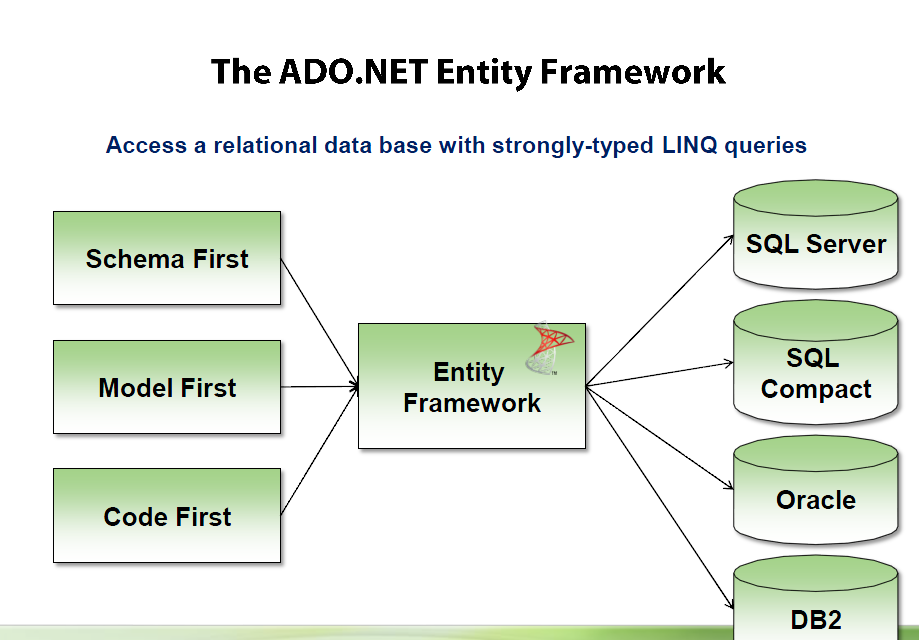
# **INTRODUCTION**

MVC stands for *model-view-controller*.  MVC is a pattern for developing applications that are well architected, testable  and easy to maintain. MVC-based applications contain:

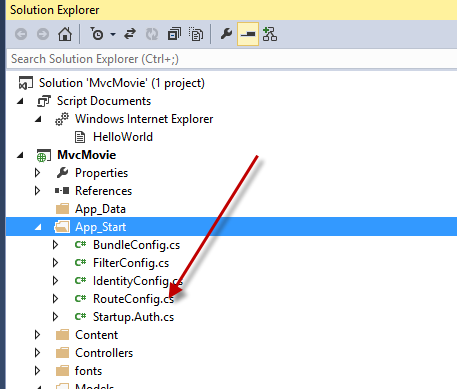
* **M**odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
* **V**iews: Template files that your application uses to dynamically generate HTML responses.
* **C**ontrollers: Classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.



ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

/[Controller]/[ActionName]/[Parameters]

You set the format for routing in the *App\_Start/RouteConfig.cs* file.



* The first part of the URL determines the controller class to execute.
* The second part of the URL determines the action method on the class to execute.  Index method was used by default. This is because a method named Index is the default method that will be called on a controller if one is not explicitly specified.
* The third part of the URL segment ( Parameters) is for route data.

n ASP.NET MVC applications, it's more typical to pass in parameters as route data ( http://localhost:xxx/HelloWorld/Welcome/3?name=Rick) than passing them as query strings.

You could also add a route to pass both the name and numtimes in  parameters as route data in the URL. In the App\_Start\RouteConfig.cs

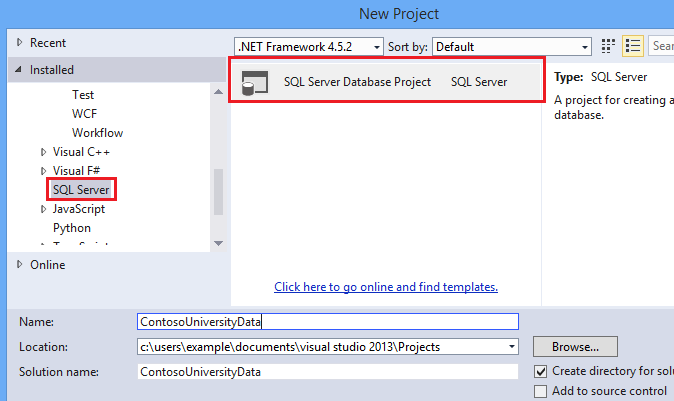
routes.MapRoute(

           name: "Hello",

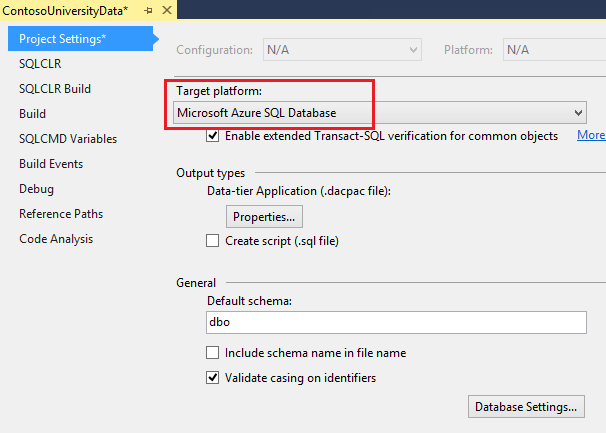
           url: "{controller}/{action}/{name}/{id}"

       );

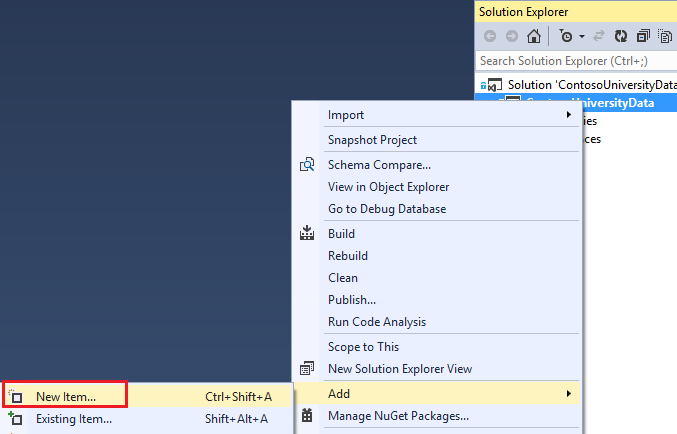
# **Entity Framework 6 Database First using MVC 5**

1. Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database.
2. The ASP.NET Scaffolding feature enables you to automatically generate code for displaying, updating, creating and deleting data. Using the publishing tools within Visual Studio, you can easily deploy the site and database to Azure.
3. 
4. You now have an empty database project. Setting the target platform does not actually deploy the database; it only means that the database project will verify that the database design is compatible with the target platform.

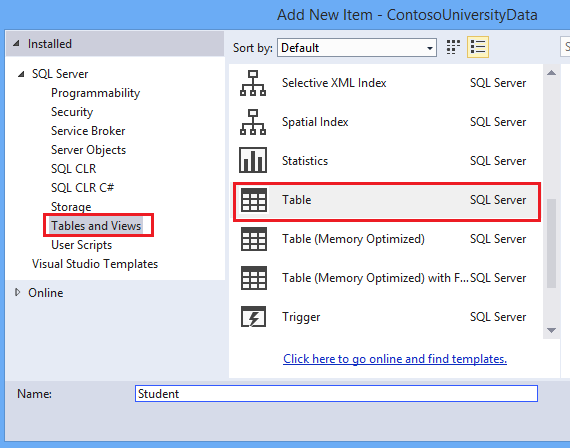
To set the target platform, open the **Properties** for the project and select **Microsoft Azure SQL Database** for the target platform.



You can create the tables needed for this tutorial by adding SQL scripts that define the tables. Right-click your project and add a new item.



Add a new table named Student.



In the table file, replace the T-SQL command with the following code to create the table.

CREATE TABLE [dbo].[Student] (

[StudentID] INT IDENTITY (1, 1) NOT NULL,

[LastName] NVARCHAR (50) NULL,

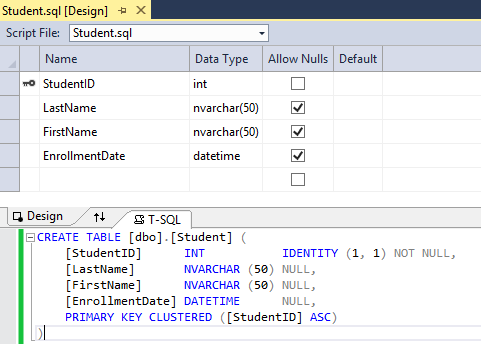
[FirstName] NVARCHAR (50) NULL,

[EnrollmentDate] DATETIME NULL,

PRIMARY KEY CLUSTERED ([StudentID] ASC)

)

Notice that the design window automatically synchronizes with the code. You can work with either the code or designer.



Add another table. This time name it Course and use the following T-SQL command.

CREATE TABLE [dbo].[Course] (

[CourseID] INT IDENTITY (1, 1) NOT NULL,

[Title] NVARCHAR (50) NULL,

[Credits] INT NULL,

PRIMARY KEY CLUSTERED ([CourseID] ASC)

)

And, repeat one more time to create a table named Enrollment.

CREATE TABLE [dbo].[Enrollment] (

[EnrollmentID] INT IDENTITY (1, 1) NOT NULL,

[Grade] DECIMAL(3, 2) NULL,

[CourseID] INT NOT NULL,

[StudentID] INT NOT NULL,

PRIMARY KEY CLUSTERED ([EnrollmentID] ASC),

CONSTRAINT [FK\_dbo.Enrollment\_dbo.Course\_CourseID] FOREIGN KEY ([CourseID])

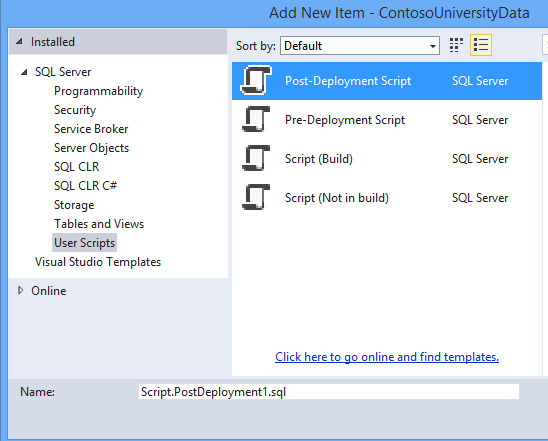
REFERENCES [dbo].[Course] ([CourseID]) ON DELETE CASCADE,

CONSTRAINT [FK\_dbo.Enrollment\_dbo.Student\_StudentID] FOREIGN KEY ([StudentID])

REFERENCES [dbo].[Student] ([StudentID]) ON DELETE CASCADE

)

You can populate your database with data through a script that is run after the database is deployed. Add a Post-Deployment Script to the project. You can use the default name.



Add the following T-SQL code to the post-deployment script. This script simply adds data to the database when no matching record is found. It does not overwrite or delete any data you may have entered into the database.

MERGE INTO Course AS Target

USING (VALUES

(1, 'Economics', 3),

(2, 'Literature', 3),

(3, 'Chemistry', 4)

)

AS Source (CourseID, Title, Credits)

ON Target.CourseID = Source.CourseID

WHEN NOT MATCHED BY TARGET THEN

INSERT (Title, Credits)

VALUES (Title, Credits);

MERGE INTO Student AS Target

USING (VALUES

(1, 'Tibbetts', 'Donnie', '2013-09-01'),

(2, 'Guzman', 'Liza', '2012-01-13'),

(3, 'Catlett', 'Phil', '2011-09-03')

)

AS Source (StudentID, LastName, FirstName, EnrollmentDate)

ON Target.StudentID = Source.StudentID

WHEN NOT MATCHED BY TARGET THEN

INSERT (LastName, FirstName, EnrollmentDate)

VALUES (LastName, FirstName, EnrollmentDate);

MERGE INTO Enrollment AS Target

USING (VALUES

(1, 2.00, 1, 1),

(2, 3.50, 1, 2),

(3, 4.00, 2, 3),

(4, 1.80, 2, 1),

(5, 3.20, 3, 1),

(6, 4.00, 3, 2)

)

AS Source (EnrollmentID, Grade, CourseID, StudentID)

ON Target.EnrollmentID = Source.EnrollmentID

WHEN NOT MATCHED BY TARGET THEN

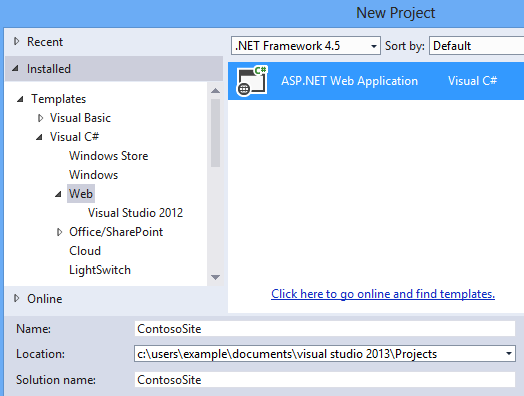
INSERT (Grade, CourseID, StudentID)

VALUES (Grade, CourseID, StudentID);

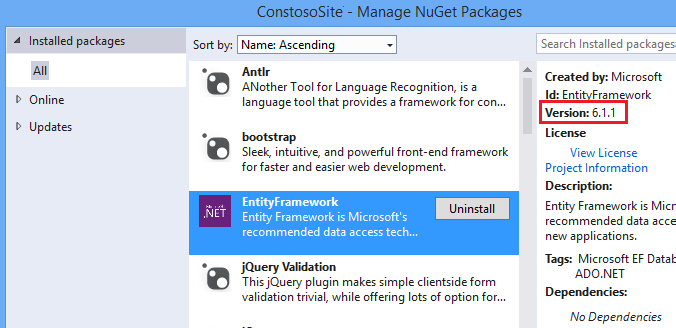
It is important to note that the post-deployment script is run every time you deploy your database project. Therefore, you need to carefully consider your requirements when writing this script

# **Create a new ASP.NET Web Application**

In either a new solution or the same solution as the database project, create a new project in Visual Studio and select the **ASP.NET Web Application** template. Name the project **ContosoSite**.



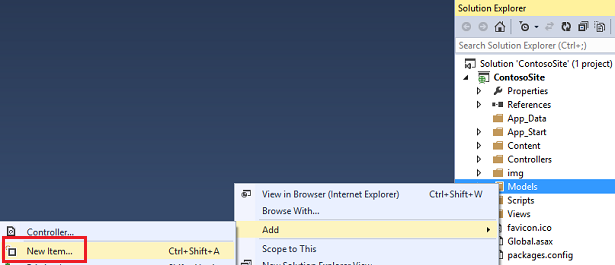
If necessary, update your version of Entity Framework.



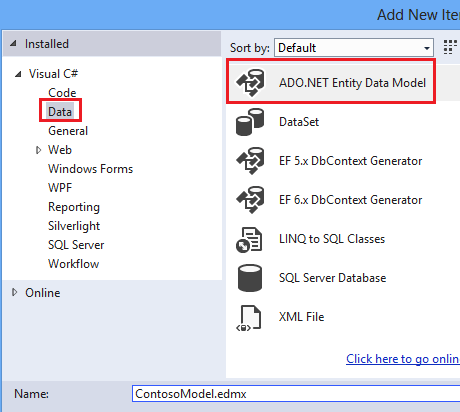
# **Generate the models**

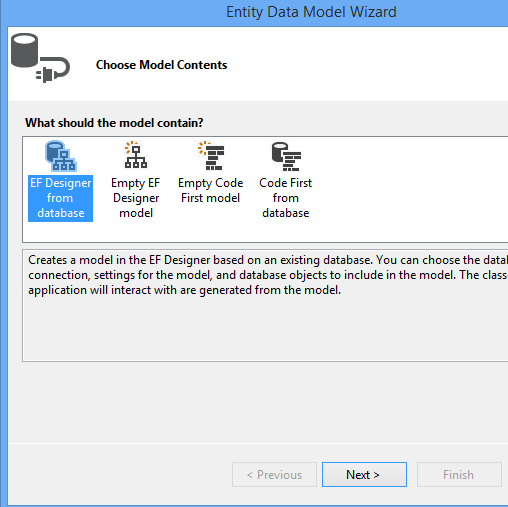
Create Entity Framework models from the database tables. These models are classes that you will use to work with the data. Each model mirrors a table in the database and contains properties that correspond to the columns in the table.

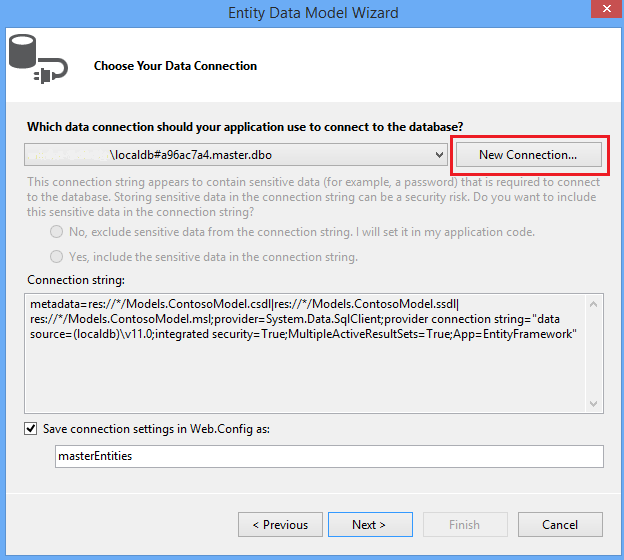
Right-click the **Models** folder, and select **Add** and **New Item**.

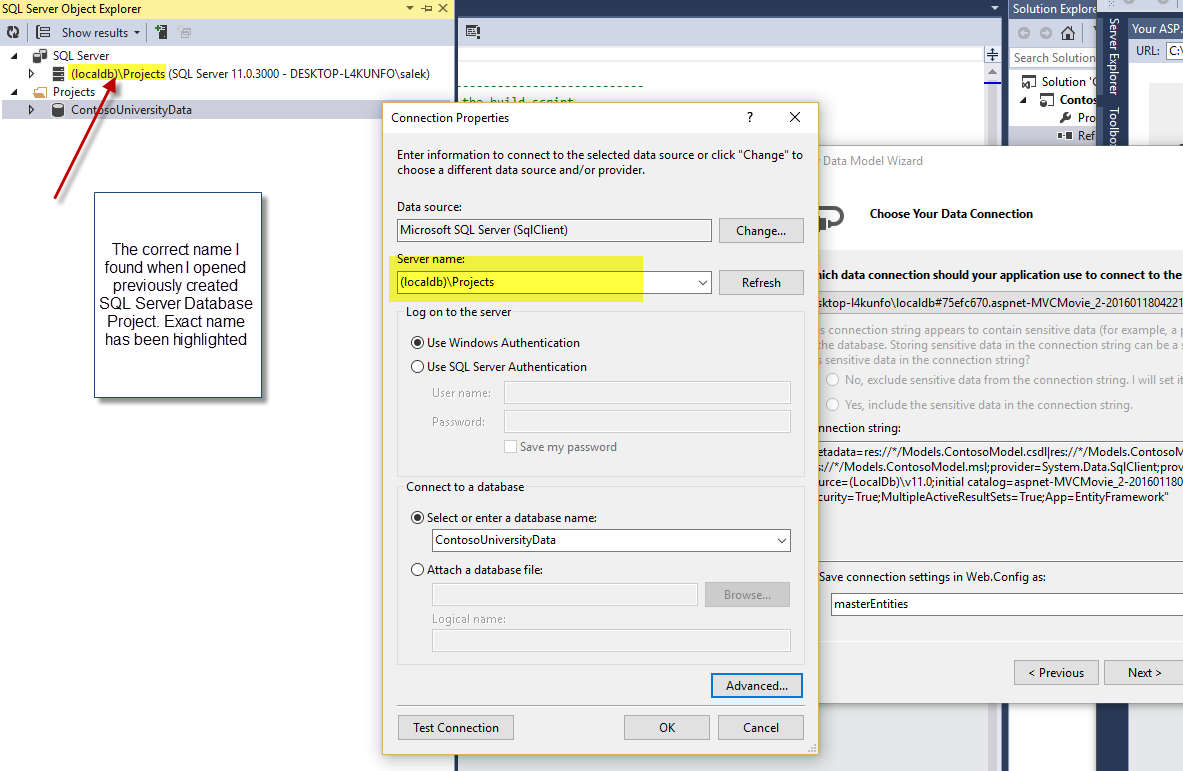


In the Add New Item window, select **Data** in the left pane and **ADO.NET Entity Data Model** from the options in the center pane. Name the new model file **ContosoModel**.

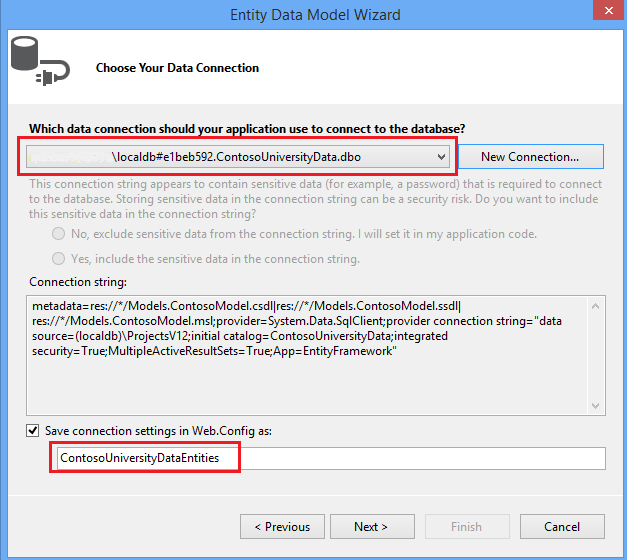






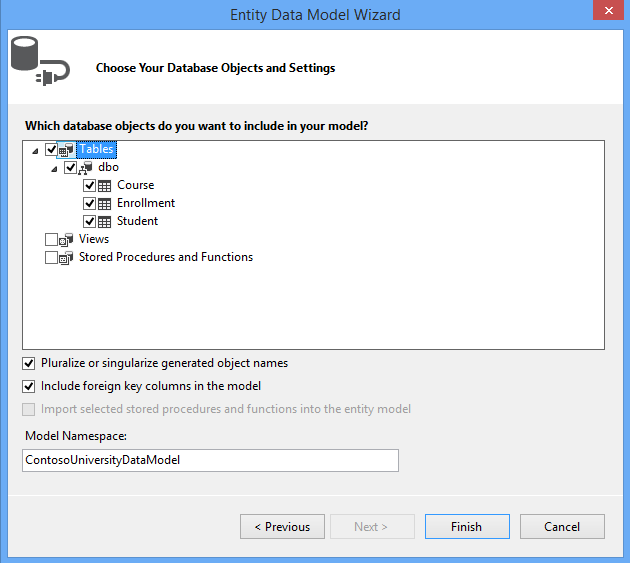


The correct connection properties are now displayed. You can use the default name for connection in the Web.Config file



Click **Next**.

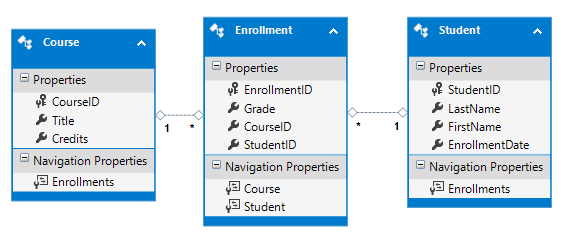
Select **Tables** to generate models for all three tables.



Click **Finish**.

If you receive a security warning, select **OK** to continue running the template.

The models are generated from the database tables, and a diagram is displayed that shows the properties and relationships between the tables.



The Models folder now includes many new files related to the models that were generated from the database.

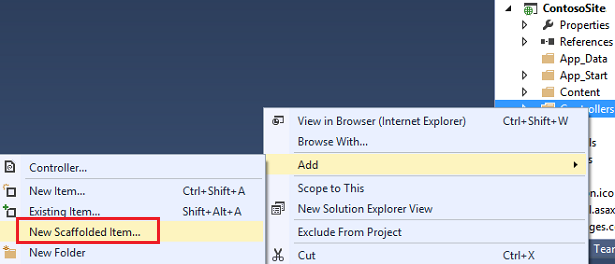
The **ContosoModel.Context.cs** file contains a class that derives from the **DbContext** class, and provides a property for each model class that corresponds to a database table. The **Course.cs**, **Enrollment.cs**, and **Student.cs** files contain the model classes that represent the databases tables.

# **Generating Views**

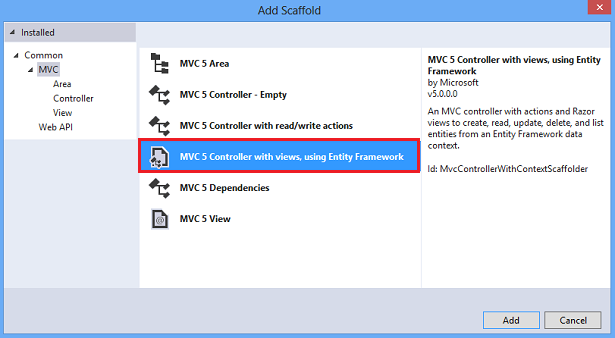
Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database.

## Add scaffold

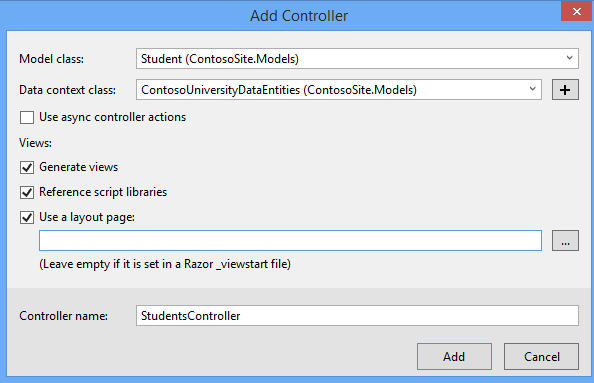
add the new controller to the existing **Controllers** folder. Right-click the **Controllers** folder, and select **Add** – **New Scaffolded Item**.



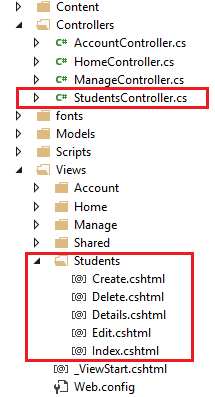
Select the **MVC 5 Controller with views, using Entity Framework** option. This option will generate the controller and views for updating, deleting, creating and displaying the data in your model.



Select **Student** for the model class, and select the **ContosoUniversityEntities** for the context class. Keep the controller name as**StudentsController**,



Click **Add**.



Perform the same steps again, but add a scaffold for the Enrollment class. When finished, you should have an **EnrollmentsController.cs** file, and a folder under **Views** named **Enrollments** with the Create, Delete, Details, Edit and Index views.

## Add links to new views

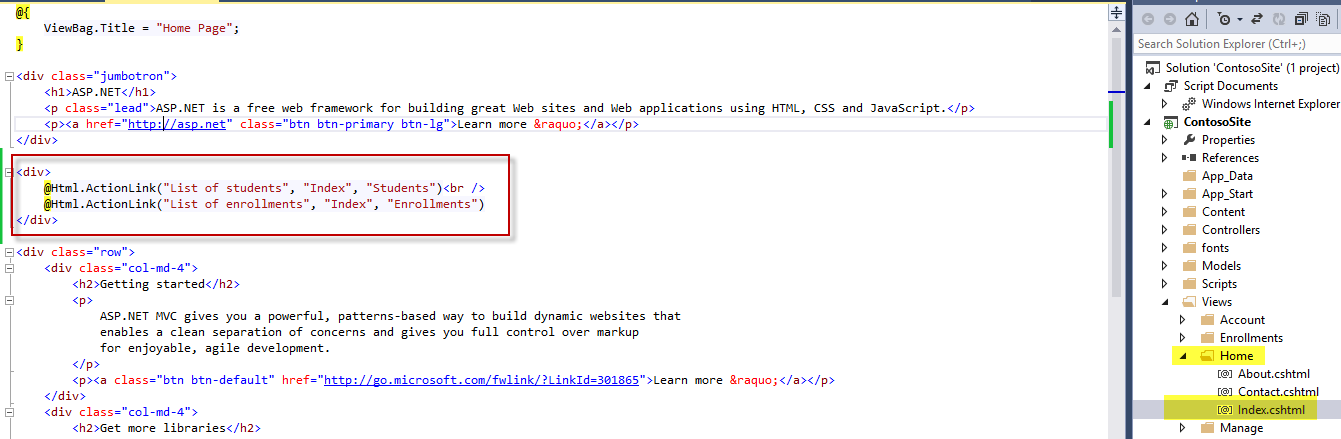
To make it easier for you to navigate to your new views, you can add a couple of hyperlinks to the Index views for students and enrollments. Open the file at **Views/Home/Index.cshtml**, which is the home page for your site. Add the following code below the jumbotron.

<div>

@Html.ActionLink("List of students", "Index", "Students")<br />

@Html.ActionLink("List of enrollments", "Index", "Enrollments")

</div>



first parameter is the text to display in the link.

The second parameter is the action and the third parameter is the name of the controller.

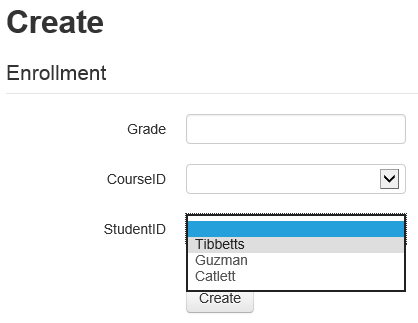
third parameter is the name of the controller

The actual hyperlink is constructed from these values. The first link ultimately takes users to the **Index.cshtml** file within the **Views/Students** folder.

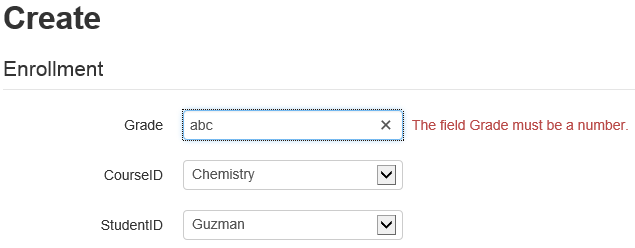
Run App. Without writing any code, you have added views that perform common operations on the data in the Student table.

## Display enrollment views

Your database includes a one-to-many relationship between the Student and Enrollment tables, and a one-to-many relationship between the Course and Enrollment tables. The views for Enrollment correctly handle these relationships. Navigate to the home page for your site and select the **List of enrollments** link and then the **Create New** link. The view displays a form for creating a new enrollment record. In particular, notice that the form contains two drop-down lists that are populated with values from the related tables.



Furthermore, validation of the provided values is automatically applied based on the data type of the field. Grade requires a number, so an error message is displayed if you try to provide an incompatible value.

 You have verified that the automatically-generated views enable users to work with the data in the database.

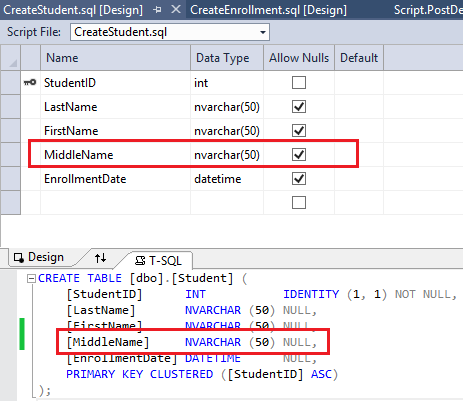
# **Changing the Database**

## Add a column

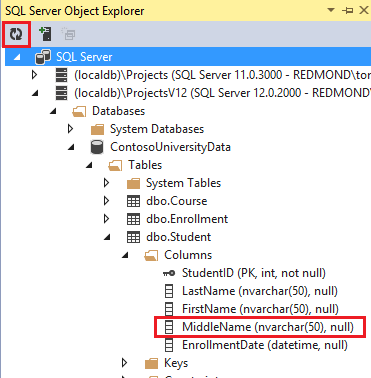
If you update the structure of a table in your database, you need to ensure that your change is propagated to the data model, views, and controller.

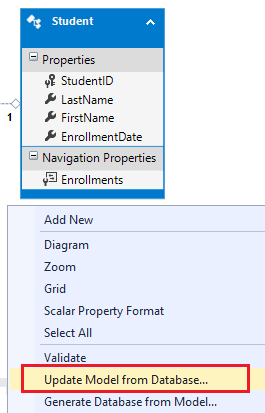
Add a new column to the Student table to record the middle name of the student.,

Open the database project, and open the Student.sql file. Through either the designer or the T-SQL code, add a column named **MiddleName** that is an NVARCHAR(50) and allows NULL values.



Deploy this change to your local database by starting your database project (or F5). The new field is added to the table. If you do not see it in the SQL Server Object Explorer, click the Refresh button in the pane.



The new column exists in the database table, but it does not currently exist in the data model class. You must update the model to include your new column. In the **Models** folder, open the **ContosoModel.edmx** file to display the model diagram. Notice that the Student model does not contain the MiddleName property. Right-click anywhere on the design surface, and select **Update Model from Database**. 

Unfortunately, the views still do not contain the new property. To update the views you have two options - you can either re-generate the views by once again adding scaffolding for the Student class, or you can - manually add the new property to your existing views.

This time we will add the scaffolding again because you have not made any customized changes to the automatically-generated views

* delete the **Students** folder under **Views**, and delete the **StudentsController**.
* Then, right-click the**Controllers** folder and add scaffolding for the **Student** model. Again, name the controller **StudentsController**. Select **OK**.

The views now contain the MiddleName property.

# **Customizing a View**

The generated code provides a good starting point for your application but it does not necessarily provide all of the functionality that you need in your application. You can customize the code to meet the particular requirements of your application. Currently, your application does not display the enrolled courses for the selected student. In this section, you will add the enrolled courses for each student to the **Details** view for the student.

Open **Students/Details.cshtml**, and below the last </dl> tab, but before the closing </div> tag, add the following code.

<table class="table">

<tr>

<th>

Course Title

</th>

<th>

Grade

</th>

<th>

Credits

</th>

</tr>

@foreach (var item in Model.Enrollments)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Course.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.Grade)

</td>

<td>

@Html.DisplayFor(modelItem => item.Course.Credits)

</td>

</tr>

}

</table>

This code creates a table that displays a row for each record in the Enrollment table for the selected student. The **Display** method renders HTML for the object (modelItem) that represents the expression. You use the Display method (rather than simply embedding the property value in the code) to make sure the value is formatted correctly based on its type and the template for that type. In this example, each expression returns a single property from the current record in the loop, and the values are primitive types which are rendered as text.

Browse to the Students/Index view again and select **Details** for one of the students. You will see the enrolled courses have been included in the view.

# **Enhancing Data Validation**

## **Add data annotations**

As you saw in an earlier topic, some data validation rules are automatically applied to the user input. For example, you can only provide a number for the Grade property**. To specify more data validation rules, you can add data annotations to your model class.** These annotations are applied throughout your web application for the specified property. You can also apply formatting attributes that change how the properties are displayed; such as, changing the value used for text labels.

We will Add data annotations to restrict the length of the values provided for the FirstName, LastName, and MiddleName properties. In the database, these values are limited to 50 characters; however, in your web application that character limit is currently not enforced. If a user provides more than 50 characters for one of those values, the page will crash when attempting to save the value to the database. You will also restrict Grade to values between 0 and 4.

Open the **Student.cs** file in the **Models** folder. Add the following highlighted code to the class.

namespace ContosoSite.Models

{

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

public partial class Student

{

public Student()

{

this.Enrollments = new HashSet<Enrollment>();

}

public int StudentID { get; set; }

[StringLength(50)]

public string LastName { get; set; }

[StringLength(50)]

public string FirstName { get; set; }

public Nullable<System.DateTime> EnrollmentDate { get; set; }

[StringLength(50)]

public string MiddleName { get; set; }

public virtual ICollection<Enrollment> Enrollments { get; set; }

}

}

In Enrollment.cs, add the following highlighted code.

namespace ContosoSite.Models

{

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

public partial class Enrollment

{

public int EnrollmentID { get; set; }

[Range(0, 4)]

public Nullable<decimal> Grade { get; set; }

public int CourseID { get; set; }

public int StudentID { get; set; }

public virtual Course Course { get; set; }

public virtual Student Student { get; set; }

}

}

Build the solution.

Browse to a page for editing or creating a student. If you attempt to enter more than 50 characters, an error message is displayed.

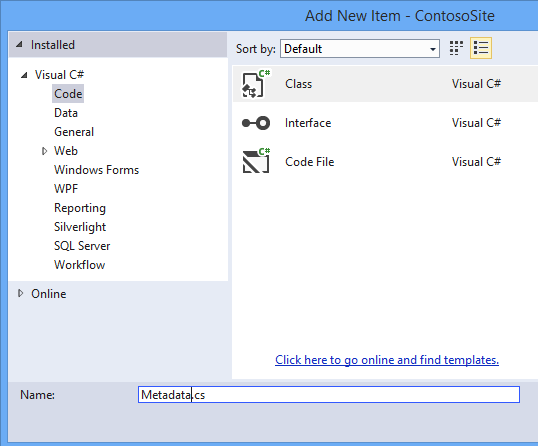
For a full list of data validation annotations you can apply to properties and classes, see [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx).

## Add metadata classes

Adding the validation attributes directly to the model class works when you do not expect the database to change; however, if your database changes and you need to regenerate the model class, you will lose all of the attributes you had applied to the model class. This approach can be very inefficient and prone to losing important validation rules.

To avoid this problem, you can add a metadata class that contains the attributes. When you associate the model class to the metadata class, those attributes are applied to the model. In this approach, the model class can be regenerated without losing all of the attributes that have been applied to the metadata class.

In the **Models** folder, add a class named **Metadata.cs**.



Replace the code in Metadata.cs with the following code.

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models

{

public class StudentMetadata

{

[StringLength(50)]

[Display(Name="Last Name")]

public string LastName;

[StringLength(50)]

[Display(Name="First Name")]

public string FirstName;

[StringLength(50)]

[Display(Name="Middle Name")]

public string MiddleName;

[Display(Name = "Enrollment Date")]

public Nullable<System.DateTime> EnrollmentDate;

}

public class EnrollmentMetadata

{

[Range(0, 4)]

public Nullable<decimal> Grade;

}

}

These metadata classes contain all of the validation attributes that you had previously applied to the model classes. The **Display** attribute is used to change the value used for text labels.

Now, you must associate the model classes with the metadata classes.

In the **Models** folder, add a class named **PartialClasses.cs**.

Replace the contents of the file with the following code.

using System;

using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models

{

[MetadataType(typeof(StudentMetadata))]

public partial class Student

{

}

[MetadataType(typeof(EnrollmentMetadata))]

public partial class Enrollment

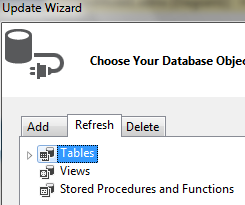
{

}

}

Notice that each class is marked as a partial class, and each matches the name and namespace as the class that is automatically generated. By applying the metadata attribute to the partial class, you ensure that the data validation attributes will be applied to the automatically-generated class. These attributes will not be lost when you regenerate the model classes because the metadata attribute is applied in partial classes that are not regenerated.

To regenerate the automatically-generated classes, open the ContosoModel.edmx file. Once again, right-click on the design surface and select**Update Model from Database**. Even though you have not changed the database, this process will regenerate the classes. In the **Refresh** tab, select **Tables** and **Finish**.



Save the ContosoModel.edmx file to apply the changes.

Open the Student.cs file or the Enrollment.cs file, and notice that the data validation attributes you applied earlier are no longer in the file. However, run the application, and notice that the validation rules are still applied when you enter data.

## Annotation Examples:

for the Date:

[Display(Name = "Enrollment Date")]

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

public Nullable<System.DateTime> EnrollmentDate;

Use regular expressions:

    [RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

    [Required]

    [StringLength(30)]

Range:

    [Range(1, 100)]

    [DataType(DataType.Currency)]