

Creating a complex data model - EF Core with ASP.NET Core MVC tutorial (5 of 10)

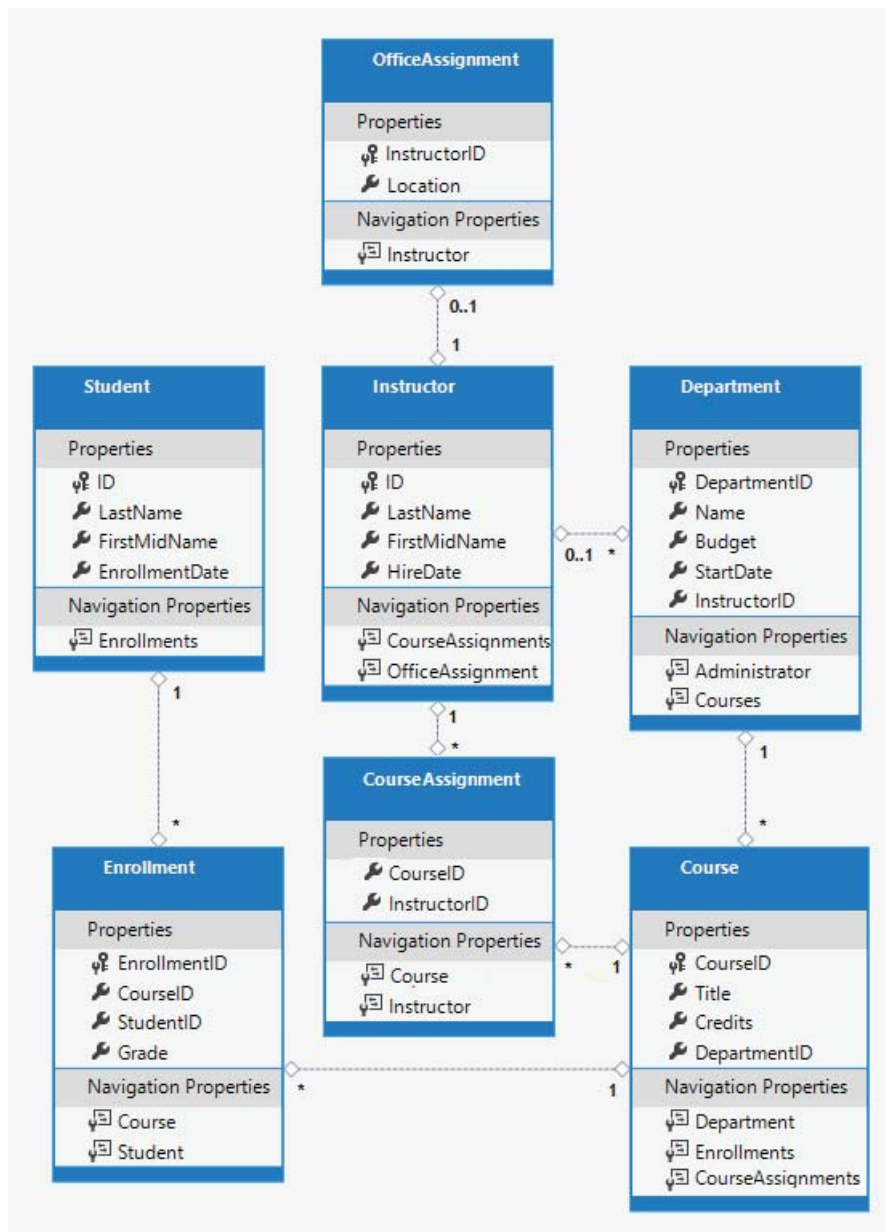
1/29/2018 • 30 min to read • [Edit Online](#)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core MVC web applications using Entity Framework Core and Visual Studio. For information about the tutorial series, see [the first tutorial in the series](#).

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.

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

The `DataType` attribute is used to specify a data type that's 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` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes don't provide any validation.

`DataType.Date` doesn't specify the format of the date that's 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)]
```

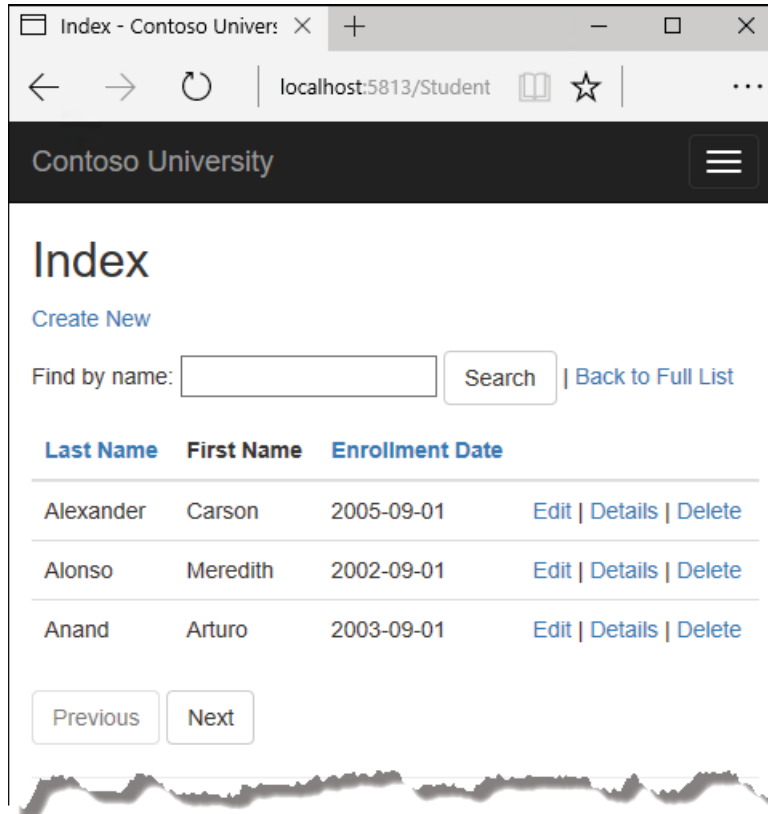
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

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.

For more information, see the [<input> tag helper documentation](#).

Run the app, go to the Students Index page 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 StringLength attribute

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 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 functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations 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 `database update` command ran that code.

The timestamp prefixed 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 app, select the **Students** tab, click **Create New**, and enter either name longer than 50 characters. When you click **Create**, client side validation shows an error message.

The screenshot shows a web browser window with the address bar at `localhost:5813/Student`. The page title is 'Contoso University'. The main heading is 'Create Student'. There are three input fields: 'LastName' with the value 'Davolio very long last name longer than !', 'FirstMidName' with the value 'Nancy very long first name longer than 5', and 'EnrollmentDate' with the value '2/15/2017'. Below the 'LastName' field is a red error message: 'The field LastName must be a string with a maximum length of 50.' Below the 'FirstMidName' field is a red error message: 'First name cannot be longer than 50 characters.' There is a 'Create' button at the bottom.

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're 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 ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

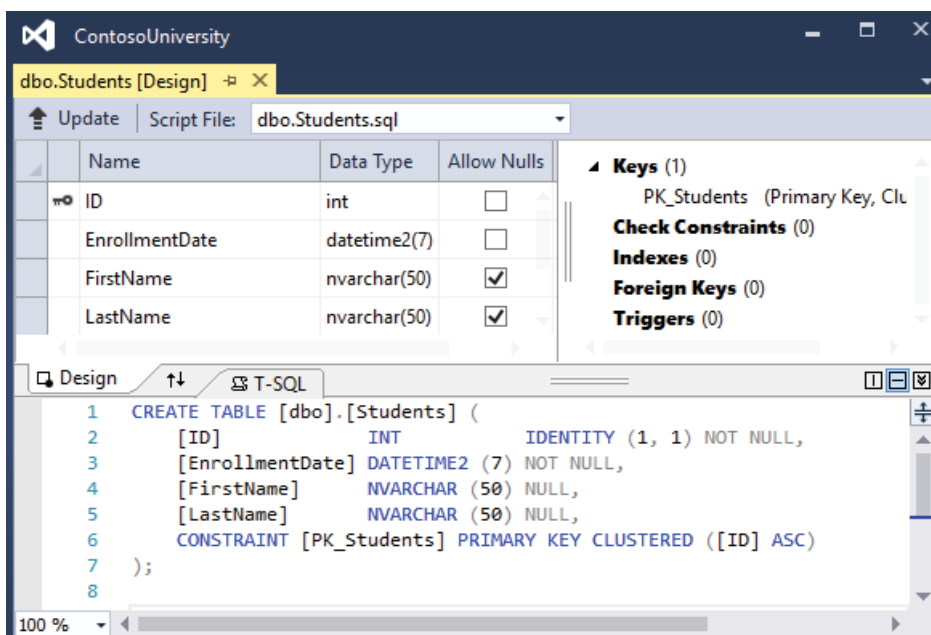
The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

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



Before you applied the first two migrations, the name columns were of type `nvarchar(MAX)`. They're now `nvarchar(50)` and the column name has changed from `FirstMidName` to `FirstName`.

NOTE

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

Final changes to the Student entity

Student
Properties
🔑 ID
🔑 LastName
🔑 FirstMidName
🔑 EnrollmentDate
Navigation Properties
🔗 Enrollments

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

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically 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

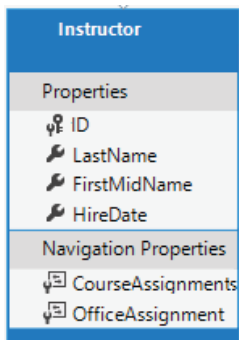
The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

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 ICollection<CourseAssignment> CourseAssignments { get; set; }
        public 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 `HireDate` attributes as follows:

```

[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]

```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

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

```

public ICollection<CourseAssignment> CourseAssignments { get; set; }

```

If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity (which may be null if no office is assigned).

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

Create the OfficeAssignment entity

OfficeAssign...
Properties
🔑 InstructorID
📍 Location
Navigation Properties
🔗 Instructor

Create `Models/OfficeAssignment.cs` with the following code:

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

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

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

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 `classnameID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
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 other than `classnameID` or `ID`.

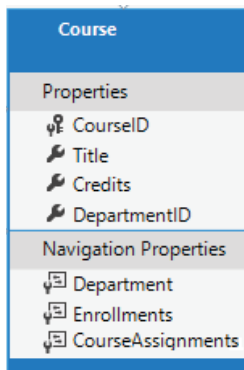
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

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. The changes are highlighted.

```
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 Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { 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're needed and creates **shadow properties** for them. 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, 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.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see [Generated Properties](#).

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 Department Department { get; set; }
```








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

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

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained [later](#)):

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create the Department entity

Department
Properties
 DepartmentID
 Name
 Budget
 StartDate
 InstructorID
Navigation Properties
 Administrator
 Courses

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 Instructor Administrator { get; set; }
        public 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 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.

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 Instructor Administrator { get; set; }

```

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

```

public 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, EF would configure a cascade delete rule to delete the instructor when you delete the department, which isn't what you want to have happen. If your business rules required the `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<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

Modify the Enrollment entity

Enrollment
Properties
• EnrollmentID
• CourseID
• StudentID
• Grade
Navigation Properties
• Course
• Student

In `Models/Enrollment.cs`, replace the code you added earlier with the following code:

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

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

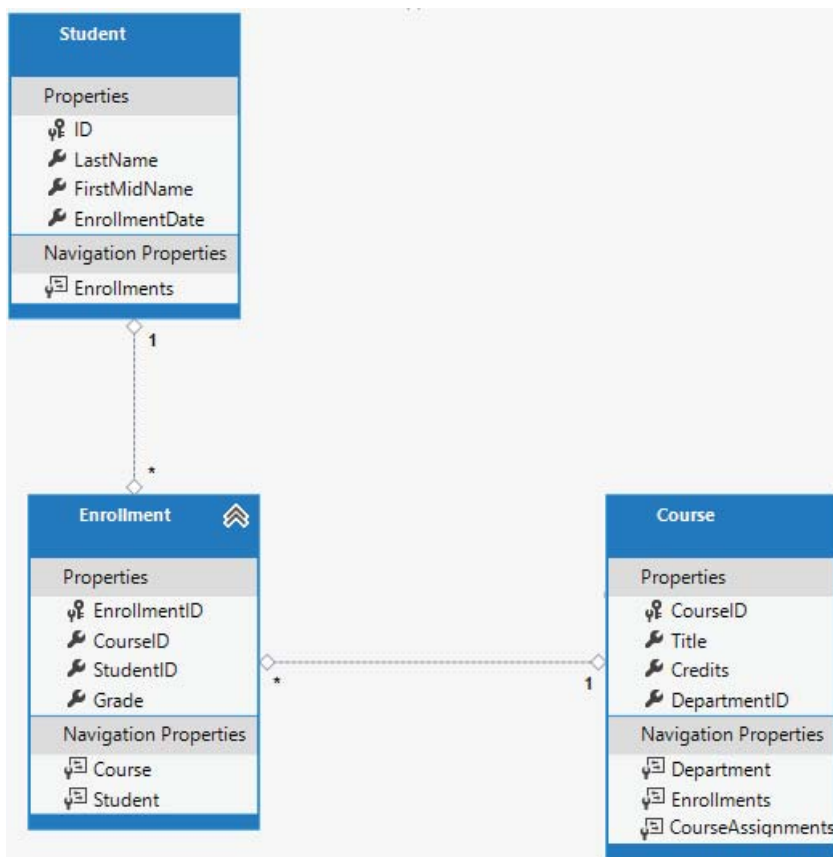
An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public 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. "With payload" 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 for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

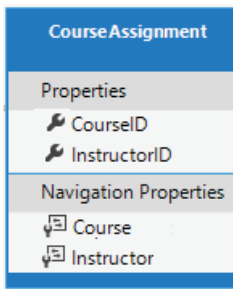


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

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 be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see the [discussion in the EF Core GitHub repository](#).)

The CourseAssignment entity



Create `Models/CourseAssignment.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 CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see [Indexes](#).

Update the database context

Add the following highlighted code to the `Data/SchoolContext.cs` file:


```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

Fluent API alternative to attributes

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the [EF Core documentation](#):

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}

```

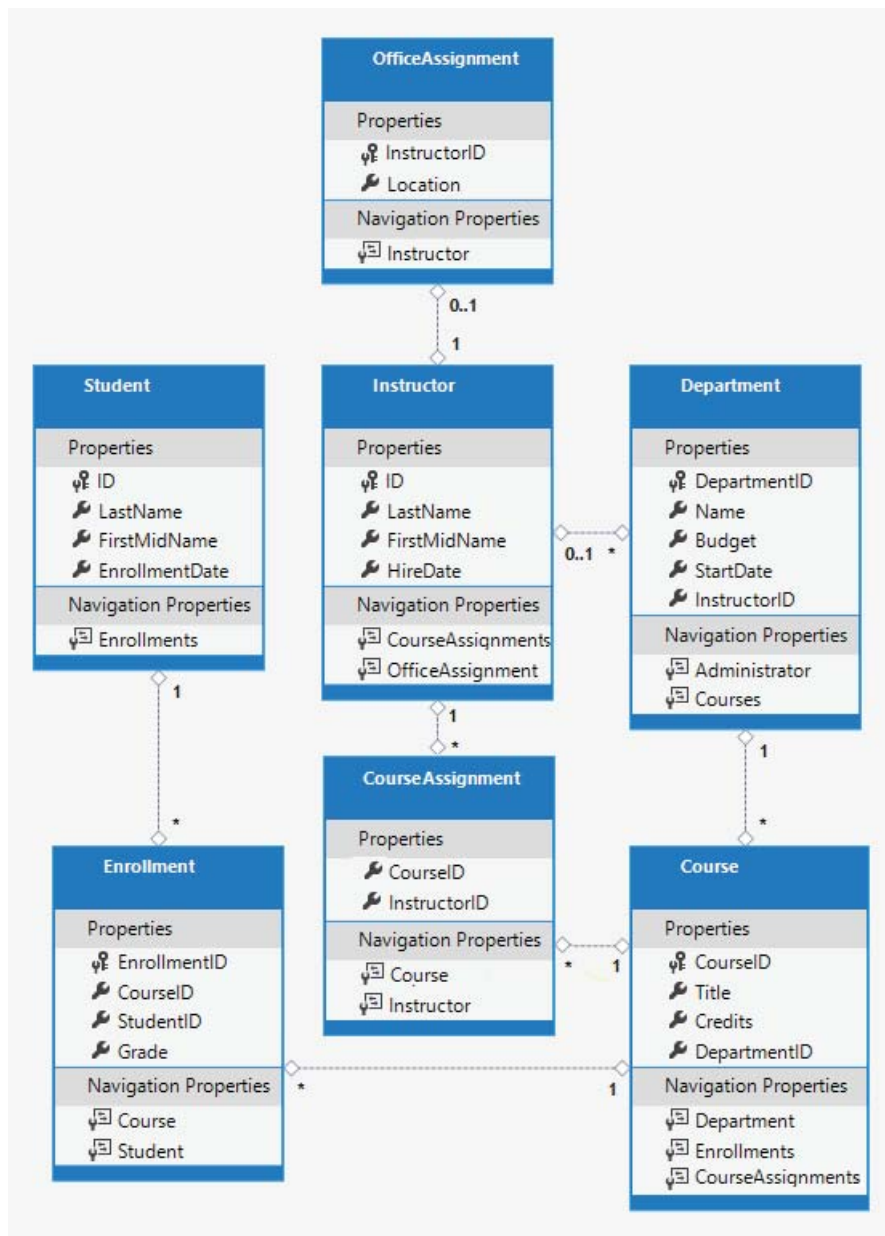
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied 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. If you do use both, note that wherever there's a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see [Methods of configuration](#).

Entity Diagram Showing Relationships

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



Besides 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.

Seed the Database with Test Data

Replace the code in the **Data/DbInitializer.cs** file with the following code in order to provide seed data for the new entities you've created.

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
```

```

{
    public static void Initialize(SchoolContext context)
    {
        //context.Database.EnsureCreated();

        // Look for any students.
        if (context.Students.Any())
        {
            return;    // DB has been seeded
        }

        var students = new 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") }
        };

        foreach (Student s in students)
        {
            context.Students.Add(s);
        }
        context.SaveChanges();

        var instructors = new 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") }
        };

        foreach (Instructor i in instructors)
        {
            context.Instructors.Add(i);
        }
        context.SaveChanges();

        var departments = new 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 }
    };

    foreach (Department d in departments)
    {
        context.Departments.Add(d);
    }
    context.SaveChanges();

    var courses = new Course[]
    {
        new Course {CourseID = 1050, Title = "Chemistry", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
    };

    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }
    context.SaveChanges();

    var officeAssignments = new 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" },
    };

    foreach (OfficeAssignment o in officeAssignments)
    {
        context.OfficeAssignments.Add(o);
    }
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        }
    }

```

```

    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    }
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}

context.SaveChanges();

var enrollments = new 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();
}
}
}

```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` 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. The generated code in the `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. If there are already rows in the `Course` table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you 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.

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(  
    name: "Title",  
    table: "Course",  
    maxLength: 50,  
    nullable: true,  
    oldClrType: typeof(string),  
    oldNullable: true);  
  
//migrationBuilder.AddColumn<int>(  
//    name: "DepartmentID",  
//    table: "Course",  
//    nullable: false,  
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the `Department` table:

```

migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
                SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
    GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);

```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string and update the database

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in *appsettings.json* to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

```

{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },

```

Save your change to *appsettings.json*.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer (SSOX)** or the `database drop` CLI command:

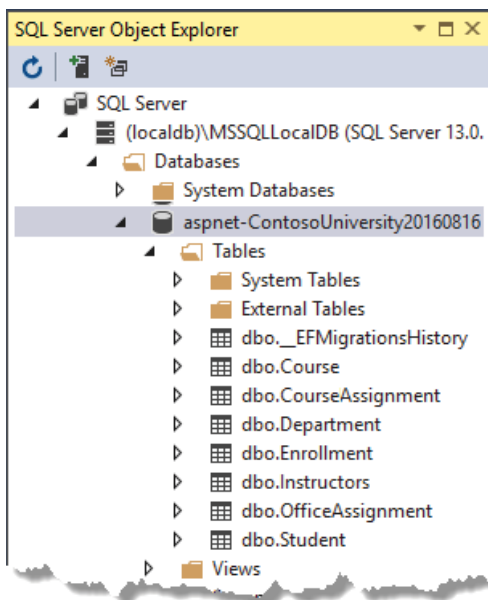
```
dotnet ef database drop
```

After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
dotnet ef database update
```

Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

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



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.

The screenshot shows the ContosoUniversity application window. The 'dbo.CourseAssignment [Data]' table is displayed, showing a list of course assignments. The table has two columns: CourseID and InstructorID. The data is as follows:

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 how to access related data.

[PREVIOUS](#)[NEXT](#)