4) template. In this tutorial you'll use a T4 template.

In the *DAL* folder, add a file using the **Text Template** template (it's under the **General** node in the **Installed Templates** list), and name it *SchoolModel.Views.tt*. Replace the existing code in the file with the following code:

```
<#
/*******************************************************************************

Copyright (c) Microsoft Corporation. All rights reserved.

THIS CODE IS PROVIDED *AS IS* WITHOUT WARRANTY OF
ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY
IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR
PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.

*******************************************************************************/
#>

<#
    //
    // TITLE: T4 template to generate views for an EDMX file in a C# project
    //
    // DESCRIPTION:
    // This is a T4 template to generate views in C# for an EDMX file in C# projects.
    // The generated views are automatically compiled into the project's output
assembly.
    //
    // This template follows a simple file naming convention to determine the EDMX
```

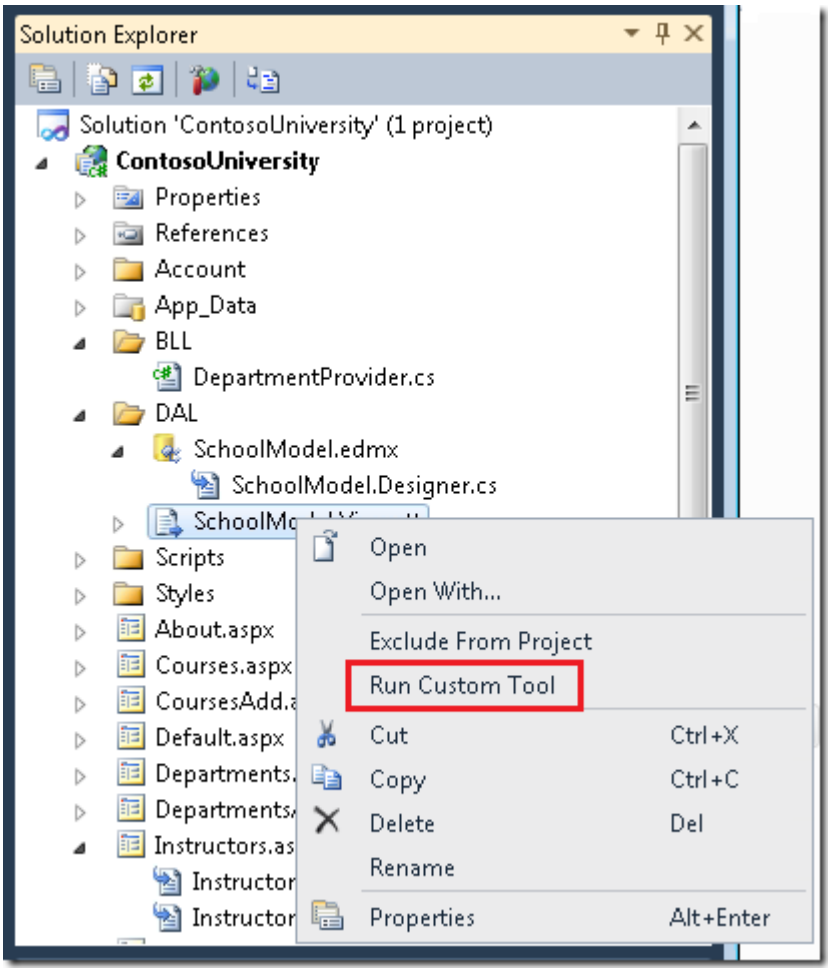
```
file to process:
    // - It assumes that [edmx-file-name].Views.tt will process and generate views
for [edmx-file-name].EDMX
    // - The views are generated in the code behind file [edmx-file-name].Views.cs
    //
    // USAGE:
    // Do the following to generate views for an EDMX file (e.g. Model1.edmx) in a C#
project
    // 1. In Solution Explorer, right-click the project node and choose
"Add...Existing...Item" from the context menu
    // 2. Browse to and choose this .tt file to include it in the project
    // 3. Ensure this .tt file is in the same directory as the EDMX file to process
    // 4. In Solution Explorer, rename this .tt file to the form [edmx-file-
name].Views.tt (e.g. Model1.Views.tt)
    // 5. In Solution Explorer, right-click Model1.Views.tt and choose "Run Custom
Tool" to generate the views
    // 6. The views are generated in the code behind file Model1.Views.cs
    //
    // TIPS:
    // If you have multiple EDMX files in your project then make as many copies of
this .tt file and rename appropriately
    // to pair each with each EDMX file.
    //
    // To generate views for all EDMX files in the solution, click the "Transform All
Templates" button in the Solution Explorer toolbar
    // (its the rightmost button in the toolbar)
    //
#>
<#
    //
    // T4 template code follows
    //
#>
<#@ template language="C#" hostspecific="true"#>
<#@ include file="EF.Utility.CS.ttinclude"#>
<#@ output extension=".cs" #>
<#
    // Find EDMX file to process: Model1.Views.tt generates views for Model1.EDMX
```

```
    string edmxFileName =
Path.GetFileNameWithoutExtension(this.Host.TemplateFile).ToLowerInvariant().Replace("
.views", "") + ".edmx";
    string edmxFilePath = Path.Combine(Path.GetDirectoryName(this.Host.TemplateFile),
edmxFileName);
    if (File.Exists(edmxFilePath))
    {
        // Call helper class to generate pre-compiled views and write to output
        this.WriteLine(GenerateViews(edmxFilePath));
    }
    else
    {
        this.Error(String.Format("No views were generated. Cannot find file {0}.
Ensure the project has an EDMX file and the file name of the .tt file is of the form
[edmx-file-name].Views.tt", edmxFilePath));
    }

    // All done!
#>

<#+
    private String GenerateViews(string edmxFilePath)
    {
        MetadataLoader loader = new MetadataLoader(this);
        MetadataWorkspace workspace;
        if(!loader.TryLoadAllMetadata(edmxFilePath, out workspace))
        {
            this.Error("Error in the metadata");
            return String.Empty;
        }

        String generatedViews = String.Empty;
        try
        {
            using (StreamWriter writer = new StreamWriter(new MemoryStream()))
            {
                StorageMappingItemCollection mappingItems =
(StorageMappingItemCollection)workspace.GetItemCollection(DataSpace.CSSpace);
```

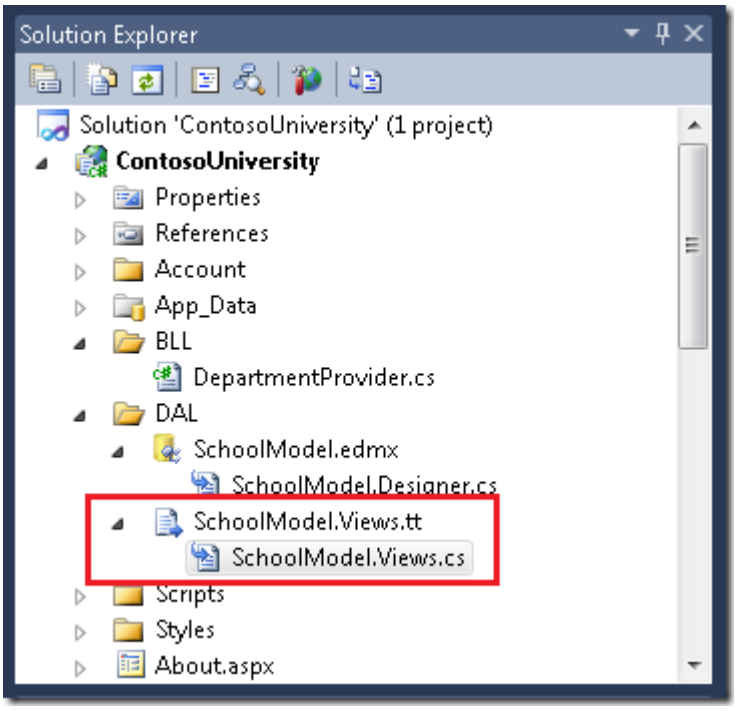
```
                // Initialize the view generator to generate views in C#
                EntityViewGenerator viewGenerator = new EntityViewGenerator();
                viewGenerator.LanguageOption = LanguageOption.GenerateCSharpCode;
                IList<EdmSchemaError> errors =
viewGenerator.GenerateViews(mappingItems, writer);

                foreach (EdmSchemaError e in errors)
                {
                    // log error
                    this.Error(e.Message);
                }

                MemoryStream memStream = writer.BaseStream as MemoryStream;
                generatedViews = Encoding.UTF8.GetString(memStream.ToArray());
            }
        }
        catch (Exception ex)
        {
            // log error
            this.Error(ex.ToString());
        }

        return generatedViews;
    }
#>
```

This code generates views for an *.edmx* file that's located in the same folder as the template and that has the same name as the template file. For example, if your template file is named *SchoolModel.Views.tt*, it will look for a data model file named *SchoolModel.edmx*.

Save the file, then right-click the file in **Solution Explorer** and select **Run Custom Tool**.



Visual Studio generates a code file that creates the views, which is named *SchoolModel.Views.cs* based on the template. (You might have noticed that the code file is generated even before you select **Run Custom Tool**, as soon as you save the template file.)



You can now run the application and verify that it works as it did before.

For more information about pre-generated views, see the following resources:

- How to: Pre-Generate Views to Improve Query Performance on the MSDN web site. Explains how to use the `EdmGen.exe` command-line tool to pre-generate views.
- Isolating Performance with Precompiled/Pre-generated Views in the Entity Framework 4 on the Windows Server AppFabric Customer Advisory Team blog.

This completes the introduction to improving performance in an ASP.NET web application that uses the Entity Framework. For more information, see the following resources:

- Performance Considerations (Entity Framework) on the MSDN web site.
- Performance-related posts on the Entity Framework Team blog.
- EF Merge Options and Compiled Queries. Blog post that explains unexpected behaviors of compiled queries and merge options such as `NoTracking`. If you plan to use compiled queries or manipulate merge option settings in your application, read this first.
- Entity Framework-related posts in the Data and Modeling Customer Advisory Team blog. Includes posts on compiled queries and using the Visual Studio 2010 Profiler to discover performance issues.
- Entity Framework forum thread with advice on improving performance of highly complex queries.
- ASP.NET State Management Recommendations.

- Using the Entity Framework and the ObjectDataSource: Custom Paging. Blog post that builds on the ContosoUniversity application created in these tutorials to explain how to implement paging in the *Departments.aspx* page.

The next tutorial reviews some of the important enhancements to the Entity Framework that are new in version 4.

What's New in the Entity Framework 4

By Tom Dykstra|January 26, 2011

This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you saw some methods for maximizing the performance of a web application that uses the Entity Framework. This tutorial reviews some of the most important new features in version 4 of the Entity Framework, and it links to resources that provide a more complete introduction to all of the new features. The features highlighted in this tutorial include the following:

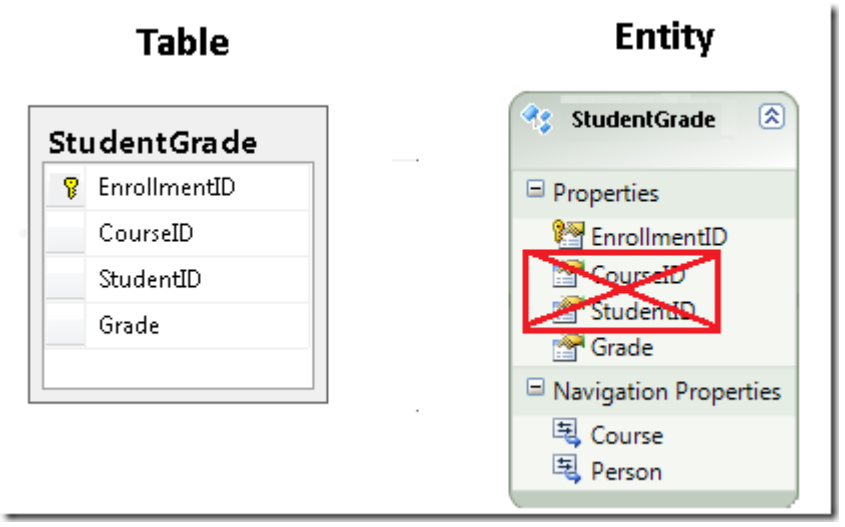
- Foreign-key associations.
- Executing user-defined SQL commands.
- Model-first development.
- POCO support.

In addition, the tutorial will briefly introduce *code-first development*, a feature that's coming in the next release of the Entity Framework.

To start the tutorial, start Visual Studio and open the Contoso University web application that you were working with in the previous tutorial.

Foreign-Key Associations

Version 3.5 of the Entity Framework included navigation properties, but it didn't include foreign-key properties in the data model. For example, the `CourseID` and `StudentID` columns of the `StudentGrade` table would be omitted from the `StudentGrade` entity.



The reason for this approach was that, strictly speaking, foreign keys are a physical implementation detail and don't belong in a conceptual data model. However, as a practical matter, it's often easier to work with entities in code when you have direct access to the foreign keys.

For an example of how foreign keys in the data model can simplify your code, consider how you would have had to code the *DepartmentsAdd.aspx* page without them. In the `Department` entity, the `Administrator` property is a foreign key that corresponds to `PersonID` in the `Person` entity. In order to establish the association between a new department and its administrator, all you had to do was set the value for the `Administrator` property in the `ItemInserting` event handler of the databound control:

```
protectedvoidDepartmentsDetailsView_ItemInserting(object
sender,DetailsViewInsertEventArgs e)
{
            e.Values["Administrator"]= administratorsDropDownList.SelectedValue;
}
```

Without foreign keys in the data model, you'd handle the `Inserting` event of the data source control instead of the `ItemInserting` event of the databound control, in order to get a reference to the entity itself before the entity is added to the entity set. When you have that reference, you establish the association using code like that in the following examples:

```
departmentEntityToBeInserted.PersonReference.EntityKey=newSystem.Data.EntityKey("Scho
olEntities.Departments","PersonID",Convert.ToInt32(administratorsDropDownList.Selecte
dValue));
```

```
            departmentEntityToBeInserted.Person= context.People.Single(p =>
p.PersonID==Convert.ToInt32(administratorsDropDownList.SelectedValue));
```

As you can see in the Entity Framework team's blog post on Foreign Key associations, there are other cases where the difference in code complexity is much greater. To meet the needs of those who prefer to live with implementation details in the conceptual data model for the sake of simpler code, the Entity Framework now gives you the option of including foreign keys in the data model.

In Entity Framework terminology, if you include foreign keys in the data model you're using *foreign key associations*, and if you exclude foreign keys you're using *independent associations*.

Executing User-Defined SQL Commands

In earlier versions of the Entity Framework, there was no easy way to create your own SQL commands on the fly and run them. Either the Entity Framework dynamically generated SQL commands for you, or you had to create a stored procedure and import it as a function. Version 4 adds `ExecuteStoreQuery` and `ExecuteStoreCommand` methods the `ObjectContext` class that make it easier for you to pass any query directly to the database.

Suppose Contoso University administrators want to be able to perform bulk changes in the database without having to go through the process of creating a stored procedure and importing it into the data model. Their first request is for a page that lets them change the number of credits for all courses in the database. On the web page, they want to be able to enter a number to use to multiply the value of every `Course` row's `Credits` column.

Create a new page that uses the *Site.Master* master page and name it *UpdateCredits.aspx*. Then add the following markup to the `Content` control named `Content2`:

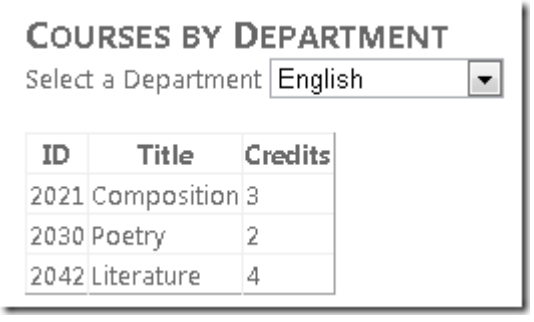
```
<h2>Update Credits</h2>
    Enter the number to multiply the current number of credits by:
<asp:TextBoxID="CreditsMultiplierTextBox"runat="server"></asp:TextBox>
<br/><br/>
<asp:ButtonID="ExecuteButton"runat="server"Text="Execute"OnClick="ExecuteButton_Click
"/><br/><br/>
    Rows affected:
<asp:LabelID="RowsAffectedLabel"runat="server"Text="0"ViewStateMode="Disabled"></asp:
Label><br/><br/>
```

This markup creates a `TextBox` control in which the user can enter the multiplier value, a `Button` control to click in order to execute the command, and a `Label` control for indicating the number of rows affected.

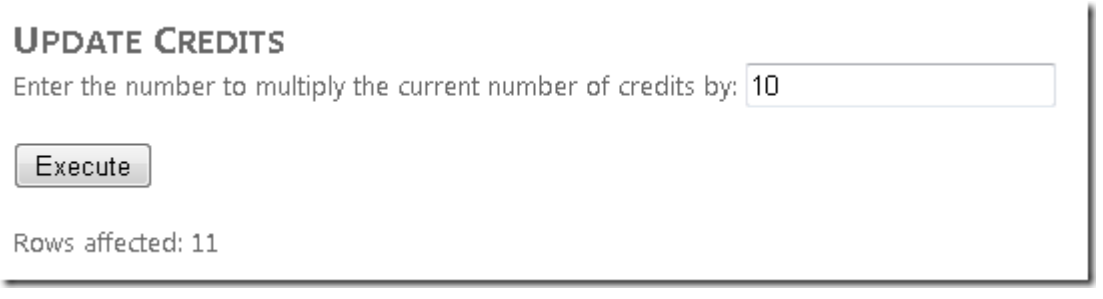
Open *UpdateCredits.aspx.cs*, and add the following `using` statement and a handler for the button's `Click` event:

```
usingContosoUniversity.DAL;


protectedvoidExecuteButton_Click(object sender,EventArgs e)
{
using(SchoolEntities context =newSchoolEntities())
{
RowsAffectedLabel.Text= context.ExecuteStoreCommand("UPDATE Course SET Credits =
Credits * {0}",CreditsMultiplierTextBox.Text).ToString();
}
}
```

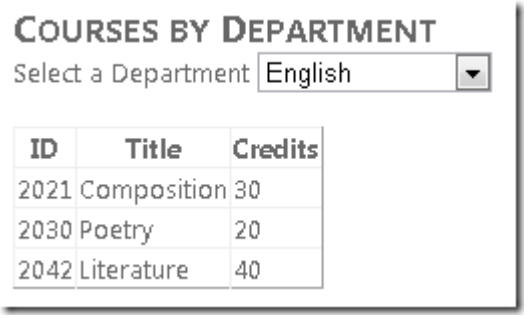
This code executes the SQL `Update` command using the value in the text box and uses the label to display the number of rows affected. Before you run the page, run the *Courses.aspx* page to get a "before" picture of some data.



Run *UpdateCredits.aspx*, enter "10" as the multiplier, and then click **Execute**.



Run the *Courses.aspx* page again to see the changed data.



(If you want to set the number of credits back to their original values, in *UpdateCredits.aspx.cs* change `Credits * {0}` to `Credits / {0}` and re-run the page, entering 10 as the divisor.)

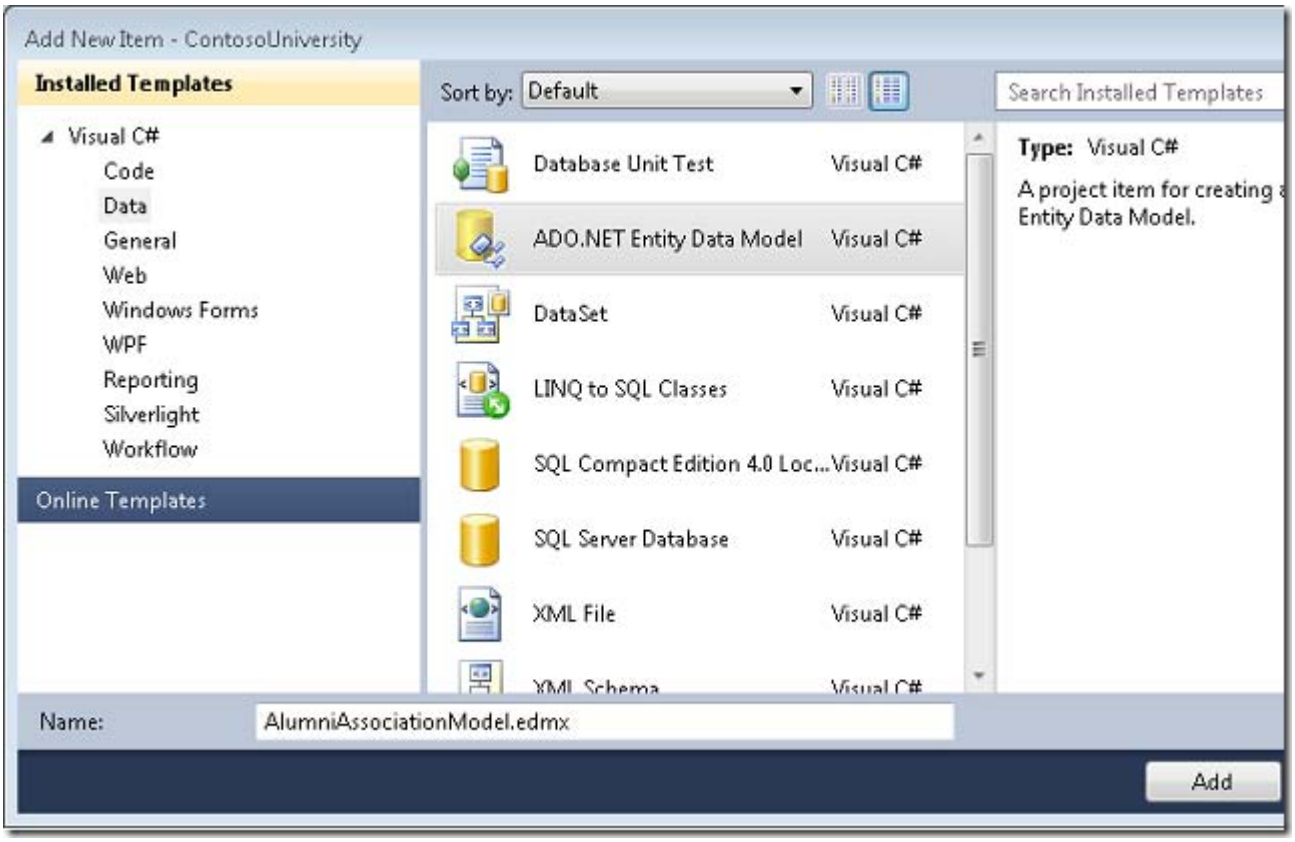
For more information about executing queries that you define in code, see How to: Directly Execute Commands Against the Data Source.

Model-First Development

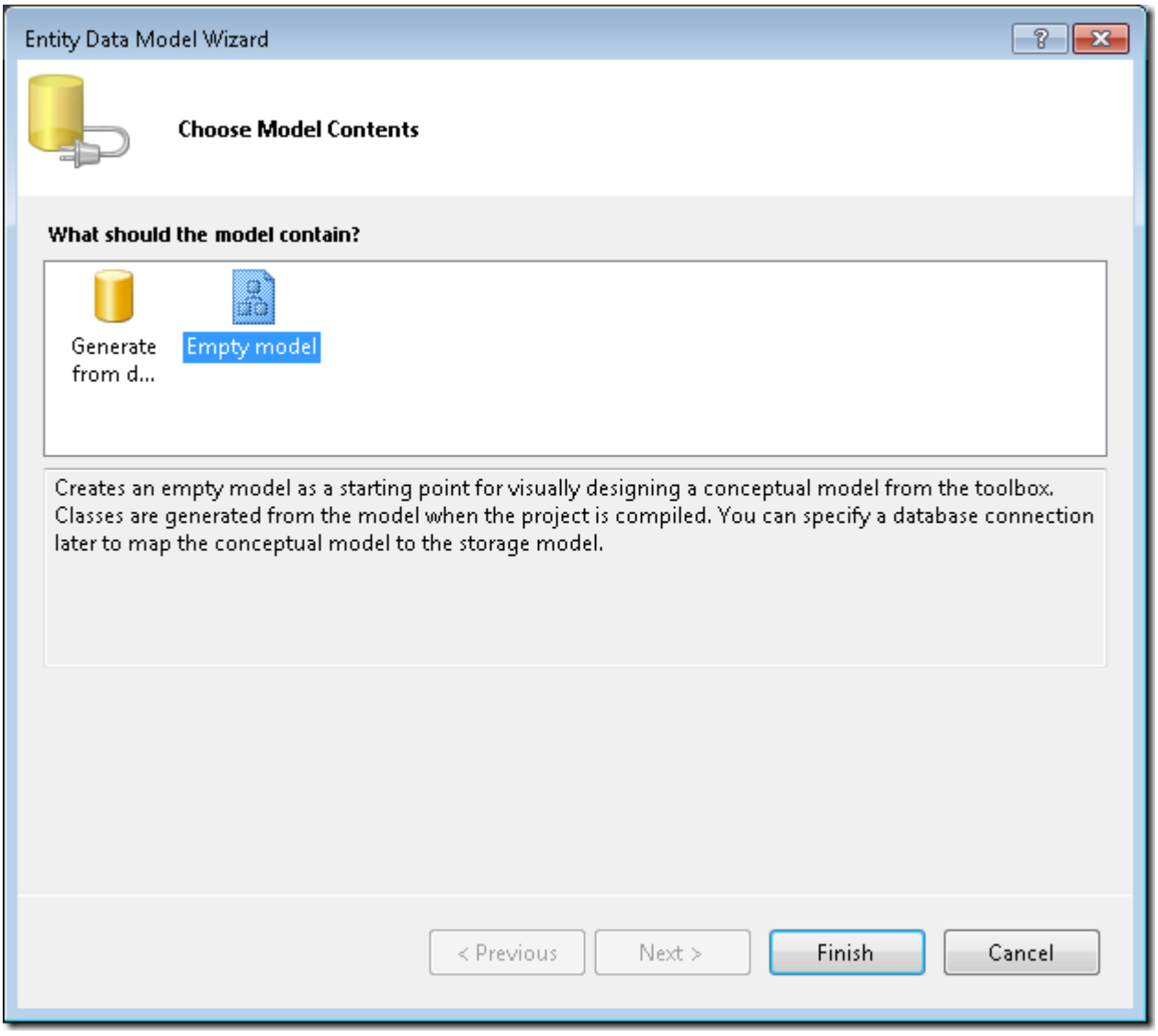
In these walkthroughs you created the database first and then generated the data model based on the database structure. In the Entity Framework 4 you can start with the data model instead and generate the database based on the data model structure. If you're creating an application for which the database doesn't already exist, the model-first approach enables you to create entities and relationships that make sense conceptually for the application, while not worrying about physical implementation details. (This remains true only through the initial stages of development, however. Eventually the database will be created and will have production data in it, and recreating it from the model will no longer be practical; at that point you'll be back to the database-first approach.)

In this section of the tutorial, you'll create a simple data model and generate the database from it.

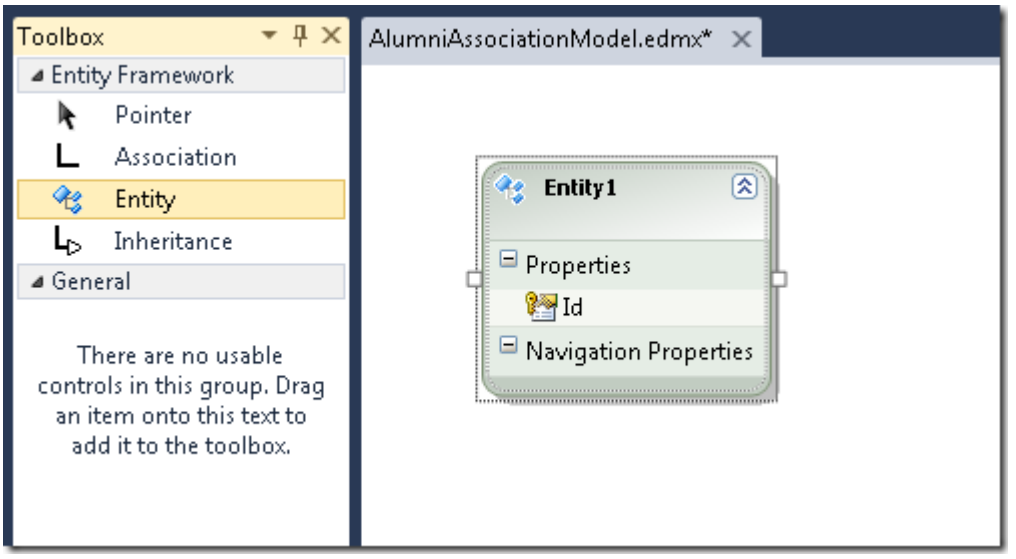
In **Solution Explorer**, right-click the *DAL* folder and select **Add New Item**. In the **Add New Item** dialog box, under **Installed Templates** select **Data** and then select the **ADO.NET Entity Data Model** template. Name the new file *AlumniAssociationModel.edmx* and click **Add**.



This launches the Entity Data Model Wizard. In the **Choose Model Contents** step, select **Empty Model** and then click **Finish**.

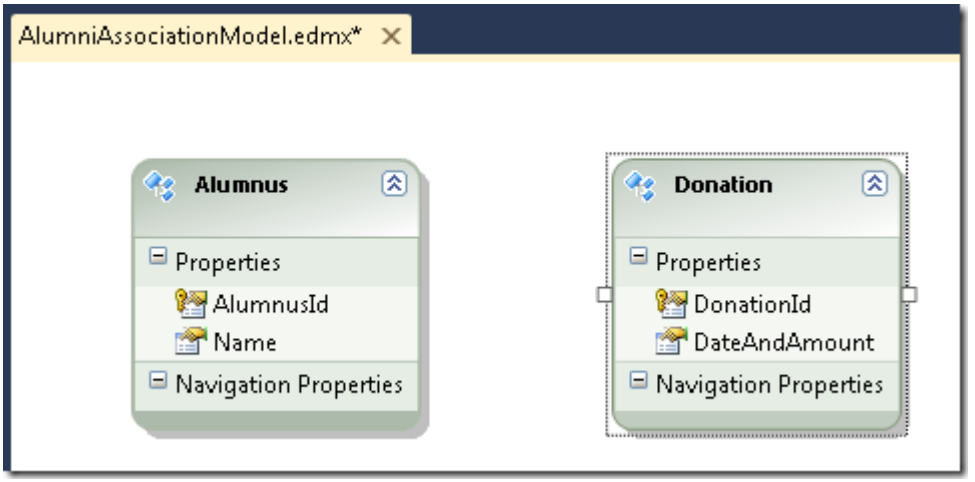


The **Entity Data Model Designer** opens with a blank design surface. Drag an **Entity** item from the **Toolbox** onto the design surface.

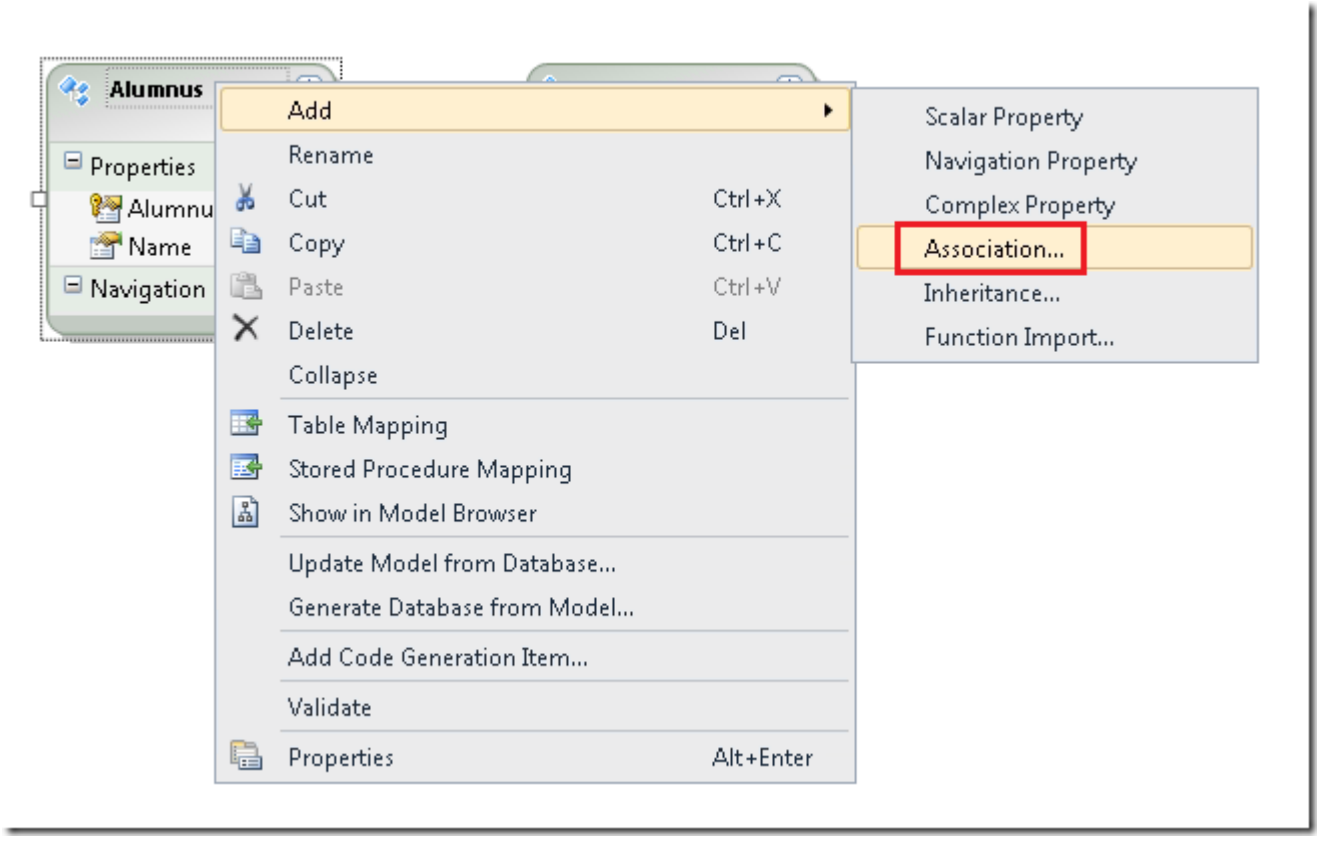


Change the entity name from `Entity1` to `Alumnus`, change the `Id` property name to `AlumnusId`, and add a new scalar property named `Name`. To add new properties you can press Enter after changing the name of the `Id` column, or right-click the entity and select **Add Scalar Property**. The default type for new properties is `String`, which is fine for this simple demonstration, but of course you can change things like data type in the **Properties** window.

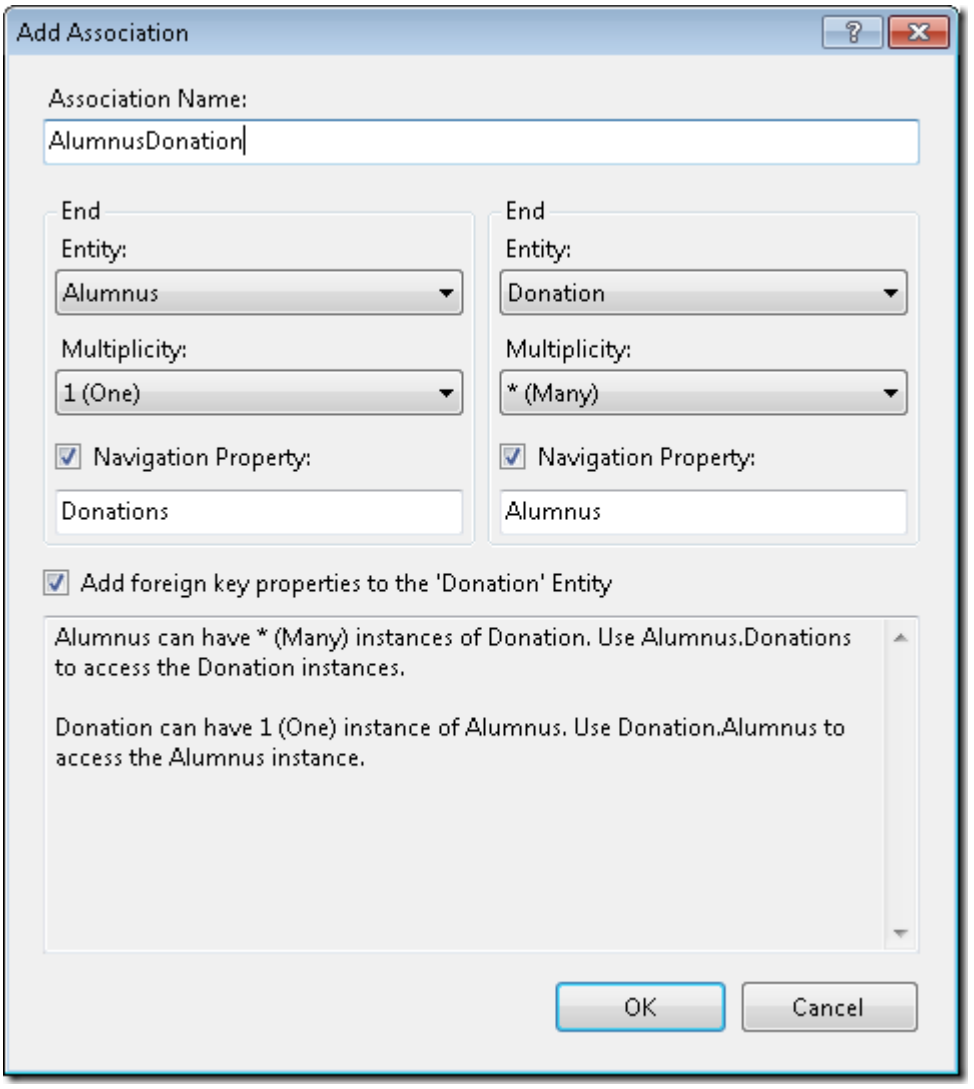
Create another entity the same way and name it `Donation`. Change the `Id` property to `DonationId` and add a scalar property named `DateAndAmount`.



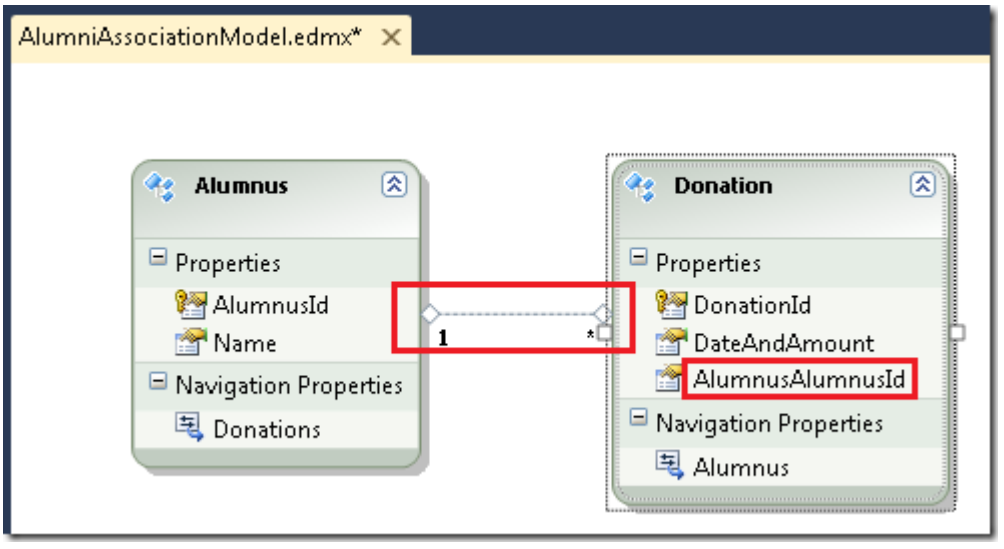
To add an association between these two entities, right-click the `Alumnus` entity, select **Add**, and then select **Association**.



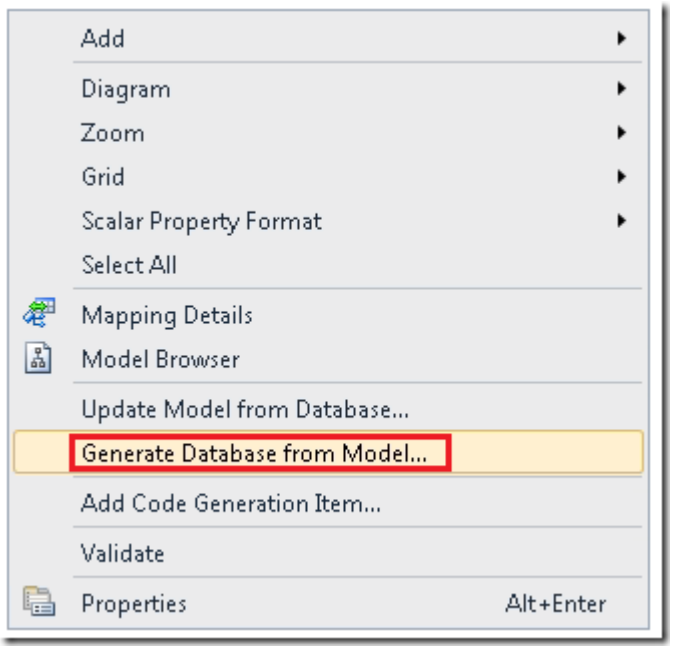
The default values in the **Add Association** dialog box are what you want (one-to-many, include navigation properties, include foreign keys), so just click **OK**.



The designer adds an association line and a foreign-key property.

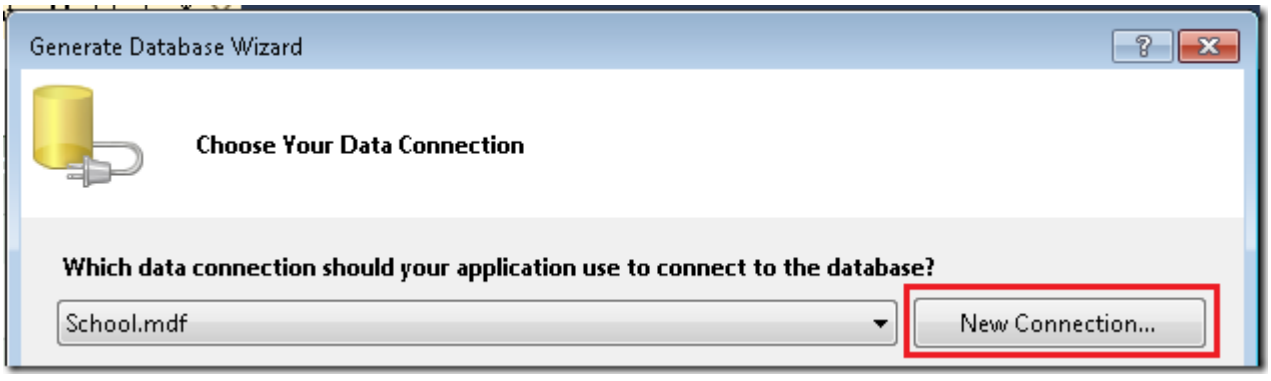


Now you're ready to create the database. Right-click the design surface and select **Generate Database from Model**.

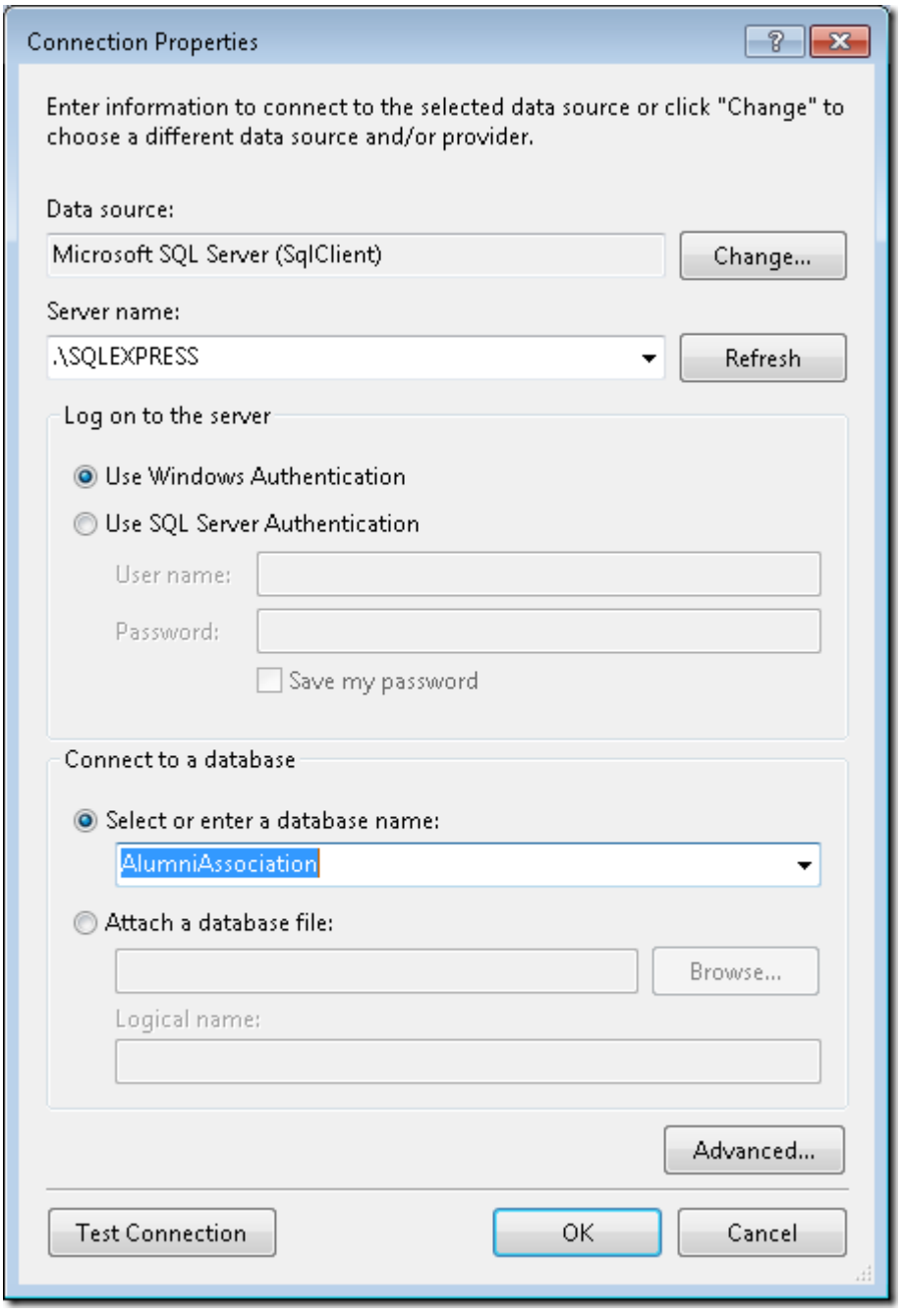


This launches the Generate Database Wizard. (If you see warnings that indicate that the entities aren't mapped, you can ignore those for the time being.)

In the **Choose Your Data Connection** step, click **New Connection**.

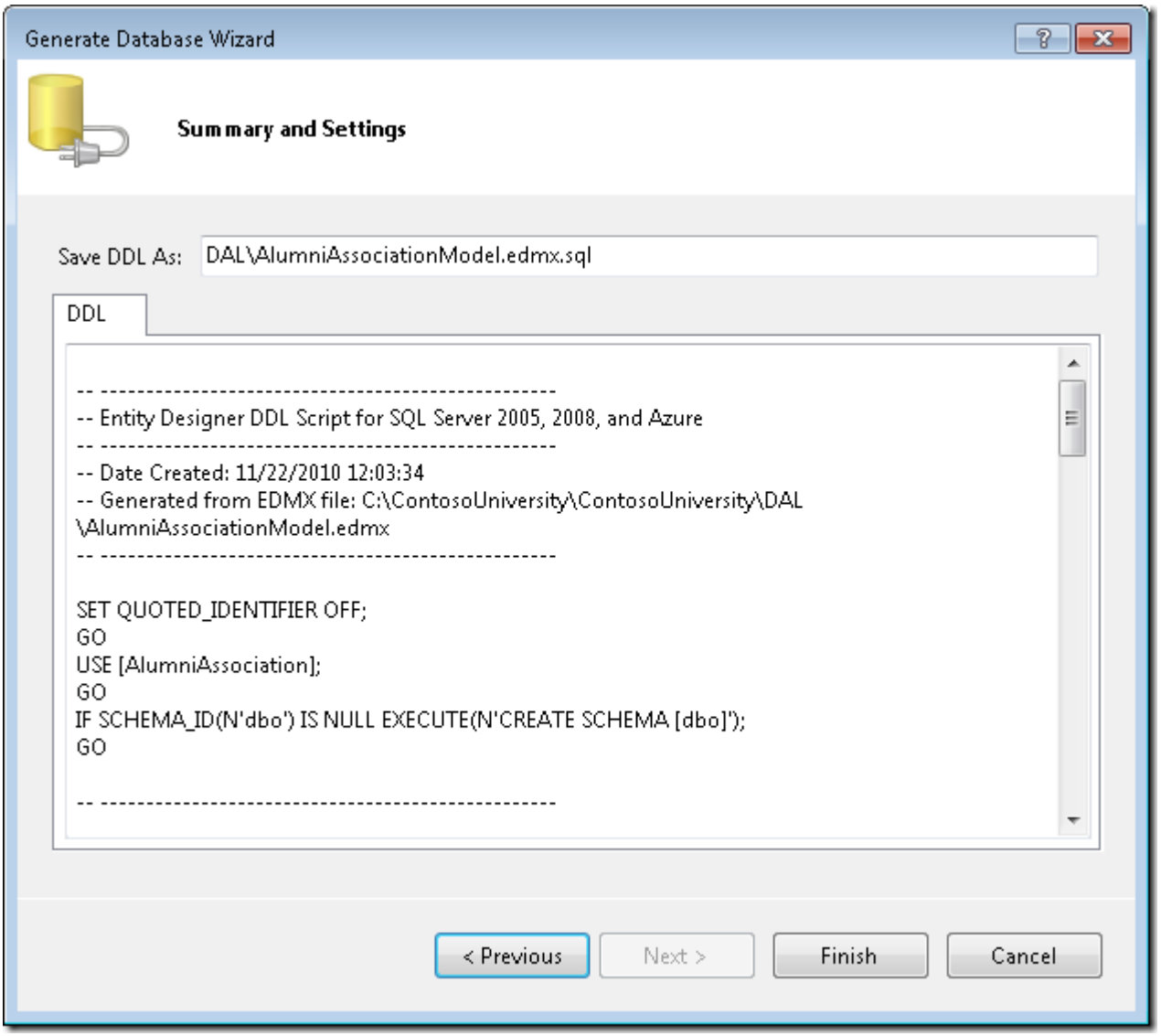


In the **Connection Properties** dialog box, select the local SQL Server Express instance and name the database `AlumniAsssociation`.



Click **Yes** when you're asked if you want to create the database. When the **Choose Your Data Connection** step is displayed again, click **Next**.

In the **Summary and Settings** step, click **Finish**.



A *.sql* file with the data definition language (DDL) commands is created, but the commands haven't been run yet.

AlumniAssociation...ql - not connected | AlumniAssociationModel.edmx*

```
-- --------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, and Azure
-- --------------------------------------------------
-- Date Created: 11/22/2010 12:03:34
-- Generated from EDMX file: C:\ContosoUniversity\ContosoUniversity\DAL\AlumniAssociat
-- --------------------------------------------------

SET QUOTED_IDENTIFIER OFF;
GO
USE [AlumniAssociation];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- --------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- --------------------------------------------------

IF OBJECT_ID(N'[dbo].[FK_AlumnusDonation]', 'F') IS NOT NULL
    ALTER TABLE [dbo].[Donations] DROP CONSTRAINT [FK_AlumnusDonation];
GO

-- --------------------------------------------------
-- Dropping existing tables
-- --------------------------------------------------
```

Use a tool such as **SQL Server Management Studio** to run the script and create the tables, as you might have done when you created the `School` database for the first tutorial. (Unless you downloaded the database.)

You can now use the `AlumniAssociation` data model in your web pages the same way you've been using the `School` model. To try this out, add some data to the tables and create a web page that displays the data.

Using **Server Explorer**, add the following rows to the `Alumnus` and `Donation` tables.

AlumnusId	Name
1	Roger Zheng
2	Candace Kapoor

DonationId	DateAndAmo...	AlumnusAlumnusId
1	1/1/2010 $100	1
2	1/2/2010 $200	1
3	1/3/2010 $300	2

Create a new web page named *Alumni.aspx* that uses the *Site.Master* master page. Add the following markup to the `Content` control named `Content2`:

```
<h2>Alumni</h2>
<asp:EntityDataSourceID="AlumniEntityDataSource"runat="server"
ContextTypeName="ContosoUniversity.DAL.AlumniAssociationModelContainer"EnableFlatteni
ng="False"
EntitySetName="Alumni">
</asp:EntityDataSource>
<asp:GridViewID="AlumniGridView"runat="server"
DataSourceID="AlumniEntityDataSource"AutoGenerateColumns="False"
OnRowDataBound="AlumniGridView_RowDataBound"
DataKeyNames="AlumnusId">
<Columns>
<asp:BoundFieldDataField="Name"HeaderText="Name"SortExpression="Name"/>
<asp:TemplateFieldHeaderText="Donations">
<ItemTemplate>
<asp:GridViewID="DonationsGridView"runat="server"AutoGenerateColumns="False">
<Columns>
<asp:BoundFieldDataField="DateAndAmount"HeaderText="Date and Amount"/>
</Columns>
</asp:GridView>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```

This markup creates nested `GridView` controls, the outer one to display alumni names and the inner one to display donation dates and amounts.

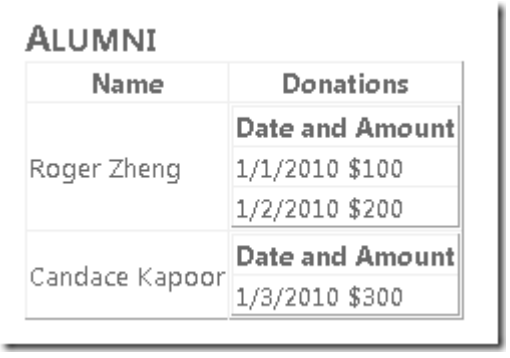
Open *Alumni.aspx.cs*. Add a `using` statement for the data access layer and a handler for the outer `GridView` control's `RowDataBound` event:

```
usingContosoUniversity.DAL;

// ...

protectedvoidAlumniGridView_RowDataBound(object sender,GridViewRowEventArgs e)
{
if(e.Row.RowType==DataControlRowType.DataRow)
```

```
{
var alumnus = e.Row.DataItemasAlumnus;
var donationsGridView =(GridView)e.Row.FindControl("DonationsGridView");
            donationsGridView.DataSource= alumnus.Donations.ToList();
            donationsGridView.DataBind();
}
}
```

This code databinds the inner `GridView` control using the `Donations` navigation property of the current row's `Alumnus` entity.

Run the page.



(Note: This page is included in the downloadable project, but to make it work you must create the database in your local SQL Server Express instance; the database isn't included as an *.mdf* file in the *App_Data* folder.)

For more information about using the model-first feature of the Entity Framework, see Model-First in the Entity Framework 4.

POCO Support

When you use domain-driven design methodology, you design data classes that represent data and behavior that's relevant to the business domain. These classes should be independent of any specific technology used to store (persist) the data; in other words, they should be *persistence ignorant*. Persistence ignorance can also make a class easier to unit test because the unit test project can use whatever persistence technology is most convenient for testing. Earlier versions of the Entity Framework offered limited support for persistence ignorance because entity classes had to inherit from the `EntityObject` class and thus included a great deal of Entity Framework-specific functionality.

The Entity Framework 4 introduces the ability to use entity classes that don't inherit from the `EntityObject` class and therefore are persistence ignorant. In the context of the Entity Framework, classes like this are typically called *plain-old CLR objects* (POCO, or POCOs). You can write POCO classes manually, or you can automatically generate them based on an existing data model using Text Template Transformation Toolkit (T4) templates provided by the Entity Framework.

For more information about using POCOs in the Entity Framework, see the following resources:

- Working with POCO Entities. This is an MSDN document that's an overview of POCOs, with links to other documents that have more detailed information.
- Walkthrough: POCO Template for the Entity Framework This is a blog post from the Entity Framework development team, with links to other blog posts about POCOs.

Code-First Development

POCO support in the Entity Framework 4 still requires that you create a data model and link your entity classes to the data model. The next release of the Entity Framework will include a feature called *code-first development*. This feature enables you to use the Entity Framework with your own POCO classes without needing to use either the data model designer or a data model XML file. (Therefore, this option has also been called *code-only*; *code-first* and *code-only* both refer to the same Entity Framework feature.)

For more information about using the code-first approach to development, see the following resources:

- Getting Started with Entity Framework Using MVC
- Code-First Development with Entity Framework 4. This is a blog post by Scott Guthrie introducing code-first development.
- Entity Framework Development Team Blog - posts tagged CodeOnly
- Entity Framework Development Team Blog - posts tagged Code First
- MVC Music Store tutorial - Part 4: Models and Data Access
- Getting Started with MVC 3 - Part 4: Entity Framework Code-First Development

More Information

This concludes this series of tutorials on Getting Started with the Entity Framework. For more resources to help you learn how to use the Entity Framework, continue with the first tutorial in the next Entity Framework tutorial series or visit the following sites:

- Entity Framework FAQ
- The Entity Framework Team Blog
- Entity Framework in the MSDN Library
- Entity Framework in the MSDN Data Developer Center
- EntityDataSource Web Server Control Overview in the MSDN Library
- EntityDataSource control API reference in the MSDN Library
- Entity Framework Forums on MSDN
- Julie Lerman's blog
- What's New in ADO.NET MSDN topic on new features in version 4 of the Entity Framework.
- Announcing the release of Entity Framework 4 The Entity Framework development team's blog post about new features in version 4.