Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10)

By Tom Dykstra | July 30, 2013

DOWNLOADS:

Download Comple...

A newer version of this tutorial series is available, for Visual Studio 2013, Entity Framework 6, and MVC 5.

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 and Visual Studio 2012. The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This tutorial series explains how to build the Contoso University sample application. You can download the completed application.

Code First

There are three ways you can work with data in the Entity Framework: *Database First*, *Model First*, and *Code First*. This tutorial is for Code First. For information about the differences between these workflows and guidance on how to choose the best one for your scenario, see Entity Framework Development Workflows.

MVC

The sample application is built on ASP.NET MVC. If you prefer to work with the ASP.NET Web Forms model, see the Model Binding and Web Forms tutorial series and ASP.NET Data Access Content Map.

Software versions

Shown in the tutorial	Also works with
Windows 8	Windows 7
Visual Studio 2012	Visual Studio 2012 Express for Web. This is automatically installed by the Windows Azure SDK if you don't already have VS 2012 or VS 2012 Express for Web. Visual Studio 2013 should work, but the tutorial has not been tested with it, and some menu selections and dialog boxes are different. The VS 2013 version of the Windows Azure SDK is required for Windows Azure deployment.
.NET 4.5	Most of the features shown will work in .NET 4, but some won't. For example, enum support in EF requires .NET 4.5.
Entity Framework 5	
Windows Azure SDK 2.1	If you skip the Windows Azure deployment steps, you don't need the SDK. When a new version of the SDK is released, the link will install the newer version. In that case, you might have to adapt some of the instructions to new UI and features.

Questions

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity Framework forum, the Entity Framework and LINQ to Entities forum, or StackOverflow.com.

Acknowledgments

See the last tutorial in the series for acknowledgments and a note about VB.

Original version of the tutorial

The original version of the tutorial is available in the the EF 4.1 / MVC 3 e-book.

The Contoso University Web Application

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

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

The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Prerequisites

The directions and screen shots in this tutorial assume that you're using Visual Studio 2012 or Visual Studio 2012 Express for Web, with the latest update and Windows Azure SDK installed as of July, 2013. You can get all of this with the following link:

Windows Azure SDK for Visual Studio 2012

If you have Visual Studio installed, the link above will install any missing components. If you don't have Visual Studio, the link will install Visual Studio 2012 Express for Web. You can use Visual Studio 2013, but some of the required procedures and screens will differ.

Create an MVC Web Application

Open Visual Studio and create a new C# project named "ContosoUniversity" using the **ASP.NET MVC 4 Web Application** template. Make sure you target **.NET Framework 4.5** (you'll be using `enum` properties, and that requires .NET 4.5).

In the **New ASP.NET MVC 4 Project** dialog box select the **Internet Application** template.

Leave the **Razor** view engine selected, and leave the **Create a unit test project** check box cleared.

Click **OK**.

Set Up the Site Style

A few simple changes will set up the site menu, layout, and home page.

Open *Views\Shared_Layout.cshtml*, and replace the contents of the file with the following code. The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>@ViewBag.Title - Contoso University</title>
        <link href="~/favicon.ico" rel="shortcut icon" type="image
        <meta name="viewport" content="width=device-width" />
        @Styles.Render("~/Content/css")
        @Scripts.Render("~/bundles/modernizr")
    </head>
    <body>
        <header>
            <div class="content-wrapper">
                <div class="float-left">
                    <p class="site-title">@Html.ActionLink("Contos
                </div>
                <div class="float-right">
                    <section id="login">
                        @Html.Partial("_LoginPartial")
                    </section>
                    <nav>
                        <ul id="menu">
                            <li>@Html.ActionLink("Home", "Index",
                            <li>@Html.ActionLink("About", "About",
                            <li>@Html.ActionLink("Students", "Inde
                            <li>@Html.ActionLink("Courses", "Index
                            <li>@Html.ActionLink("Instructors", "I
                            <li>@Html.ActionLink("Departments", "I
                        </ul>
                    </nav>
                </div>
            </div>
        </header>
        <div id="body">
            @RenderSection("featured", required: false)
            <section class="content-wrapper main-content clear-fix
                @RenderBody()
            </section>
        </div>
        <footer>
            <div class="content-wrapper">
                <div class="float-left">
                    <p>&copy; @DateTime.Now.Year - Contoso Univers
                </div>
            </div>
        </footer>

        @Scripts.Render("~/bundles/jquery")
        @RenderSection("scripts", required: false)
    </body>
</html>
```

This code makes the following changes:

Replaces the template instances of "My ASP.NET MVC Application" and "your logo here" with "Contoso University".

Adds several action links that will be used later in the tutorial.

In *Views\Home\Index.cshtml*, replace the contents of the file with the following code to eliminate the template paragraphs about ASP.NET and MVC:

```
@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
        </div>
    </section>
}
```

In *Controllers\HomeController.cs*, change the value for `ViewBag.Message` in the `Index` Action method to "Welcome to Contoso University!", as shown in the following example:

```
public ActionResult Index()
{
    ViewBag.Message = "Welcome to Contoso University";

    return View();
}
```

Press CTRL+F5 to run the site. You see the home page with the main menu.

Create the Data Model

Next you'll create entity classes for the Contoso University application. You'll start with the following three entities:

There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

> **Note** If you try to compile the project before you finish creating all of these entity classes, you'll get compiler errors.

The Student Entity

In the *Models* folder, create *Student.cs* and replace the existing code with the following code:

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int StudentID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

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

The `StudentID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or *classname*`ID` as the primary key.

The `Enrollments` property is a *navigation property*. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given `Student` row in the database has two related `Enrollment` rows (rows that contain that student's primary key value in their `StudentID` foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

Navigation properties are typically defined as `virtual` so that they can take advantage of certain Entity Framework functionality such as *lazy loading*. (Lazy loading will be explained later, in the Reading Related Data tutorial later in this series.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection`.

The Enrollment Entity

In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

The Grade property is an enum. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade — null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

The Course Entity

In the *Models* folder, create *Course.cs*, replacing the existing code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

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

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute in the next tutorial. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

The main class that coordinates Entity Framework functionality for a given data model is the *database context* class. You create this class by deriving from the System.Data.Entity.DbContext class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

Create a folder named *DAL* (for Data Access Layer). In that folder create a new class file named *SchoolContext.cs*, and replace the existing code with the following code:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(DbModelBuilder mod
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameCo
        }
    }
}
```

This code creates a DbSet property for each entity set. In Entity Framework terminology, an *entity set* typically corresponds to a database table, and an *entity* corresponds to a row in the table.

The `modelBuilder.Conventions.Remove` statement in the OnModelCreating method prevents table names from being pluralized. If you didn't do this, the generated tables would be named `Students`, `Courses`, and `Enrollments`. Instead, the table names will be `Student`, `Course`, and `Enrollment`. Developers disagree about whether table names should be pluralized or not. This tutorial uses the singular form, but the important point is that you can select whichever form you prefer by including or omitting this line of code.

SQL Server Express LocalDB

LocalDB is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution

mode of SQL Server Express that enables you to work with databases as *.mdf* files. Typically, LocalDB database files are kept in the *App_Data* folder of a web project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files.

Typically SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it is not designed to work with IIS.

In Visual Studio 2012 and later versions, LocalDB is installed by default with Visual Studio. In Visual Studio 2010 and earlier versions, SQL Server Express (without LocalDB) is installed by default with Visual Studio; you have to install it manually if you're using Visual Studio 2010.

In this tutorial you'll work with LocalDB so that the database can be stored in the *App_Data* folder as an *.mdf* file. Open the root *Web.config* file and add a new connection string to the `connectionStrings` collection, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file is in the *Views* subfolder that you don't need to update.)

```
<add name="SchoolContext" connectionString="Data Source=(LocalDb)
```

By default, the Entity Framework looks for a connection string named the same as the `DbContext` class (`SchoolContext` for this project). The connection string you've added specifies a LocalDB database named *ContosoUniversity.mdf* located in the *App_Data* folder. For more information, see SQL Server Connection Strings for ASP.NET Web Applications.

You don't actually need to specify the connection string. If you don't supply a connection string, Entity Framework will create one for you; however, the database might not be in the *App_data* folder of your app. For information on where the database will be created, see Code First to a New Database.

The `connectionStrings` collection also has a connection string named `DefaultConnection` which is used for the membership database. You won't be using the membership database in this tutorial. The only difference between the two connection strings is the database name and the name attribute value.

Set up and Execute a Code First Migration

When you first start to develop an application, your data model changes frequently, and each time the model changes it gets out of sync with the database. You can configure the Entity Framework to automatically drop and re-create the database each time you change the data model. This is not a problem early in development because test data is easily re-created, but after you have deployed to production you usually want to update the database schema without dropping the database. The Migrations feature enables Code First to update the database without dropping and re-creating it. Early in the development cycle of a new project you might want to use DropCreateDatabaseIfModelChanges to drop, recreate and re-seed the database each time the model changes. One you get ready to deploy your application, you can convert to the migrations approach. For this tutorial you'll only use migrations. For more information, see Code First Migrations and Migrations Screencast Series.

Enable Code First Migrations

1. From the **Tools** menu, click **Library Package Manager** and then **Package Manager Console**.

2. At the `PM>` prompt enter the following command:

```
enable-migrations -contexttypename SchoolContext
```

This command creates a *Migrations* folder in the ContosoUniversity project, and it puts in that folder a *Configuration.cs* file that you can edit to configure Migrations.

The `Configuration` class includes a `Seed` method that is called when the database is created and every time it is updated after a data model change.

```
internal sealed class Configuration :
DbMigrationsConfiguration<ContosoUniversity.Models.SchoolC

{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    protected override void
Seed(ContosoUniversity.Models.SchoolContext context)
    {
        //  This method will be called after migrating to
the latest version.

        //  You can use the DbSet<T>.AddOrUpdate() helper
extension method
        //  to avoid creating duplicate seed data. E.g.
        //
        //    context.People.AddOrUpdate(
        //      p => p.FullName,
        //      new Person { FullName = "Andrew Peters"
},
        //      new Person { FullName = "Brice Lambson"
},
        //      new Person { FullName = "Rowan Miller" }
        //    );
        //
    }
}
```

The purpose of this `Seed` method is to enable you to insert test data into the database after Code First creates it or updates it.

Set up the Seed Method

The Seed method runs when Code First Migrations creates the database and every time it updates the database to the latest migration. The purpose of the Seed method is to enable you to insert data into your tables before the application accesses the database for the first time.

In earlier versions of Code First, before Migrations was released, it was common for `Seed` methods to insert test data, because with every model change during development the database had to be completely deleted and re-created from

scratch. With Code First Migrations, test data is retained after database changes, so including test data in the Seed method is typically not necessary. In fact, you don't want the Seed method to insert test data if you'll be using Migrations to deploy the database to production, because the Seed method will run in production. In that case you want the Seed method to insert into the database only the data that you want to be inserted in production. For example, you might want the database to include actual department names in the Department table when the application becomes available in production.

For this tutorial, you'll be using Migrations for deployment, but your Seed method will insert test data anyway in order to make it easier to see how application functionality works without having to manually insert a lot of data.

1. Replace the contents of the *Configuration.cs* file with the following code, which will load test data into the new database.

```
namespace ContosoUniversity.Migrations
{
    using System;
    using System.Collections.Generic;
    using System.Data.Entity.Migrations;
    using System.Linq;
    using ContosoUniversity.Models;

    internal sealed class Configuration :
DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolCont

    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void
Seed(ContosoUniversity.DAL.SchoolContext context)
        {
            var students = new List<Student>
                {
                    new Student { FirstMidName = "Carson",
LastName = "Alexander",
                        EnrollmentDate =
DateTime.Parse("2010-09-01") },
                    new Student { FirstMidName = "Meredith",
LastName = "Alonso",
                        EnrollmentDate =
DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Arturo",
LastName = "Anand",
                        EnrollmentDate =
DateTime.Parse("2013-09-01") },
                    new Student { FirstMidName = "Gytis",
LastName = "Barzdukas",
                        EnrollmentDate =
DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Yan",
LastName = "Li",
                        EnrollmentDate =
DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Peggy",
LastName = "Justice",
                        EnrollmentDate =
DateTime.Parse("2011-09-01") },
                    new Student { FirstMidName = "Laura",
LastName = "Norman",
                        EnrollmentDate =
DateTime.Parse("2013-09-01") },
                    new Student { FirstMidName = "Nino",
LastName = "Olivetto",
                        EnrollmentDate =
DateTime.Parse("2005-08-11") }
                };
            students.ForEach(s =>
context.Students.AddOrUpdate(p => p.LastName, s));
            context.SaveChanges();

            var courses = new List<Course>
```

```
            {
                new Course {CourseID = 1050, Title =
"Chemistry",      Credits = 3, },
                new Course {CourseID = 4022, Title =
"Microeconomics", Credits = 3, },
                new Course {CourseID = 4041, Title =
"Macroeconomics", Credits = 3, },
                new Course {CourseID = 1045, Title =
"Calculus",       Credits = 4, },
                new Course {CourseID = 3141, Title =
"Trigonometry",   Credits = 4, },
                new Course {CourseID = 2021, Title =
"Composition",    Credits = 3, },
                new Course {CourseID = 2042, Title =
"Literature",     Credits = 4, }
            };
         courses.ForEach(s =>
context.Courses.AddOrUpdate(p => p.Title, s));
         context.SaveChanges();

         var enrollments = new List<Enrollment>
            {
                new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Alexander").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Chemistry" ).CourseID,
                    Grade = Grade.A
                },
                 new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Alexander").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Microeconomics" ).CourseID,
                    Grade = Grade.C
                 },
                 new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Alexander").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Macroeconomics" ).CourseID,
                    Grade = Grade.B
                },
                 new Enrollment {
                     StudentID = students.Single(s =>
s.LastName == "Alonso").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Calculus" ).CourseID,
                    Grade = Grade.B
                },
                 new Enrollment {
                     StudentID = students.Single(s =>
s.LastName == "Alonso").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Trigonometry" ).CourseID,
                    Grade = Grade.B
                },
                 new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Alonso").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Composition" ).CourseID,
                    Grade = Grade.B
                },
                 new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Anand").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Chemistry" ).CourseID
                },
                 new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Anand").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Microeconomics").CourseID,
                    Grade = Grade.B
```

```
                },
                new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Barzdukas").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Li").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Composition").CourseID,
                    Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s =>
s.LastName == "Justice").StudentID,
                    CourseID = courses.Single(c =>
c.Title == "Literature").CourseID,
                    Grade = Grade.B
                }
            };

         foreach (Enrollment e in enrollments)
         {
            var enrollmentInDataBase =
context.Enrollments.Where(
                s =>
                     s.Student.StudentID == e.StudentID
&&
                     s.Course.CourseID ==
e.CourseID).SingleOrDefault();
            if (enrollmentInDataBase == null)
            {
               context.Enrollments.Add(e);
            }
         }
         context.SaveChanges();
      }
   }
}
```

The Seed method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate DbSet property, and then saves the changes to the database. It isn't necessary to call the SaveChanges method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

Some of the statements that insert data use the AddOrUpdate method to perform an "upsert" operation. Because the `Seed` method runs with every migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it ***overrides*** any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist. The Seed method uses both approaches.

The first parameter passed to the AddOrUpdate method specifies the property to use to check if a row already exists. For the test student data that you are providing, the `LastName` property can be used for this purpose since each last name in the list is unique:

```
context.Students.AddOrUpdate(p => p.LastName, s)
```

This code assumes that last names are unique. If you manually add a student with a duplicate last name, you'll get the following exception the next time you perform a migration.

Sequence contains more than one element

For more information about the `AddOrUpdate` method, see Take care with EF 4.3 AddOrUpdate Method on Julie Lerman's blog.

The code that adds `Enrollment` entities doesn't use the `AddOrUpdate` method. It checks if an entity already exists and inserts the entity if it doesn't exist. This approach will preserve changes you make to an enrollment grade when migrations run. The code loops through each member of the `Enrollment` List and if the enrollment is not found in the database, it adds the enrollment to the database. The first time you update the database, the database will be empty, so it will add each enrollment.

```
foreach (Enrollment e in enrollments)
{
    var enrollmentInDataBase = context.Enrollments.Where(
        s => s.Student.StudentID == e.Student.StudentID
&&
             s.Course.CourseID ==
e.Course.CourseID).SingleOrDefault();
    if (enrollmentInDataBase == null)
    {
        context.Enrollments.Add(e);
    }
}
```

For information about how to debug the `Seed` method and how to handle redundant data such as two students named "Alexander Carson", see Seeding and Debugging Entity Framework (EF) DBs on Rick Anderson's blog.

2. Build the project.

Create and Execute the First Migration

1. In the Package Manager Console window, enter the following commands:

```
add-migration InitialCreate
update-database
```

The `add-migration` command adds to the Migrations folder a *[DateStamp]_InitialCreate.cs* file that contains code which creates the database. The first parameter (`InitialCreate`) is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them. Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method. The following code shows the contents of the *`InitialCreate`* file:

```
namespace ContosoUniversity.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class InitialCreate : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Student",
                c => new
                    {
                        StudentID = c.Int(nullable:
false, identity: true),
                        LastName = c.String(),
                        FirstMidName = c.String(),
                        EnrollmentDate =
c.DateTime(nullable: false),
                    })
                .PrimaryKey(t => t.StudentID);

            CreateTable(
                "dbo.Enrollment",
                c => new
                    {
                        EnrollmentID = c.Int(nullable:
false, identity: true),
                        CourseID = c.Int(nullable:
false),
                        StudentID = c.Int(nullable:
false),
                        Grade = c.Int(),
                    })
                .PrimaryKey(t => t.EnrollmentID)
                .ForeignKey("dbo.Course", t =>
t.CourseID, cascadeDelete: true)
                .ForeignKey("dbo.Student", t =>
t.StudentID, cascadeDelete: true)
                .Index(t => t.CourseID)
                .Index(t => t.StudentID);

            CreateTable(
                "dbo.Course",
                c => new
                    {
                        CourseID = c.Int(nullable:
false),
                        Title = c.String(),
                        Credits = c.Int(nullable: false),
                    })
                .PrimaryKey(t => t.CourseID);

        }

        public override void Down()
        {
            DropIndex("dbo.Enrollment", new[] {
"StudentID" });
            DropIndex("dbo.Enrollment", new[] {
"CourseID" });
            DropForeignKey("dbo.Enrollment", "StudentID",
"dbo.Student");
            DropForeignKey("dbo.Enrollment", "CourseID",
"dbo.Course");
            DropTable("dbo.Course");
            DropTable("dbo.Enrollment");
            DropTable("dbo.Student");
        }
    }
}
```

The `update-database` command runs the `Up` method to create the database and then it runs the `Seed` method to populate the database.

A SQL Server database has now been created for your data model. The name of the database is *ContosoUniversity*, and the *.mdf* file is in your project's *App_Data* folder because that's what you specified in your connection string.

You can use either **Server Explorer** or **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio. For this tutorial you'll use **Server Explorer**. In Visual Studio Express 2012 for Web, **Server Explorer** is called **Database Explorer**.

1. From the **View** menu, click **Server Explorer**.

2. Click the **Add Connection** icon.

3. If you are prompted with the **Choose Data Source** dialog, click **Microsoft SQL Server**, and then click **Continue**.

4. In the **Add Connection** dialog box, enter **(localdb)\v11.0** for the **Server Name**. Under **Select or enter a database name**, select

ContosoUniversity.

5. Click **OK.**

6. Expand **SchoolContext** and then expand **Tables**.

7. Right-click the **Student** table and click **Show Table Data** to see the columns that were created and the rows that were inserted into the table.

Creating a Student Controller and Views

The next step is to create an ASP.NET MVC controller and views in your application that can work with one of these tables.

1. To create a `Student` controller, right-click the **Controllers** folder in **Solution Explorer**, select **Add**, and then click **Controller**. In the **Add Controller** dialog box, make the following selections and then click **Add**:
Controller name: **StudentController**.

Template: **MVC controller with read/write actions and views, using Entity Framework**.

Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)

Data context class: **SchoolContext (ContosoUniversity.Models)**.

Views: **Razor (CSHTML)**. (The default.)

2. Visual Studio opens the *Controllers\StudentController.cs* file. You see a class variable has been created that instantiates a database context object:

```
private SchoolContext db = new SchoolContext();
```

The `Index` action method gets a list of students from the *Students* entity set by reading the `Students` property of the database context instance:

```
 public ViewResult Index()
{
    return View(db.Students.ToList());
}
```

The *Student\Index.cshtml* view displays this list in a table:

```
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.LastName)
        </th>
        <th>
            @Html.DisplayNameFor(model =>
model.FirstMidName)
        </th>
        <th>
            @Html.DisplayNameFor(model =>
model.EnrollmentDate)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem =>
item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem =>
item.EnrollmentDate)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new {
id=item.StudentID }) |
            @Html.ActionLink("Details", "Details", new {
id=item.StudentID }) |
            @Html.ActionLink("Delete", "Delete", new {
id=item.StudentID })
        </td>
    </tr>
}
```

3. Press CTRL+F5 to run the project.

Click the **Students** tab to see the test data that the `Seed` method inserted.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of *conventions*, or assumptions that the Entity Framework makes. Some of them have already been noted:

The pluralized forms of entity class names are used as table names.

Entity property names are used for column names.

Entity properties that are named `ID` or *classname*`ID` are recognized as primary key properties.

You've seen that conventions can be overridden (for example, you specified that table names shouldn't be pluralized), and you'll learn more about conventions and how to override them in the Creating a More Complex Data Model tutorial later in this series. For more information, see Code First Conventions.

Summary

You've now created a simple application that uses the Entity Framework and SQL Server Express to store and display data. In the following tutorial you'll learn how to perform basic CRUD (create, read, update, delete) operations. You can leave feedback at the bottom of this page. Please let us know how you liked this portion of the tutorial and how we could improve it.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

This article was originally created on July 30, 2013

Author Information

Tom Dykstra – Tom Dykstra is a Senior Programming Writer on Microsoft's Web Platform & Tools Content Team...

You're Viewing
Creating an Entity Framework Data Model

→ Next
Implementing Basic CRUD Functionality

SHOW COMMENTS