Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

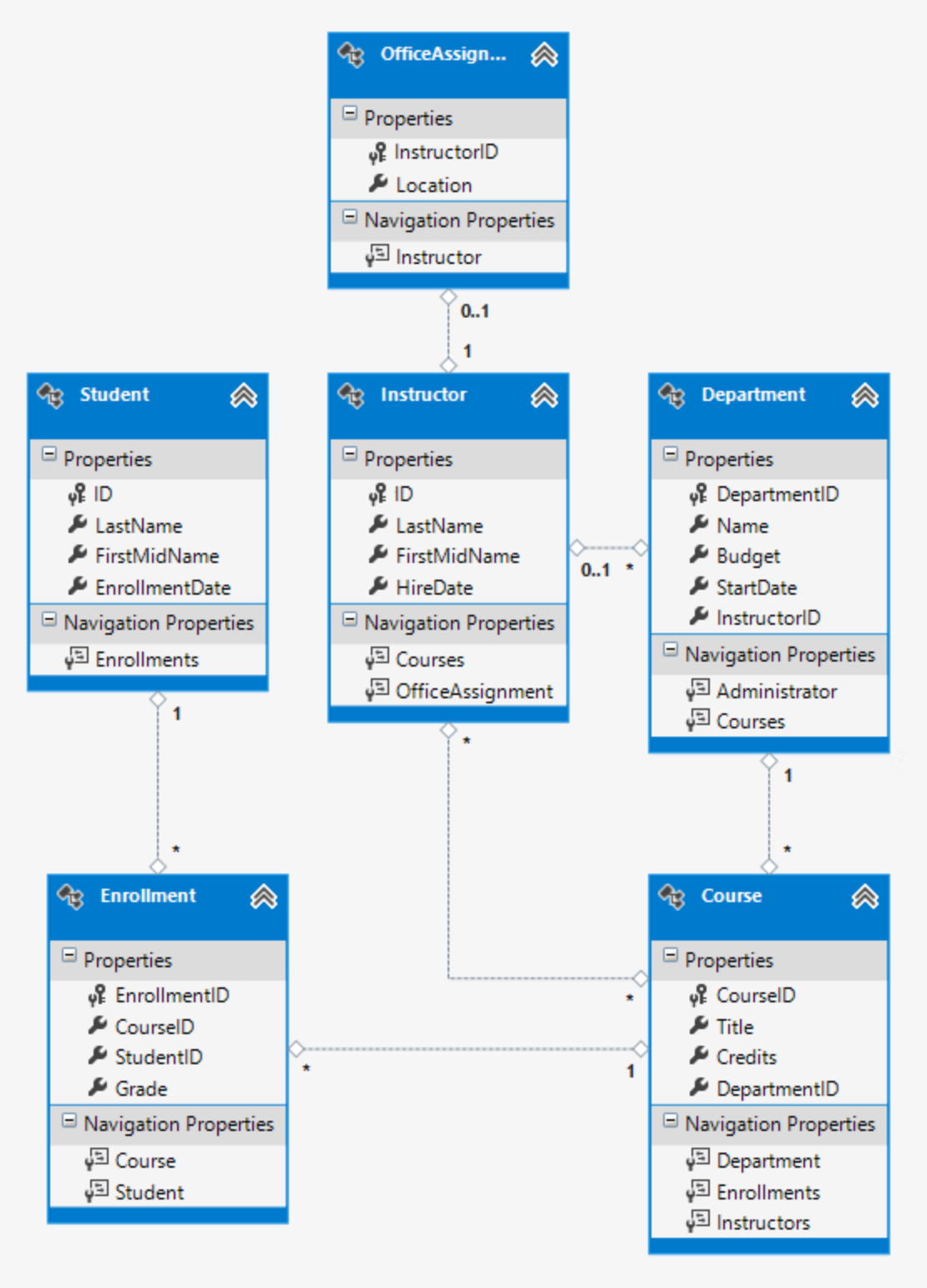
Step By Step, Guide

Microsoft

Creating a More Complex Data Model for an ASP.NET MVC Application

In the previous tutorials you worked with a simple data model that was composed of three entities. In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules. You'll see two ways to customize the data model: by adding attributes to entity classes and by adding code to the database context class.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



Customize the Data Model by Using Attributes

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete `School` data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType Attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models\Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

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

The DataType attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The DataType Enumeration provides for many data types, such as *Date, Time, PhoneNumber, Currency, EmailAddress* and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emits HTML 5 data- (pronounced *data dash*) attributes that HTML 5 browsers can understand. The DataType attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

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

The `ApplyFormatInEditMode` setting specifies that the specified formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you might not want the currency symbol in the text box for editing.)

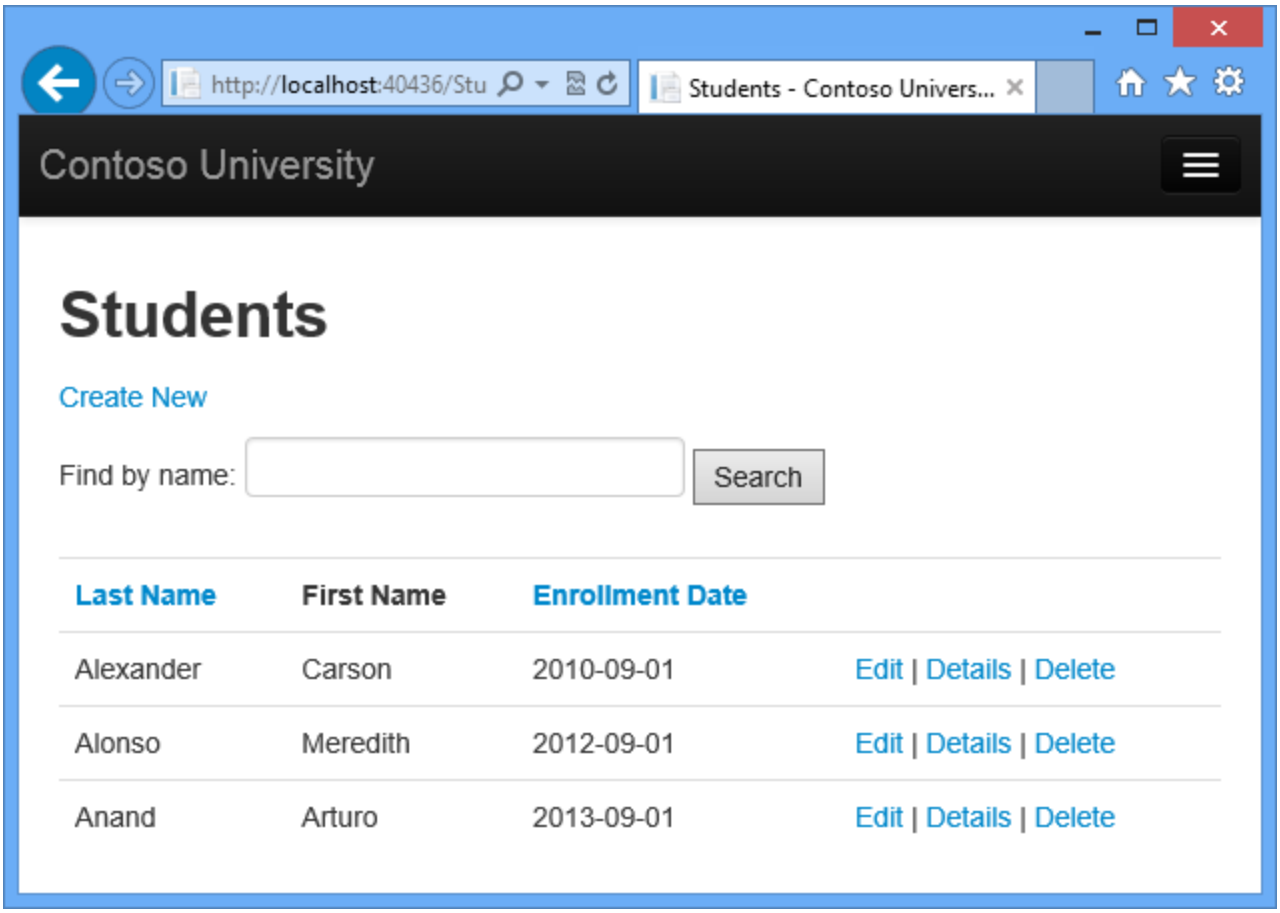
You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute also. The `DataType` attribute conveys the *semantics* of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat uses the string template). For more information, see Brad Wilson's ASP.NET MVC 2 Templates. (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the `DataType` attribute with a date field, you have to specify the `DisplayFormat` attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see this StackOverflow thread.

For more information about how to handle other date formats in MVC, go to MVC 5 Introduction: Examining the Edit Methods and Edit View and search in the page for "internationalization".

Run the Student Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the `Student` model.



The StringLengthAttribute

You can also specify data validation rules and validation error messages using attributes. The StringLength attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add StringLength attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
```

```
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

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

The StringLength attribute won't prevent a user from entering white space for a name. You can use the RegularExpression attribute to apply restrictions to the input. For example the following code requires the first character to be upper case and the remaining characters to be alphabetical:

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

The MaxLength attribute provides similar functionality to the StringLength attribute but doesn't provide client side validation.

Run the application and click the **Students** tab. You get the following error:

The model backing the 'SchoolContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).

The database model has changed in a way that requires a change in the database schema, and Entity Framework detected that. You'll use migrations to update the schema without losing any data that you added to the database by using the UI. If you changed data that was created by the `Seed` method, that will be changed back to its original state because of the AddOrUpdate method that you're using in the `Seed` method. (AddOrUpdate is equivalent to an "upsert" operation from database terminology.)

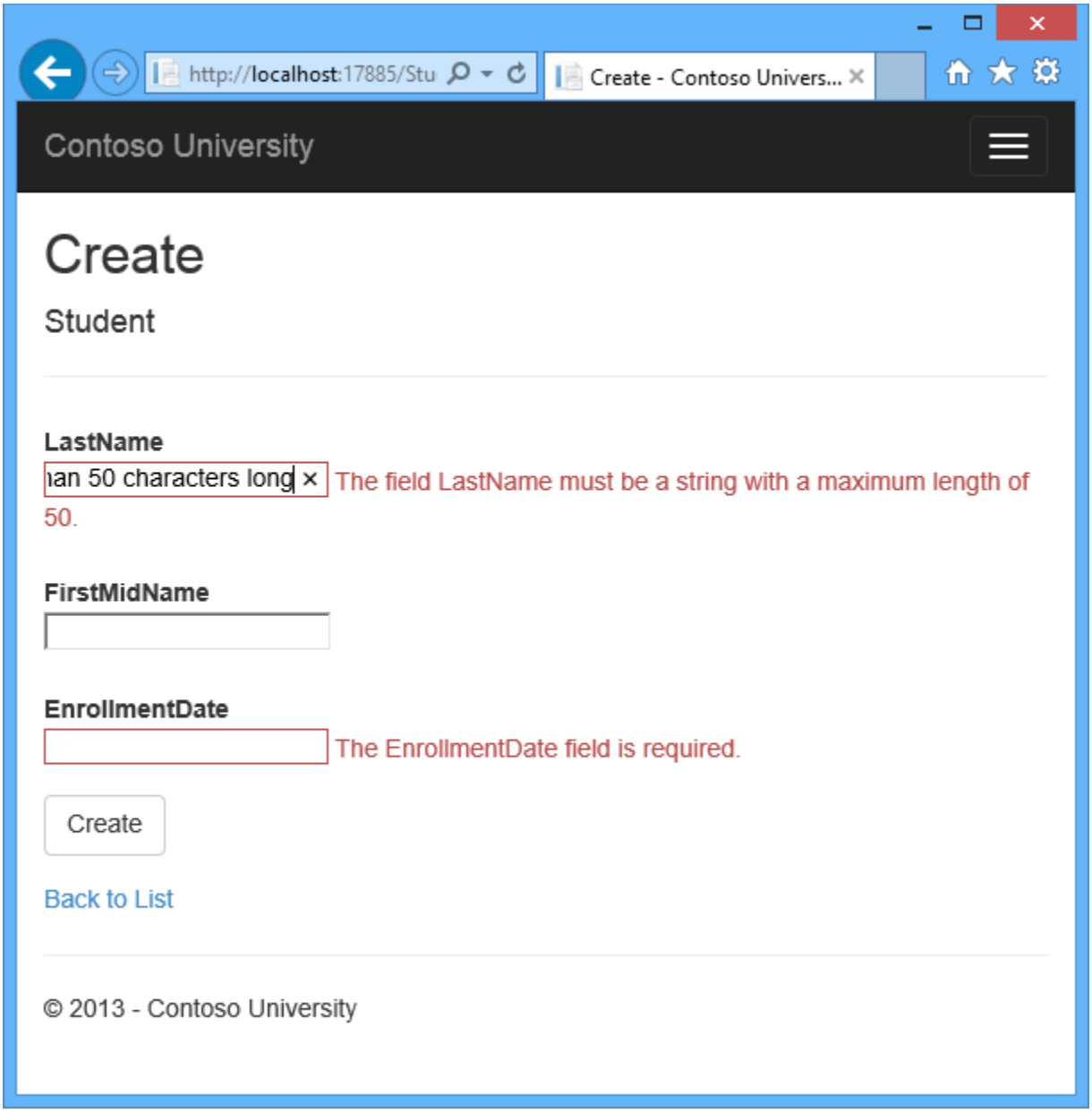
In the Package Manager Console (PMC), enter the following commands:

```
add-migration MaxLengthOnNames
update-database
```

The `add-migration` command creates a file named *<timeStamp>_MaxLengthOnNames.cs*. This file contains code in the `Up` method that will update the database to match the current data model. The `update-database` command ran that code.

The timestamp prepended to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the `update-database` command, and then all of the migrations are applied in the order in which they were created.

Run the **Create** page, and enter either name longer than 50 characters. When you click **Create**, client side validation shows an error message.



The Column Attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

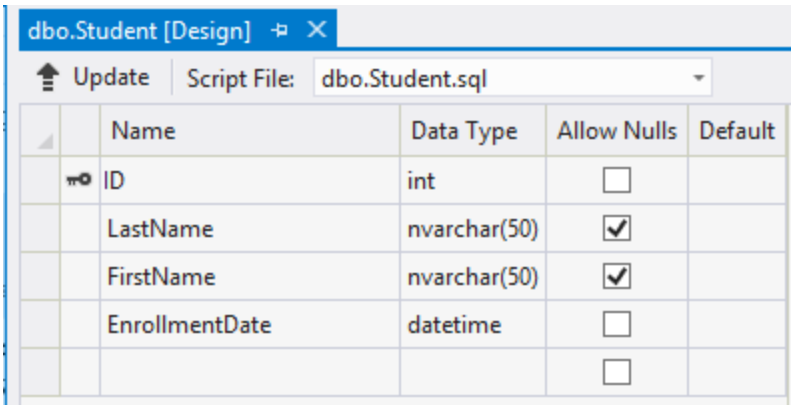
In the *Student.cs* file, add a `using` statement for System.ComponentModel.DataAnnotations.Schema and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]

        public DateTime EnrollmentDate { get; set; }

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

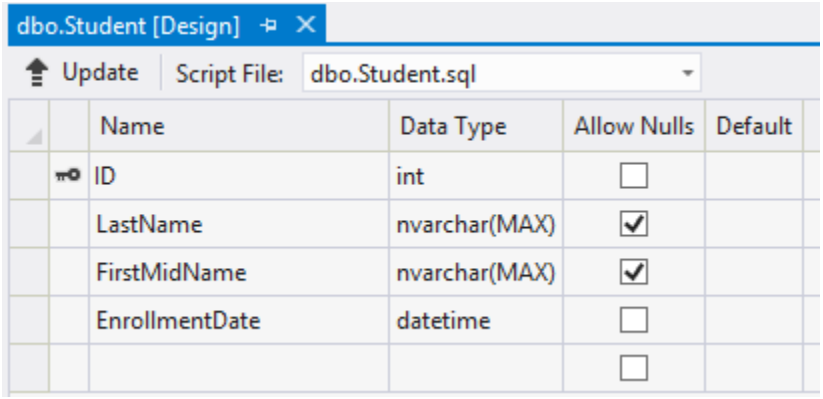
The addition of the Column attribute changes the model backing the SchoolContext, so it won't match the database. Enter the following commands in the PMC to create another migration:

```
add-migration ColumnFirstName
update-database
```

In **Server Explorer**, open the *Student* table designer by double-clicking the *Student* table.



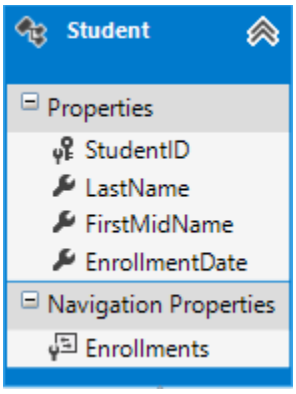
The following image shows the original column name as it was before you applied the first two migrations. In addition to the column name changing from `FirstMidName` to `FirstName`, the two name columns have changed from `MAX` length to 50 characters.



You can also make database mapping changes using the Fluent API, as you'll see later in this tutorial.

Note If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Complete Changes to the Student Entity



In *Models\Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

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

The Required Attribute

The Required attribute makes the name properties required fields. The `Required attribute` is not needed for value types such as DateTime, int, double, and float. Value types cannot be assigned a null value, so they are inherently treated as required fields. You could remove the Required attribute and replace it with a minimum length parameter for the `StringLength` attribute:

```
        [Display(Name = "Last Name")]
        [StringLength(50, MinimumLength=1)]
        public string LastName { get; set; }
```

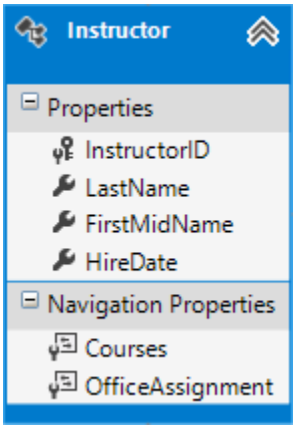
The Display Attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName Calculated Property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a `get` accessor, and no `FullName` column will be generated in the database.

Create the Instructor Entity



Create *Models\Instructor.cs*, replacing the template code with the following code:

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

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }
```

```
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public virtual ICollection<Course> Courses { get; set; }
        public virtual OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the `Student` and `Instructor` entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the instructor class as follows:

```
public class Instructor
{
   public int ID { get; set; }

   [Display(Name = "Last Name"),StringLength(50, MinimumLength=1)]
   public string LastName { get; set; }

   [Column("FirstName"),Display(Name = "First Name"),StringLength(50,
MinimumLength=1)]
   public string FirstMidName { get; set; }

   [DataType(DataType.Date),Display(Name = "Hire Date")]
   public DateTime HireDate { get; set; }

   [Display(Name = "Full Name")]
   public string FullName
   {
      get { return LastName + ", " + FirstMidName; }
   }

   public virtual ICollection<Course> Courses { get; set; }
   public virtual OfficeAssignment OfficeAssignment { get; set; }
}
```

The Courses and OfficeAssignment Navigation Properties

The `Courses` and `OfficeAssignment` properties are navigation properties. As was explained earlier, they are typically defined as virtual so that they can take advantage of an Entity

Framework feature called lazy loading. In addition, if a navigation property can hold multiple entities, its type must implement the ICollection<T> Interface. For example IList<T> qualifies but not IEnumerable<T> because `IEnumerable<T>` doesn't implement Add.

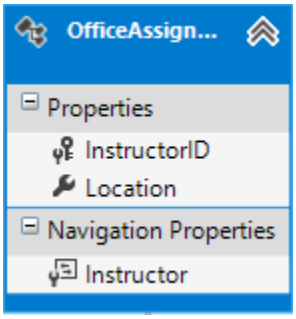
An instructor can teach any number of courses, so `Courses` is defined as a collection of `Course` entities.

```
public virtual ICollection<Course> Courses { get; set; }
```

Our business rules state an instructor can only have at most one office, so `OfficeAssignment` is defined as a single `OfficeAssignment` entity (which may be `null` if no office is assigned).

```
public virtual OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment Entity



Create *Models\OfficeAssignment.cs* with the following code:

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

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        [ForeignKey("Instructor")]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public virtual Instructor Instructor { get; set; }
    }
}
```

Build the project, which saves your changes and verifies you haven't made any copy and paste errors the compiler can catch.

The Key Attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or *classname*`ID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
[ForeignKey("Instructor")]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something different than *classname*`ID` or `ID`. By default EF treats the key as non-database-generated because the column is for an identifying relationship.

The ForeignKey Attribute

When there is a one-to-zero-or-one relationship or a one-to-one relationship between two entities (such as between `OfficeAssignment` and `Instructor`), EF can't work out which end of the relationship is the principal and which end is dependent. One-to-one relationships have a reference navigation property in each class to the other class. The ForeignKey Attribute can be applied to the dependent class to establish the relationship. If you omit the ForeignKey Attribute, you get the following error when you try to create the migration:

Unable to determine the principal end of an association between the types 'ContosoUniversity.Models.OfficeAssignment' and 'ContosoUniversity.Models.Instructor'. The principal end of this association must be explicitly configured using either the relationship fluent API or data annotations.

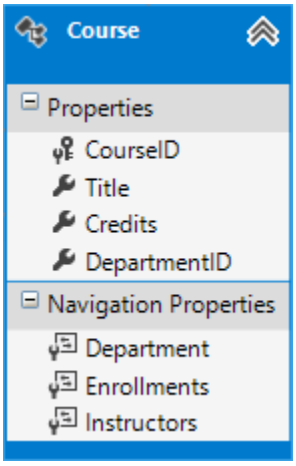
Later in the tutorial you'll see how to configure this relationship with the fluent API.

The Instructor Navigation Property

The `Instructor` entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the `OfficeAssignment` entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an `Instructor` entity has a related `OfficeAssignment` entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the InstructorID foreign key (which is also the key to this table) is non-nullable.

Modify the Course Entity



In *Models\Course.cs*, replace the code you added earlier with the following code:

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

namespace ContosoUniversity.Models
{
   public class Course
   {
      [DatabaseGenerated(DatabaseGeneratedOption.None)]
      [Display(Name = "Number")]
      public int CourseID { get; set; }

      [StringLength(50, MinimumLength = 3)]
      public string Title { get; set; }

      [Range(0, 5)]
      public int Credits { get; set; }

      public int DepartmentID { get; set; }

      public virtual Department Department { get; set; }
      public virtual ICollection<Enrollment> Enrollments { get; set; }
      public virtual ICollection<Instructor> Instructors { get; set; }
   }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related `Department` entity and it has a `Department` navigation property. The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the `Department` entity is null if you don't load it, so when you update the course entity, you would have to first fetch the

`Department` entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the `Department` entity before you update.

The DatabaseGenerated Attribute

The DatabaseGenerated attribute with the None parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, the Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for `Course` entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

Foreign Key and Navigation Properties

The foreign key properties and navigation properties in the `Course` entity reflect the following relationships:

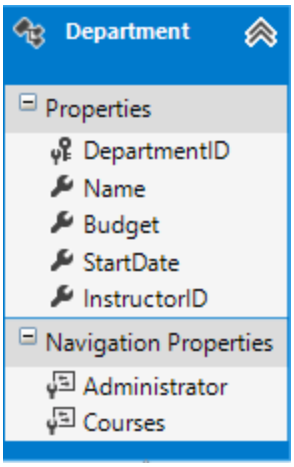
- A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.
- ```
  public int DepartmentID { get; set; }
  public virtual Department Department { get; set; }
  ```

- A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

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

- A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

  ```
  public virtual ICollection<Instructor> Instructors { get; set; }
  ```

Create the Department Entity
  ```



Create *Models\Department.cs* with the following code:

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

namespace ContosoUniversity.Models
{
   public class Department
   {
      public int DepartmentID { get; set; }

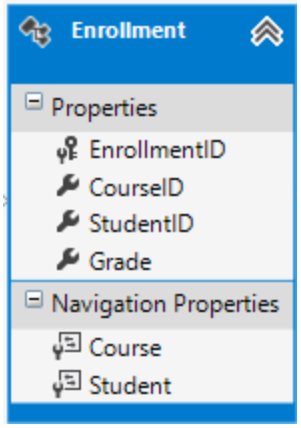
      [StringLength(50, MinimumLength=3)]
      public string Name { get; set; }

      [DataType(DataType.Currency)]
      [Column(TypeName = "money")]
      public decimal Budget { get; set; }

      [DataType(DataType.Date)]
      [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
      [Display(Name = "Start Date")]
      public DateTime StartDate { get; set; }

      public int? InstructorID { get; set; }

      public virtual Instructor Administrator { get; set; }
      public virtual ICollection<Course> Courses { get; set; }
   }
}
```

The Column Attribute

Earlier you used the Column attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework usually chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that. For more information about CLR data types and how they match to SQL Server data types, see SqlClient for Entity FrameworkTypes.

Foreign Key and Navigation Properties

The foreign key and navigation properties reflect the following relationships:

- A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the `Instructor` entity, and a question mark is added after the `int` type designation to mark the property as nullable.The navigation property is named `Administrator` but holds an `Instructor` entity:
- ```
  public int? InstructorID { get; set; }
  public virtual Instructor Administrator { get; set; }
  ```

- A department may have many courses, so there's a `Courses` navigation property:

  ```
  public virtual ICollection<Course> Courses { get; set; }
  ```

Note By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, you'd get the following exception message: "The referential relationship will result in a cyclical reference that's not allowed." If your business rules required `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

```
modelBuilder.Entity().HasRequired(d =>
d.Administrator).WithMany().WillCascadeOnDelete(false);
```

Modify the Enrollment Entity
```



In *Models\Enrollment.cs*, replace the code you added earlier with the following code
`using System.ComponentModel.DataAnnotations;`

```
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; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

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

Foreign Key and Navigation Properties

The foreign key properties and navigation properties reflect the following relationships:

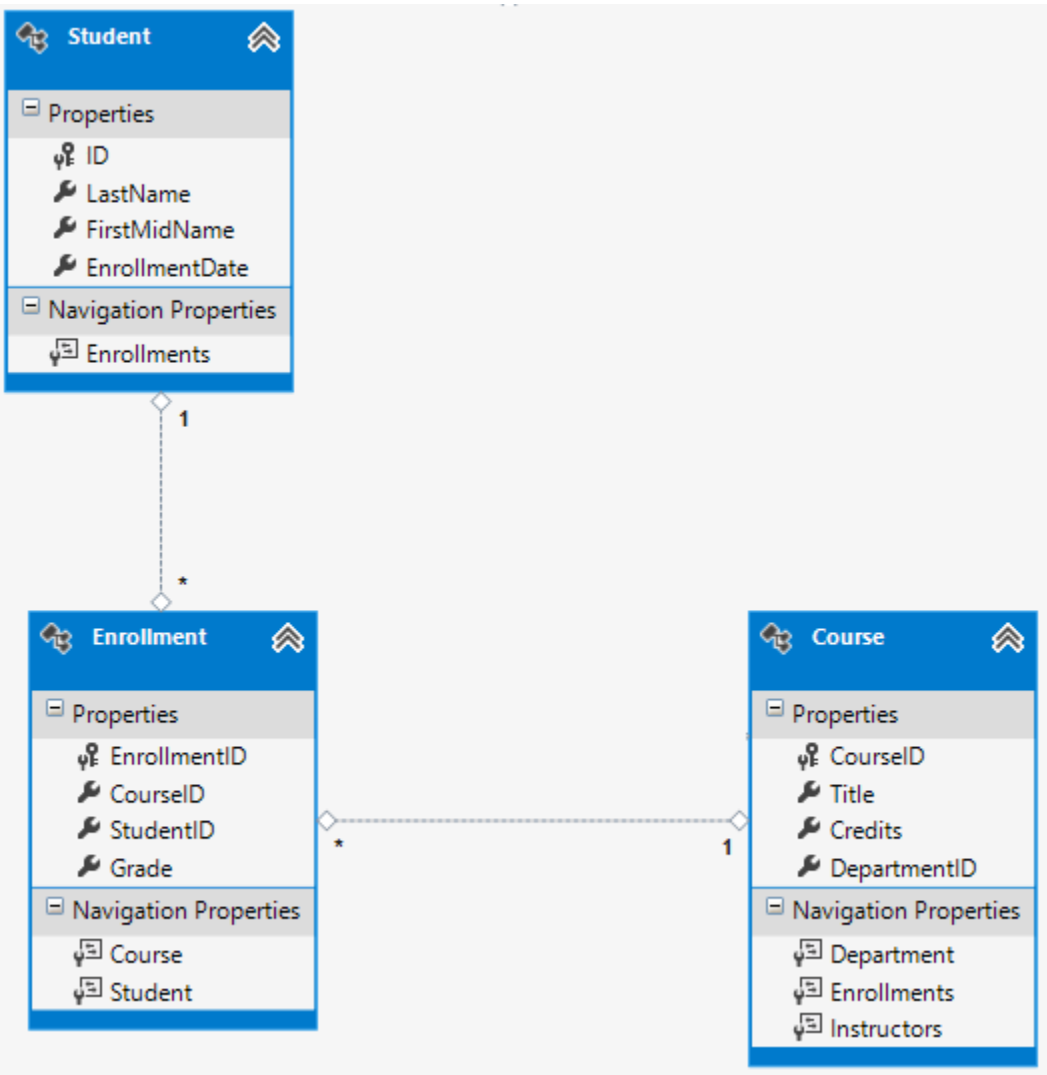
- An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:
- `public int CourseID { get; set; }`
  `public virtual Course Course { get; set; }`

- An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:
- `public int StudentID { get; set; }`
  `public virtual Student Student { get; set; }`

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity functions as a many-to-many join table *with payload* in the database. This means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a `Grade` property).

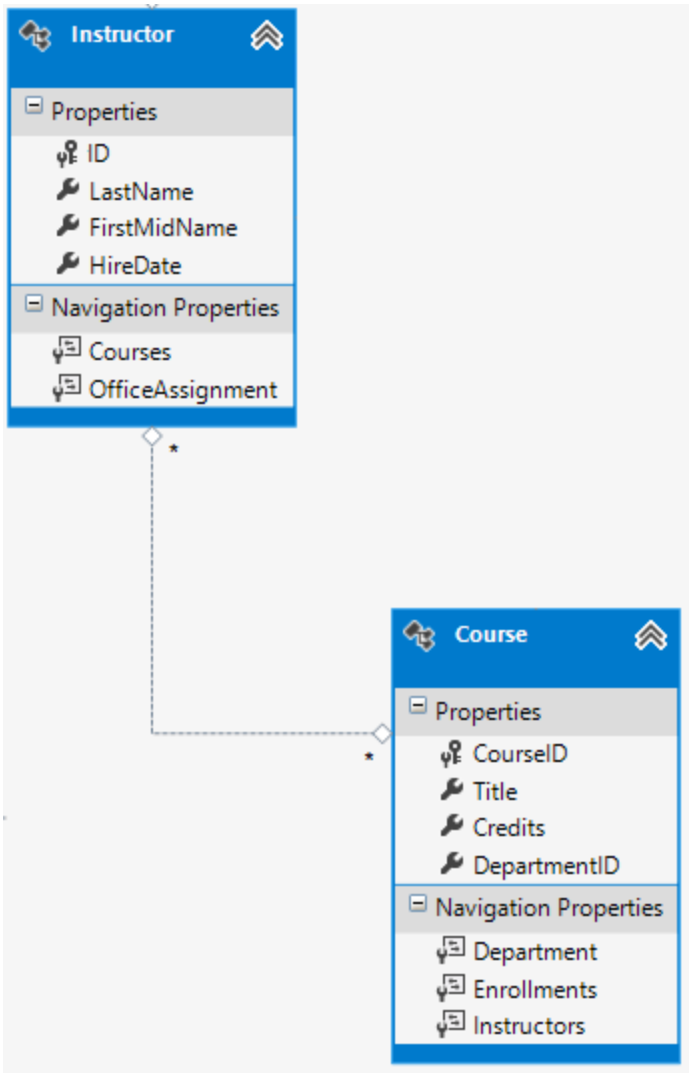
The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)



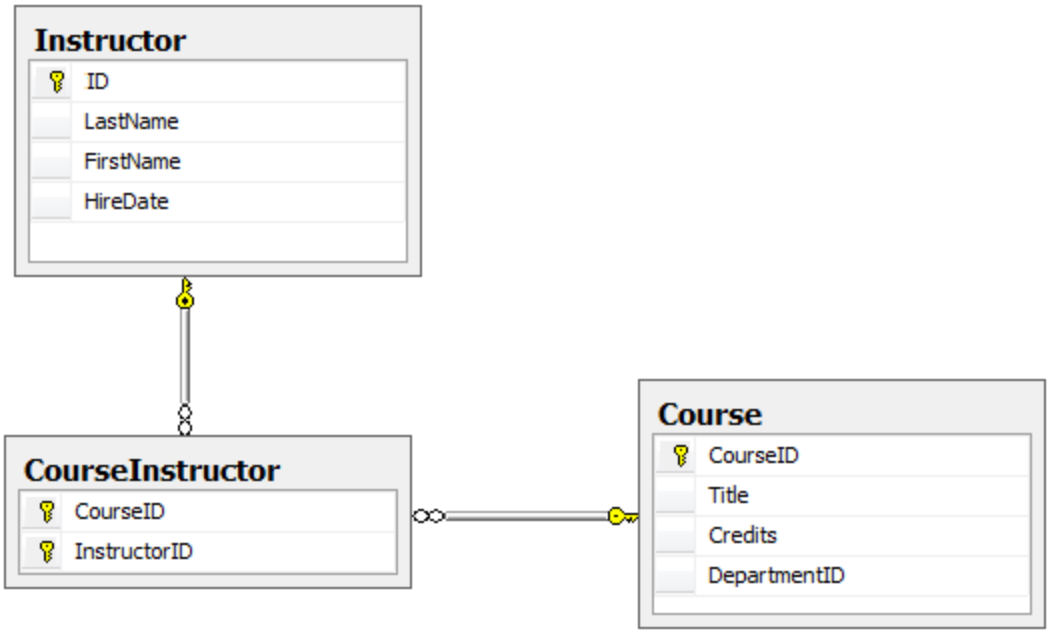
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would correspond to a many-to-many join table *without payload* (or a *pure join table*) in the database, and you wouldn't have to create a

model class for it at all. The `Instructor` and `Course` entities have that kind of many-to-many relationship, and as you can see, there is no entity class between them:



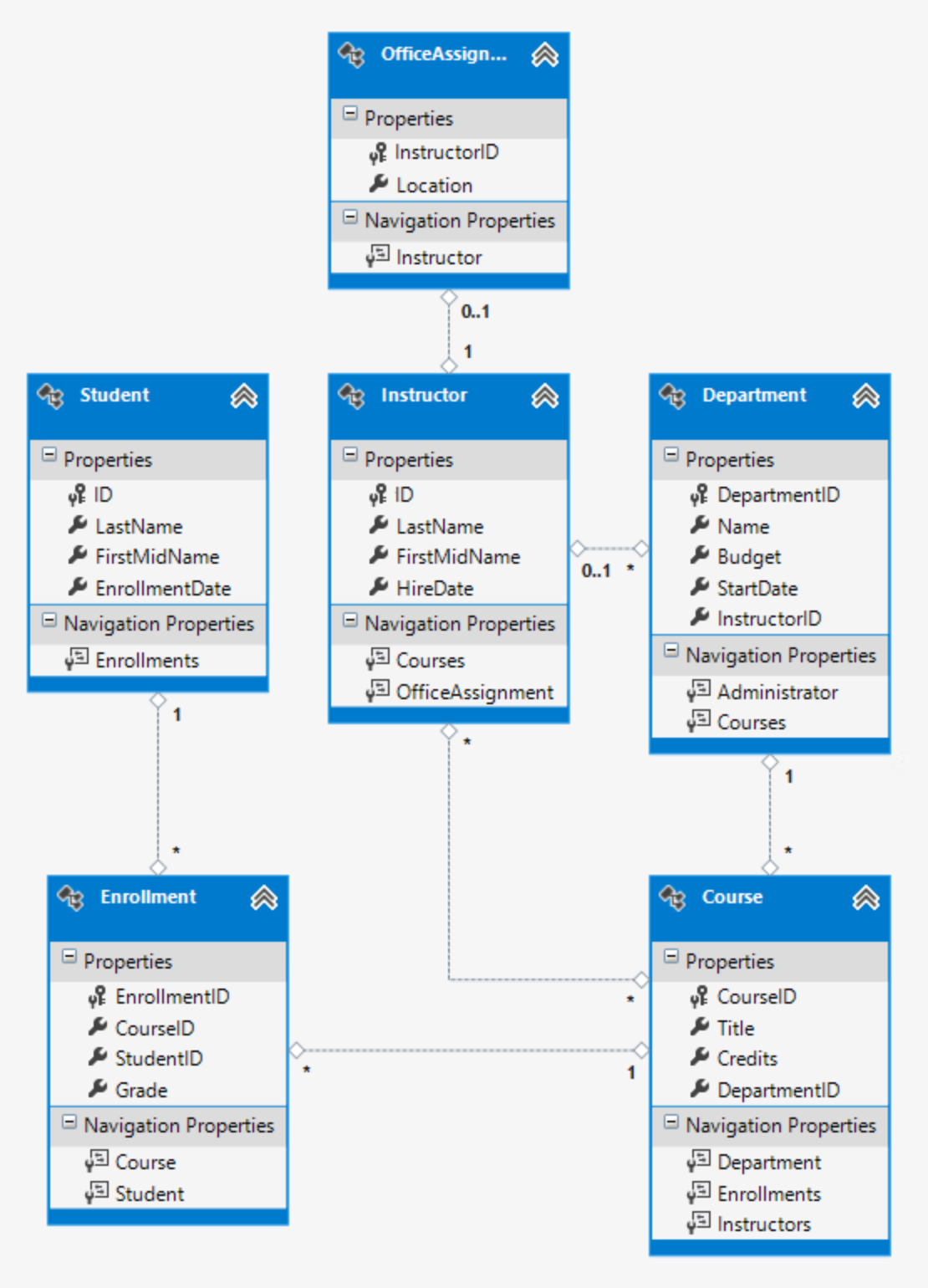
A join table is required in the database, however, as shown in the following database diagram:



The Entity Framework automatically creates the `CourseInstructor` table, and you read and update it indirectly by reading and updating the `Instructor.Courses` and `Course.Instructors` navigation properties.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the many-to-many relationship lines (* to *) and the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor`

and `OfficeAssignment` entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Customize the Data Model by adding Code to the Database Context

Next you'll add the new entities to the `SchoolContext` class and customize some of the mapping using fluent API calls. The API is "fluent" because it's often used by stringing a series of method calls together into a single statement, as in the following example:

```
modelBuilder.Entity<Course>()
    .HasMany(c => c.Instructors).WithMany(i => i.Courses)
    .Map(t => t.MapLeftKey("CourseID")
        .MapRightKey("InstructorID")
        .ToTable("CourseInstructor"));
```

In this tutorial you'll use the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible.

To add the new entities to the data model and perform database mapping that you didn't do by using attributes, replace the code in *DAL\SchoolContext.cs* 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<Course> Courses { get; set; }
      public DbSet<Department> Departments { get; set; }
      public DbSet<Enrollment> Enrollments { get; set; }
      public DbSet<Instructor> Instructors { get; set; }
      public DbSet<Student> Students { get; set; }
      public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

      protected override void OnModelCreating(DbModelBuilder modelBuilder)
      {
         modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

         modelBuilder.Entity<Course>()
```

```
            .HasMany(c => c.Instructors).WithMany(i => i.Courses)
            .Map(t => t.MapLeftKey("CourseID")
                .MapRightKey("InstructorID")
                .ToTable("CourseInstructor"));
    }
  }
}
```

The new statement in the OnModelCreating method configures the many-to-many join table:

- For the many-to-many relationship between the `Instructor` and `Course` entities, the code specifies the table and column names for the join table. Code First can configure the many-to-many relationship for you without this code, but if you don't call it, you will get default names such as `InstructorInstructorID` for the `InstructorID` column.
- `modelBuilder.Entity<Course>()`
- `    .HasMany(c => c.Instructors).WithMany(i => i.Courses)`
- `    .Map(t => t.MapLeftKey("CourseID")`
- `        .MapRightKey("InstructorID")`
  `        .ToTable("CourseInstructor"));`

The following code provides an example of how you could have used fluent API instead of attributes to specify the relationship between the `Instructor` and `OfficeAssignment` entities:

```
modelBuilder.Entity<Instructor>()
    .HasOptional(p => p.OfficeAssignment).WithRequired(p => p.Instructor);
```

For information about what "fluent API" statements are doing behind the scenes, see the Fluent API blog post.

Seed the Database with Test Data

Replace the code in the *Migrations\Configuration.cs* file with the following code in order to provide seed data for the new entities you've created.

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

    internal sealed class Configuration :
DbMigrationsConfiguration<SchoolContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }
```

```
        protected override void Seed(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("2005-09-01") }
            };


            students.ForEach(s => context.Students.AddOrUpdate(p =>
p.LastName, s));
            context.SaveChanges();

            var instructors = new List<Instructor>
            {
                new Instructor { FirstMidName = "Kim",     LastName =
"Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName =
"Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName =
"Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };
            instructors.ForEach(s => context.Instructors.AddOrUpdate(p =>
p.LastName, s));
            context.SaveChanges();

            var departments = new List<Department>
            {
                new Department { Name = "English",     Budget = 350000,
```

```
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Kapoor").ID }
            };
            departments.ForEach(s => context.Departments.AddOrUpdate(p =>
p.Name, s));
            context.SaveChanges();

            var courses = new List<Course>
            {
                new Course {CourseID = 1050, Title = "Chemistry",
Credits = 3,
                  DepartmentID = departments.Single( s => s.Name ==
"Engineering").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 4022, Title = "Microeconomics",
Credits = 3,
                  DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 4041, Title = "Macroeconomics",
Credits = 3,
                  DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 1045, Title = "Calculus",
Credits = 4,
                  DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 3141, Title = "Trigonometry",
Credits = 4,
                  DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 2021, Title = "Composition",
Credits = 3,
                  DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID,
                  Instructors = new List<Instructor>()
```

```
                },
                new Course {CourseID = 2042, Title = "Literature",
Credits = 4,
                  DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID,
                  Instructors = new List<Instructor>()
                },
            };
            courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID,
s));
            context.SaveChanges();

            var officeAssignments = new List<OfficeAssignment>
            {
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Fakhouri").ID,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Harui").ID,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Kapoor").ID,
                    Location = "Thompson 304" },
            };
            officeAssignments.ForEach(s =>
context.OfficeAssignments.AddOrUpdate(p => p.InstructorID, s));
            context.SaveChanges();

            AddOrUpdateInstructor(context, "Chemistry", "Kapoor");
            AddOrUpdateInstructor(context, "Chemistry", "Harui");
            AddOrUpdateInstructor(context, "Microeconomics", "Zheng");
            AddOrUpdateInstructor(context, "Macroeconomics", "Zheng");

            AddOrUpdateInstructor(context, "Calculus", "Fakhouri");
            AddOrUpdateInstructor(context, "Trigonometry", "Harui");
            AddOrUpdateInstructor(context, "Composition", "Abercrombie");
            AddOrUpdateInstructor(context, "Literature", "Abercrombie");

            context.SaveChanges();

            var enrollments = new List<Enrollment>
            {
                new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    Grade = Grade.A
                },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
```

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

```
                    StudentID = students.Single(s => s.LastName ==
"Justice").ID,
                    CourseID = courses.Single(c => c.Title ==
"Literature").CourseID,
                    Grade = Grade.B
                 }
            };

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

        void AddOrUpdateInstructor(SchoolContext context, string courseTitle,
string instructorName)
        {
            var crs = context.Courses.SingleOrDefault(c => c.Title ==
courseTitle);
            var inst = crs.Instructors.SingleOrDefault(i => i.LastName ==
instructorName);
            if (inst == null)
                crs.Instructors.Add(context.Instructors.Single(i =>
i.LastName == instructorName));
        }
    }
}
```

As you saw in the first tutorial, most of this code simply updates or creates new entity objects and loads sample data into properties as required for testing. However, notice how the `Course` entity, which has a many-to-many relationship with the `Instructor` entity, is handled:

```
var courses = new List<Course>
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
      DepartmentID = departments.Single( s => s.Name ==
"Engineering").DepartmentID,
      Instructors = new List<Instructor>()
    },
    ...
};
courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID, s));
context.SaveChanges();
```

When you create a `Course` object, you initialize the `Instructors` navigation property as an empty collection using the code `Instructors = new List<Instructor>()`. This makes it possible to add `Instructor` entities that are related to this `Course` by using the

`Instructors.Add` method. If you didn't create an empty list, you wouldn't be able to add these relationships, because the `Instructors` property would be null and wouldn't have an `Add` method. You could also add the list initialization to the constructor.

Add a Migration and Update the Database

From the PMC, enter the `add-migration` command (don't do the `update-database` command yet):

```
add-Migration ComplexDataModel
```

If you tried to run the `update-database` command at this point (don't do it yet), you would get the following error:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints, and that's what you have to do now. The generated code in the ComplexDataModel `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. Because there are already rows in the `Course` table when the code runs, the `AddColumn` operation will fail because SQL Server doesn't know what value to put in the column that can't be null. Therefore have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing `Course` rows will all be related to the "Temp" department after the `Up` method runs. You can relate them to the correct departments in the `Seed` method.

Edit the *<timestamp>_ComplexDataModel.cs* file, comment out the line of code that adds the DepartmentID column to the Course table, and add the following highlighted code (the commented line is also highlighted):

```
   CreateTable(
        "dbo.CourseInstructor",
        c => new
            {
                CourseID = c.Int(nullable: false),
                InstructorID = c.Int(nullable: false),
            })
        .PrimaryKey(t => new { t.CourseID, t.InstructorID })
        .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
        .ForeignKey("dbo.Instructor", t => t.InstructorID, cascadeDelete:
true)
        .Index(t => t.CourseID)
        .Index(t => t.InstructorID);

    // Create  a department for course to point to.
    Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp',
0.00, GETDATE())");
```

```
    //  default value for FK points to department created above.
    AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false,
defaultValue: 1));
    //AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false));

    AlterColumn("dbo.Course", "Title", c => c.String(maxLength: 50));
```

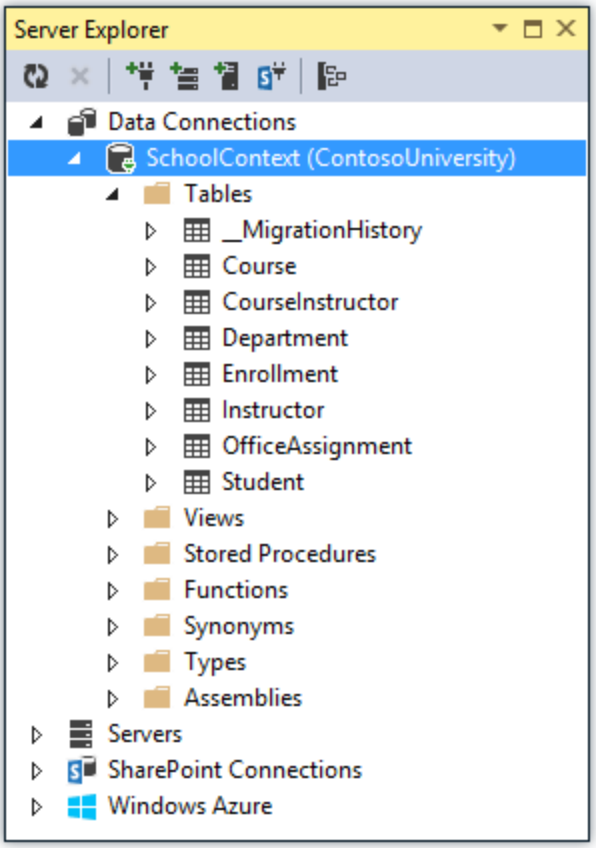
When the `Seed` method runs, it will insert rows in the `Department` table and it will relate existing `Course` rows to those new `Department` rows. If you haven't added any courses in the UI, you would then no longer need the "Temp" department or the default value on the `Course.DepartmentID` column. To allow for the possibility that someone might have added courses by using the application, you'd also want to update the `Seed` method code to ensure that all `Course` rows (not just the ones inserted by earlier runs of the `Seed` method) have valid `DepartmentID` values before you remove the default value from the column and delete the "Temp" department.

After you have finished editing the *<timestamp>_ComplexDataModel.cs* file, enter the `update-database` command in the PMC to execute the migration.

```
update-database
```

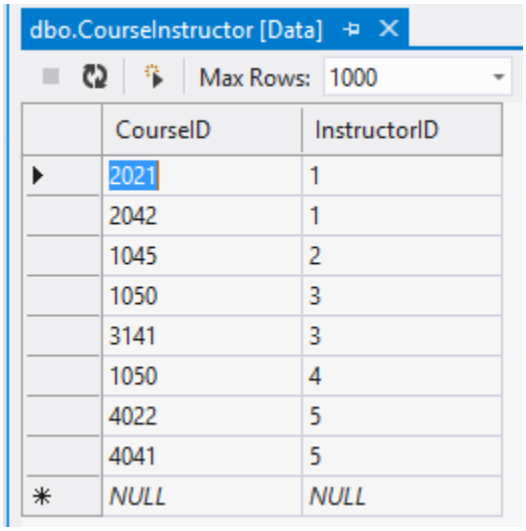
> **Note:** It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. The simplest approach is to rename the database in *Web.config* file. The following example shows the name changed to CU_Test:
>
> ```
>  <add name="SchoolContext" connectionString="Data
> Source=(LocalDb)\v11.0;Initial Catalog=CU_Test;Integrated
> Security=SSPI;"
>       providerName="System.Data.SqlClient" />
> ```
>
> With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see [How to Drop a Database from Visual Studio 2012].
>
> If that fails, another thing you can try is re-initialize the database by entering the following command in the PMC:
>
> ```
> update-database -TargetMigration:0
> ```

Open the database in **Server Explorer** as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have **Server Explorer** open from the earlier time, click the **Refresh** button.)



You didn't create a model class for the `CourseInstructor` table. As explained earlier, this is a join table for the many-to-many relationship between the `Instructor` and `Course` entities.

Right-click the `CourseInstructor` table and select **Show Table Data** to verify that it has data in it as a result of the `Instructor` entities you added to the `Course.Instructors` navigation property.



CourseID	InstructorID
2021	1
2042	1
1045	2
1050	3
3141	3
1050	4
4022	5
4041	5
NULL	*NULL*

Summary

You now have a more complex data model and corresponding database. In the following tutorial you'll learn more about different ways to access related data.