

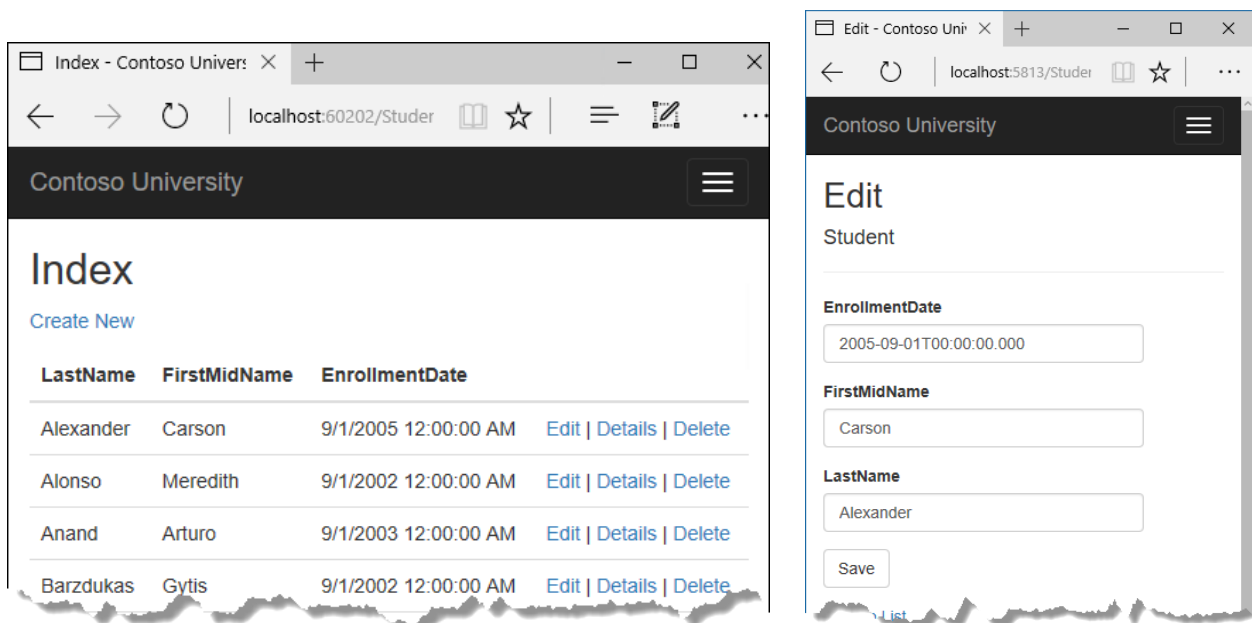
# Entity Framework Core

L'application est un site web pour une université Contoso fictive. Il comprend des fonctionnalités telles que l'admission des étudiants, la création des cours et les affectations des formateurs. Ce tutorial explique comment générer à partir de zéro l'application Contoso University. L'application utilise Entity Framework Core. EF Core 2.0 est la dernière version d'EF mais elle ne dispose pas encore de toutes les fonctionnalités d'EF 6.x.

## Application web Contoso University

L'application que vous allez générer dans ce tutoriel est un site web simple d'université.

Les utilisateurs peuvent afficher et mettre à jour les informations relatives aux étudiants, aux cours et aux formateurs. Voici quelques écrans que vous allez créer.

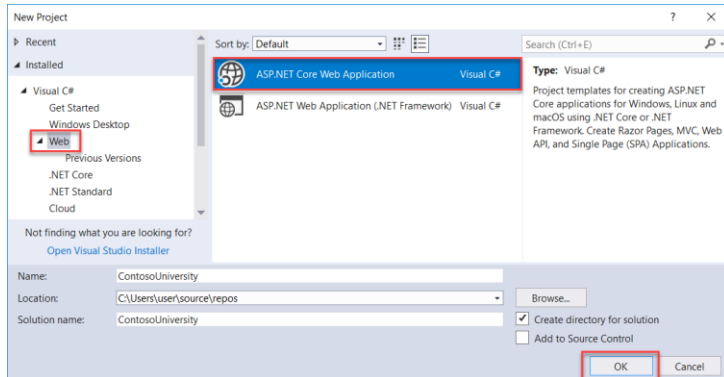


Le style d'interface utilisateur de ce site a été maintenu proche de ce qui est généré par les modèles intégrés, afin que le tutoriel puisse principalement se concentrer sur la façon d'utiliser Entity Framework.

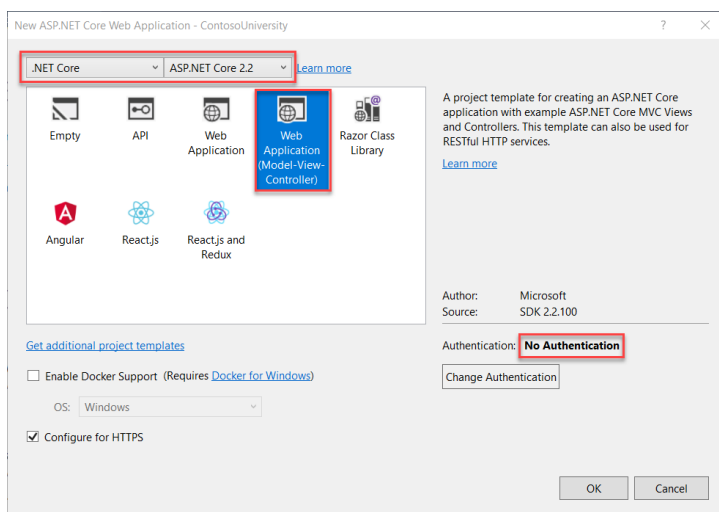
## Créer une application web ASP.NET Core MVC

Ouvrez Visual Studio et créez un nouveau projet web C# ASP.NET Core, nommé « ContosoUniversity ».

- Dans le menu **Fichier**, sélectionnez **Nouveau > Projet**.
- Dans le volet gauche, sélectionnez **Installé > Visual C# > Web**.
- Sélectionnez le modèle de projet **Application web ASP.NET Core**.
- Entrez **ContosoUniversity** comme nom et cliquez sur **OK**.



Attendez que la boîte de dialogue **Nouvelle application web ASP.NET Core (.NET Core)** s'affiche.



- Sélectionnez **ASP.NET Core 2.2**, et le modèle **Application web (Model-View-Controller)**.

**Remarque :** Ce tutoriel nécessite ASP.NET Core 2.2 et EF Core 2.0 ou version ultérieure.

- Vérifiez que le paramètre **Authentification** a pour valeur **Aucune authentification**.
- Cliquez sur **OK**.

# Configurer le style du site

Quelques changements simples configureront le menu, la disposition et la page d'accueil du site.

Ouvrez *Views/Shared/\_Layout.cshtml* et apportez les modifications suivantes :

- Remplacez chaque occurrence de « ContosoUniversity » par « Contoso University ». Il y a trois occurrences.
- Ajoutez des entrées de menu pour **About**, **Students**, **Courses**, **Instructors**, et **Departments**, et supprimez l'entrée de menu **Privacy**.

Les modifications sont mises en surbrillance.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - Contoso University</title>

  <environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
  </environment>
  <environment exclude="Development">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
      asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
      asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
  </environment>
</head>
<body>
  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">Contoso University</a>
      </div>
      <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
          <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
          <li class="nav-item"><a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="About">About</a></li>
          <li class="nav-item"><a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-action="Index">Students</a></li>
          <li class="nav-item"><a class="nav-link text-dark" asp-area="" asp-controller="Courses" asp-action="Index">Courses</a></li>
          <li class="nav-item"><a class="nav-link text-dark" asp-area="" asp-controller="Instructors" asp-action="Index">Instructors</a></li>
          <li class="nav-item"><a class="nav-link text-dark" asp-area="" asp-controller="Departments" asp-action="Index">Departments</a></li>
        </ul>
      </div>
    </div>
  </nav>

  <partial name="CookieConsentPartial" />

  <div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; 2019 - Contoso University - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a></p>
    </footer>
  </div>

  <environment include="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
  </environment>
  <environment exclude="Development">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
      asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
      asp-fallback-test="window.jQuery"
      crossorigin="anonymous"
      integrity="sha384-Ts5Qib877qvYJh04KfkvA9v1WsNxZUpnICJ7L2MCMIP99mGCD8W6GICPD7Txa">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
      asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
      asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
      crossorigin="anonymous"
      integrity="sha384-Tc5Qib877qvYJh04KfkvA9v1WsNxZUpnICJ7L2MCMIP99mGCD8W6GICPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
  </environment>

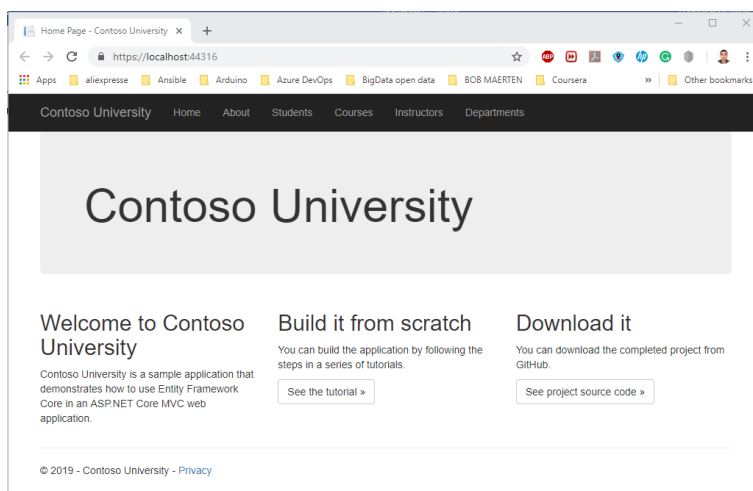
  @RenderSection("Scripts", required: false)
</body>
</html>
```

Dans *Views/Home/Index.cshtml*, remplacez le contenu du fichier par le code suivant, afin de remplacer le texte relatif à ASP.NET et MVC par le texte relatif à cette application :

```
@{
    ViewData["Title"] = "Home Page" ;
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of
        tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-
        mvc/intro.html">See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default
        href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
        mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

Appuyez sur Ctrl+F5 pour exécuter le projet ou choisissez **Déboguer > Exécuter sans débogage** dans le menu. Vous voyez la page d'accueil avec des onglets pour les pages que vous allez créer dans ce tutoriel.



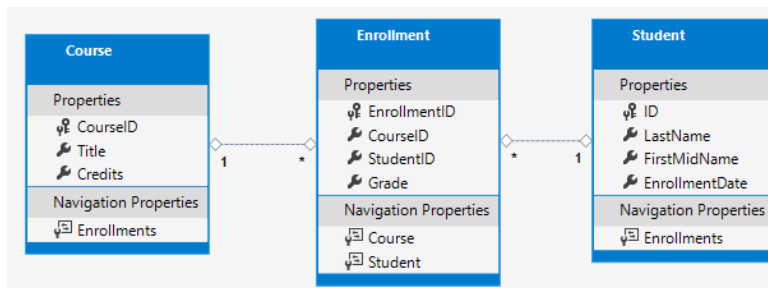
## À propos des packages NuGet EF Core

Pour ajouter la prise en charge d'EF Core à un projet, installez le fournisseur de bases de données que vous souhaitez cibler. Ce tutoriel utilise SQL Server et le package de fournisseur est `Microsoft.EntityFrameworkCore.SqlServer`. Ce package étant inclus dans le méta paquet `Microsoft.AspNetCore.App`, vous n'avez pas besoin de référencer le package si votre application comporte une référence pour le package `Microsoft.AspNetCore.App`.

Ce package et ses dépendances (`Microsoft.EntityFrameworkCore` et `Microsoft.EntityFrameworkCore.Relational`) fournissent la prise en charge du runtime pour EF. Vous ajouterez un package d'outils ultérieurement, dans le tutoriel Migrations.

## Créer le modèle de données

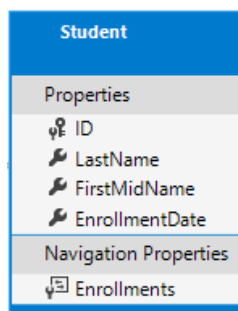
Ensuite, vous allez créer des classes d'entités pour l'application Contoso University. Vous commencerez avec les trois entités suivantes.



Il existe une relation un-à-plusieurs entre les entités `Student` et `Enrollment`, et une relation un-à-plusieurs entre les entités `Course` et `Enrollment`. En d'autres termes, un étudiant peut être inscrit dans un nombre quelconque de cours et un cours peut avoir un nombre quelconque d'élèves inscrits.

Dans les sections suivantes, vous allez créer une classe pour chacune de ces entités.

### Entité Student



Dans le dossier *Models*, créez un fichier de classe nommé *Student.cs* et remplacez le code du modèle par le code suivant.

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

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







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

La propriété `ID` devient la colonne de clé primaire de la table de base de données qui correspond à cette classe. Par défaut, Entity Framework interprète une propriété nommée `ID` ou `classnameID` comme clé primaire.

La propriété `Enrollments` est une propriété de navigation. Les propriétés de navigation contiennent d'autres entités qui sont associées à cette entité. Dans ce cas, la propriété `Enrollments` d'un `Student` entity contient toutes les entités `Enrollment` associées à l'entité `Student`. En d'autres termes, si une ligne `Student` donnée dans la base de données a deux lignes `Enrollment` associées (lignes qui contiennent la valeur de clé primaire de cet étudiant dans la colonne de clé étrangère `StudentID`), la propriété de navigation `Enrollments` de cette entité `Student` contiendra ces deux entités `Enrollment`.

Si une propriété de navigation peut contenir plusieurs entités (comme dans des relations plusieurs à plusieurs ou un -à-plusieurs), son type doit être une liste dans laquelle les entrées peuvent être ajoutées, supprimées et mises à jour, telle que `ICollection<T>`. Vous pouvez spécifier `ICollection<T>` ou un type tel que `List<T>` ou `HashSet<T>`. Si vous spécifiez `ICollection<T>`, EF crée une collection `HashSet<T>` par défaut.

## Entité Enrollment

Enrollment	
Properties	
	EnrollmentID
	CourseID
	StudentID
	Grade
Navigation Properties	
	Course
	Student

Dans le dossier *Models*, créez *Enrollment.cs* et remplacez le code existant par le code suivant :

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

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

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

La propriété `EnrollmentID` sera la clé primaire. Cette entité utilise le modèle `classnameID` à la place de `ID` par lui-même, comme vous l'avez vu dans l'entité `Student`. En général, vous choisissez un modèle et l'utilisez dans tout votre modèle de données. Ici, la variante illustre que vous pouvez utiliser l'un ou l'autre modèle. Dans une prochaine section, vous verrez comment l'utilisation de l'`ID` sans `classname` simplifie l'implémentation de l'héritage dans le modèle de données.

La propriété `Grade` est un `enum`. Le point d'interrogation après la déclaration de type `Grade` indique que la propriété `Grade` est nullable. Une note (Grade) qui a la valeur Null est différente d'une note égale à zéro : la valeur Null signifie qu'une note n'est pas connue ou n'a pas encore été affectée.

La propriété `StudentID` est une clé étrangère et la propriété de navigation correspondante est `Student`. Une entité `Enrollment` est associée à une entité `Student`, donc la propriété peut contenir uniquement une entité `Student` unique (contrairement à la propriété de navigation `Student.Enrollments` que vous avez vue précédemment, qui peut contenir plusieurs entités `Enrollment`).

La propriété `CourseID` est une clé étrangère et la propriété de navigation correspondante est `Course`. Une entité `Enrollment` est associée à une entité `Course`.

Entity Framework interprète une propriété comme une propriété de clé étrangère si elle est nommée `<navigation property name><primary key property name>` (par exemple, `StudentID` pour la propriété de navigation `Student`, puisque la clé primaire de l'entité `Student` est `ID`). Les propriétés de clé étrangère peuvent également être nommées simplement `<primary key property name>` (par exemple, `CourseID`, puisque la clé primaire de l'entité `Course` est `CourseID`).

## Entité Course

Course
Properties
CourseID
Title
Credits
Navigation Properties
Enrollments

Dans le dossier *Models*, créez *cs* et remplacez le code existant par le code suivant :

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

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

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



La propriété `Enrollments` est une propriété de navigation. Une entité `Course` peut être associée à un nombre quelconque d'entités `Enrollment`.

Nous fournirons plus de détails sur l'attribut `DatabaseGenerated` dans la suite de ce tutorial. En fait, cet attribut vous permet d'entrer la clé primaire pour le cours plutôt que de laisser la base de données la générer.

## Créer le contexte de base de données

La classe principale qui coordonne les fonctionnalités d'Entity Framework pour un modèle de données spécifié est la classe de contexte de base de données. Vous créez cette classe en dérivant de la classe `Microsoft.EntityFrameworkCore.DbContext`. Dans votre code, vous spécifiez les entités qui sont incluses dans le modèle de données. Vous pouvez également personnaliser un certain comportement d'Entity Framework. Dans ce projet, la classe est nommée `SchoolContext`.

Dans le dossier du projet, créez un dossier nommé *Data*.

Dans ce dossier *Data*, créez un nouveau fichier de classe nommé *SchoolContext.cs* et remplacez le code du modèle par le code suivant :

```
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; }
    }
}
```

Ce code crée une propriété `DbSet` pour chaque jeu d'entités. Dans la terminologie Entity Framework, un jeu d'entités correspond généralement à une table de base de données, et une entité correspond à une ligne dans la table.

Vous pourriez avoir omis les instructions `DbSet<Enrollment>` et `DbSet<Course>`, et cela fonctionnerait de la même façon. Entity Framework les inclurait implicitement, car l'entité `Student` référence l'entité `Enrollment`, et l'entité `Enrollment` référence l'entité `Course`.

Quand la base de données est créée, EF crée des tables dont les noms sont identiques aux noms de propriété `DbSet`. Les noms de propriété pour les collections sont généralement pluriels (Students plutôt que Student), mais les développeurs ne sont pas tous d'accord sur la nécessité d'utiliser des noms de tables au pluriel. Pour ce tutoriel, vous remplacerez le comportement par défaut en spécifiant des noms de tables au singulier dans le contexte `DbContext`. Pour ce faire, ajoutez le code en surbrillance suivant après la dernière propriété `DbSet`.

```
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; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

## Inscrire SchoolContext

ASP.NET Core implémente l'injection de dépendance par défaut. Des services (tels que le contexte de base de données EF) sont inscrits avec l'injection de dépendance au démarrage de l'application. Ces services sont affectés aux composants qui les nécessitent (tels que les contrôleurs MVC) par le biais de paramètres de constructeur. Vous verrez le

code de constructeur de contrôleur qui obtient une instance de contexte plus loin dans ce tutoriel.

Pour inscrire `SchoolContext` en tant que service, ouvrez *Startup.cs* et ajoutez les lignes en surbrillance à la méthode `ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

Le nom de la chaîne de connexion est transmis au contexte en appelant une méthode sur un objet `DbContextOptionsBuilder`. Pour le développement local, le système de configuration ASP.NET Core lit la chaîne de connexion à partir du fichier *appsettings.json*.

Ajoutez des instructions `using` pour les espaces de noms `ContosoUniversity.Data` et `Microsoft.EntityFrameworkCore`, puis générez le projet.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

Ouvrez le fichier *appsettings.json* et ajoutez une chaîne de connexion comme indiqué dans l'exemple suivant.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

## SQL Server Express LocalDB

La chaîne de connexion spécifie une base de données SQL Server LocalDB. LocalDB est une version allégée du moteur de base de données SQL Server Express. Elle est destinée au développement d'applications, et non à une utilisation en production. LocalDB démarre à la demande et s'exécute en mode utilisateur, ce qui n'implique aucune configuration complexe. Par défaut, LocalDB crée des fichiers de base de données *.mdf* dans le répertoire `C:/Users/<user>`.

## Initialiser la base de données avec des données de test

Entity Framework créera une base de données vide pour vous. Dans cette section, vous écrivez une méthode qui est appelée après la création de la base de données pour la remplir avec des données de test.

Là, vous allez utiliser la méthode `EnsureCreated` pour créer automatiquement la base de données. Dans une section ultérieure, vous verrez comment traiter les modifications des modèles à l'aide des migrations Code First pour modifier le schéma de base de données au lieu de supprimer et de recréer la base de données.

Dans le dossier *Data*, créez un nouveau fichier de classe nommé *DbInitializer.cs* et remplacez le code de modèle par le code suivant, qui entraîne la création d'une base de données, si nécessaire, et charge les données de test dans la nouvelle base de données.

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    0 references
    public static class DbInitializer
    {
        0 references | 0 exceptions
        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("2005-09-01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();
        }
    }
}
```

```

var enrollments = new Enrollment[]
{
    new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
    new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
    new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
    new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
    new Enrollment{StudentID=3,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
    new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
    new Enrollment{StudentID=6,CourseID=1045},
    new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
    context.Enrollments.Add(e);
}
context.SaveChanges();
}
}

```

Le code vérifie si des étudiants figurent dans la base de données et, dans la négative, il suppose que la base de données est nouvelle et doit être initialement peuplée avec des données de test. Il charge les données de test dans les tableaux plutôt que dans les collections `List<T>` afin d'optimiser les performances.

Dans *Program.cs*, modifiez la méthode `Main` pour effectuer les opérations suivantes au démarrage de l'application :

- Obtenir une instance de contexte de base de données à partir du conteneur d'injection de dépendance.
- Appeler la méthode de remplissage initial, en lui transmettant le contexte.
- Supprimer le contexte une fois l'exécution de la méthode de peuplement initial terminée.

```

public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}

```

Ajoutez des instructions `using` :

```
using Microsoft.Extensions.DependencyInjection;  
using ContosoUniversity.Data;
```


Dans d'autres tutoriels, vous pouvez voir un code similaire dans la méthode `Configure`, dans *Startup.cs*. Nous vous recommandons d'utiliser la méthode `Configure` uniquement pour configurer le pipeline de la requête. Le code de démarrage d'application doit être figuré dans la méthode `Main`.

À présent, la première fois que vous exécutez l'application, la base de données est créée et initialement peuplée avec les données de test. Chaque fois que vous changez votre modèle de données, vous pouvez supprimer la base de données, mettre à jour votre méthode de peuplement initial et repartir de la même façon avec une nouvelle base de données. Dans la suite de ce tutoriel, vous verrez comment modifier la base de données quand le modèle de données change, sans supprimer et recréer la base de données.

## Créer un contrôleur et des vues

Ensuite, vous utiliserez le moteur de génération de modèles automatique dans Visual Studio pour ajouter un contrôleur MVC et les vues qu'utilisera EF pour exécuter des requêtes de données et enregistrer les données.

La création automatique de vues et de méthodes d'action CRUD porte le nom de génération de modèles automatique. La génération de modèles automatique diffère de la génération de code dans la mesure où le code obtenu par génération de modèles automatique est un point de départ que vous pouvez modifier pour prendre en compte vos propres exigences, tandis qu'en général vous ne modifiez pas le code généré. Lorsque vous avez besoin de personnaliser le code généré, vous utilisez des classes partielles ou vous régénérez le code en cas de changements.

- Cliquez avec le bouton droit sur le dossier **Contrôleurs** dans l'**Explorateur de solutions**, puis sélectionnez **Ajouter > Nouvel élément généré automatiquement** 

Si la boîte de dialogue **Ajouter des dépendances MVC** apparaît :

- Sélectionnez **ADD**, puis suivez à nouveau les étapes pour ajouter un contrôleur.
- Dans la boîte de dialogue **Ajouter un modèle automatique** :

- Sélectionnez **Contrôleur MVC avec vues, utilisant Entity Framework**.
- Cliquez sur **Ajouter**. La boîte de dialogue **Ajouter un contrôleur MVC avec vues, utilisant Entity Framework** s'affiche.

Add MVC Controller with views, using Entity Framework

Model class: Student (ContosoUniversity.Models)

Data context class: SchoolContext (ContosoUniversity.Data)

Views:

☒ Generate views

☒ Reference script libraries

☒ Use a layout page:

(Leave empty if it is set in a Razor \_viewstart file)

Controller name: StudentsController

Add Cancel

- Dans **Classe de modèle**, sélectionnez **Student**.
- Dans **Classe du contexte de données**, sélectionnez **SchoolContext**.
- Acceptez la valeur par défaut **StudentsController** comme nom.
- Cliquez sur **Ajouter**.

Lorsque vous cliquez sur **Ajouter**, le moteur de génération de modèles automatique de Visual Studio crée un fichier *StudentsController.cs* et un ensemble de vues (fichiers *.cshtml*) qui fonctionnent avec le contrôleur.



(Le moteur de génération de modèles automatique peut également créer le contexte de base de données pour vous, si vous ne l'avez pas déjà créé manuellement, comme vous l'avez fait précédemment pour ce tutoriel. Vous pouvez spécifier une nouvelle classe de contexte dans la zone **Ajouter un contrôleur** en cliquant sur le signe plus à droite de **Classe du contexte de données**. Visual Studio crée ensuite votre classe `DbContext` ainsi que le contrôleur et les vues.)

Vous pouvez remarquer que le contrôleur accepte un `SchoolContext` comme paramètre de constructeur.

```

namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
    }
}

```

L'injection de dépendance ASP.NET Core s'occupe de la transmission d'une instance de `SchoolContext` dans le contrôleur. Vous avez configuré cela dans le fichier *Startup.cs* précédemment.

Le contrôleur contient une méthode d'action `Index`, qui affiche tous les étudiants dans la base de données. La méthode obtient la liste des étudiants du jeu d'entités `Students` en lisant la propriété `Students` de l'instance de contexte de base de données :

```

public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}

```

Vous découvrirez les éléments de programmation asynchrones dans ce code, plus loin dans ce tutoriel.

La vue *Views/Students/Index.cshtml* affiche cette liste dans une table :



```

@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

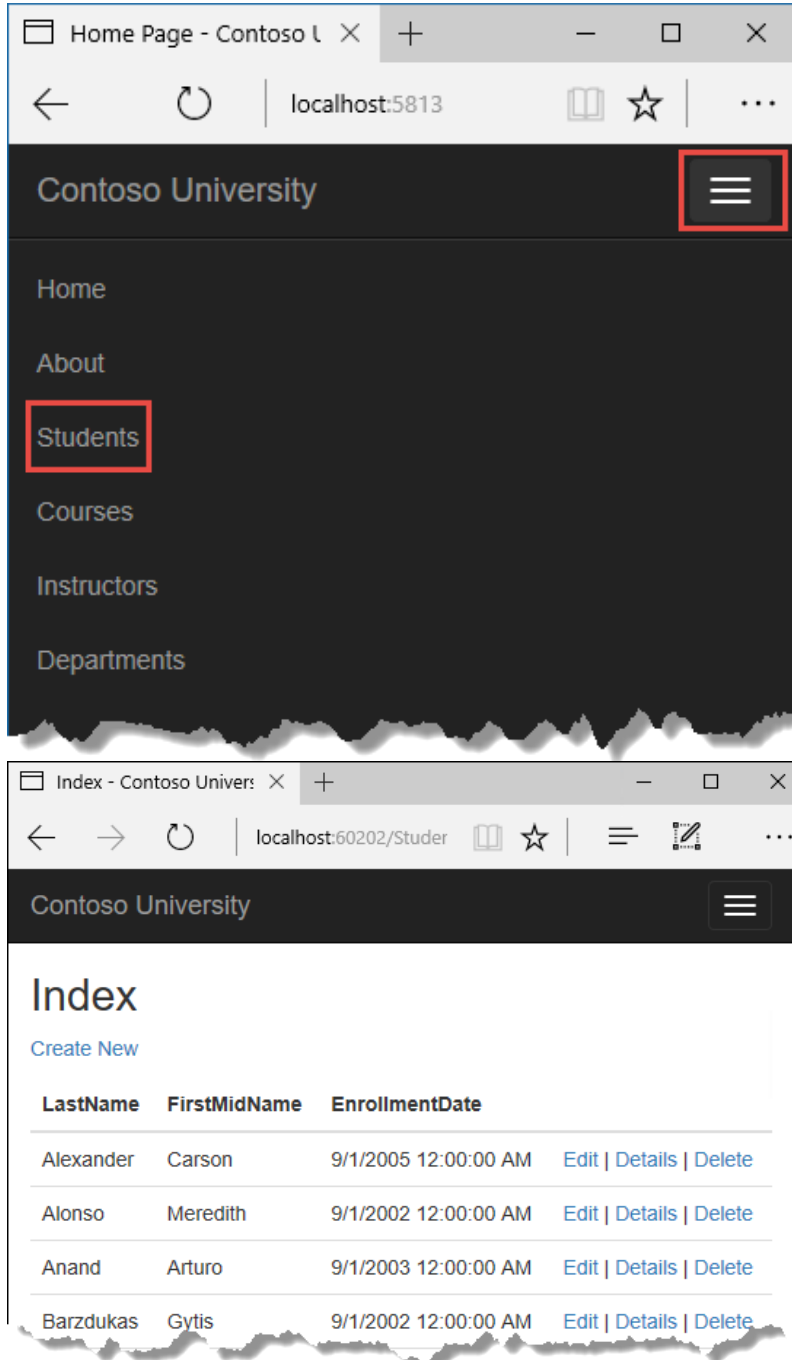
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.EnrollmentDate)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

Appuyez sur Ctrl+F5 pour exécuter le projet ou choisissez **Déboguer > Exécuter sans débogage** dans le menu.

Cliquez sur l'onglet Students pour afficher les données de test que la méthode `DbInitializer.Initialize` a insérées. Selon l'étréitresse de votre fenêtre de navigateur, vous verrez le lien de l'onglet `Student` en haut de la page ou vous devrez cliquer sur l'icône de navigation dans le coin supérieur droit pour afficher le lien.



## Afficher la base de données

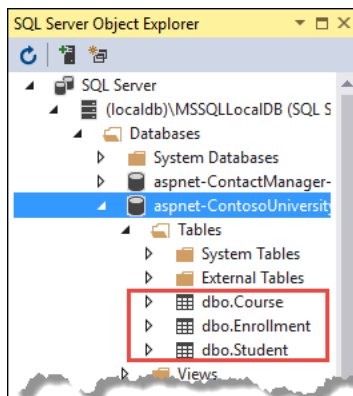
Lorsque vous avez démarré l'application, la méthode `DbInitializer.Initialize` appelle `EnsureCreated`. EF a vu qu'il n'y avait pas de base de données et en a donc créé une, puis le reste du code de la méthode `Initialize` a rempli la base de données avec des données. Vous pouvez utiliser l'**Explorateur d'objets SQL Server** (SSOX) pour afficher la base de données dans Visual Studio.

Fermez le navigateur.

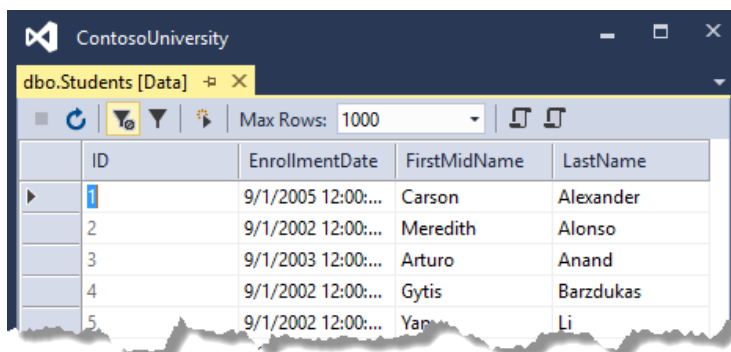
Si la fenêtre SSOX n'est pas déjà ouverte, sélectionnez-la dans le menu **Affichage** de Visual Studio.

Dans SSOX, cliquez sur **(localdb)\MSSQLLocalDB > Bases de données**, puis cliquez sur l'entrée pour le nom de la base de données qui se trouve dans la chaîne de connexion, dans votre fichier *appsettings.json*.

Développez le nœud **Tables** pour afficher les tables de votre base de données.



Cliquez avec le bouton droit sur la table **Student** et cliquez sur **Afficher les données** pour voir les colonnes qui ont été créées et les lignes qui ont été insérées dans la table.



	ID	EnrollmentDate	FirstMidName	LastName
▶	1	9/1/2005 12:00:...	Carson	Alexander
	2	9/1/2002 12:00:...	Meredith	Alonso
	3	9/1/2003 12:00:...	Arturo	Anand
	4	9/1/2002 12:00:...	Gytis	Barzdukas
	5	9/1/2002 12:00:...	Yan	Li

Les fichiers de base de données *.mdf* et *.ldf* se trouvent dans le dossier *C:\Utilisateurs\*.

Étant donné que vous appelez `EnsureCreated` dans la méthode d'initialiseur qui s'exécute au démarrage de l'application, vous pouvez maintenant apporter une modification à la classe `Student`, supprimer la base de données ou réexécuter l'application, et la base de données serait automatiquement recrée conformément à votre modification. Par exemple, si vous ajoutez une propriété `EmailAddress` à la classe `Student`, vous voyez une nouvelle colonne `EmailAddress` dans la table recrée.

## Conventions

La quantité de code que vous deviez écrire pour qu'Entity Framework puisse créer une base de données complète pour vous est minimale en raison de l'utilisation de conventions ou d'hypothèses effectuées par Entity Framework.

- Les noms des propriétés `DbSet` sont utilisés comme noms de tables. Pour les entités non référencées par une propriété `DbSet`, les noms des classes d'entités sont utilisés comme noms de tables.
- Les noms des propriétés d'entités sont utilisés comme noms de colonnes.
- Les propriétés d'entité nommées `ID` ou `classnameID` sont reconnues comme propriétés de clé primaire.
- Une propriété est interprétée comme propriété de clé étrangère si elle se nomme (par exemple `StudentID` pour la propriété de navigation `Student`, puisque la clé primaire de l'entité `Student` est `ID`). Les propriétés de clé étrangère peuvent également être nommées simplement (par exemple, `EnrollmentID`, puisque la clé primaire de l'entité `Enrollment` est `EnrollmentID`).

Le comportement conventionnel peut être remplacé. Par exemple, vous pouvez spécifier explicitement les noms de tables, comme vous l'avez vu précédemment dans ce tutoriel. De plus, vous pouvez définir des noms de colonne et définir une propriété quelconque en tant que clé primaire ou clé étrangère, comme vous le verrez dans une section ultérieure.

## Code asynchrone

La programmation asynchrone est le mode par défaut pour ASP.NET Core et EF Core.

Un serveur web a un nombre limité de threads disponibles et, dans les situations de forte charge, tous les threads disponibles peuvent être utilisés. Quand cela se produit, le serveur

ne peut pas traiter de nouvelle requête tant que les threads ne sont pas libérés. Avec le code synchrone, plusieurs threads peuvent être bloqués alors qu'ils n'effectuent en fait aucun travail, car ils attendent que des E/S se terminent. Avec le code asynchrone, quand un processus attend que des E/S se terminent, son thread est libéré afin d'être utilisé par le serveur pour traiter d'autres demandes. Le code asynchrone permet ainsi d'utiliser plus efficacement les ressources serveur, et le serveur peut gérer plus de trafic sans retard.

Le code asynchrone introduit néanmoins une petite surcharge au moment de l'exécution, mais dans les situations de faible trafic, la baisse de performances est négligeable, alors qu'en cas de trafic élevé, l'amélioration potentielle des performances est importante.

Dans le code suivant, le mot clé `async`, la valeur renvoyée `Task<T>`, le mot clé `await` et la méthode `ToListAsync` provoquent l'exécution asynchrone du code.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- Le mot clé `async` indique au compilateur de générer des rappels pour les parties du corps de la méthode et pour créer automatiquement l'objet `Task<IActionResult>` qui est renvoyé.
- Le type de retour `Task<IActionResult>` représente le travail en cours avec un résultat de type `IActionResult`.
- Le mot clé `await` indique au compilateur de fractionner la méthode en deux parties. La première partie se termine par l'opération qui est démarrée de façon asynchrone. La seconde partie est placée dans une méthode de rappel qui est appelée quand l'opération se termine.
- `ToListAsync` est la version asynchrone de la méthode d'extension `ToList`.

Voici quelques éléments à connaître lorsque vous écrivez un code asynchrone qui utilise Entity Framework :

- Seules les instructions qui provoquent l'envoi de requêtes ou de commandes vers la base de données sont exécutées de façon asynchrone. Cela inclut, par exemple, `ToListAsync`, `SingleOrDefaultAsync` et `SaveChangesAsync`, mais pas les instructions qui ne font, par exemple, que changer `IQueryable`, telles que `var students = context.Students.Where(s => s.LastName == "Davolio")`.

- Un contexte EF n'est pas thread-safe : n'essayez pas d'effectuer plusieurs opérations en parallèle. Lorsque vous appelez une méthode EF asynchrone quelconque, utilisez toujours le mot clé `await`.
- Si vous souhaitez tirer profit des meilleures performances du code asynchrone, assurez-vous que tous les packages de bibliothèque que vous utilisez (par exemple pour changer de page) utilisent également du code asynchrone s'ils appellent des méthodes Entity Framework qui provoquent l'envoi des requêtes à la base de données.

## Implémenter la fonctionnalité CRUD - ASP.NET MVC avec EF Core

### Personnaliser la page Details

Le code du modèle généré automatiquement pour la page Index des étudiants exclut la propriété `Enrollments`, car elle contient une collection. Dans la page **Details**, vous affichez le contenu de la collection dans un tableau HTML.

Dans `Controllers/StudentsController.cs`, la méthode d'action pour la vue Details utilise la méthode `SingleOrDefaultAsync` pour récupérer une seule entité `Student`. Ajoutez du code qui appelle `Include`. Les méthodes `ThenInclude` et `AsNoTracking`, comme indiqué dans le code en surbrillance suivant.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

Les méthodes `Include` et `ThenInclude` font que le contexte charge la propriété de navigation `Student.Enrollments` et dans chaque inscription, la propriété de navigation `Enrollment.Course`.

La méthode `AsNoTracking` améliore les performances dans les scénarios où les entités retournées ne sont pas mises à jour pendant la durée de vie du contexte actif. Vous pouvez découvrir plus d'informations sur `AsNoTracking` à la fin de ce tutoriel.

## Données de route

La valeur de clé qui est passée à la méthode `Details` provient des *données de route*. Les données de route sont des données que le classeur de modèles a trouvées dans un segment de l'URL. Par exemple, la route par défaut spécifie les segments contrôleur, action et ID :

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Dans l'URL suivante, la route par défaut mappe Instructor en tant que contrôleur, Index en tant qu'action et 1 en tant qu'ID ; il s'agit des valeurs des données de route.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```



La dernière partie de l'URL (« ?courseID=2021 ») est une valeur de chaîne de requête. Le classeur de modèles passe aussi la valeur d'ID au paramètre `id` de la méthode `Details` si vous le passez en tant que valeur de chaîne de requête :

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

Dans la page Index, les URL des liens hypertexte sont créées par des instructions Tag Helper dans la vue Razor. Dans le code Razor suivant, le paramètre `id` correspond à la route par défaut : `id` est donc ajouté aux données de route.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

Ceci génère le code HTML suivant quand `item.ID` vaut 6 :

```
<a href="/Students/Edit/6">Edit</a>
```

Dans le `code Razor` suivant, `studentID` ne correspond pas à un paramètre dans la route par défaut : il est donc ajouté en tant que chaîne de requête.

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

Ceci génère le code HTML suivant quand `item.ID` vaut 6 :

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

## Ajouter des inscriptions à la vue Details

Ouvrez `Views/Students/Details.cshtml`. Chaque champ est affiché avec les helpers `DisplayNameFor` et `DisplayFor`, comme montré dans l'exemple suivant :

```
<dt>  
    @Html.DisplayNameFor(model => model.LastName)  
</dt>  
<dd>  
    @Html.DisplayFor(model => model.LastName)  
</dd>
```

Après le dernier champ et immédiatement avant la balise de fermeture `</dl>`, ajoutez le code suivant pour afficher une liste d'inscriptions :



```

<dt>
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd>
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>

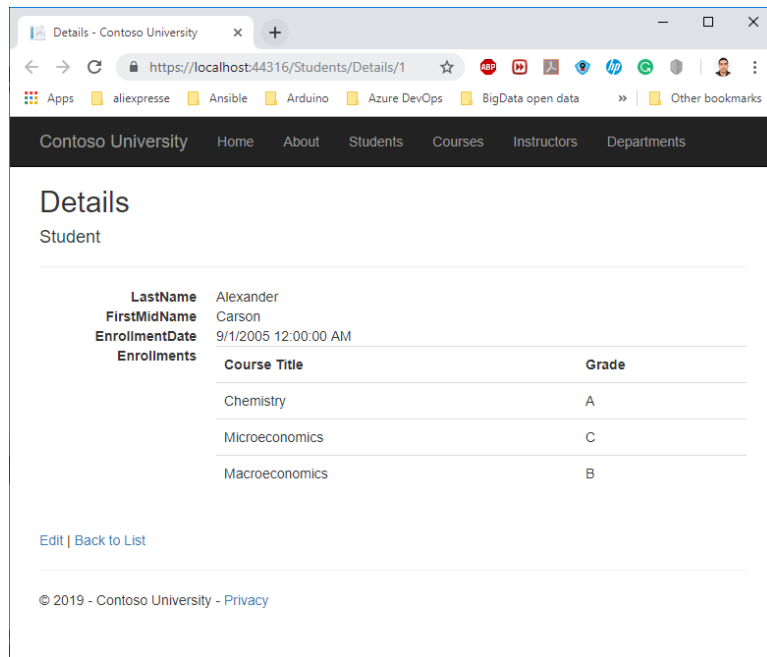
```

Si l'indentation du code est incorrecte une fois le code collé, appuyez sur Ctrl-K-D pour la corriger.

Ce code parcourt en boucle les entités dans la propriété de navigation `Enrollments`. Pour chaque inscription, il affiche le titre du cours et la note. Le titre du cours est récupéré à partir de l'entité de cours qui est stockée dans la propriété de navigation `Course` de l'entité `Enrollments`.



Exécutez l'application, sélectionnez l'onglet **Students**, puis cliquez sur le lien **Details** pour un étudiant. Vous voyez la liste des cours et des notes de l'étudiant sélectionné :



## Mettre à jour la page Create

Dans *StudentsController.cs*, modifiez la méthode `HttpPost Create` en ajoutant un bloc try-catch et en supprimant l'ID de l'attribut `Bind`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Ce code ajoute l'entité Student créée par le classeur de modèles ASP.NET Core MVC au jeu d'entités Students, puis enregistre les modifications dans la base de données. (Le classeur de modèles référence la fonctionnalité d'ASP.NET Core MVC qui facilite l'utilisation des données envoyées par un formulaire ; un classeur de modèles convertit les valeurs de formulaire envoyées en types CLR et les passe à la méthode d'action dans des paramètres. Dans ce cas, le classeur de modèles instancie une entité Student pour vous avec des valeurs de propriété provenant de la collection Form.)

Vous avez supprimé `ID` de l'attribut `Bind`, car ID est la valeur de clé primaire définie automatiquement par SQL Server lors de l'insertion de la ligne. L'entrée de l'utilisateur ne définit pas la valeur de l'ID.

À part l'attribut `Bind`, le bloc try-catch est la seule modification que vous avez apportée au code du modèle généré automatiquement. Si une exception qui dérive de `DbUpdateException` est interceptée lors de l'enregistrement des modifications, un message d'erreur générique est affiché. Les exceptions `DbUpdateException` sont parfois dues à quelque chose d'externe à l'application et non pas à une erreur de programmation : il est donc conseillé à l'utilisateur de réessayer. Bien que ceci ne soit pas implémenté dans cet exemple, une application destinée à la production doit consigner l'exception.

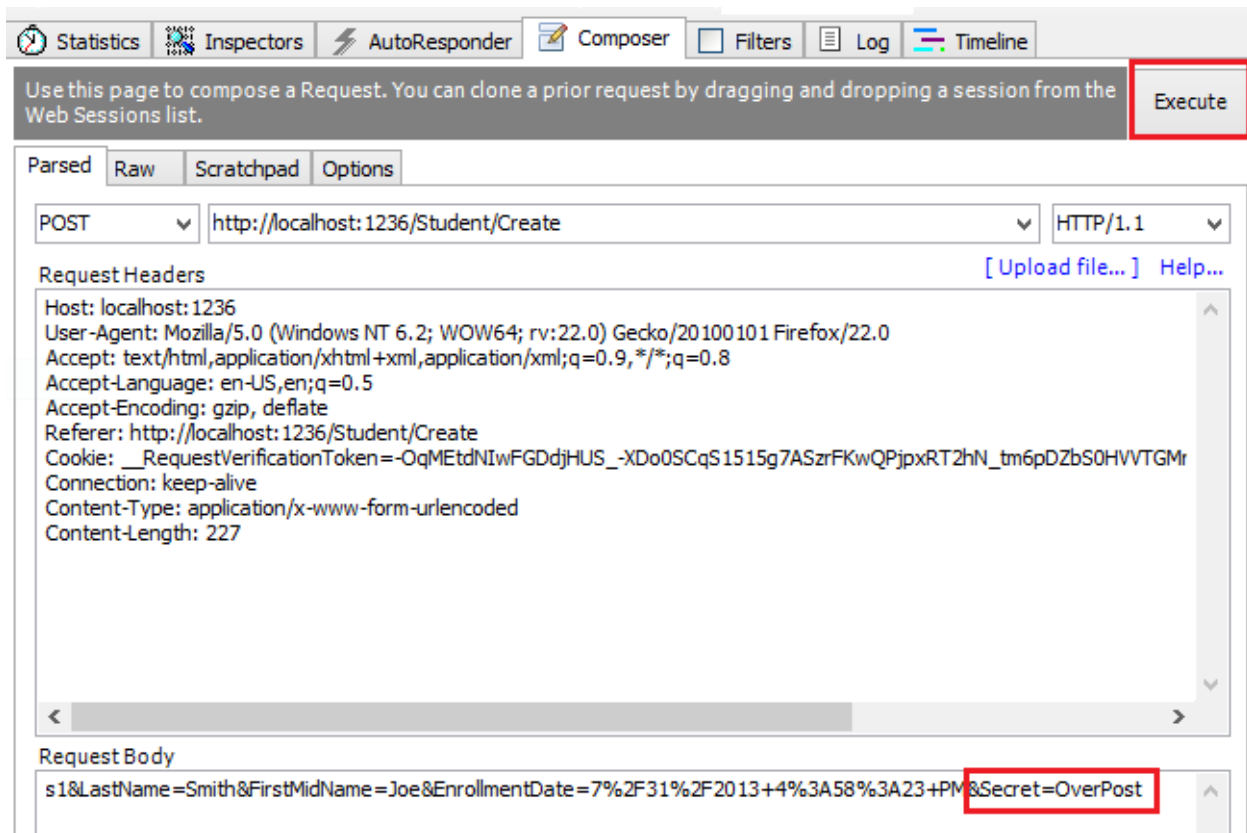
L'attribut `ValidateAntiForgeryToken` aide à éviter les attaques par falsification de requête intersites (CSRF, Cross-Site Request Forgery). Le jeton est automatiquement injecté dans la vue par le `FormTagHelper` et est inclus quand le formulaire est envoyé par l'utilisateur. Le jeton est validé par l'attribut `ValidateAntiForgeryToken`.

## Remarque sur la sécurité concernant la survalidation

L'attribut `Bind` inclus dans le code du modèle généré automatiquement sur la méthode `Create` est un moyen de protéger contre la survalidation dans les scénarios de création. Par exemple, supposons que l'entité Student comprend une propriété `Secret` et que vous ne voulez pas que cette page web définisse sa valeur.

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Même si vous n'avez pas de champ `Secret` dans la page web, un hacker pourrait utiliser un outil comme Fiddler, ou écrire du JavaScript, pour envoyer une valeur de formulaire pour `Secret`. Sans l'attribut `Bind` limitant les champs utilisés par le classeur de modèles quand il crée une instance de Student, le classeur de modèles choisit la valeur de formulaire pour `Secret` et l'utilise pour créer l'instance de l'entité Student. Ensuite, la valeur spécifiée par le hacker pour le champ de formulaire `Secret`, quelle qu'elle soit, est mise à jour dans la base de données. L'illustration suivante montre l'outil Fiddler en train d'ajouter le champ `Secret` (avec la valeur « OverPost ») aux valeurs du formulaire envoyé.



La valeur « OverPost » serait correctement ajoutée à la propriété `Secret` de la ligne insérée, même si vous n'aviez jamais prévu que la page web puisse définir cette propriété.

Vous pouvez empêcher la survalidation dans les scénarios de modification en lisant d'abord l'entité à partir de la base de données, puis en appelant `TryUpdateModel`, en passant une liste des propriétés explicitement autorisées. Il s'agit de la méthode utilisée dans ce tutoriel.

Une autre façon d'empêcher la survalidation qui est préférée par de nombreux développeurs consiste à utiliser les afficher des modèles de vues au lieu de classes

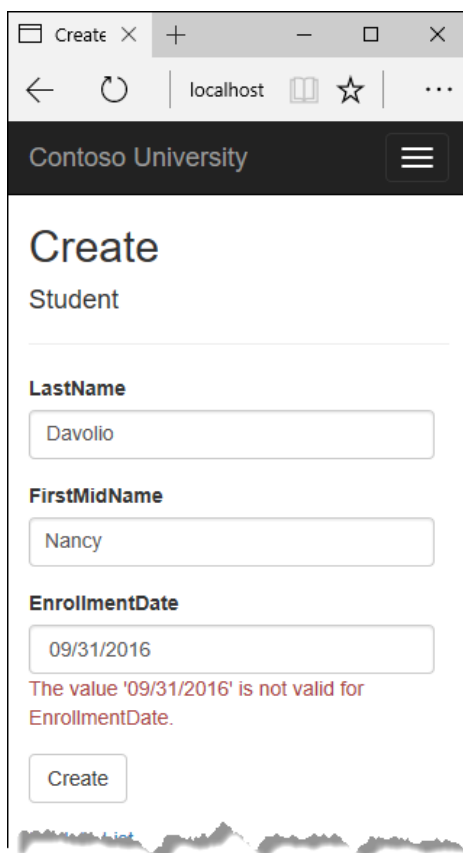
d'entités avec la liaison de modèle. Incluez seulement les propriétés que vous voulez mettre à jour dans le modèle de vue. Une fois le classeur de modèles MVC a terminé, copiez les propriétés du modèle de vue vers l'instance de l'entité, en utilisant si vous le souhaitez un outil comme AutoMapper. Utilisez `_context.Entry` sur l'instance de l'entité pour définir son état sur `Unchanged`, puis définissez `Property("PropertyName").IsModified` sur true sur chaque propriété d'entité qui est incluse dans le modèle de vue. Cette méthode fonctionne à la fois dans les scénarios de modification et de création.

## Tester la page Create

Le code dans *Views/Students/Create.cshtml* utilise les tag helpers `label`, `input` et `span` (pour les messages de validation) pour chaque champ.

Exécutez l'application, sélectionnez l'onglet **Students**, puis cliquez sur **Create New**.

Entrez des noms et une date. Si votre navigateur vous le permet, essayez d'entrer une date non valide. (Certains navigateurs vous obligent à utiliser un sélecteur de dates.) Cliquez ensuite sur **Create** pour voir le message d'erreur.



The screenshot shows a web browser window with the address bar at 'localhost'. The page title is 'Contoso University'. The main heading is 'Create Student'. The form contains three input fields: 'LastName' (containing 'Davolio'), 'FirstMidName' (containing 'Nancy'), and 'EnrollmentDate' (containing '09/31/2016'). Below the 'EnrollmentDate' field, there is a red error message: 'The value '09/31/2016' is not valid for EnrollmentDate.' At the bottom of the form is a 'Create' button.

Il s'agit de la validation côté serveur que vous obtenez par défaut ; dans un tutoriel suivant, vous verrez comment ajouter des attributs qui génèrent du code également pour la validation côté client. Le code en surbrillance suivant montre la vérification de validation du modèle dans la méthode `Create`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Changez la date en une valeur valide, puis cliquez sur **Create** pour voir apparaître le nouvel étudiant dans la page **Index**.

## Mettre à jour la page Edit

Dans *StudentController.cs*, la méthode `HttpGet Edit` (celle sans l'attribut `HttpPost`) utilise la méthode `SingleOrDefaultAsync` pour récupérer l'entité `Student` sélectionnée, comme vous l'avez vu dans la méthode `Details`. Vous n'avez pas besoin de modifier cette méthode.

Code `HttpPost Edit` recommandé : Lire et mettre à jour

Remplacez la méthode d'action `HttpPost Edit` par le code suivant.

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.SingleOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}

```

Ces modifications implémentent une bonne pratique de sécurité pour empêcher la **survalidation**. Le générateur de modèles automatique a généré un attribut `Bind` et a ajouté l'entité créée par le classeur de modèles au jeu d'entités avec un indicateur `Modified`. Ce code n'est pas recommandé dans de nombreux scénarios, car l'attribut `Bind` efface toutes les données préexistantes dans les champs non répertoriés dans le paramètre `Include`.

Le nouveau code lit l'entité existante et appelle `TryUpdateModel` pour mettre à jour les champs dans l'entité récupérée en fonction de l'entrée d'utilisateur dans les données du formulaire envoyé. Le suivi automatique des modifications d'Entity Framework définit l'indicateur `Modified` sur les champs qui sont modifiés via une entrée dans le formulaire. Quand la méthode `SaveChanges` est appelée, Entity Framework crée des instructions SQL pour mettre à jour la ligne de la base de données. Les conflits d'accès concurrentiel sont ignorés, et seules les colonnes de table qui ont été mises à jour par l'utilisateur sont mises à jour dans la base de données. (Un tutoriel suivant montre comment gérer les conflits d'accès concurrentiel.)

Au titre de bonne pratique pour empêcher la survalidation, les champs dont vous voulez qu'ils puissent être mis à jour par la page **Edit** sont placés en liste verte dans les paramètres de `TryUpdateModel`. (La chaîne vide précédant la liste de champs dans la liste de paramètres est pour un préfixe à utiliser avec les noms des champs du formulaire.) Actuellement, vous ne protégez aucun champ supplémentaire, mais le fait de répertorier les champs que vous voulez que le classeur de modèles lie garantit que si vous ajoutez ultérieurement des champs au modèle de données, ils seront automatiquement protégés jusqu'à ce que vous les ajoutiez explicitement ici.

À la suite de ces modifications, la signature de méthode de la méthode `HttpPost Edit` est la même que celle de la méthode `HttpGet Edit`; par conséquent, vous avez renommé la méthode `EditPost`.

### Autre possibilité pour le code `HttpPost Edit` : Créer et attacher

Le code de `HttpPost Edit` recommandé garantit que seules les colonnes modifiées sont mises à jour et conserve les données dans les propriétés que vous ne voulez pas inclure pour la liaison de modèle. Cependant, l'approche « lecture en premier » nécessite une lecture supplémentaire de la base de données et peut aboutir à un code plus complexe pour la gestion des conflits d'accès concurrentiel. Une alternative consiste à attacher une entité créée par le classeur de modèles au contexte EF et à la marquer comme étant modifiée. (Ne mettez pas à jour votre projet avec ce code, il figure ici seulement pour illustrer une approche facultative.)

```
public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastNa
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```



Vous pouvez utiliser cette approche quand l'interface utilisateur de la page web inclut tous les champs de l'entité et peut les mettre à jour.

Le code du modèle généré automatiquement utilise l'approche « créer et attacher », mais il intercepte seulement les exceptions `DbUpdateConcurrencyException` et retourne des codes d'erreur 404. L'exemple suivant intercepte toutes les exceptions de mise à jour de la base de données et affiche un message d'erreur.

## États des entités

Le contexte de base de données effectue le suivi de la synchronisation ou non des entités en mémoire avec leurs lignes correspondantes dans la base de données, et ces informations déterminent ce qui se passe quand vous appelez la méthode `SaveChanges`. Par exemple, quand vous passez une nouvelle entité à la méthode `Add`, l'état de cette entité est défini sur `Added`. Ensuite, quand vous appelez la méthode `SaveChanges`, le contexte de base de données émet une commande SQL INSERT.

Une entité peut être dans un des états suivants :

- `Added`. L'entité n'existe pas encore dans la base de données. La méthode `SaveChanges` émet une instruction INSERT.
- `Unchanged`. La méthode `SaveChanges` ne doit rien faire avec cette entité. Quand vous lisez une entité dans la base de données, l'entité a d'abord cet état.
- `Modified`. Tout ou partie des valeurs de propriété de l'entité ont été modifiées. La méthode `SaveChanges` émet une instruction UPDATE.
- `Deleted`. L'entité a été marquée pour suppression. La méthode `SaveChanges` émet une instruction DELETE.
- `Detached`. L'entité n'est pas suivie par le contexte de base de données.

Dans une application de poste de travail, les changements d'état sont généralement définis automatiquement. Vous lisez une entité et vous apportez des modifications à certaines de ses valeurs de propriété. Son état passe alors automatiquement à `Modified`. Quand vous appelez `SaveChanges`, Entity Framework génère une instruction SQL UPDATE qui met à jour seulement les propriétés que vous avez modifiées.

Dans une application web, le `DbContext` qui lit initialement une entité et affiche ses données pour permettre leur modification est supprimé après le rendu d'une page. Quand la méthode d'action `HttpPost Edit` est appelée, une nouvelle requête web est effectuée

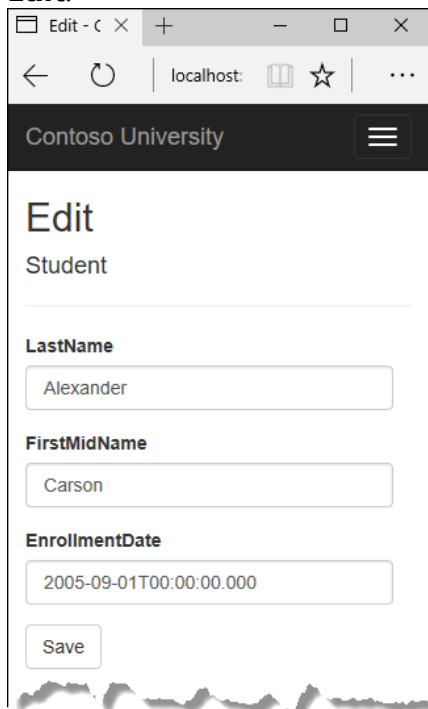
et vous disposez d'une nouvelle instance de `DbContext`. Si vous relisez l'entité dans ce nouveau contexte, vous simulez le traitement du poste de travail.

Mais si vous ne voulez pas effectuer l'opération de lecture supplémentaire, vous devez utiliser l'objet entité créé par le classeur de modèles. Le moyen le plus simple consiste à définir l'état de l'entité en Modified, comme cela est fait dans l'alternative pour le code `HttpPost Edit` illustrée précédemment. Ensuite, quand vous appelez `SaveChanges`, Entity Framework met à jour toutes les colonnes de la ligne de la base de données, car le contexte n'a aucun moyen de savoir quelles propriétés vous avez modifiées.

Si vous voulez éviter l'approche « lecture en premier », mais que vous voulez aussi que l'instruction SQL UPDATE mette à jour seulement les champs que l'utilisateur a réellement changés, le code est plus complexe. Vous devez enregistrer les valeurs d'origine d'une façon ou d'une autre (par exemple en utilisant des champs masqués) afin qu'ils soient disponibles quand la méthode `HttpPost Edit` est appelée. Vous pouvez ensuite créer une entité Student en utilisant les valeurs d'origine, appeler la méthode `Attach` avec cette version d'origine de l'entité, mettre à jour les valeurs de l'entité avec les nouvelles valeurs, puis appeler `SaveChanges`.

## Tester la page Edit

Exécutez l'application, sélectionnez l'onglet **Students**, puis cliquez sur un lien hypertexte **Edit**.



The screenshot shows a web browser window with the address bar at 'localhost'. The page title is 'Edit' and the header is 'Contoso University'. The main content area is titled 'Edit Student' and contains three input fields: 'LastName' with the value 'Alexander', 'FirstMidName' with the value 'Carson', and 'EnrollmentDate' with the value '2005-09-01T00:00:00.000'. A 'Save' button is located at the bottom of the form.

Changez quelques données et cliquez sur **Save**. La page **Index** s'ouvre et affiche les données modifiées.

## Mettre à jour la page Delete

Dans *StudentController.cs*, le modèle de code pour la méthode `HttpGet Delete` utilise la méthode `SingleOrDefaultAsync` pour récupérer l'entité *Student* sélectionnée, comme vous l'avez vu dans les méthodes *Details* et *Edit*. Cependant, pour implémenter un message d'erreur personnalisé quand l'appel à `SaveChanges` échoue, vous devez ajouter des fonctionnalités à cette méthode et à sa vue correspondante.

Comme vous l'avez vu pour les opérations de mise à jour et de création, les opérations de suppression nécessitent deux méthodes d'action. La méthode qui est appelée en réponse à une demande GET affiche une vue qui permet à l'utilisateur d'approuver ou d'annuler l'opération de suppression. Si l'utilisateur l'approuve, une demande POST est créée. Quand cela se produit, la méthode `HttpPost Delete` est appelée, puis cette méthode effectue ensuite l'opération de suppression.

Vous allez ajouter un bloc try-catch à la méthode `HttpPost Delete` pour gérer les erreurs qui peuvent se produire quand la base de données est mise à jour. Si une erreur se produit, la méthode `HttpPost Delete` appelle la méthode `HttpGet Delete`, en lui passant un paramètre qui indique qu'une erreur s'est produite. La méthode `HttpGet Delete` réaffiche ensuite la page de confirmation, ainsi que le message d'erreur, donnant à l'utilisateur la possibilité d'annuler ou de recommencer.

Remplacez la méthode d'action `HttpGet Delete` par le code suivant, qui gère le signalement des erreurs.

```

public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}

```

Ce code accepte un paramètre facultatif qui indique si la méthode a été appelée après une erreur d'enregistrement des modifications. Ce paramètre a la valeur `false` quand la méthode `HttpGet Delete` est appelée sans une erreur antérieure. Quand elle est appelée par la méthode `HttpPost Delete` en réponse à une erreur de mise à jour de la base de données, le paramètre a la valeur `true` et un message d'erreur est passé à la vue.

### L'approche « lecture en premier » pour `HttpPost Delete`

Remplacez la méthode d'action `HttpPost Delete` (nommée `DeleteConfirmed`) par le code suivant, qui effectue l'opération de suppression réelle et intercepte les erreurs de mise à jour de la base de données.

```

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}

```

Ce code récupère l'entité sélectionnée, puis appelle la méthode `Remove` pour définir l'état de l'entité sur `Deleted`. Quand `SaveChanges` est appelée, une commande SQL DELETE est générée.

### L'approche « créer et attacher » pour HttpPost Delete

Si l'amélioration des performances dans une application traitant des volumes importants est une priorité, vous pouvez éviter une requête SQL inutile en instanciant une entité de Student en utilisant seulement la valeur de la clé primaire, puis en définissant l'état de l'entité sur `Deleted`. C'est tout ce dont a besoin Entity Framework pour pouvoir supprimer l'entité. (Ne placez pas de ce code dans votre projet ; il figure ici seulement pour illustrer une solution alternative.)

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}

```

Si l'entité a des données connexes qui doivent également être supprimées, vérifiez que la suppression en cascade est configurée dans la base de données. Avec cette approche pour la suppression de l'entité, EF peut ne pas réaliser que des entités connexes doivent être supprimées.

## Mettre à jour la vue Delete

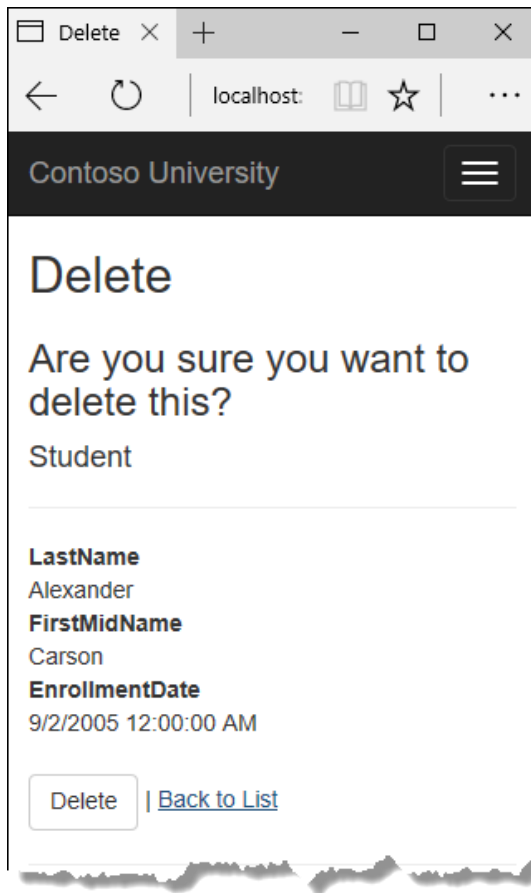
Dans *Views/Student/Delete.cshtml*, ajoutez un message d'erreur entre le titre h2 et le titre h3, comme indiqué dans l'exemple suivant :

```

<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>

```

Exécutez l'application, sélectionnez l'onglet **Students**, puis cliquez sur un lien hypertexte **Delete** :



Cliquez sur **Delete**. La page Index s'affiche sans l'étudiant supprimé. (Vous verrez un exemple du code de gestion des erreurs en action dans le tutoriel sur l'accès concurrentiel.)

## Fermer les connexions de base de données

Pour libérer les ressources détenues par une connexion de base de données, l'instance du contexte doit être supprimée dès que possible quand vous en avez terminé avec celle-ci. L'injection de **dépendances intégrée** d'ASP.NET Core prend en charge cette tâche pour vous.

Dans *Startup.cs*, vous appelez la méthode d'extension **AddDbContext** pour provisionner la classe `DbContext` dans le conteneur d'injection de dépendances d'ASP.NET Core. Cette méthode définit par défaut la durée de vie du service sur `Scoped`. `Scoped` signifie que la durée de vie de l'objet de contexte coïncide avec la durée de vie de la demande web, et que la méthode `Dispose` sera appelée automatiquement à la fin de la requête web.

## Gérer les transactions

Par défaut, Entity Framework implémente implicitement les transactions. Dans les scénarios où vous apportez des modifications à plusieurs lignes ou plusieurs tables, puis que vous appelez `SaveChanges`, Entity Framework garantit automatiquement que soit toutes vos modifications réussissent soit elles échouent toutes. Si certaines modifications sont effectuées en premier puis qu'une erreur se produit, ces modifications sont automatiquement annulées.

## Pas de suivi des requêtes

Quand un contexte de base de données récupère des lignes de table et crée des objets entité qui les représentent, par défaut, il effectue le suivi du fait que les entités en mémoire sont ou non synchronisées avec ce qui se trouve dans la base de données. Les données en mémoire agissent comme un cache et sont utilisées quand vous mettez à jour une entité. Cette mise en cache est souvent inutile dans une application web, car les instances de contexte ont généralement une durée de vie courte (une instance est créée puis supprimée pour chaque requête) et le contexte qui lit une entité est généralement supprimé avant que cette entité soit réutilisée.

Vous pouvez désactiver le suivi des objets entité en mémoire en appelant la méthode `AsNoTracking`. Voici des scénarios classiques où vous voulez procéder ainsi :

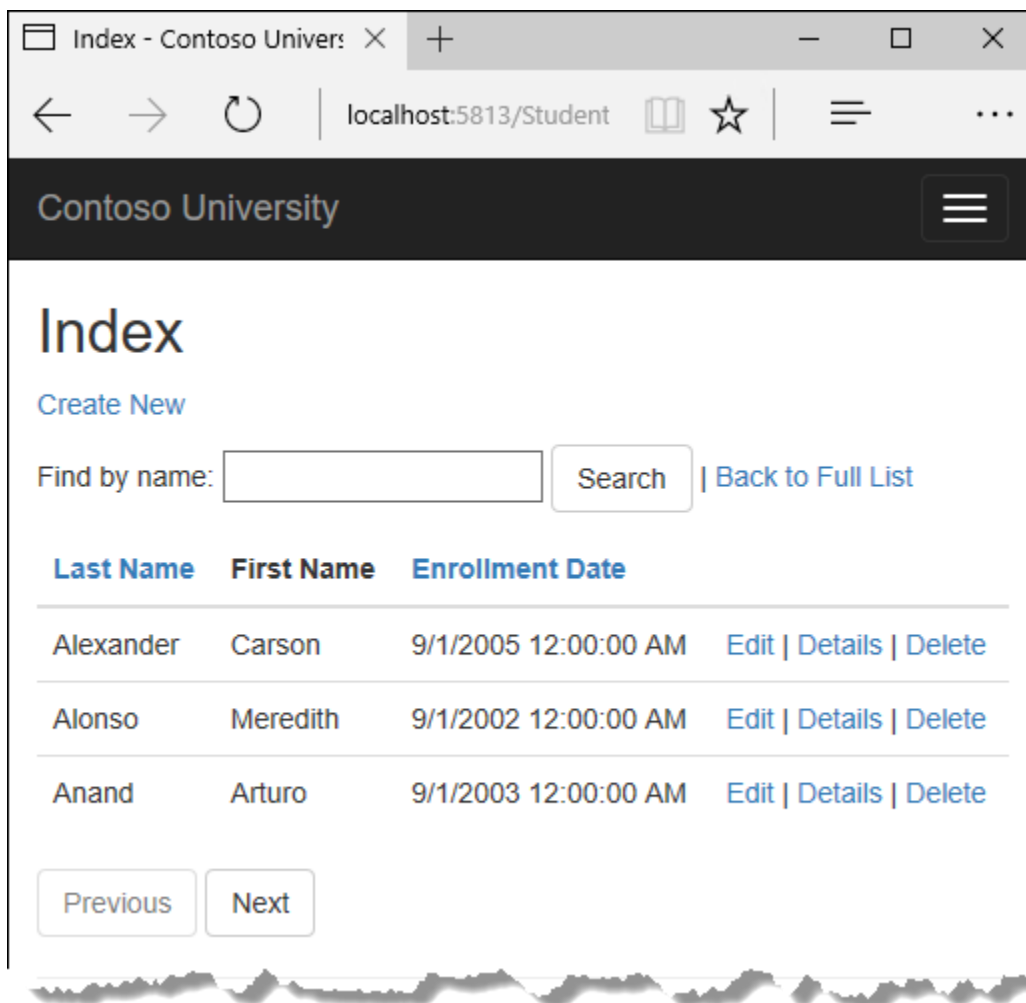
- Pendant la durée de vie du contexte, vous n'avez besoin de mettre à jour aucune entité et il n'est pas nécessaire qu'EF charge automatiquement les propriétés de navigation avec les entités récupérées par des requêtes distinctes. Ces conditions sont souvent rencontrées dans les méthodes d'action `HttpGet` d'un contrôleur.
- Vous exécutez une requête qui récupère un gros volume de données, et seule une petite partie des données retournées sont mises à jour. Il peut être plus efficace de désactiver le suivi pour la requête retournant un gros volume de données et d'exécuter une requête plus tard pour les quelques entités qui doivent être mises à jour.
- Vous voulez attacher une entité pour pouvoir la mettre à jour, mais vous avez auparavant récupéré la même entité à d'autres fins. Comme l'entité est déjà suivie par le contexte de base de données, vous ne pouvez pas attacher l'entité que vous voulez modifier. Une façon de gérer cette situation est d'appeler `AsNoTracking` sur la requête précédente.



## Ajouter le tri, le filtrage et la pagination

Dans cette section, vous allez ajouter les fonctionnalités de tri, de filtrage et de changement de page à la page d'index des étudiants. Vous allez également créer une page qui effectue un regroupement simple.

L'illustration suivante montre à quoi ressemblera la page quand vous aurez terminé. Les en-têtes des colonnes sont des liens sur lesquels l'utilisateur peut cliquer pour trier selon les colonnes. Cliquer de façon répétée sur un en-tête de colonne permet de changer l'ordre de tri (croissant ou décroissant).



## Ajouter des liens de tri de colonne

Pour ajouter le tri à la page d'index des étudiants, vous allez modifier la méthode `Index` du contrôleur `Students` et ajouter du code à la vue de l'index des étudiants.

## Ajouter la fonctionnalité de tri à la méthode Index

Dans *StudentsController.cs*, remplacez la méthode `Index` par le code suivant :

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

Ce code reçoit un paramètre `sortOrder` à partir de la chaîne de requête dans l'URL. La valeur de chaîne de requête est fournie par ASP.NET Core MVC en tant que paramètre à la méthode d'action. Le paramètre sera la chaîne « Name » ou « Date », éventuellement suivie d'un trait de soulignement et de la chaîne « desc » pour spécifier l'ordre décroissant. L'ordre de tri par défaut est le tri croissant.

La première fois que la page d'index est demandée, il n'y a pas de chaîne de requête. Les étudiants sont affichés dans l'ordre croissant par leur nom, ce qui correspond au paramétrage par défaut de l'instruction `switch`. Quand l'utilisateur clique sur un lien hypertexte d'en-tête de colonne, la valeur `sortOrder` appropriée est fournie dans la chaîne de requête.

Les deux éléments `ViewData` (`NameSortParm` et `DateSortParm`) sont utilisés par la vue pour configurer les liens hypertexte d'en-tête de colonne avec les valeurs de chaîne de requête appropriées.

```

public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

Il s'agit d'instructions ternaires. La première spécifie que si le paramètre `sortOrder` est null ou vide, NameSortParm doit être défini sur « name\_desc » ; sinon, il doit être défini sur une chaîne vide. Ces deux instructions permettent à la vue de définir les liens hypertexte d'en-tête de colonne comme suit :

Ordre de tri actuel	Lien hypertexte Nom de famille	Lien hypertexte Date
Nom de famille croissant	descending	ascending
Nom de famille décroissant	ascending	ascending
Date croissante	ascending	descending
Date décroissante	ascending	ascending

La méthode utilise LINQ to Entities pour spécifier la colonne d'après laquelle effectuer le tri. Le code crée une variable `IQueryable` avant l'instruction `switch`, la modifie dans l'instruction `switch` et appelle la méthode `ToListAsync` après l'instruction `switch`. Lorsque vous créez et modifiez des variables `IQueryable`, aucune requête n'est envoyée à la base de données. La requête n'est pas exécutée tant que vous ne convertissez pas l'objet `IQueryable` en collection en appelant une méthode telle que `ToListAsync`. Par conséquent, ce code génère une requête unique qui n'est pas exécutée avant l'instruction `return View`.

Ce code peut devenir très détaillé avec un grand nombre de colonnes. La dernière section de ce tutoriel montre comment écrire du code qui vous permet de transmettre le nom de la colonne `OrderBy` dans une variable chaîne.

## Ajouter des liens hypertexte d'en-tête de colonne dans la vue de l'index des étudiants

Remplacez le code dans `Views/Students/Index.cshtml` par le code suivant pour ajouter des liens hypertexte d'en-tête de colonne. Les lignes modifiées apparaissent en surbrillance.

```
@model IEnumerable<ContosoUniversity.Models.Student>

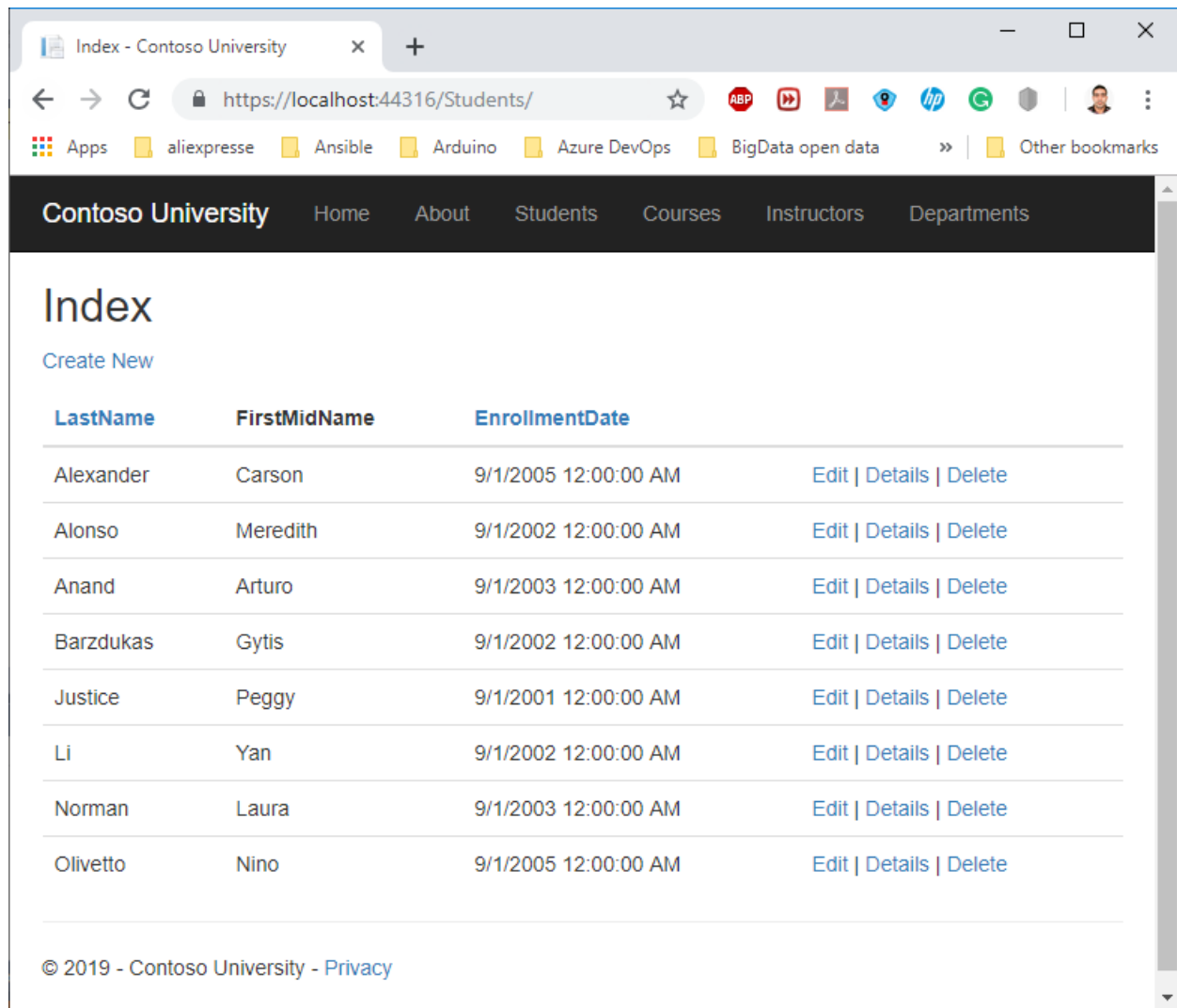
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
```

Ce code utilise les informations figurant dans les propriétés `ViewData` pour configurer des liens hypertexte avec les valeurs de chaîne de requête appropriées.

Exécutez l'application, sélectionnez l'onglet **Students**, puis cliquez sur les en-têtes des colonnes **Last Name** et **Enrollment Date** pour vérifier que le tri fonctionne.



## Ajouter une zone Recherche

Pour ajouter le filtrage à la page d'index des étudiants, vous allez ajouter une zone de texte et un bouton d'envoi à la vue et apporter les modifications correspondantes dans la méthode `Index`. La zone de texte vous permet d'entrer une chaîne à rechercher dans les champs de prénom et de nom.

### Ajouter la fonctionnalité de filtrage à la méthode Index

Dans *StudentsController.cs*, remplacez la méthode `Index` par le code suivant (les modifications apparaissent en surbrillance).

```

public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

Vous avez ajouté un paramètre `searchString` à la méthode `Index`. La valeur de chaîne de recherche est reçue à partir d'une zone de texte que vous ajouterez à la vue Index. Vous avez également ajouté à l'instruction LINQ une clause `where` qui sélectionne uniquement les étudiants dont le prénom ou le nom contient la chaîne de recherche. L'instruction qui ajoute la clause `where` est exécutée uniquement s'il existe une valeur à rechercher.

**Notes :** Ici, vous appelez la méthode `Where` sur un objet `IQueryable`, et le filtre sera traité sur le serveur. Dans certains scénarios, vous pouvez appeler la méthode `Where` en tant que méthode d'extension sur une collection en mémoire. (Par exemple, supposons que vous changez la référence à `_context.Students` de sorte qu'à la place d'un `DbSet` EF, elle fasse référence à une méthode de référentiel qui renvoie une collection `IEnumerable`.) Le résultat serait normalement le même, mais pourrait être différent dans certains cas.

Par exemple, l'implémentation par le .NET Framework de la méthode `Contains` effectue une comparaison respectant la casse par défaut, mais dans SQL Server, cela est déterminé par le paramètre de classement de l'instance SQL Server. Ce paramètre définit par défaut le non-respect de la casse. Vous pouvez appeler la méthode `ToUpper` pour que le test ne respecte pas la casse de manière explicite : `Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))`. Cela garantit que les résultats resteront les mêmes si vous modifiez le code ultérieurement pour utiliser un référentiel qui renverra une collection `IEnumerable` à la place d'un objet `IQueryable`. (Lorsque vous appelez la méthode `Contains` sur une collection `IEnumerable`, vous obtenez l'implémentation du .NET Framework ; lorsque vous l'appellez sur un objet `IQueryable`, vous obtenez l'implémentation du fournisseur de base de données.) Toutefois, cette solution présente un coût en matière de performances. Le code `ToUpper` place une fonction dans la clause WHERE de l'instruction TSQL SELECT. Elle empêche l'optimiseur d'utiliser un index. Étant donné que SQL est généralement installé comme non sensible à la casse, il est préférable d'éviter le code `ToUpper` jusqu'à ce que vous ayez migré vers un magasin de données qui respecte la casse.

### Ajouter une zone de recherche à la vue de l'index des étudiants

Dans `Views/Student/Index.cshtml`, ajoutez le code en surbrillance immédiatement avant la balise d'ouverture de table afin de créer une légende, une zone de texte et un bouton de **recherche**.

```
<p>
  <a asp-action="Create">Create New</a>
</p>

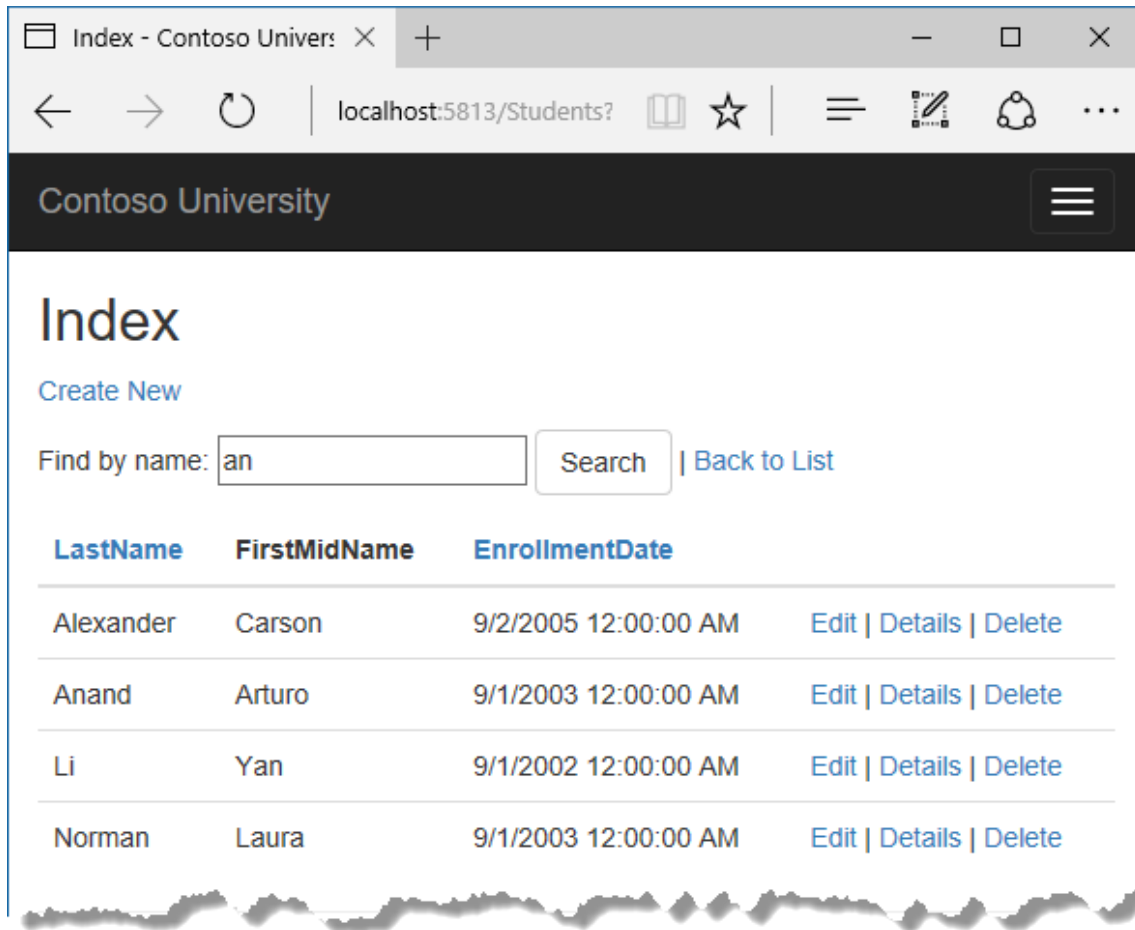
<form asp-action="Index" method="get">
  <div class="form-actions no-color">
    <p>
      Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
      <input type="submit" value="Search" class="btn btn-default" /> |
      <a asp-action="Index">Back to Full List</a>
    </p>
  </div>
</form>

<table class="table">
```

Ce code utilise le Tag Helper `<form>` pour ajouter le bouton et la zone de texte de recherche. Par défaut, le Tag Helper `<form>` envoie les données de formulaire avec un POST, ce qui signifie que les paramètres sont transmis dans le corps du message HTTP et non pas dans l'URL sous forme de chaînes de requête. Lorsque vous spécifiez HTTP GET,

les données de formulaire sont transmises dans l'URL sous forme de chaînes de requête, ce qui permet aux utilisateurs d'ajouter l'URL aux favoris. Les recommandations du W3C stipulent que vous devez utiliser GET quand l'action ne produit pas de mise à jour.

Exécutez l'application, sélectionnez l'onglet **Students**, entrez une chaîne de recherche, puis cliquez sur Rechercher pour vérifier que le filtrage fonctionne.



Notez que l'URL contient la chaîne de recherche.

```
http://localhost:5813/Students?SearchString=an
```

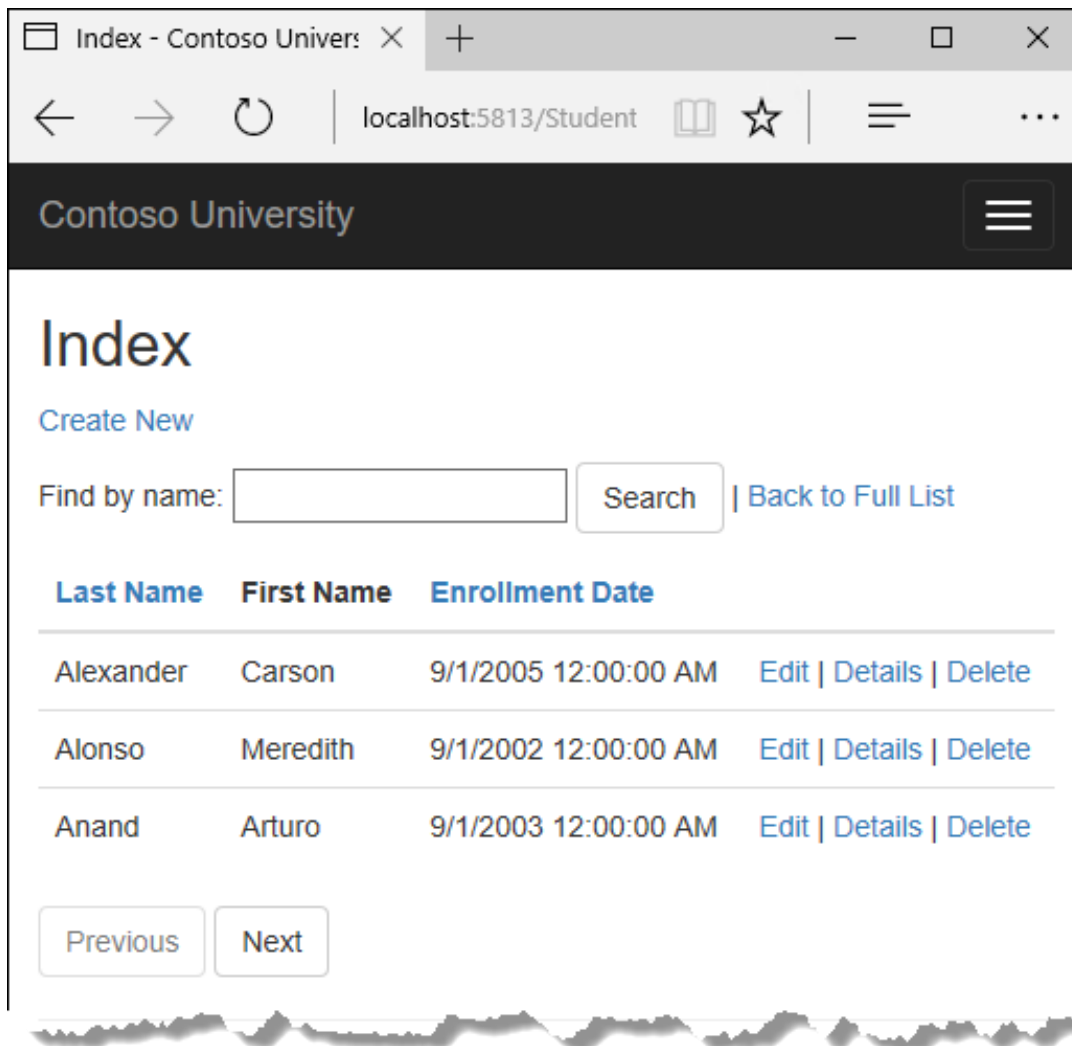
Si vous marquez cette page d'un signet, vous obtenez la liste filtrée lorsque vous utilisez le signet. L'ajout de `method="get"` dans la balise `form` est ce qui a provoqué la génération de la chaîne de requête.

À ce stade, si vous cliquez sur un lien de tri d'en-tête de colonne, vous perdez la valeur de filtre que vous avez entrée dans la zone **Rechercher**. Vous corrigerez cela dans la section suivante.



## Ajouter la pagination aux Index des étudiants

Pour ajouter le changement de page à la page d'index des étudiants, vous allez créer une classe `PaginatedList` qui utilise les instructions `Skip` et `Take` pour filtrer les données sur le serveur au lieu de toujours récupérer toutes les lignes de la table. Ensuite, vous apporterez des modifications supplémentaires dans la méthode `Index` et ajouterez des boutons de changement de page dans la vue `Index`. L'illustration suivante montre les boutons de changement de page.



Dans le dossier du projet, créez `PaginatedList.cs`, puis remplacez le code du modèle par le code suivant.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    3 references
    public class PaginatedList<T> : List<T>
    {
        3 references | 0 exceptions
        public int PageIndex { get; private set; }
        2 references | 0 exceptions
        public int TotalPages { get; private set; }

        1 reference | 0 exceptions
        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        0 references | 0 exceptions
        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        0 references | 0 exceptions
        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        0 references | 0 exceptions
        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}

```

La méthode `CreateAsync` de ce code accepte la taille de page et le numéro de page, et applique les instructions `Skip` et `Take` appropriées à `IQueryable`. Quand la méthode `ToListAsync` est appelée sur `IQueryable`, elle renvoie une liste contenant uniquement la page demandée. Les propriétés `HasPreviousPage` et `HasNextPage` peuvent être utilisées pour activer ou désactiver les boutons de changement de page **Précédent** et **Suivant**.

Une méthode `CreateAsync` est utilisée à la place d'un constructeur pour créer l'objet `PaginatedList<T>`, car les constructeurs ne peuvent pas exécuter de code asynchrone.

## Ajouter la pagination à la méthode Index

Dans *StudentsController.cs*, remplacez la méthode `Index` par le code suivant.

```
// GET: Students
4 references | 0 requests | 0 exceptions
public async Task<IActionResult> Index(string sortOrder, string currentFilter, string searchString, int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    var students = from s in _context.Students
                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
}
```

Ce code ajoute un paramètre de numéro de page, un paramètre d'ordre de tri actuel et un paramètre de filtre actuel à la signature de la méthode.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
```

La première fois que la page s'affiche, ou si l'utilisateur n'a pas cliqué sur un lien de changement de page ni de tri, tous les paramètres sont Null. Si l'utilisateur clique sur un lien de changement de page, la variable de page contient le numéro de page à afficher.

L'élément  `ViewData`  nommé  `CurrentSort`  fournit à l'affichage l'ordre de tri actuel, car il doit être inclus dans les liens de changement de page pour que l'ordre de tri soit conservé lors du changement de page.

L'élément  `ViewData`  nommé  `CurrentFilter`  fournit à la vue la chaîne de filtre actuelle. Cette valeur doit être incluse dans les liens de changement de page pour que les paramètres de filtre soient conservés lors du changement de page, et elle doit être restaurée dans la zone de texte lorsque la page est réaffichée.

Si la chaîne de recherche est modifiée au cours du changement de page, la page doit être réinitialisée à 1, car le nouveau filtre peut entraîner l'affichage de données différentes. La chaîne de recherche est modifiée quand une valeur est entrée dans la zone de texte et que le bouton d'envoi est enfoncé. Dans ce cas, le paramètre  `searchString`  n'est pas Null.

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

À la fin de la méthode  `Index` , la méthode  `PaginatedList.CreateAsync`  convertit la requête d'étudiant en une page individuelle d'étudiants dans un type de collection qui prend en charge le changement de page. Cette page individuelle d'étudiants est alors transmise à la vue.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, p
```

La méthode  `PaginatedList.CreateAsync`  accepte un numéro de page. Les deux points d'interrogation représentent l'opérateur de fusion Null. L'opérateur de fusion Null définit une valeur par défaut pour un type nullable ; l'expression  `(page ?? 1)`  indique de renvoyer la valeur de  `page`  si elle a une valeur, ou de renvoyer 1 si  `page`  a la valeur Null.

## Ajouter des liens de pagination

Dans  `Views/Instructors/Index.cshtml` , remplacez le code existant par le code suivant. Les modifications apparaissent en surbrillance.

```

@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex + 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @nextDisabled">
    Next
</a>

```

L'instruction `@model` en haut de la page spécifie que la vue obtient désormais un objet `PaginatedList<T>` à la place d'un objet `List<T>`.

Les liens d'en-tête de colonne utilisent la chaîne de requête pour transmettre la chaîne de recherche actuelle au contrôleur afin que l'utilisateur puisse trier les résultats de filtrage :

```

<a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>

```

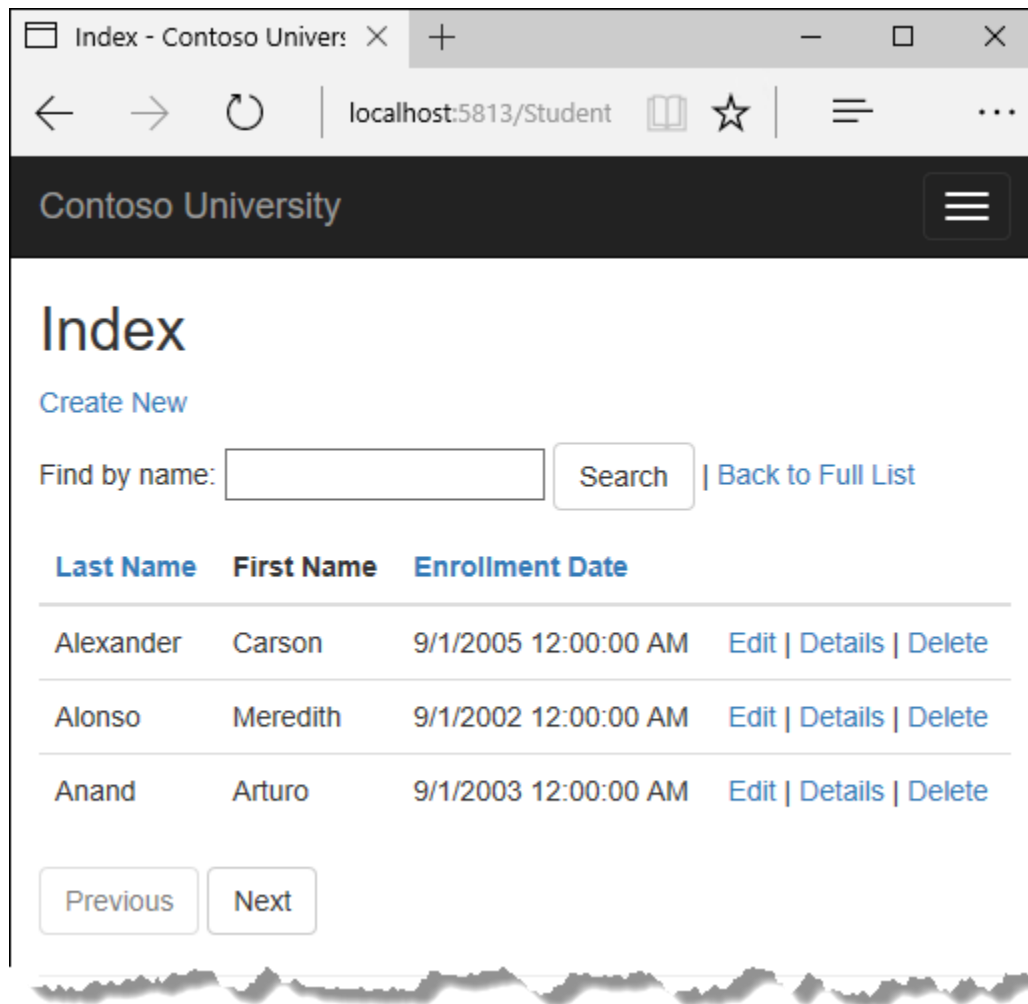
Les boutons de changement de page sont affichés par des Tag Helpers :

```

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>

```

Exécutez l'application et accédez à la page des étudiants.



Cliquez sur les liens de changement de page dans différents ordres de tri pour vérifier que le changement de page fonctionne. Ensuite, entrez une chaîne de recherche et essayez de changer de page à nouveau pour vérifier que le changement de page fonctionne correctement avec le tri et le filtrage.

## Créer une page About

Pour la page **About** du site web de Contoso University, vous afficherez le nombre d'étudiants inscrits pour chaque date d'inscription. Cela nécessite un regroupement et des calculs simples sur les groupes. Pour ce faire, vous devez effectuer les opérations suivantes :

- Créez une classe de modèle de vue pour les données que vous devez transmettre à la vue.
- Modifiez la méthode About dans le contrôleur Home.
- Modifiez la vue About.

### Créer le modèle de vue

Créez un dossier *SchoolViewModels* dans le dossier *Models*.

Dans le nouveau dossier, ajoutez un fichier de classe *EnrollmentDateGroup.cs* et remplacez le code du modèle par le code suivant :

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

### Modifier le contrôleur Home

Dans *HomeController.cs*, ajoutez les instructions using suivantes en haut du fichier :

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

Ajoutez une variable de classe pour le contexte de base de données immédiatement après l'accolade ouvrante de la classe et obtenez une instance du contexte à partir d'ASP.NET Core DI :

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
}
```

Remplacez la méthode `About` par le code suivant :

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

L'instruction LINQ regroupe les entités Student par date d'inscription, calcule le nombre d'entités dans chaque groupe et stocke les résultats dans une collection d'objets de modèle de vue `EnrollmentDateGroup`.

**Notes :** Dans la version 1.0 d'Entity Framework Core, le jeu de résultats entier est renvoyé au client et le regroupement est effectué sur le client. Dans certains scénarios, cela peut générer des problèmes de performances. Veuillez à tester les performances avec des volumes de données de production et, si nécessaire, utilisez des requêtes SQL brutes pour effectuer le regroupement sur le serveur.

Remplacez le code du fichier *Views/Home/About.cshtml* par le code suivant :



```

@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>

```

Exécutez l'application et accédez à la page About. Le nombre d'étudiants pour chaque date d'inscription s'affiche dans une table.

## Utilisation de la fonctionnalité de migrations

### À propos des migrations

Quand vous développez une nouvelle application, votre modèle de données change fréquemment et, chaque fois que le modèle change, il n'est plus en synchronisation avec la base de données. Vous avez démarré ce tutorial en configurant Entity Framework pour créer la base de données si elle n'existait pas. Ensuite, chaque fois que vous modifiez le modèle de données, en ajoutant, supprimant ou changeant des classes d'entité ou votre

classe DbContext, vous pouvez supprimer la base de données : EF en crée alors une nouvelle qui correspond au modèle et l'alimente avec des données de test.

Cette méthode pour conserver la base de données en synchronisation avec le modèle de données fonctionne bien jusqu'au déploiement de l'application en production. Quand l'application s'exécute en production, elle stocke généralement les données que vous voulez conserver, et vous ne voulez pas tout perdre chaque fois que vous apportez une modification, comme ajouter une nouvelle colonne. La fonctionnalité Migrations d'EF Core résout ce problème en permettant à EF de mettre à jour le schéma de base de données au lieu de créer une nouvelle base de données.

## À propos des packages NuGet de migration

Pour effectuer des migrations, vous pouvez utiliser la **console du Gestionnaire de package** ou l'interface de ligne de commande (CLI). Cette section montre comment utiliser des commandes CLI.

Les outils EF de l'interface de ligne de commande (CLI) sont fournis dans Microsoft.EntityFrameworkCore.Tools.DotNet. Pour installer ce package, ajoutez-le à la collection `DotNetCliToolReference` dans le fichier `.csproj`, comme indiqué.

**Remarque :** Vous devez installer ce package en modifiant le fichier `.csproj`. Vous ne pouvez pas utiliser la commande `install-package` ou le GUI (interface graphique utilisateur) du Gestionnaire de package. Vous pouvez modifier le fichier `.csproj` en cliquant sur le nom du projet dans **l'Explorateur de solutions** et en sélectionnant **Modifier ContosoUniversity.csproj**.

```
1 <Project Sdk="Microsoft.NET.Sdk.Web">
2
3   <PropertyGroup>
4     <TargetFramework>netcoreapp2.1</TargetFramework>
5   </PropertyGroup>
6
7   <ItemGroup>
8     <PackageReference Include="Microsoft.AspNetCore.App" />
9     <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.1.2" PrivateAssets="All" />
10    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1" />
11    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
12    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
13  </ItemGroup>
14
15 </Project>
16
```

## Changer la chaîne de connexion

Dans le fichier *appsettings.json*, remplacez le nom de la base de données dans la chaîne de connexion par ContosoUniversity2 ou par un autre nom que vous n'avez pas utilisé sur l'ordinateur que vous utilisez.

```
"ConnectionStrings": {  
  "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"  
},
```

Cette modification configure le projet de façon à ce que la première migration crée une nouvelle base de données. Ce n'est pas obligatoire pour commencer à utiliser les migrations, mais vous verrez plus tard pourquoi c'est judicieux.

**Notes :** Au lieu de changer le nom de la base de données, vous pouvez la supprimer. Utilisez l'**Explorateur d'objets SQL Server** (SSOX) ou la commande CLI `database drop` :

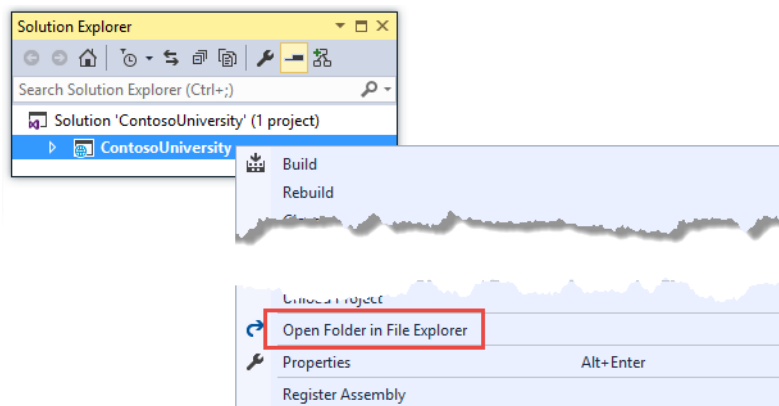
```
dotnet ef database drop
```

La section suivante explique comment exécuter des commandes CLI.

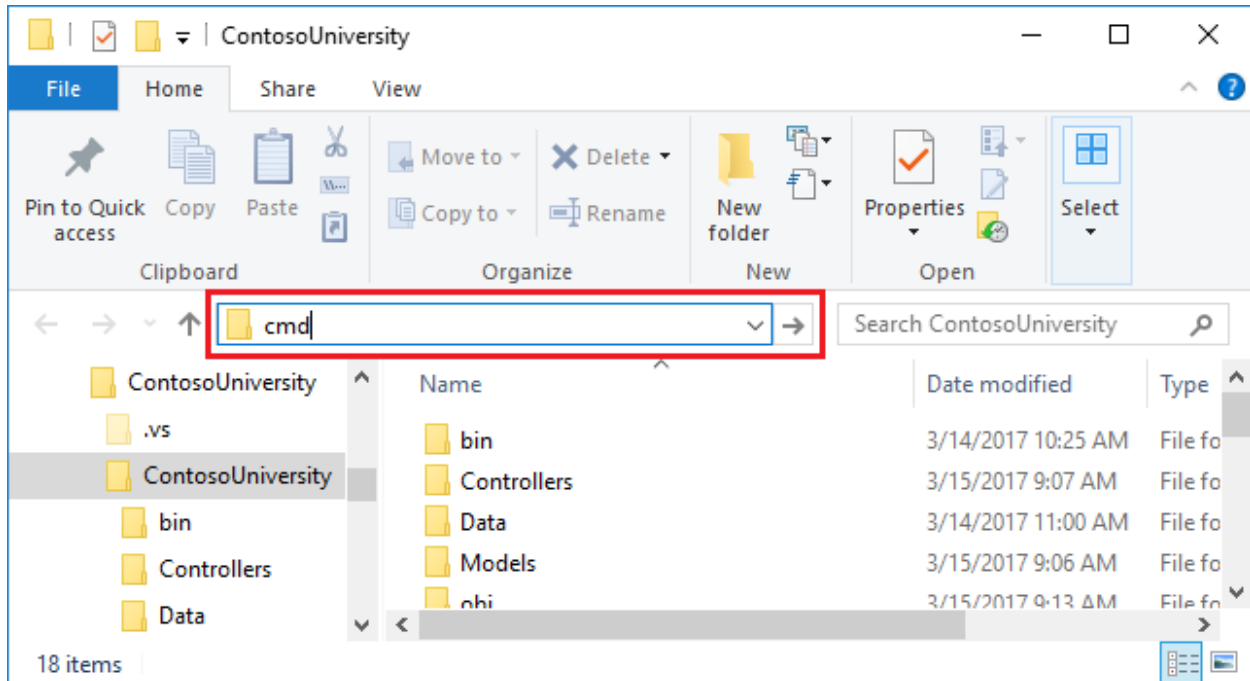
## Créer une migration initiale

Enregistrez vos modifications et générez le projet. Ouvrez ensuite une fenêtre Commande et accédez au dossier du projet. Voici un moyen rapide pour le faire :

- Dans l'**Explorateur de solutions**, cliquez sur le projet et choisissez **Ouvrir le dossier dans l'Explorateur de fichiers** dans le menu contextuel.



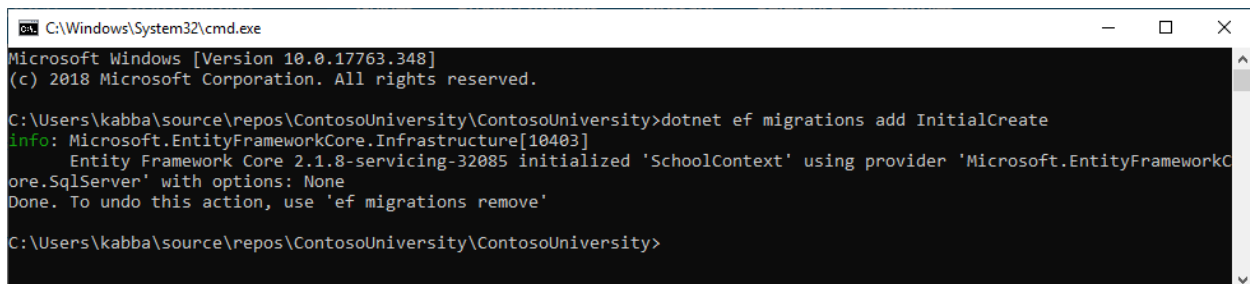
- Entrez « cmd » dans la barre d'adresses et appuyez sur Entrée.



- Entrez la commande suivante dans la fenêtre Commande :

```
dotnet ef migrations add InitialCreate
```

- Vous voyez une sortie similaire à celle-ci dans la fenêtre Commande :



Si vous voyez un message d'erreur « *Impossible d'accéder au fichier... ContosoUniversity.dll, car il est utilisé par un autre processus.* », recherchez l'icône IIS Express dans la barre d'état système de Windows, cliquez avec le bouton droit, puis cliquez sur **ContosoUniversity > Arrêter le Site**.

## Examiner les méthodes Up et Down

Quand vous avez exécuté la commande `migrations add`, EF a généré le code qui crée la base de données à partir de zéro. Ce code se trouve dans le dossier *Migrations*, dans le fichier nommé `<horodatage>_InitialCreate.cs`. La méthode `Up` de la classe `InitialCreate` crée les tables de base de données qui correspondent aux jeux d'entités du modèle de données, et la méthode `Down` les supprime, comme indiqué dans l'exemple suivant.

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        // Additional code not shown
    }
}
```

La fonctionnalité Migrations appelle la méthode `Up` pour implémenter les modifications du modèle de données pour une migration. Quand vous entrez une commande pour annuler la mise à jour, Migrations appelle la méthode `Down`.

Ce code est celui de la migration initiale qui a été créé quand vous avez entré la commande `migrations add InitialCreate`. Le paramètre de nom de la migration (« `InitialCreate` » dans l'exemple) est utilisé comme nom de fichier ; vous pouvez le choisir librement. Nous vous conseillons néanmoins de choisir un mot ou une expression qui résume ce qui est effectué dans la migration. Par exemple, vous pouvez nommer une migration ultérieure « `AjouterTableDépartement` ».

Si vous avez créé la migration initiale alors que la base de données existait déjà, le code de création de la base de données est généré, mais il n'est pas nécessaire de l'exécuter, car la base de données correspond déjà au modèle de données. Quand vous déployez

l'application sur un autre environnement où la base de données n'existe pas encore, ce code est exécuté pour créer votre base de données : il est donc judicieux de le tester au préalable. C'est la raison pour laquelle vous avez précédemment changé le nom de la base de données dans la chaîne de connexion : les migrations doivent pouvoir créer une base de données à partir de zéro.

## Capture instantanée du modèle de données

Migrations crée une *capture instantanée* du schéma de base de données actuel dans *Migrations/SchoolContextModelSnapshot.cs*. Quand vous ajoutez une migration, EF détermine ce qui a changé en comparant le modèle de données au fichier de capture instantanée.

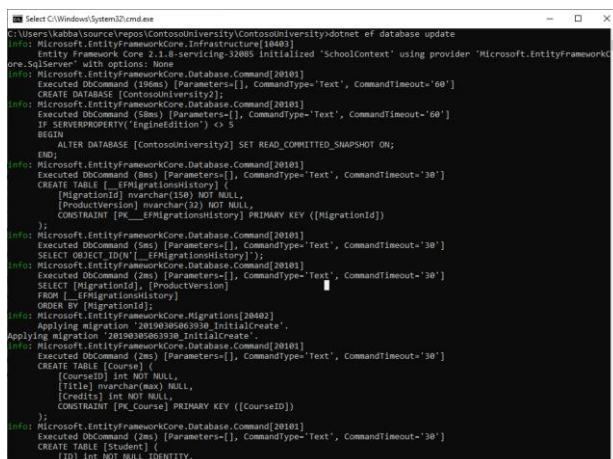
Lors de la suppression d'une migration, utilisez la commande `dotnet ef migrations remove`. `dotnet ef migrations remove` supprime la migration et garantit que la capture instantanée est correctement réinitialisée.

## Appliquer la migration

Dans la fenêtre Commande, entrez la commande suivante pour créer la base de données et ses tables.

```
dotnet ef database update
```

La sortie de la commande est similaire à la commande `migrations add`, à ceci près que vous voyez des journaux pour les commandes SQL qui configurent la base de données. La plupart des journaux sont omis dans l'exemple de sortie suivant. Si vous préférez ne pas voir ce niveau de détail dans les messages des journaux, vous pouvez changer le niveau de journalisation dans le fichier *appsettings.Development.json*.

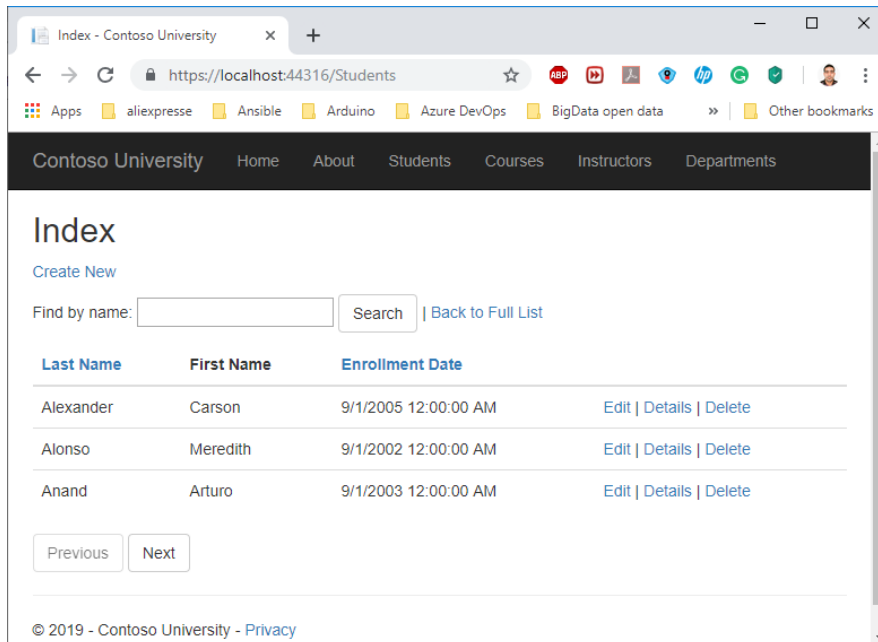


```

C:\Users\kabbal\source\repos\ContosoUniversity>dotnet ef database update
[info]: Microsoft.EntityFrameworkCore.Infrastructure[10400]
      Entity Framework Core 2.1.8 servicing 32085 initialized 'SchoolContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (149ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (58ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('Edition') < 5
      BEGIN
      ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT ON;
      END;
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
      [MigrationId] nvarchar(150) NOT NULL,
      [ProductVersion] nvarchar(32) NOT NULL,
      CONSTRAINT [PK_.__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT OBJECT_ID('.__EFMigrationsHistory');
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [MigrationId], [ProductVersion]
      FROM [__EFMigrationsHistory]
      ORDER BY [MigrationId];
[info]: Microsoft.EntityFrameworkCore.Migrations[20402]
      Applying migration '20190305063930_InitialCreate'.
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [Courses] (
      [CourseId] int NOT NULL,
      [Title] nvarchar(max) NULL,
      [Credits] int NOT NULL,
      CONSTRAINT [PK_Course] PRIMARY KEY ([CourseId])
      );
[info]: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [Students] (
      [ID] int NOT NULL IDENTITY,
```

Utilisez l'**Explorateur d'objets SQL Server** pour inspecter la base de données, comme vous l'avez fait dans la première section de ce tutorial. Vous pouvez noter l'ajout d'une table `__EFMigrationsHistory`, qui fait le suivi des migrations qui ont été appliquées à la base de données. Visualisez les données de cette table : vous y voyez une ligne pour la première migration. (Le dernier journal dans l'exemple de sortie CLI précédent montre l'instruction INSERT qui crée cette ligne.)

Exécutez l'application pour vérifier que tout fonctionne toujours comme avant.



## Comparer l'interface CLI et PMC

Les outils EF pour la gestion des migrations sont disponibles à partir de commandes CLI .NET Core ou d'applets de commande PowerShell dans la fenêtre **Console du Gestionnaire de package** de Visual Studio. Ce tutorial montre comment utiliser l'interface CLI, mais vous pouvez utiliser la console du Gestionnaire de package si vous préférez.

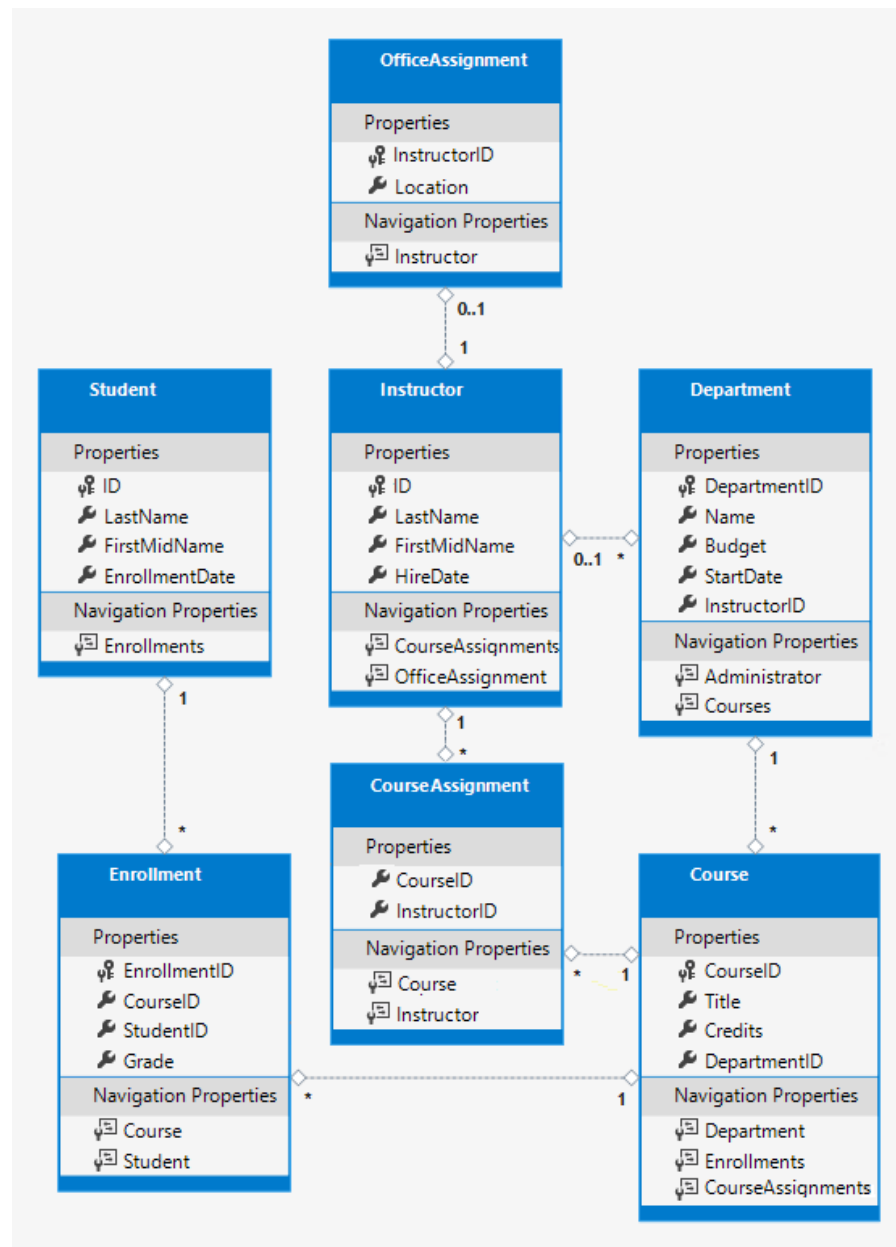
Les commandes EF pour la console du Gestionnaire de package se trouvent dans le package `Microsoft.EntityFrameworkCore.Tools`. Ce package étant inclus dans le métapackage `Microsoft.AspNetCore.App`, vous n'avez pas besoin d'ajouter une référence de package si votre application en comporte une pour `Microsoft.AspNetCore.App`.

**Important :** Il ne s'agit pas du même package que celui que vous installez pour l'interface CLI en modifiant le fichier `.csproj`. Le nom de celui-ci se termine par `Tools`, contrairement au nom du package CLI qui se termine par `Tools.DotNet`.

## Créer un modèle de données complexe

Dans les sections précédentes, vous avez travaillé avec un modèle de données simple composé de trois entités. Dans cette section, vous allez ajouter des entités et des relations, et vous personnaliserez le modèle de données en spécifiant des règles de mise en forme, de validation et de mappage de base de données.

Lorsque vous aurez terminé, les classes d'entité composeront le modèle de données complet indiqué dans l'illustration suivante :





## Personnaliser le modèle de données

Dans cette section, vous allez apprendre à personnaliser le modèle de données en utilisant des attributs qui spécifient des règles de mise en forme, de validation et de mappage de base de données. Ensuite, dans plusieurs des sections suivantes, vous allez créer le modèle de données School complet en ajoutant des attributs aux classes que vous avez déjà créées et en créant de nouvelles classes pour les autres types d'entités dans le modèle.

### Attribut `DataType`

Pour les dates d'inscription des étudiants, toutes les pages web affichent l'heure avec la date, alors que seule la date vous intéresse dans ce champ. Vous pouvez avoir recours aux attributs d'annotation de données pour apporter une modification au code, permettant de corriger le format d'affichage dans chaque vue qui affiche ces données. Pour voir un exemple de la procédure à suivre, vous allez ajouter un attribut à la propriété `EnrollmentDate` dans la classe `Student`.

Dans `Models/Student.cs`, ajoutez une instruction `using` pour l'espace de noms `System.ComponentModel.DataAnnotations` et ajoutez les attributs `DataType` et `DisplayFormat` à la propriété `EnrollmentDate`, comme indiqué dans l'exemple suivant :  
C#

```
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; }
    }
}
```

L'attribut `DataType` sert à spécifier un type de données qui est plus spécifique que le type intrinsèque de la base de données. Dans le cas présent, nous voulons uniquement effectuer le suivi de la date, pas de la date et de l'heure. L'énumération `DataType` fournit

de nombreux types de données, tels que `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress`, etc. L'attribut `DataType` peut également permettre à l'application de fournir automatiquement des fonctionnalités propres au type. Par exemple, vous pouvez créer un lien `mailto:` pour `DataType.EmailAddress`, et vous pouvez fournir un sélecteur de date pour `DataType.Date` dans les navigateurs qui prennent en charge HTML5. L'attribut `DataType` émet des attributs HTML 5 `data-`compréhensibles par les navigateurs HTML 5. Les attributs `DataType` ne fournissent aucune validation.

`DataType.Date` ne spécifie pas le format de la date qui s'affiche. Par défaut, le champ de données est affiché conformément aux formats par défaut basés sur l'objet `CultureInfo` du serveur.

L'attribut `DisplayFormat` est utilisé pour spécifier explicitement le format de date :

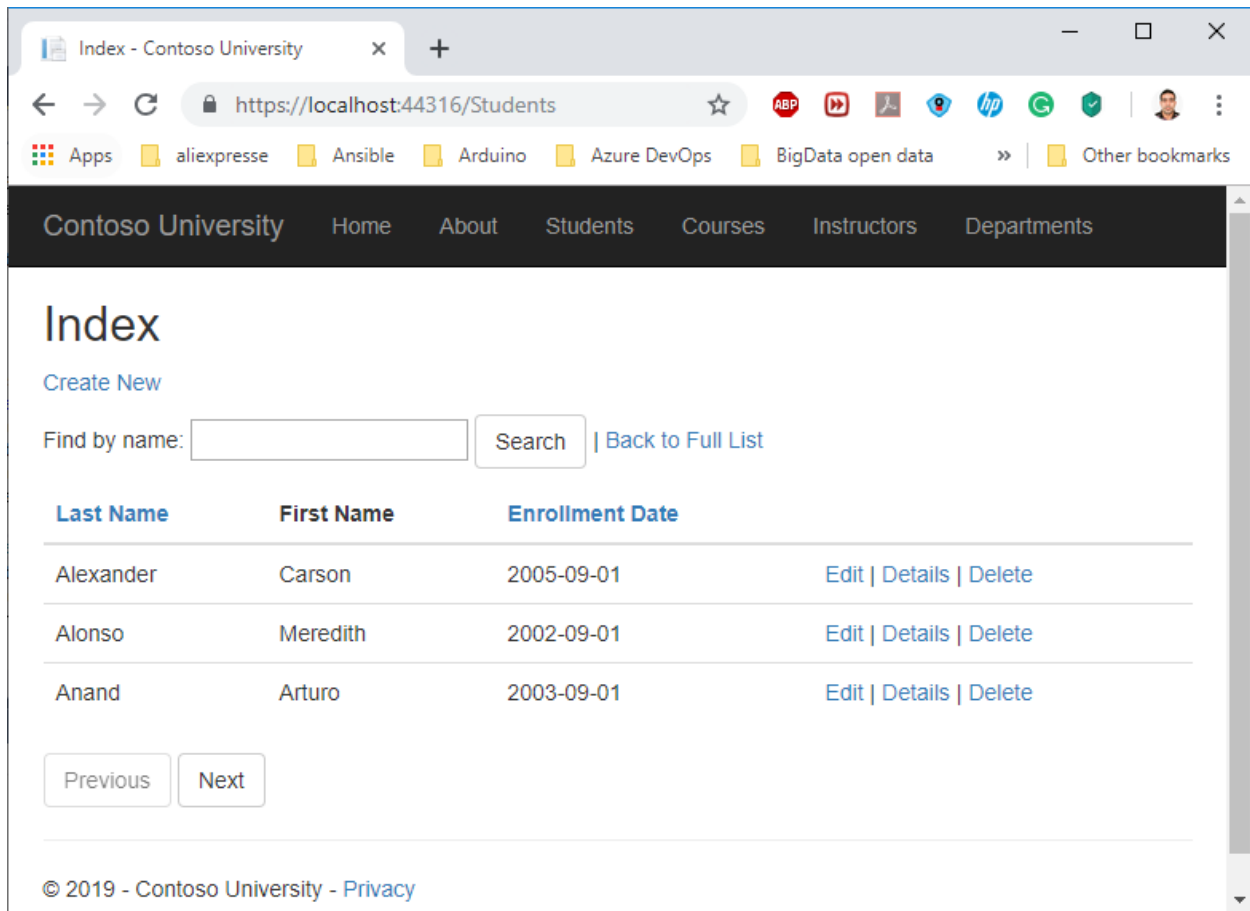
```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

Le paramètre `ApplyFormatInEditMode` indique que la mise en forme doit également être appliquée quand la valeur est affichée dans une zone de texte à des fins de modification. (Ceci peut ne pas être souhaitable pour certains champs ; par exemple, pour les valeurs monétaires, vous ne souhaitez peut-être pas que le symbole monétaire figure dans la zone de texte.)

Vous pouvez utiliser l'attribut `DisplayFormat` seul, mais il est généralement judicieux d'utiliser également l'attribut `DataType`. L'attribut `DataType` donne la sémantique des données au lieu d'expliquer comment les afficher à l'écran. Il présente, par ailleurs, les avantages suivants, dont vous ne bénéficiez pas avec `DisplayFormat` :

- Le navigateur peut activer des fonctionnalités HTML5 (par exemple pour afficher un contrôle de calendrier, le symbole monétaire correspondant aux paramètres régionaux, des liens de messagerie, une certaine validation des entrées côté client, etc.).
- Par défaut, le navigateur affiche les données à l'aide du format correspondant à vos paramètres régionaux.

Exécutez l'application, accédez à la page d'index des étudiants et notez que les heures ne sont plus affichées pour les dates d'inscription. La même chose est vraie pour toute vue qui utilise le modèle `Student`.



## Attribut StringLength

Vous pouvez également spécifier les règles de validation de données et les messages d'erreur de validation à l'aide d'attributs. L'attribut `StringLength` définit la longueur maximale dans la base de données et assure la validation côté client et côté serveur pour ASP.NET Core MVC. Vous pouvez également spécifier la longueur de chaîne minimale dans cet attribut, mais la valeur minimale n'a aucun impact sur le schéma de base de données.

Supposons que vous voulez garantir que les utilisateurs n'entrent pas plus de 50 caractères pour un nom. Pour ajouter cette limitation, ajoutez des attributs `StringLength` aux propriétés `LastName` et `FirstMidName`, comme indiqué dans l'exemple suivant :

```

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)]
        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; }
    }
}

```

L'attribut `StringLength` n'empêche pas un utilisateur d'entrer un espace blanc comme nom. Vous pouvez utiliser l'attribut `RegularExpression` pour appliquer des restrictions à l'entrée. Par exemple, le code suivant exige que le premier caractère soit en majuscule et que les autres caractères soient alphabétiques :

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

L'attribut `MaxLength` fournit des fonctionnalités similaires à l'attribut `StringLength`, mais n'assure pas la validation côté client.

Le modèle de base de données a maintenant changé d'une manière qui nécessite la modification du schéma de base de données. Vous allez utiliser des migrations pour mettre à jour le schéma sans perdre les données que vous avez éventuellement ajoutées à la base de données via l'interface utilisateur de l'application.

Enregistrez vos modifications et générez le projet. Ensuite, ouvrez la fenêtre de commande dans le dossier du projet et entrez les commandes suivantes :

```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

La commande `migrations add` vous avertit qu'une perte de données peut se produire, car la modification raccourcit la longueur maximale de deux colonnes. Migrations crée un fichier nommé `<timeStamp>_MaxLengthOnNames.cs`. Ce fichier contient du code dans la méthode `Up` qui met à jour la base de données pour qu'elle corresponde au modèle de données actuel. La commande `database update` a exécuté ce code.

L'horodatage utilisé comme préfixe du nom de fichier migrations est utilisé par Entity Framework pour ordonner les migrations. Vous pouvez créer plusieurs migrations avant d'exécuter la commande de mise à jour de base de données, puis toutes les migrations sont appliquées dans l'ordre où elles ont été créées.

Exécutez l'application, sélectionnez l'onglet **Students**, cliquez sur **Create New** et essayez d'entrer un nom de plus de 50 caractères. L'application doit empêcher cette opération.

## Attribut Column

Vous pouvez également utiliser des attributs pour contrôler la façon dont les classes et les propriétés sont mappées à la base de données. Supposons que vous aviez utilisé le nom `FirstMidName` pour le champ de prénom, car le champ peut également contenir un deuxième prénom. Mais vous souhaitez que la colonne de base de données soit nommée `FirstName`, car les utilisateurs qui écriront des requêtes ad-hoc par rapport à la base de données sont habitués à ce nom. Pour effectuer ce mappage, vous pouvez utiliser l'attribut `Column`.

L'attribut `Column` spécifie que lorsque la base de données sera créée, la colonne de la table `Student` qui est mappée sur la propriété `FirstMidName` sera nommée `FirstName`. En d'autres termes, lorsque votre code fait référence à `Student.FirstMidName`, les données proviennent de la colonne `FirstName` de la table `Student` ou y sont mises à jour. Si vous ne nommez pas les colonnes, elles obtiennent le nom de la propriété.

Dans le fichier `Student.cs`, ajoutez une instruction `using` pour `System.ComponentModel.DataAnnotations.Schema` et ajoutez l'attribut de nom de colonne à la propriété `FirstMidName`, comme indiqué dans le code en surbrillance suivant :

```

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)]
        [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; }
    }
}

```

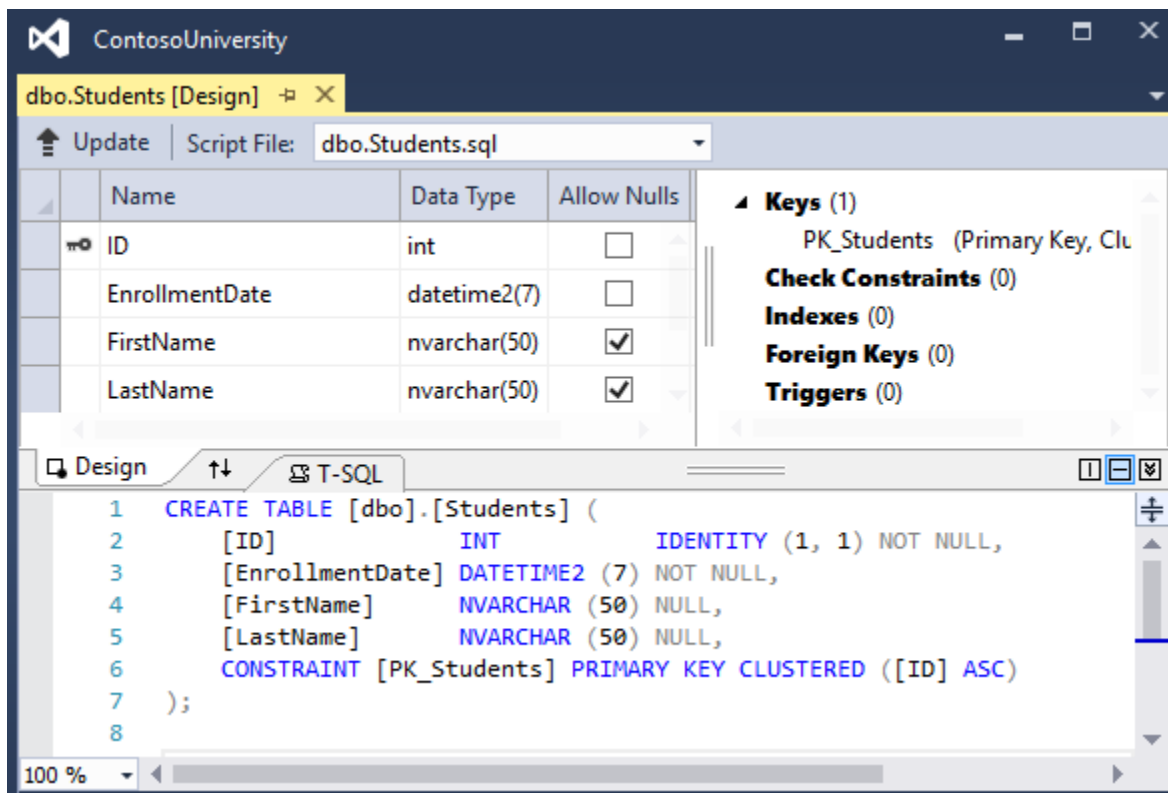
L'ajout de l'attribut `Column` change le modèle sur lequel repose `SchoolContext`, donc il ne correspond pas à la base de données.

Enregistrez vos modifications et générez le projet. Ensuite, ouvrez la fenêtre de commande dans le dossier du projet et entrez les commandes suivantes pour créer une autre migration :

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

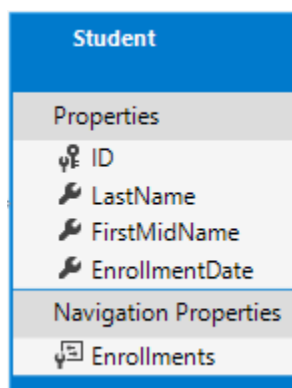
Dans l'**Explorateur d'objets SQL Server**, ouvrez le concepteur de tables `Student` en double-cliquant sur la table **Student**.



Avant d'appliquer les deux premières migrations, les colonnes de nom étaient de type nvarchar(MAX). Elles sont maintenant de type nvarchar(50) et le nom de colonne FirstMidName a été remplacé par FirstName.

**Notes :** Si vous essayez de compiler avant d'avoir fini de créer toutes les classes d'entité dans les sections suivantes, vous pouvez obtenir des erreurs de compilation.

## Modifications apportées à l'entité Student



Dans *Models/Student.cs*, remplacez le code que vous avez ajouté précédemment par le code suivant. Les modifications apparaissent en surbrillance.

```

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)]
        [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; }
    }
}

```

## Attribut Required

L'attribut `Required` fait des propriétés de nom des champs obligatoires. L'attribut `Required` n'est pas requis pour les types non nullables tels que les types valeur (`DateTime`, `int`, `double`, `float`, etc.). Les types qui n'acceptent pas les valeurs `Null` sont traités automatiquement comme des champs requis.

Vous pouvez supprimer l'attribut `Required` et le remplacer par un paramètre de longueur minimale pour l'attribut `StringLength` :

```

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

```

## Attribut Display

L'attribut `Display` spécifie que la légende pour les zones de texte doit être « First Name », « Last Name », « Full Name » et « Enrollment Date », au lieu du nom de propriété dans chaque instance (qui n'a pas d'espace pour séparer les mots).



## Propriété calculée FullName

`FullName` est une propriété calculée qui retourne une valeur créée par concaténation de deux autres propriétés. Par conséquent, elle a uniquement un accesseur `get` et aucune colonne `FullName` n'est générée dans la base de données.

## Créer une entité Instructor

Instructor	
Properties	
PK ID	
LastName	
FirstMidName	
HireDate	
Navigation Properties	
CourseAssignments	
OfficeAssignment	

Créez *Models/Instructor.cs*, en remplaçant le code du modèle par le code suivant :

```
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; }
    }
}
```

Notez que plusieurs propriétés sont identiques dans les entités `Student` et `Instructor`. Dans une section suivante de ce tutorial, vous allez refactoriser ce code pour éliminer la redondance.

Vous pouvez placer plusieurs attributs sur une seule ligne et écrire les attributs `HireDate` comme suit :

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

## Propriétés de navigation `CourseAssignments` et `OfficeAssignment`

Les propriétés `CourseAssignments` et `OfficeAssignment` sont des propriétés de navigation.

Un formateur peut animer un nombre quelconque de cours, de sorte que `CourseAssignments` est défini comme une collection.

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




Si une propriété de navigation peut contenir plusieurs entités, son type doit être une liste dans laquelle les entrées peuvent être ajoutées, supprimées et mises à jour. Vous pouvez spécifier `ICollection<T>` ou un type tel que `List<T>` ou `HashSet<T>`. Si vous spécifiez `ICollection<T>`, EF crée une collection `HashSet<T>` par défaut.

La raison pour laquelle ce sont des entités `CourseAssignment` est expliquée ci-dessous dans la section sur les relations plusieurs-à-plusieurs.

Les règles d'entreprise de Contoso University stipulent qu'un formateur peut avoir au plus un bureau, de sorte que la propriété `OfficeAssignment` contient une seule entité `OfficeAssignment` (qui peut être null si aucun bureau n'est affecté).

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

## Créer une entité OfficeAssignment

OfficeAssign...	
Properties	
	InstructorID
	Location
Navigation Properties	
	Instructor

Créez *Models/OfficeAssignment.cs* avec le code suivant :

```
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; }
    }
}
```

### Attribut Key

Il existe une relation un-à-zéro-ou-un entre les entités Instructor et OfficeAssignment. Une affectation de bureau existe uniquement en relation avec le formateur auquel elle est affectée. Par conséquent, sa clé primaire est également sa clé étrangère pour l'entité Instructor. Mais Entity Framework ne peut pas reconnaître automatiquement InstructorID comme clé primaire de cette entité, car son nom ne suit pas la convention de nommage d'ID ou de classnameID. Par conséquent, l'attribut `Key` est utilisé pour l'identifier comme clé :

```
[Key]
public int InstructorID { get; set; }
```

Vous pouvez également utiliser l'attribut `Key` si l'entité a sa propre clé primaire, mais que vous souhaitez nommer la propriété autrement que `classNameID` ou `ID`.

Par défaut, EF traite la clé comme n'étant pas générée par la base de données, car la colonne est utilisée pour une relation d'identification.

## Propriété de navigation du formateur

L'entité `Instructor` a une propriété de navigation `OfficeAssignment` nullable (parce qu'un formateur n'a peut-être pas d'affectation de bureau) et l'entité `OfficeAssignment` a une propriété de navigation `Instructor` non nullable (comme une affectation de bureau ne peut pas exister sans formateur, `InstructorID` est non nullable). Lorsqu'une entité `Instructor` a une entité `OfficeAssignment` associée, chaque entité a une référence à l'autre dans sa propriété de navigation.

Vous pouvez placer un attribut `[Required]` sur la propriété de navigation du formateur pour spécifier qu'il doit y avoir un formateur associé, mais vous n'êtes pas obligé de le faire, car la clé étrangère `InstructorID` (qui est également la clé pour cette table) est non nullable.

## Modifier l'entité `Course`

Course
Properties
ψ CourseID
🔑 Title
🔑 Credits
🔑 DepartmentID
Navigation Properties
ψ Department
ψ Enrollments
ψ CourseAssignments

Dans `Models/Course.cs`, remplacez le code que vous avez ajouté précédemment par le code suivant. Les modifications apparaissent en surbrillance.

```

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; }
    }
}

```

L'entité de cours a une propriété de clé étrangère `DepartmentID` qui pointe sur l'entité `Department` associée et elle a une propriété de navigation `Department`.

Entity Framework ne vous demande pas d'ajouter une propriété de clé étrangère à votre modèle de données lorsque vous avez une propriété de navigation pour une entité associée. EF crée automatiquement des clés étrangères dans la base de données partout où elles sont nécessaires et crée des propriétés **Shadow** pour elles. Mais le fait d'avoir la clé étrangère dans le modèle de données peut rendre les mises à jour plus simples et plus efficaces. Par exemple, lorsque vous récupérez une entité de cours à modifier, l'entité `Department` a la valeur `Null` si vous ne la chargez pas. Par conséquent, lorsque vous mettez à jour l'entité de cours, vous devriez tout d'abord récupérer l'entité `Department`. Lorsque la propriété de clé étrangère `DepartmentID` est incluse dans le modèle de données, vous n'avez pas besoin de récupérer l'entité `Department` avant de mettre à jour.

## Attribut DatabaseGenerated

L'attribut `DatabaseGenerated` avec le paramètre `None` sur la propriété `CourseID` spécifie que les valeurs de clé primaire sont fournies par l'utilisateur au lieu d'être générées par la base de données.

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

Par défaut, Entity Framework suppose que les valeurs de clé primaire sont générées par la base de données. C'est ce que vous souhaitez dans la plupart des scénarios. Toutefois, pour les entités `Course`, vous allez utiliser un numéro de cours spécifié par l'utilisateur comme une série de 1000 pour un département, une série de 2000 pour un autre département, etc.

L'attribut `DatabaseGenerated` peut également être utilisé pour générer des valeurs par défaut, comme dans le cas des colonnes de base de données utilisées pour enregistrer la date à laquelle une ligne a été créée ou mise à jour.

## Propriétés de clé étrangère et de navigation

Les propriétés de clé étrangère et les propriétés de navigation dans l'entité `Course` reflètent les relations suivantes :

Un cours est affecté à un seul département, donc il existe une clé étrangère `DepartmentID` et une propriété de navigation `Department` pour les raisons mentionnées ci-dessus.

```
public int DepartmentID { get; set; }  
public Department Department { get; set; }
```

Un cours peut avoir un nombre quelconque d'étudiants inscrits, si bien que la propriété de navigation `Enrollments` est une collection :

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

Un cours peut être animé par plusieurs formateurs, si bien que la propriété de navigation `CourseAssignments` est une collection (le type `CourseAssignment` est expliqué ultérieurement) :

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

## Créer l'entité Department

Department
Properties
DepartmentID
Name
Budget
StartDate
InstructorID
Navigation Properties
Administrator
Courses

Créez *Models/Department.cs* avec le code suivant :

```
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; }
    }
}
```

### Attribut Column

Précédemment, vous avez utilisé l'attribut `Column` pour changer le mappage de noms de colonne. Dans le code de l'entité `Department`, l'attribut `Column` sert à modifier le mappage des types de données SQL afin que la colonne soit définie à l'aide du type monétaire (money) SQL Server dans la base de données :

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

Le mappage de colonnes n'est généralement pas nécessaire, car Entity Framework choisit le type de données SQL Server approprié en fonction du type CLR que vous définissez pour la propriété. Le type CLR `decimal` est mappé à un type SQL Server `decimal`. Toutefois, dans ce cas, vous savez que la colonne contiendra des montants en devise et que le type de données monétaire est plus approprié pour cela.

## Propriétés de clé étrangère et de navigation

Les propriétés de clé étrangère et de navigation reflètent les relations suivantes :

Un département peut ou non avoir un administrateur, et un administrateur est toujours un formateur. Par conséquent, la propriété `InstructorID` est incluse en tant que clé étrangère à l'entité `Instructor`, et un point d'interrogation est ajouté après la désignation du type `int` pour marquer la propriété comme nullable. La propriété de navigation est nommée `Administrator`, mais elle contient une entité `Instructor` :

```
public int? InstructorID { get; set; }  
public Instructor Administrator { get; set; }
```

Un département peut avoir de nombreux cours, si bien qu'il existe une propriété de navigation `Courses` :

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

**Notes :** Par convention, Entity Framework permet la suppression en cascade pour les clés étrangères non nullables et pour les relations plusieurs à plusieurs. Cela peut entraîner des règles de suppression en cascade circulaires, qui provoqueront une exception lorsque vous essaierez d'ajouter une migration. Par exemple, si vous n'avez pas défini la propriété `Department.InstructorID` comme nullable, EF configure une règle de suppression en cascade pour supprimer le formateur lorsque vous supprimez le département, ce qui n'est pas ce que vous voulez. Si vos règles d'entreprise exigent que la propriété `InstructorID` soit non nullable, vous devez utiliser l'instruction d'API Fluent suivante pour désactiver la suppression en cascade sur la relation :



```

modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)

```

## Modifier l'entité Enrollment

Enrollment
Properties
<ul style="list-style-type: none"> <li>EnrollmentID</li> <li>CourseID</li> <li>StudentID</li> <li>Grade</li> </ul>
Navigation Properties
<ul style="list-style-type: none"> <li>Course</li> <li>Student</li> </ul>

Dans *Models/Enrollment.cs*, remplacez le code que vous avez ajouté précédemment par le code suivant :

```

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; }
    }
}

```

## Propriétés de clé étrangère et de navigation

Les propriétés de clé étrangère et de navigation reflètent les relations suivantes :

Un enregistrement d'inscription est utilisé pour un cours unique, si bien qu'il existe une propriété de clé étrangère `CourseID` et une propriété de navigation `Course` :

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

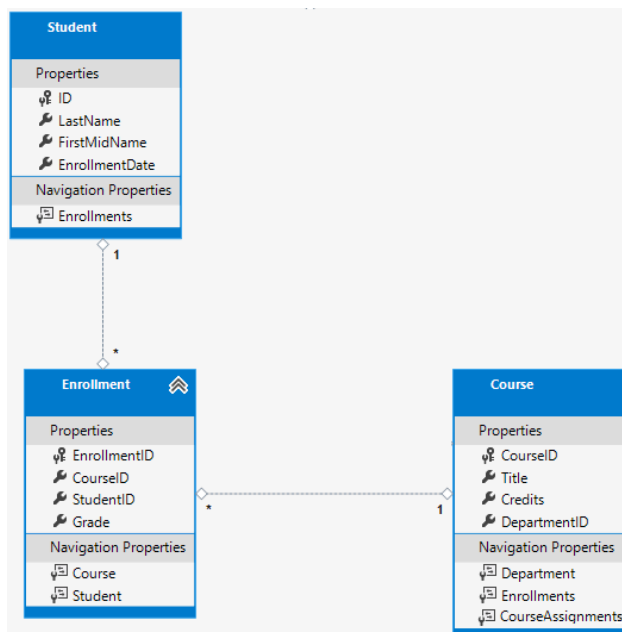
Un enregistrement d'inscription est utilisé pour un étudiant unique, si bien qu'il existe une propriété de clé étrangère `StudentID` et une propriété de navigation `Student` :

```
public int StudentID { get; set; }  
public Student Student { get; set; }
```

## Relations plusieurs-à-plusieurs

Il existe une relation plusieurs-à-plusieurs entre les entités `Student` et `Course`, et l'entité `Enrollment` fonctionne comme une table de jointure plusieurs-à-plusieurs *avec une charge utile* dans la base de données. « Avec une charge utile » signifie que la table `Enrollment` contient des données supplémentaires en plus des clés étrangères pour les tables jointes (dans ce cas, une clé primaire et une propriété `Grade`).

L'illustration suivante montre à quoi ressemblent ces relations dans un diagramme d'entité. (Ce diagramme a été généré à l'aide d'Entity Framework Power Tools pour EF 6.x ; la création du diagramme ne fait pas partie de ce tutorial, elle est uniquement utilisée ici à titre d'illustration.)







Chaque ligne de relation comporte un 1 à une extrémité et un astérisque (\*) à l'autre, ce qui indique une relation un-à-plusieurs.

Si la table Enrollment n'incluait pas d'informations de notes, elle aurait uniquement besoin de contenir les deux clés étrangères CourseID et StudentID. Dans ce cas, ce serait une table de jointure plusieurs-à-plusieurs sans charge utile (ou une table de jointure pure) dans la base de données. Les entités Instructor and Course ont ce type de relation plusieurs-à-plusieurs, et l'étape suivante consiste à créer une classe d'entité qui fonctionnera comme une table de jointure sans charge utile.

(EF 6.x prend en charge les tables de jointure implicites pour les relations plusieurs-à-plusieurs, mais EF Core ne le fait pas.)

## Entité CourseAssignment

CourseAssignment	
Properties	
	CourseID
	InstructorID
Navigation Properties	
	Course
	Instructor

Créez *Models/CourseAssignment.cs* avec le code suivant :

```
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; }
    }
}
```

## Noms des entités de jointure

Une table de jointure est requise dans la base de données pour la relation plusieurs-à-plusieurs entre formateurs et cours, et elle doit être représentée par un jeu d'entités. Il est courant de nommer une entité de jointure `EntityName1EntityName2`, ce qui donnerait dans ce cas `CourseInstructor`. Toutefois, nous vous recommandons de choisir un nom qui décrit la relation. Les modèles de données sont simples au départ, puis croissent, avec des jointures sans charge utile qui obtiennent souvent des charges utiles plus tard. Si vous commencez avec un nom d'entité descriptif, vous n'aurez pas à le modifier par la suite. Dans l'idéal, l'entité de jointure aura son propre nom (éventuellement un mot unique) naturel dans le domaine d'entreprise. Par exemple, les livres et les clients pourraient être liés par le biais d'évaluations. Pour cette relation, `CourseAssignment` est un meilleur choix que `CourseInstructor`.

## Clé composite

Étant donné que les clés étrangères ne sont pas nullables et qu'elles identifient ensemble de façon unique chaque ligne de la table, une clé primaire distincte n'est pas requise. Les propriétés `InstructorID` et `CourseID` doivent fonctionner comme une clé primaire composite. La seule façon d'identifier des clés primaires composites pour EF consiste à utiliser l'*API Fluent* (ce n'est pas possible à l'aide d'attributs). Vous allez voir comment configurer la clé primaire composite dans la section suivante.

La clé composite garantit qu'en ayant plusieurs lignes pour un cours et plusieurs lignes pour un formateur, vous ne puissiez pas avoir plusieurs lignes pour les mêmes formateur et cours. L'entité de jointure `Enrollment` définit sa propre clé primaire, si bien que les doublons de ce type sont possibles. Pour éviter ces doublons, vous pourriez ajouter un index unique sur les champs de clé étrangère ou configurer `Enrollment` avec une clé composite primaire similaire à `CourseAssignment`.

## Mettre à jour le contexte de base de données

Ajoutez le code en surbrillance suivant au fichier `Data/SchoolContext.cs` :

```

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 });
        }
    }
}

```

Ce code ajoute les nouvelles entités et configure la clé primaire composite de l'entité `CourseAssignment`.

## À propos de l'alternative d'API Fluent

Le code dans la méthode `OnModelCreating` de la classe `DbContext` utilise l'*API Fluent* pour configurer le comportement EF. L'API est appelée « fluent », car elle est souvent utilisée pour enchaîner une série d'appels de méthode en une seule instruction, comme dans cet exemple tiré de la documentation d'EF Core :

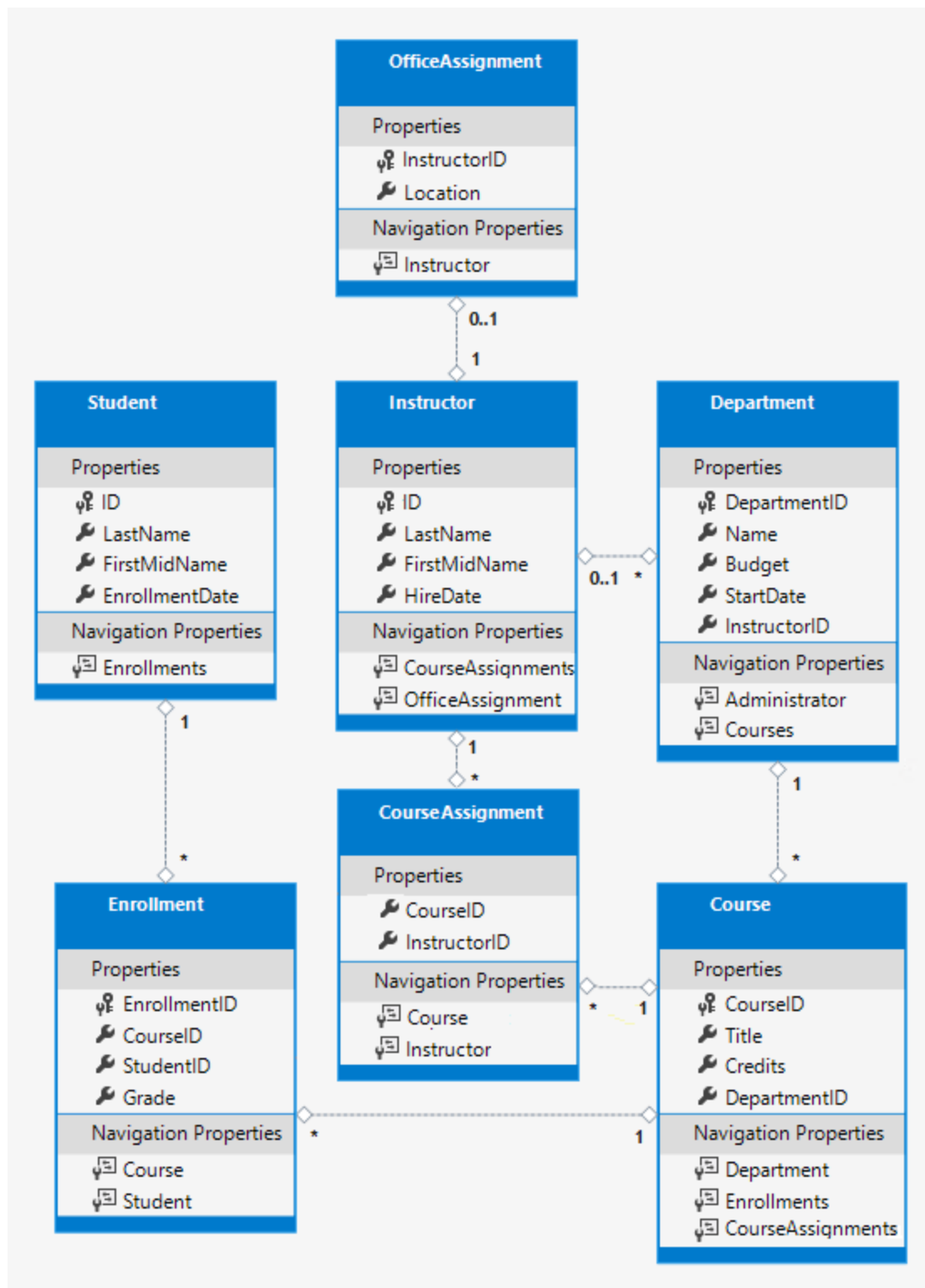
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

Dans ce tutorial, vous utilisez l'API Fluent uniquement pour le mappage de base de données que vous ne pouvez pas faire avec des attributs. Toutefois, vous pouvez également utiliser l'API Fluent pour spécifier la majorité des règles de mise en forme, de validation et de mappage que vous pouvez spécifier à l'aide d'attributs. Certains attributs, tels que `MinimumLength`, ne peuvent pas être appliqués avec l'API Fluent. Comme mentionné précédemment, `MinimumLength` ne change pas le schéma, il applique uniquement une règle de validation côté client et côté serveur.

Certains développeurs préfèrent utiliser exclusivement l'API Fluent afin de conserver des classes d'entité « propres ». Vous pouvez combiner les attributs et l'API Fluent si vous le voulez, et il existe quelques personnalisations qui peuvent être effectuées uniquement à l'aide de l'API Fluent, mais en général la pratique recommandée consiste à choisir l'une de ces deux approches et à l'utiliser constamment, autant que possible. Si vous utilisez ces deux approches, notez que partout où il existe un conflit, l'API Fluent a priorité sur les attributs.

## Diagramme des entités montrant les relations

L'illustration suivante montre le diagramme que les outils Entity Framework Power Tools créent pour le modèle School complet.



Outre les lignes de relation un-à-plusieurs (1 à \*), vous pouvez voir ici la ligne de relation un-à-zéro-ou-un (1 à 0..1) entre les entités Instructor et OfficeAssignment et la ligne de relation zéro-ou-un-à-plusieurs (0..1 à \*) entre les entités Instructor et Department.

## Peupler la base de données avec des données de test

Remplacez le code dans le fichier *Data/DbInitializer.cs* par le code suivant afin de fournir des données initiales pour les nouvelles entités que vous avez créées.

```
9  public static class DbInitializer
10 {
11     public static void Initialize(SchoolContext context)
12     {
13         //context.Database.EnsureCreated();
14
15         // Look for any students.
16         if (context.Students.Any())
17         {
18             return; // DB has been seeded
19         }
20
21         var students = new Student[]
22         {
23             new Student { FirstMidName = "Carson", LastName = "Alexander",
24                 EnrollmentDate = DateTime.Parse("2010-09-01") },
25         };
26
27         foreach (Student s in students)
28         {
29             context.Students.Add(s);
30         }
31         context.SaveChanges();
32
33         var instructors = new Instructor[]
34         {
35             new Instructor { FirstMidName = "Kim", LastName = "Abercrombie",
36                 HireDate = DateTime.Parse("1995-03-11") },
37         };
38
39         foreach (Instructor i in instructors)
40         {
41             context.Instructors.Add(i);
42         }
43         context.SaveChanges();
44
45         var departments = new Department[]
46         {
47             new Department { Name = "English", Budget = 350000,
48                 StartDate = DateTime.Parse("2007-09-01"),
49                 InstructorID = instructors.Single( i => i.LastName == "Abercrombie").ID },
50         };
51
52         foreach (Department d in departments)
53         {
54             context.Departments.Add(d);
55         }
56         context.SaveChanges();
57
58         var courses = new Course[]
59         {
60             new Course { CourseID = 1050, Title = "Chemistry", Credits = 3,
61                 DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
62             },
63         };
64     }
```



```

65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
}

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" },
};

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
    },
};

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
    },
};

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();
}
}
}

```

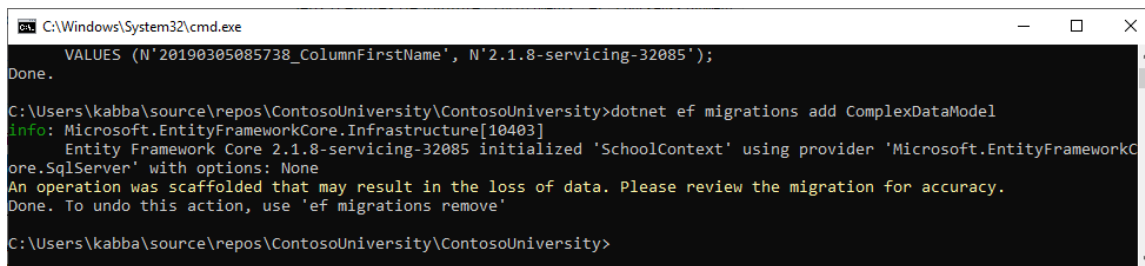
Comme vous l'avez vu dans la première section de ce tutorial, la majeure partie de ce code crée simplement de nouveaux objets d'entité et charge des exemples de données dans les propriétés requises pour les tests. Notez la façon dont les relations plusieurs à plusieurs sont gérées : le code crée des relations en créant des entités dans les jeux d'entités de jointure `Enrollments` et `CourseAssignment`.

## Ajouter une migration

Enregistrez vos modifications et générez le projet. Ensuite, ouvrez la fenêtre de commande dans le dossier du projet et entrez la commande `migrations add` (n'exécutez pas encore la commande de mise à jour de base de données) :

```
dotnet ef migrations add ComplexDataModel
```

Vous obtenez un avertissement concernant une perte possible de données.



```
C:\Windows\System32\cmd.exe
VALUES (N'20190305085738_ColumnFirstName', N'2.1.8-servicing-32085');
Done.

C:\Users\kabba\source\repos\ContosoUniversity\ContosoUniversity>dotnet ef migrations add ComplexDataModel
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.1.8-servicing-32085 initialized 'SchoolContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
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'

C:\Users\kabba\source\repos\ContosoUniversity\ContosoUniversity>
```

Si vous tentiez d'exécuter la commande `database update` à ce stade (ne le faites pas encore), vous obtiendriez l'erreur suivante :

L'instruction `ALTER TABLE` est en conflit avec la contrainte `FOREIGN KEY` « `FK_dbo.Course_dbo.Department_DepartmentID` ». Le conflit s'est produit dans la base de données « `ContosoUniversity` », table « `dbo.Department` », colonne « `DepartmentID` ».

Parfois, lorsque vous exécutez des migrations avec des données existantes, vous devez insérer des données stub dans la base de données pour répondre aux contraintes de clé étrangère. Le code généré dans la méthode `Up` ajoute une clé étrangère `DepartmentID` non nullable à la table `Course`. S'il existe déjà des lignes dans la table `Course` lorsque le code s'exécute, l'opération `AddColumn` échoue car SQL Server ne sait pas quelle valeur placer dans la colonne qui ne peut pas être null. Pour ce tutorial, vous allez exécuter la migration sur une nouvelle base de données. Toutefois, dans une application de production, vous devriez faire en sorte que la migration traite les données existantes, si bien que les instructions suivantes montrent un exemple de la procédure à suivre pour ce faire.

Pour faire en sorte que cette migration fonctionne avec les données existantes, vous devez modifier le code pour attribuer à la nouvelle colonne une valeur par défaut et créer un département stub nommé « `Temp` » qui agira en tant que département par défaut. Par conséquent, les lignes `Course` existantes seront toutes associées au département « `Temp` » après l'exécution de la méthode `Up`.

- Ouvrez le fichier *{timestamp}\_ComplexDataModel.cs*.
- Commentez la ligne de code qui ajoute la colonne DepartmentID à la table Course.

```
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);
```

Ajoutez le code en surbrillance suivant après le code qui crée la table Department :

```
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, '2017-01-01')");
// 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);
```

Dans une application de production, vous devez écrire un code ou des scripts pour ajouter

des lignes Department et associer des lignes Course aux nouvelles lignes Department. Vous n'avez alors plus besoin du département « Temp » ni de la valeur par défaut sur la colonne Course.DepartmentID.

Enregistrez vos modifications et générez le projet.

## Changer la chaîne de connexion

Vous avez maintenant un nouveau code dans la classe `DbInitializer` qui ajoute des données initiales pour les nouvelles entités à une base de données vide. Pour faire en sorte qu'EF crée une nouvelle base de données vide, remplacez le nom de la base de données dans la chaîne de connexion, dans *appsettings.json*, par `ContosoUniversity3` ou un autre nom que vous n'avez pas utilisé sur l'ordinateur que vous utilisez.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=true;TrustServerCertificate=true;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

Enregistrez les modifications dans *appsettings.json*.

**Notes :** Comme alternative au changement de nom de la base de données, vous pouvez supprimer la base de données. Utilisez **l'Explorateur d'objets SQL Server (SSOX)** ou la commande CLI `database drop` :

```
dotnet ef database drop
```

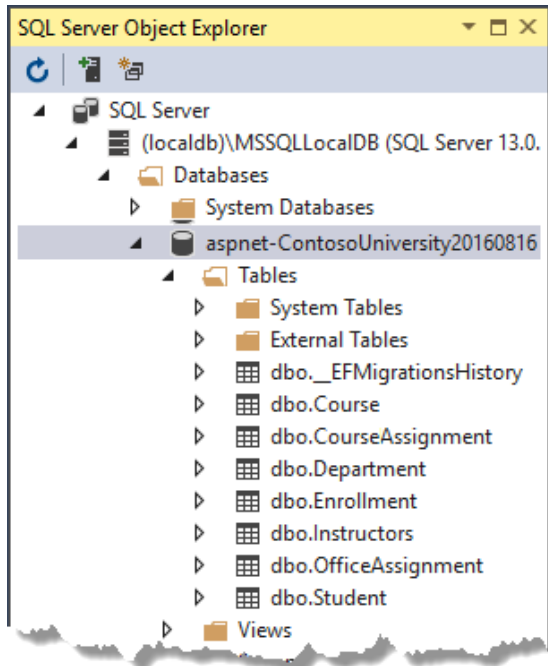
## Mettre à jour la base de données

Une fois que vous avez modifié le nom de la base de données ou supprimé la base de données, exécutez la commande `database update` dans la fenêtre de commande pour exécuter les migrations.

```
dotnet ef database update
```

Exécutez l'application pour que la méthode `DbInitializer.Initialize` exécute la nouvelle base de données et la remplisse.

Ouvrez la base de données dans SSOX comme vous l'avez fait précédemment, puis développez le nœud **Tables** pour voir que toutes les tables ont été créées. (Si SSOX est resté ouvert, cliquez sur le bouton **Actualiser**.)



Exécutez l'application pour déclencher le code d'initialiseur qui peuple la base de données.

Cliquez avec le bouton droit sur la table **CourseAssignment** et sélectionnez **Afficher les données** pour vérifier qu'elle comporte des données.

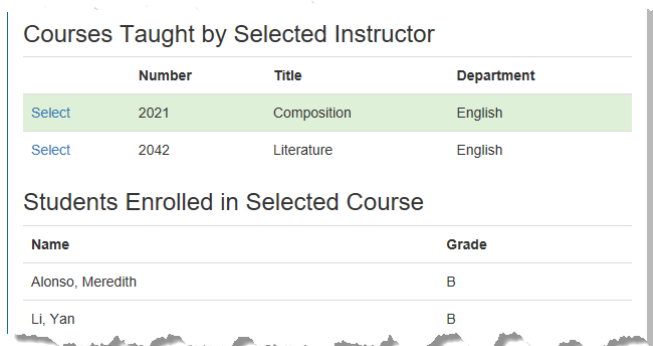
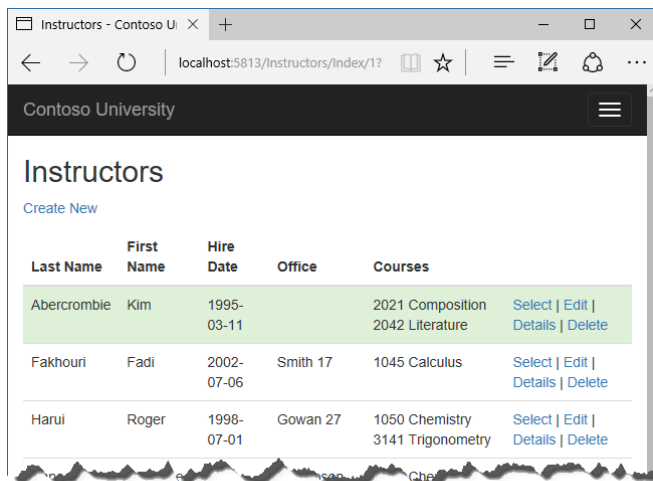
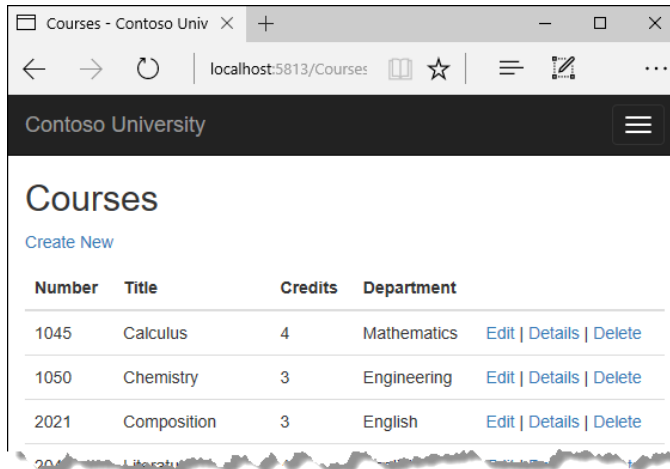
The screenshot shows the ContosoUniversity application window. The 'dbo.CourseAssignment [Data]' tab is selected, displaying a table with two columns: CourseID and InstructorID. The table contains the following data:

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

## Lire les données associées

Dans la section précédente, vous a élaboré le modèle de données School. Dans cette section, vous allez lire et afficher les données associées, à savoir les données qu'Entity Framework charge dans les propriétés de navigation.

Les illustrations suivantes montrent les pages que vous allez utiliser.



## Découvrir comment charger les données associées

Il existe plusieurs façons de permettre à un logiciel de mappage relationnel objet (ORM) comme Entity Framework de charger les données associées dans les propriétés de navigation d'une entité :

- Chargement hâtif (Eager loading). Quand l'entité est lue, ses données associées sont également récupérées. Cela génère en général une requête de jointure unique qui récupère toutes les données nécessaires. Vous spécifiez un chargement hâtif dans Entity Framework Core à l'aide des méthodes `Include` et `ThenInclude`.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Vous pouvez récupérer une partie des données dans des requêtes distinctes et EF « corrige » les propriétés de navigation. Autrement dit, EF ajoute automatiquement les entités récupérées séparément là où elles doivent figurer dans les propriétés de navigation des entités précédemment récupérées. Pour la requête qui récupère les données associées, vous pouvez utiliser la méthode `Load` à la place d'une méthode renvoyant une liste ou un objet, telle que `ToList` ou `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Chargement explicite (Explicit loading). Quand l'entité est lue pour la première fois, les données associées ne sont pas récupérées. Vous écrivez un code qui récupère les données associées si elles sont nécessaires. Comme dans le cas du chargement hâtif avec des requêtes distinctes, le chargement explicite génère plusieurs requêtes envoyées à la base de données. La différence tient au fait qu'avec le chargement explicite, le code spécifie les propriétés de navigation à charger. Dans Entity Framework Core 1.1, vous pouvez utiliser la méthode `Load` pour effectuer le chargement explicite. Par exemple :

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Chargement différé (Lazy loading). Quand l'entité est lue pour la première fois, les données associées ne sont pas récupérées. Toutefois, la première fois que vous essayez d'accéder à une propriété de navigation, les données requises pour cette propriété de navigation sont récupérées automatiquement. Une requête est envoyée à la base de données chaque fois que vous essayez d'obtenir des données à partir d'une propriété de navigation pour la première fois. Entity Framework Core 1.0 ne prend pas en charge le chargement différé.

## Considérations sur les performances

Si vous savez que vous avez besoin des données associées pour toutes les entités récupérées, le chargement hâtif souvent offre des performances optimales, car une seule requête envoyée à la base de données est généralement plus efficace que les requêtes distinctes pour chaque entité récupérée. Par exemple, supposons que chaque département a dix cours associés. Le chargement hâtif de toutes les données associées générerait une seule requête (de jointure) et un seul aller-retour à la base de données. Une requête distincte pour les cours pour chaque département entraînerait onze allers-retours à la base de données. Les allers-retours supplémentaires à la base de données sont particulièrement nuisibles pour les performances lorsque la latence est élevée.

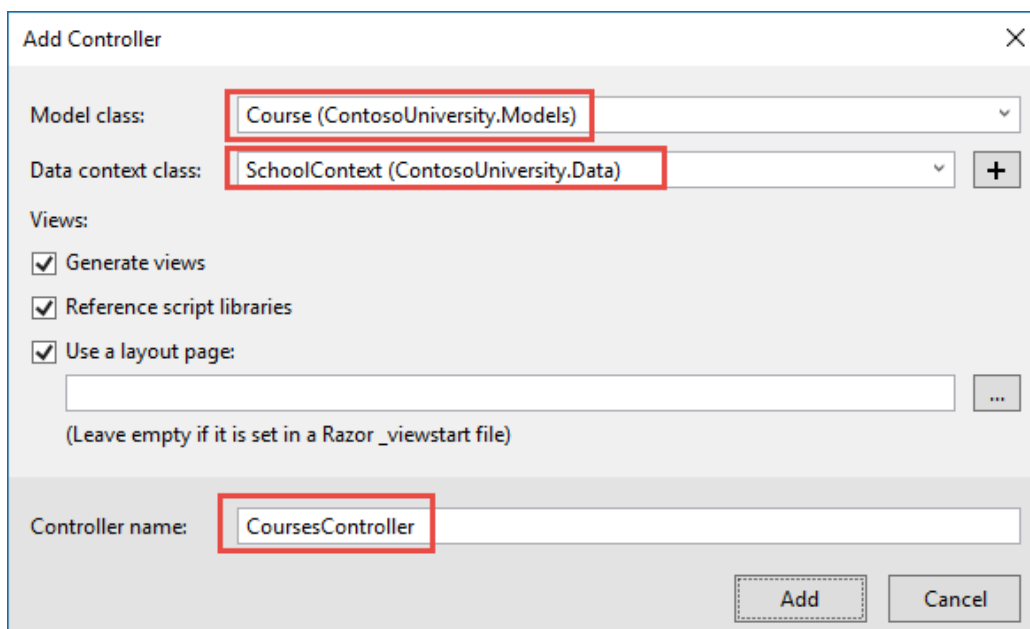
En revanche, dans certains scénarios, les requêtes distinctes s'avèrent plus efficaces. Le chargement hâtif de toutes les données associées dans une seule requête peut entraîner une jointure très complexe à générer, que SQL Server ne peut pas traiter efficacement. Ou, si vous avez besoin d'accéder aux propriétés de navigation d'entité uniquement pour un sous-ensemble des entités que vous traitez, des requêtes distinctes peuvent être plus performantes, car le chargement hâtif de tous les éléments en amont entraînerait la récupération de plus de données qu'il vous faut. Si les performances sont essentielles, il est préférable de tester les performances des deux façons afin d'effectuer le meilleur choix.



## Créer une page Courses

L'entité `Course` inclut une propriété de navigation qui contient l'entité `Department` du service auquel le cours est affecté. Pour afficher le nom du service affecté dans une liste de cours, vous devez obtenir la propriété `Name` de l'entité `Department` qui figure dans la propriété de navigation `Course.Department`.

Créez un contrôleur nommé `CoursesController` pour le type d'entité `Course`, en utilisant les mêmes options pour le générateur de modèles automatique **Contrôleur MVC avec vues, utilisant Entity Framework** que vous avez utilisées précédemment pour le contrôleur `Students`, comme indiqué dans l'illustration suivante :



The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Course (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. The 'Views' section has three checked options: 'Generate views', 'Reference script libraries', and 'Use a layout page'. The 'Controller name' text box contains 'CoursesController'. The 'Add' button is highlighted with a red dashed border.

Ouvrez `CoursesController.cs` et examinez la méthode `Index`. La génération de modèles automatique a spécifié un chargement hâtif pour la propriété de navigation `Department` à l'aide de la méthode `Include`.

Remplacez la méthode `Index` par le code suivant qui utilise un nom plus approprié pour `IQueryable` qui renvoie les entités `Course` (`courses` à la place de `schoolContext`) :

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Ouvrez *Views/Courses/Index.cshtml* et remplacez le code de modèle par le code suivant. Les modifications apparaissent en surbrillance :

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

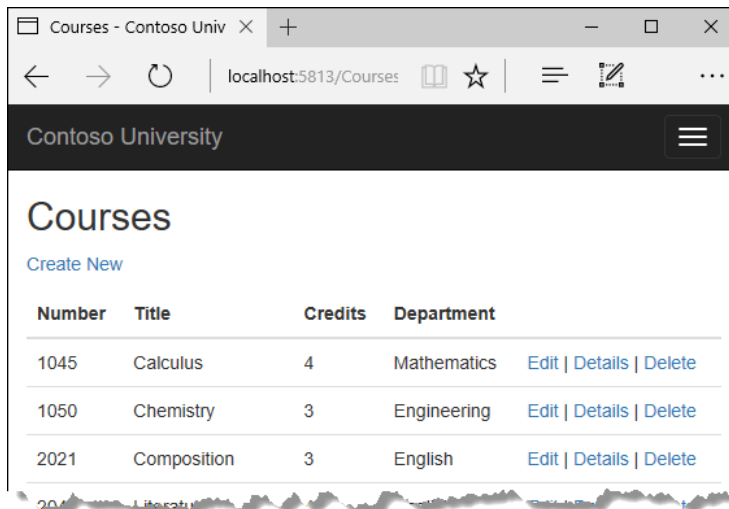
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
        </tr>
    </thead>
```

Vous avez apporté les modifications suivantes au code généré automatiquement :

- Changement de l'en-tête : Index a été remplacé par Courses.
- Ajout d'une colonne **Number** qui affiche la valeur de la propriété `CourseID`. Par défaut, les clés primaires ne sont pas générées automatiquement, car elles ne sont normalement pas significatives pour les utilisateurs finaux. Toutefois, dans le cas présent, la clé primaire est significative et vous voulez l'afficher.
- Modification de la colonne **Department** afin d'afficher le nom du département. Le code affiche la propriété `Name` de l'entité `Department` qui est chargée dans la propriété de navigation `Department` :

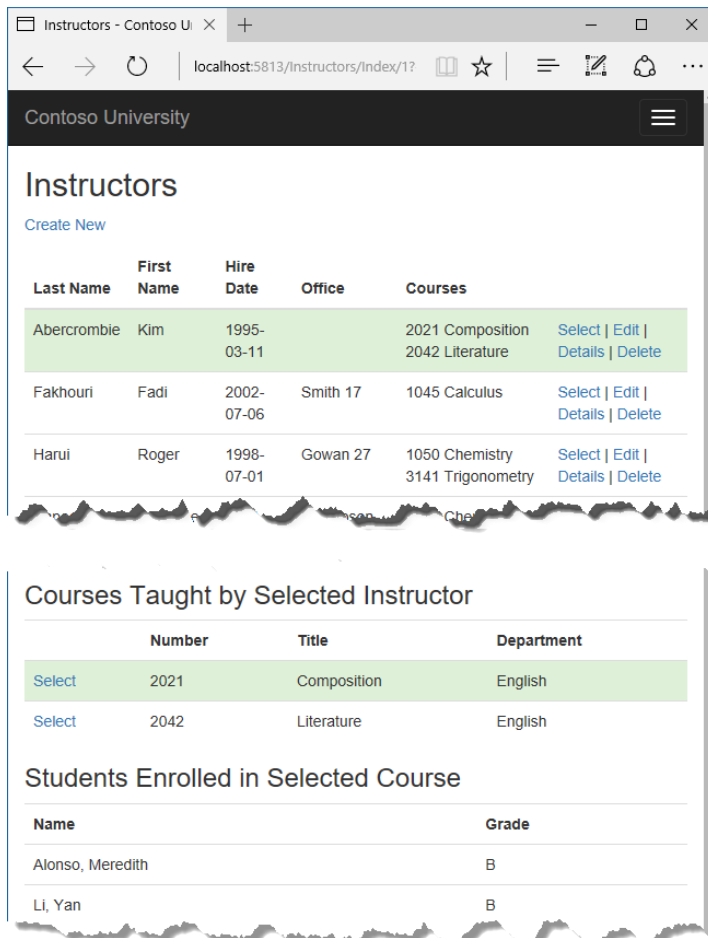
```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Exécutez l'application et sélectionnez l'onglet **Courses** pour afficher la liste avec les noms des départements.



## Créer une page Instructors

Dans cette section, vous allez créer un contrôleur et une vue pour l'entité Instructor afin d'afficher la page Instructors :



Cette page lit et affiche les données associées comme suit :

- La liste des formateurs affiche les données associées de l'entité OfficeAssignment. Il existe une relation un-à-zéro-ou-un entre les entités Instructor et OfficeAssignment. Vous allez utiliser un chargement hâtif pour les entités OfficeAssignment. Comme expliqué précédemment, le chargement hâtif est généralement plus efficace lorsque vous avez besoin des données associées pour toutes les lignes extraites de la table primaire. Dans ce cas, vous souhaitez afficher les affectations de bureaux pour tous les formateurs affichés.
- Lorsque l'utilisateur sélectionne un formateur, les entités Course associées sont affichées. Il existe une relation plusieurs à plusieurs entre les entités Instructor et Course. Vous allez utiliser un chargement hâtif pour les entités Course et les entités Department qui leur sont associées. Dans ce cas, des requêtes distinctes peuvent être plus efficaces, car vous avez besoin de cours uniquement pour le formateur sélectionné. Toutefois, cet exemple montre comment utiliser le chargement hâtif pour des propriétés de navigation dans des entités qui se trouvent elles-mêmes dans des propriétés de navigation.
- Lorsque l'utilisateur sélectionne un cours, les données associées dans le jeu d'entités Enrollment s'affichent. Il existe une relation un-à-plusieurs entre les entités Course et Enrollment. Vous allez utiliser des requêtes distinctes pour les entités Enrollment et les entités Student qui leur sont associées.

## Créer un modèle de vue pour la vue d'index des formateurs

La page Instructors affiche des données de trois tables différentes. Par conséquent, vous allez créer un modèle de vue qui comprend trois propriétés, chacune contenant les données d'une des tables.

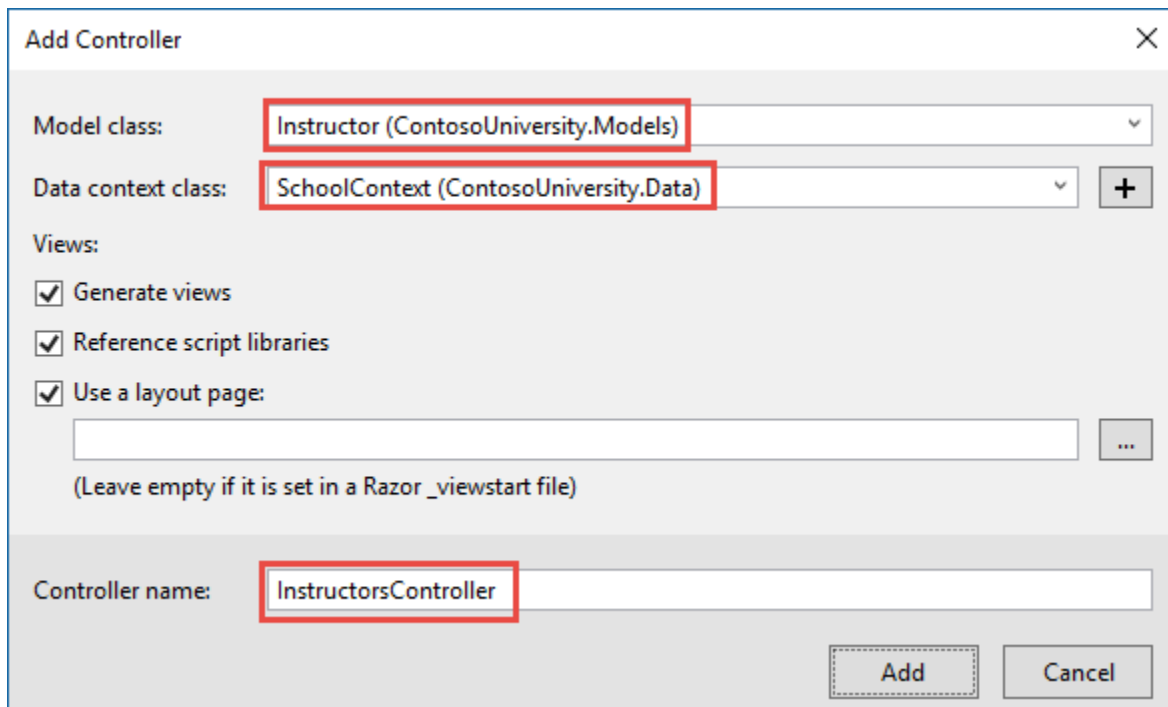
Dans le dossier *SchoolViewModels*, créez *InstructorIndexData.cs* et remplacez le code existant par le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

## Créer les vues et le contrôleur de formateurs

Créez un contrôleur de formateurs avec des actions de lecture/écriture EF comme indiqué dans l'illustration suivante :



The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Instructor (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below the 'Use a layout page' checkbox is an empty text box and a button with three dots. The 'Controller name' text box contains 'InstructorsController'. At the bottom right, there are 'Add' and 'Cancel' buttons. The 'Add' button has a dashed border.

Ouvrez *InstructorsController.cs* et ajoutez une instruction using pour l'espace de noms ViewModel :

```
using ContosoUniversity.Models.SchoolViewModels;
```

Remplacez la méthode Index par le code suivant pour effectuer un chargement hâtif des données associées et le placer dans le modèle de vue.

```

public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
        .ThenInclude(i => i.Student)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}

```

La méthode accepte des données de route facultatives (`id`) et un paramètre de chaîne de requête (`courseID`) qui fournissent les valeurs d'ID du formateur sélectionné et du cours sélectionné. Ces paramètres sont fournis par les liens hypertexte **Select** dans la page.

Le code commence par créer une instance du modèle de vue et la placer dans la liste des formateurs. Le code spécifie un chargement hâtif pour les propriétés de navigation `Instructor.OfficeAssignment` et `Instructor.CourseAssignments`. Dans la propriété `CourseAssignments`, la propriété `Course` est chargée et, dans ce cadre, les propriétés `Enrollments` et `Department` sont chargées, et dans chaque entité `Enrollment`, la propriété `Student` est chargée.

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

Étant donné que la vue nécessite toujours l'entité OfficeAssignment, il est plus efficace de l'extraire dans la même requête. Les entités Course sont requises lorsqu'un formateur est sélectionné dans la page web, de sorte qu'une requête individuelle est meilleure que plusieurs requêtes seulement si la page s'affiche plus souvent avec un cours sélectionné que sans.

Le code répète CourseAssignments et Course, car vous avez besoin de deux propriétés de Course. La première chaîne d'appels ThenInclude obtient CourseAssignment.Course, Course.Enrollments et Enrollment.Student.

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

À ce stade dans le code, un autre ThenInclude serait pour les propriétés de navigation de Student, dont vous n'avez pas besoin. Toutefois, l'appel de Include recommence avec les propriétés Instructor, donc vous devez parcourir la chaîne à nouveau, cette fois en spécifiant Course.Department à la place de Course.Enrollments.

```

viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

Le code suivant s'exécute quand un formateur a été sélectionné. Le formateur sélectionné est récupéré à partir de la liste des formateurs dans le modèle de vue. La propriété `Courses` du modèle de vue est alors chargée avec les entités `Course` de la propriété de navigation `CourseAssignments` de ce formateur.

C#Copier

```

if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}

```

La méthode `Where` renvoie une collection, mais dans ce cas, les critères transmis à cette méthode entraînent le renvoi d'une seule entité `Instructor`. La méthode `Single` convertit la collection en une seule entité `Instructor`, ce qui vous permet d'accéder à la propriété `CourseAssignments` de cette entité. La propriété `CourseAssignments` contient des entités `CourseAssignment`, à partir desquelles vous souhaitez uniquement les entités `Course` associées.

Vous utilisez la méthode `Single` sur une collection lorsque vous savez que la collection aura un seul élément. La méthode `Single` lève une exception si la collection transmise est vide ou s'il y a plusieurs éléments. Une alternative est `SingleOrDefault`, qui renvoie une valeur par défaut (`Null` dans ce cas) si la collection est vide. Toutefois, dans ce cas, cela entraînerait encore une exception (en tentant de trouver une propriété `Courses` sur une référence `null`) et le message d'exception indiquerait moins clairement la cause du problème. Lorsque vous appelez la méthode `Single`, vous pouvez également transmettre la condition `Where` au lieu d'appeler séparément la méthode `Where` :



```
.Single(i => i.ID == id.Value)
```

À la place de :

```
.Where(i => i.ID == id.Value).Single()
```

Ensuite, si un cours a été sélectionné, le cours sélectionné est récupéré à partir de la liste des cours dans le modèle de vue. Ensuite, la propriété `Enrollments` du modèle de vue est chargée avec les entités `Enrollment` à partir de la propriété de navigation `Enrollments` de ce cours.

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

## Modifier la vue d'index des formateurs

Dans *Views/Instructors/Index.cshtml*, remplacez le code du modèle par le code suivant. Les modifications apparaissent en surbrillance.

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
```

```

<tbody>
    @foreach (var item in Model.Instructors)
    {
        string selectedRow = "";
        if (item.ID == (int?)ViewData["InstructorID"])
        {
            selectedRow = "success";
        }
        <tr class="@selectedRow">
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.HireDate)
            </td>
            <td>
                @if (item.OfficeAssignment != null)
                {
                    @item.OfficeAssignment.Location
                }
            </td>
            <td>
                @foreach (var course in item.CourseAssignments)
                {
                    @course.Course.CourseID @: @course.Course.Title <br />
                }
            </td>
            <td>
                <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

```

Vous avez apporté les modifications suivantes au code existant :

- Vous avez changé la classe de modèle en `InstructorIndexData`.
- Vous avez changé le titre de la page en remplaçant **Index** par **Instructors**.
- Vous avez ajouté une colonne **Office** qui affiche `item.OfficeAssignment.Location` seulement si `item.OfficeAssignment` n'est pas Null. (Comme il s'agit d'une relation un-à-zéro-ou-un, il se peut qu'il n'y ait pas d'entité `OfficeAssignment` associée.)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Vous avez ajouté une colonne **Courses** qui affiche les cours animés par chaque formateur.
- Vous avez ajouté un code qui ajoute dynamiquement `class="success"` à l'élément `tr` du formateur sélectionné. Cela définit une couleur d'arrière-plan pour la ligne sélectionnée à l'aide d'une classe d'amorçage.

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Vous avez ajouté un nouveau lien hypertexte étiqueté **Select** immédiatement avant les autres liens dans chaque ligne, ce qui entraîne l'envoi de l'ID du formateur sélectionné à la méthode `Index`.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Exécutez l'application et sélectionnez l'onglet **Instructors**. La page affiche la propriété Location des entités OfficeAssignment associées et une cellule de table vide lorsqu'il n'existe aucune entité OfficeAssignment associée.

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select   Edit   Details   Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select   Edit   Details   Delete

Dans le fichier *Views/Instructors/Index.cshtml*, après l'élément de fermeture de table (à la fin du fichier), ajoutez le code suivant. Ce code affiche la liste des cours associés à un formateur quand un formateur est sélectionné.

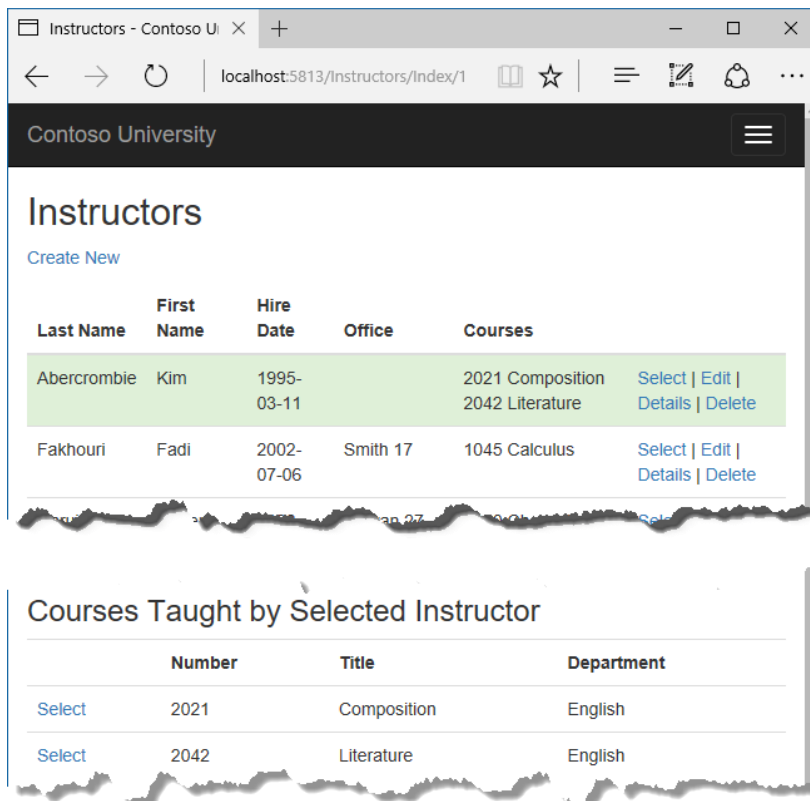
```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

Ce code lit la propriété `Courses` du modèle de vue pour afficher la liste des cours. Il fournit également un lien hypertexte **Select** qui envoie l'ID du cours sélectionné à la méthode d'action `Index`.

Actualisez la page et sélectionnez un formateur. Vous voyez à présent une grille qui affiche les cours affectés au formateur sélectionné et, pour chaque cours, vous voyez le nom du département affecté.



Après le bloc de code que vous venez d'ajouter, ajoutez le code suivant. Ceci affiche la liste des étudiants qui sont inscrits à un cours quand ce cours est sélectionné.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

Ce code lit la propriété Enrollments du modèle de vue pour afficher la liste des étudiants inscrits dans ce cours.

Actualisez la page à nouveau et sélectionnez un formateur. Ensuite, sélectionnez un cours pour afficher la liste des étudiants inscrits et leurs notes.

Instructors - Contoso UI

localhost:5813/Instructors/Index/1?

## Contoso University

### Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	<a href="#">Select</a>   <a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	<a href="#">Select</a>   <a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	<a href="#">Select</a>   <a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

### Courses Taught by Selected Instructor

	Number	Title	Department
<a href="#">Select</a>	2021	Composition	English
<a href="#">Select</a>	2042	Literature	English

### Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

## À propos du chargement explicite

Lorsque vous avez récupéré la liste des formateurs dans *InstructorsController.cs*, vous avez spécifié un chargement hâtif pour la propriété de navigation `CourseAssignments`.

Supposons que vous vous attendiez à ce que les utilisateurs ne souhaitent que rarement voir les inscriptions pour un formateur et un cours sélectionnés. Dans ce cas, vous pouvez charger les données d'inscription uniquement si elles sont demandées. Pour voir un exemple illustrant comment effectuer un chargement explicite, remplacez la méthode `Index` par le code suivant, qui supprime le chargement hâtif pour `Enrollments` et charge explicitement cette propriété. Les modifications du code apparaissent en surbrillance.

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

Le nouveau code supprime les appels de la méthode *ThenInclude* pour les données d'inscription à partir du code qui extrait les entités de formateur. Si un formateur et un

cours sont sélectionnés, le code en surbrillance récupère les entités Enrollment pour le cours sélectionné et les entités Student pour chaque entité Enrollment.

Exécutez l'application, accédez à la page d'index des formateurs et vous ne verrez aucune différence pour ce qui est affiché dans la page, bien que vous ayez modifié la façon dont les données sont récupérées.

## Mettre à jour les données associées

Dans le didacticiel précédent, vous avez affiché des données associées ; dans ce didacticiel, vous mettez à jour des données associées en mettant à jour des champs de clé étrangère et des propriétés de navigation.

Les illustrations suivantes montrent quelques-unes des pages que vous allez utiliser.

The left screenshot shows the 'Edit Course' page. The browser address bar indicates 'localhost:58'. The page title is 'Edit Course'. The form contains the following fields: 'Number' (1000), 'Title' (Algebra 2), 'Credits' (5), and 'Department' (Mathematics). A 'Save' button is at the bottom.

The right screenshot shows the 'Edit Instructor' page. The browser address bar indicates 'localhost:5813/Instruct'. The page title is 'Edit Instructor'. The form contains the following fields: 'Last Name' (Abercrombie), 'First Name' (Kim), 'Hire Date' (3/11/1995), and 'Office Location' (44/3P). Below these fields is a list of courses with checkboxes: 1000 Algebra 2, 1045 Calculus, 1050 Chemistry, 2021 Composition, 2042 Literature, 3141 Trigonometry, 4022 Microeconomics, and 4041 Macroeconomics. The '2021 Composition' checkbox is checked. A 'Save' button is at the bottom.

## Personnaliser les pages de cours

Quand une entité Course est créée, elle doit avoir une relation avec un département existant. Pour faciliter cela, le code du modèle généré automatiquement inclut des



méthodes de contrôleur, et des vues Create et Edit qui incluent une liste déroulante pour sélectionner le département. La liste déroulante définit la propriété de clé étrangère `Course.DepartmentID`, qui est tout ce dont Entity Framework a besoin pour charger la propriété de navigation `Department` avec l'entité `Department` appropriée. Vous utilisez le code du modèle généré automatiquement, mais que vous modifiez un peu pour ajouter la gestion des erreurs et trier la liste déroulante.

Dans *CoursesController.cs*, supprimez les quatre méthodes Create et Edit, et remplacez-les par le code suivant :

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .SingleOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}

```

Après la méthode `HttpPost Edit`, créez une méthode qui charge les informations des départements pour la liste déroulante.

```

4 references | 0 exceptions
private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name", selectedDepartment);
}

```

La méthode `PopulateDepartmentsDropDownList` obtient une liste de tous les départements triés par nom, crée une collection `SelectList` pour une liste déroulante et passe la collection à la vue dans `ViewBag`. La méthode accepte le paramètre facultatif `selectedDepartment` qui permet au code appelant de spécifier l'élément sélectionné lors de l'affichage de la liste déroulante. La vue passe le nom « `DepartmentID` » pour le tag

helper `<select>` : le helper peut alors rechercher dans l'objet `ViewBag` une `SelectList` nommée « `DepartmentID` ».

La méthode `HttpGet Create` appelle la méthode `PopulateDepartmentsDropDownList` sans définir l'élément sélectionné, car pour un nouveau cours, le département n'est pas encore établi :

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

La méthode `HttpGet Edit` définit l'élément sélectionné, en fonction de l'ID du département qui est déjà affecté au cours à modifier :

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

Les méthodes `HttpPost` pour `Create` et pour `Edit` incluent également du code qui définit l'élément sélectionné quand elles réaffichent la page après une erreur. Ceci garantit que quand la page est réaffichée pour montrer le message d'erreur, le département qui a été sélectionné le reste.

## Ajouter `.AsNoTracking` aux méthodes `Details` et `Delete`

Pour optimiser les performances des pages `Details` et `Delete` pour les cours, ajoutez des appels de `AsNoTracking` dans les méthodes `Details` et `HttpGet Delete`.

```

public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

```

public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

## Modifier les vues des cours

Dans *Views/Courses/Create.cshtml*, ajoutez une option « Select Department » à la liste déroulante **Department**, changez la légende de **DepartmentID** en **Department** et ajoutez un message de validation.

```

<div class="form-group">
  <label asp-for="Department" class="control-label"></label>
  <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
    <option value="">-- Select Department --</option>
  </select>
  <span asp-validation-for="DepartmentID" class="text-danger" />

```

Dans *Views/Courses/Edit.cshtml*, faites les mêmes modifications pour le champ Department que ce que vous venez de faire dans *Create.cshtml*.

Également dans *Views/Courses/Edit.cshtml*, ajoutez un champ de numéro de cours avant le champ **Title**. Comme le numéro de cours est la clé primaire, il est affiché mais ne peut pas être modifié.

```

<div class="form-group">
  <label asp-for="CourseID" class="control-label"></label>
  <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>

```

Il existe déjà un champ masqué (`<input type="hidden">`) pour le numéro de cours dans la vue Edit. L'ajout d'un tag helper `<label>` n'élimine la nécessité d'avoir le champ masqué, car cela n'a pas comme effet que le numéro de cours est inclut dans les données envoyées quand l'utilisateur clique sur **Save** dans la page **Edit**.

Dans *Views/Courses/Delete.cshtml*, ajoutez un champ pour le numéro de cours en haut et changez l'ID de département en nom de département.

```

@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

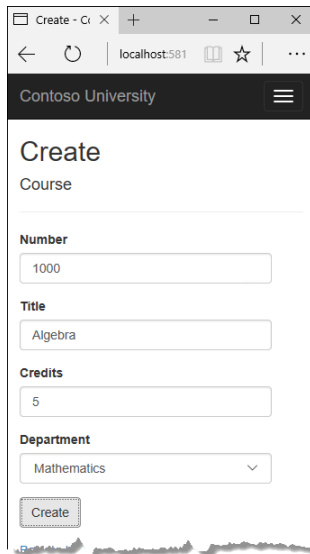
    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>

```

Dans *Views/Courses/Details.cshtml*, faites la même modification que celle que vous venez de faire pour *Delete.cshtml*.

## Tester les pages des cours

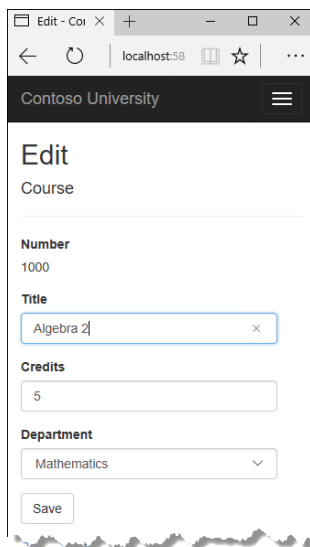
Exécutez l'application, sélectionnez l'onglet **Courses**, cliquez sur **Create New** et entrez les données pour un nouveau cours :



The screenshot shows a web browser window with the address bar at localhost:581. The page title is 'Contoso University'. The main heading is 'Create Course'. Below this, there are four input fields: 'Number' with the value '1000', 'Title' with the value 'Algebra', 'Credits' with the value '5', and 'Department' with a dropdown menu showing 'Mathematics'. At the bottom of the form is a 'Create' button.

Cliquez sur **Créer**. La page Index des cours est affichée avec le nouveau cours ajouté à la liste. Le nom du département dans la liste de la page Index provient de la propriété de navigation, ce qui montre que la relation a été établie correctement.

Cliquez sur **Edit** pour un cours dans la page Index des cours.



The screenshot shows a web browser window with the address bar at localhost:58. The page title is 'Contoso University'. The main heading is 'Edit Course'. Below this, there are four input fields: 'Number' with the value '1000', 'Title' with the value 'Algebra 2', 'Credits' with the value '5', and 'Department' with a dropdown menu showing 'Mathematics'. At the bottom of the form is a 'Save' button.

Modifiez les données dans la page et cliquez sur **Save**. La page Index des cours est affichée avec les données du cours mises à jour.

## Ajouter une page de modification de formateur

Quand vous modifiez un enregistrement de formateur, vous voulez avoir la possibilité de mettre à jour l'attribution du bureau du formateur. L'entité `Instructor` a une relation un-à-zéro ou un-à-un avec l'entité `OfficeAssignment`, ce qui signifie que votre code doit gérer les situations suivantes :

- Si l'utilisateur efface l'attribution du bureau et qu'il existait une valeur à l'origine, supprimez l'entité `OfficeAssignment`.
- Si l'utilisateur entre une attribution de bureau et qu'elle était vide à l'origine, créez une entité `OfficeAssignment`.
- Si l'utilisateur change la valeur d'une attribution de bureau, changez la valeur dans une entité `OfficeAssignment` existante.

### Mettre à jour le contrôleur `Instructors`

Dans `InstructorsController.cs`, modifiez le code de la méthode `HttpGet Edit` afin qu'elle charge la propriété de navigation `OfficeAssignment` de l'entité `Instructor` et appelle `AsNoTracking` :

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Remplacez la méthode `HttpPost Edit` par le code suivant pour gérer les mises à jour des attributions de bureau :



```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}

```

Le code effectue les actions suivantes :

- Il change le nom de la méthode `EditPost`, car la signature est maintenant la même que celle de la méthode `HttpGet Edit` (l'attribut `ActionName` spécifie que l'URL `/Edit/` est encore utilisée).
- Obtient l'entité `Instructor` actuelle auprès de la base de données en utilisant le chargement hâtif pour la propriété de navigation `OfficeAssignment`. C'est identique à ce que vous avez fait dans la méthode `HttpGet Edit`.
- Elle met à jour l'entité `Instructor` récupérée avec des valeurs dans le classeur de modèles. La surcharge de `TryUpdateModel` vous permet de mettre en liste verte les propriétés que vous voulez inclure. Ceci empêche la survalidation.

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- Si l'emplacement du bureau est vide, il définit la propriété `Instructor.OfficeAssignment` sur null, de façon que la ligne correspondante dans la table `OfficeAssignment` soit supprimée.

```
if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Il enregistre les modifications dans la base de données.

## Mettre à jour la vue de modification des formateurs

Dans `Views/Instructors/Edit.cshtml`, ajoutez un nouveau champ pour la modification de l'emplacement du bureau, à la fin et avant le bouton **Save** :

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

Exécutez l'application, sélectionnez l'onglet **Instructors**, puis cliquez sur **Edit** pour un formateur. Modifiez **Office Location** et cliquez sur **Save**.

Contoso University

## Edit

Instructor

**Last Name**  
Abercrombie

**First Name**  
Kim

**Hire Date**  
3/11/1995

**Office Location**  
44/3P

Save

## Ajouter des cours à la page de modification

Les instructeurs peuvent enseigner dans n'importe quel nombre de cours. Maintenant, vous allez améliorer la page de modification des formateurs en ajoutant la possibilité de modifier les affectations de cours avec un groupe de cases à cocher, comme le montre la capture d'écran suivante :

Contoso University

## Edit

Instructor

**Last Name**  
Abercrombie

**First Name**  
Kim

**Hire Date**  
3/11/1995

**Office Location**  
44/3P

☐ 1000 Algebra 2   ☐ 1045 Calculus   ☐ 1050 Chemistry  
☒ 2021 Composition   ☒ 2042 Literature   ☐ 3141 Trigonometry  
☐ 4022 Microeconomics   ☐ 4041 Macroeconomics

Save

La relation entre les entités *Course* et *Instructor* est plusieurs-à-plusieurs. Pour ajouter et supprimer des relations, vous ajoutez et vous supprimez des entités dans le jeu d'entités de la jointure *CourseAssignments*.

L'interface utilisateur qui vous permet de changer les cours auxquels un formateur est affecté est un groupe de cases à cocher. Une case à cocher est affichée pour chaque cours de la base de données, et ceux auxquels le formateur est actuellement affecté sont sélectionnés. L'utilisateur peut cocher ou décocher les cases pour changer les affectations de cours. Si le nombre de cours était beaucoup plus important, vous pourriez utiliser une autre méthode de présentation des données dans la vue, mais vous utiliseriez la même méthode de manipulation d'une entité de jointure pour créer ou supprimer des relations.

## Mettre à jour le contrôleur *Instructors*

Pour fournir des données à la vue pour la liste de cases à cocher, vous utilisez une classe de modèle de vue.

Créez *AssignedCourseData.cs* dans le dossier *SchoolViewModels* et remplacez le code existant par le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

Dans *InstructorsController.cs*, remplacez la méthode `HttpGet Edit` par le code suivant. Les modifications apparaissent en surbrillance.

```

public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}

```

Le code ajoute un chargement hâtif pour la propriété de navigation `Courses` et appelle la nouvelle méthode `PopulateAssignedCourseData` pour fournir des informations pour le tableau de cases à cocher avec la classe de modèle de vue `AssignedCourseData`.

Le code de la méthode `PopulateAssignedCourseData` lit toutes les entités `Course` pour charger une liste de cours avec la classe de modèle de vue. Pour chaque cours, le code vérifie s'il existe dans la propriété de navigation `Courses` du formateur. Pour créer une recherche efficace quand il est vérifié si un cours est affecté au formateur, les cours affectés au formateur sont placés dans une collection `HashSet`. La propriété `Assigned` est définie sur `true` pour les cours auxquels le formateur est affecté. La vue utilise cette propriété pour déterminer quelles cases doivent être affichées cochées. Enfin, la liste est passée à la vue dans `ViewData`.

Ensuite, ajoutez le code qui est exécuté quand l'utilisateur clique sur **Save**. Remplacez la méthode `EditPost` par le code suivant et ajoutez une nouvelle méthode qui met à jour la propriété de navigation `Courses` de l'entité `Instructor`.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```

2 references | 0 exceptions
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i => i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

La signature de la méthode diffère maintenant de celle de la méthode `HttpGet Edit` : le nom de la méthode change donc de `EditPost` en `Edit`.

Comme la vue n'a pas de collection d'entités `Course`, le classeur de modèles ne peut pas mettre à jour automatiquement la propriété de navigation `CourseAssignments`. Au lieu d'utiliser le classeur de modèles pour mettre à jour la propriété de navigation `CourseAssignments`, vous faites cela dans la nouvelle méthode `UpdateInstructorCourses`. Par conséquent, vous devez exclure la propriété `CourseAssignments` de la liaison de modèle. Ceci ne nécessite aucune modification du code qui appelle `TryUpdateModel`, car vous utilisez la surcharge de mise en liste verte et `CourseAssignments` n'est pas dans la liste des éléments à inclure.

Si aucune case n'a été cochée, le code de `UpdateInstructorCourses` initialise la propriété de navigation `CourseAssignments` avec une collection vide et retourne :

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = inst
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefa
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

Le code boucle ensuite à travers tous les cours dans la base de données, et vérifie chaque cours par rapport à ceux actuellement affectés au formateur relativement à ceux qui ont été sélectionnés dans la vue. Pour faciliter des recherches efficaces, les deux dernières collections sont stockées dans des objets `HashSet`.

Si la case pour un cours a été cochée mais que le cours n'est pas dans la propriété de navigation `Instructor.CourseAssignments`, le cours est ajouté à la collection dans la propriété de navigation.



```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instr
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefau
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

Si la case pour un cours a été cochée mais que le cours est dans la propriété de navigation `Instructor.CourseAssignments`, le cours est supprimé de la propriété de navigation.

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID = instr
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefau
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

## Mettre à jour les vues des formateurs

Dans *Views/Instructors/Edit.cshtml*, ajoutez un champ **Courses** avec un tableau de cases à cocher, en ajoutant le code suivant immédiatement après le code les éléments `div` pour le champ **Office** et avant l'élément `div` pour le bouton **Save**.

**Notes :** Quand vous collez le code dans Visual Studio, les sauts de ligne seront changés d'une façon qui va déstructurer le code. Appuyez une fois sur Ctrl+Z pour annuler la mise en forme automatique. Ceci permet de corriger les sauts de ligne de façon à ce qu'ils apparaissent comme ce que vous voyez ici. L'indentation ne doit pas nécessairement être parfaite, mais les lignes `@</tr><tr>`, `@:<td>`, `@:</td>` et `@:</tr>` doivent chacune tenir sur une seule ligne comme dans l'illustration, sinon vous recevrez une erreur d'exécution. Avec le bloc de nouveau code sélectionné, appuyez trois fois sur la touche Tab pour aligner le nouveau code avec le code existant.

```

<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <table>
      <tr>
        @{
          int cnt = 0;
          List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses = ViewBag.Courses;

          foreach (var course in courses)
          {
            if (cnt++ % 3 == 0)
            {
              @:</tr><tr>
            }
            @:<td>
              <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
              @course.CourseID @: @course.Title
            @:</td>
          }
        @:</tr>
      </tr>
    </table>
  </div>
</div>

```

Ce code crée un tableau HTML qui a trois colonnes. Dans chaque colonne se trouve une case à cocher, suivie d'une légende qui est constituée du numéro et du titre du cours. Toutes les cases à cocher ont le même nom (« selectedCourses »), qui indique au classeur de modèles qu'ils doivent être traités comme un groupe. L'attribut de valeur de chaque case à cocher est défini sur la valeur de `CourseID`. Quand la page est envoyée, le classeur de modèles passe un tableau au contrôleur, constitué des valeurs de `CourseID` seulement pour les cases qui sont cochées.

Quand les cases à cocher sont affichées à l'origine, celles qui correspondent à des cours affectés au formateur ont des attributs cochés, qui les sélectionnent (ils les affichent cochées).

Exécutez l'application, sélectionnez l'onglet **Instructors**, puis cliquez sur **Edit** pour un formateur pour voir la page **Edit**.

Contoso University

## Edit

Instructor

**Last Name**  
Abercrombie

**First Name**  
Kim

**Hire Date**  
3/11/1995

**Office Location**  
44/3P

☐ 1000 Algebra 2
 ☐ 1045 Calculus
 ☐ 1050 Chemistry  
☒ 2021 Composition
 ☒ 2042 Literature
 ☐ 3141 Trigonometry  
☐ 4022 Microeconomics
 ☐ 4041 Macroeconomics

Save

Changez quelques affectations de cours et cliquez sur Save. Les modifications que vous apportez sont reflétées dans la page Index.

**Notes :** L'approche adoptée ici pour modifier les données des cours des formateurs fonctionne bien le nombre de cours est limité. Pour les collections qui sont beaucoup plus volumineuses, une autre interface utilisateur et une autre méthode de mise à jour seraient nécessaires.

## Mettre à jour la page Delete

Dans *InstructorsController.cs*, supprimez la méthode `DeleteConfirmed` et insérez à la place le code suivant.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Ce code apporte les modifications suivantes :

- Il effectue un chargement hâtif pour la propriété de navigation `CourseAssignments`. Vous devez inclure ceci car sinon, EF ne dispose pas d'informations sur les entités `CourseAssignment` associées et ne les supprime pas. Pour éviter de devoir les lire ici, vous pouvez configurer une suppression en cascade dans la base de données.
- Si le formateur à supprimer est attribué en tant qu'administrateur d'un département, supprime l'attribution de l'instructeur de ces départements.

## Ajouter des emplacements de bureau et des cours à la page Create

Dans *InstructorsController.cs*, supprimez les méthodes `HttpGet` et `HttpPost Create`, puis ajoutez le code suivant à leur place :

```
0 references | 0 requests | 0 exceptions
public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
0 references | 0 requests | 0 exceptions
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID = int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

Ce code est similaire à ce que vous avez vu pour les méthodes `Edit`, excepté qu'initialement, aucun cours n'est sélectionné. La méthode `HttpGet Create` appelle la méthode `PopulateAssignedCourseData`, non pas en raison du fait qu'il peut exister des cours sélectionnés, mais pour pouvoir fournir une collection vide pour la boucle `foreach` dans la vue (sinon, le code de la vue lèverait une exception de référence null).

La méthode `HttpPost Create` ajoute chaque cours sélectionné à la propriété de navigation `CourseAssignments` avant de vérifier s'il y a des erreurs de validation et ajoute le nouveau formateur à la base de données. Les cours sont ajoutés même s'il existe des erreurs de modèle : ainsi, quand c'est le cas (par exemple si l'utilisateur a tapé une date non valide)

et que la page est réaffichée avec un message d'erreur, les sélections de cours qui ont été faites sont automatiquement restaurées.

Notez que pour pouvoir ajouter des cours à la propriété de navigation `CourseAssignments`, vous devez initialiser la propriété en tant que collection vide :

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

Comme alternative à cette opération dans le code du contrôleur, vous pouvez l'effectuer dans le modèle `Instructor` en modifiant le getter de propriété pour créer automatiquement la collection si elle n'existe pas, comme le montre l'exemple suivant :

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

Si vous modifiez la propriété `CourseAssignments` de cette façon, vous pouvez supprimer le code d'initialisation explicite de la propriété dans le contrôleur.

Dans `Views/Instructor/Create.cshtml`, ajoutez une zone de texte pour l'emplacement du bureau et des cases à cocher pour les cours avant le bouton Envoyer.

```

<div class="form-group">
  <label asp-for="OfficeAssignment.Location" class="control-label"></label>
  <input asp-for="OfficeAssignment.Location" class="form-control" />
  <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <table>
      <tr>
        @if
        {
          int cnt = 0;
          List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses = ViewBag.Courses;

          foreach (var course in courses)
          {
            if (cnt++ % 3 == 0)
            {
              @:</tr><tr>
            }
            @:<td>
              <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ? "checked=\"checked\" : \"\"")) />
              @course.CourseID @: @course.Title
            @:</td>
          }
        }
      @:</tr>
    </table>
  </div>
</div>

```

Testez en exécutant l'application et en créant un formateur.

## Gérer l'accès concurrentiel

Dans les didacticiels précédents, vous avez découvert comment mettre à jour des données. Ce didacticiel montre comment gérer les conflits quand plusieurs utilisateurs mettent à jour la même entité en même temps.

Vous allez créer des pages web qui utilisent l'entité Department et gérer les erreurs d'accès concurrentiel. Les illustrations suivantes montrent les pages Edit et Delete, notamment certains messages qui sont affichés si un conflit d'accès concurrentiel se produit.

Departmenten x Edit - Conto x + - □ x

localhost:5813/Departments

Contoso University

## Edit

### Department

- The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.

**Name**

English

**Budget**

200000.00

Current value: \$50,000.00

**Start Date**

9/1/2007

**InstructorID**

Abercrombie, Kim

Save

Departments - Delete - Co x + - □ x

localhost:5813/Departments

Contoso University

## Delete

The record you attempted to delete was modified by another user after you got the original values. The delete operation was canceled and the current values in the database have been displayed. If you still want to delete this record, click the Delete button again. Otherwise click the Back to List hyperlink.

Are you sure you want to delete this?

Department

**Name**  
English

**Budget**  
\$200,000.00

**Start Date**  
2017-02-16

**Administrator**  
Abercrombie, Kim

Delete | Back to List

## Conflits d'accès concurrentiel

Un conflit d'accès concurrentiel se produit quand un utilisateur affiche les données d'une entité pour la modifier, puis qu'un autre utilisateur met à jour les données de la même entité avant que les modifications du premier utilisateur soient écrites dans la base de données. Si vous n'activez pas la détection de ces conflits, la personne qui met à jour la base de données en dernier remplace les modifications de l'autre utilisateur. Dans de nombreuses applications, ce risque est acceptable : s'il n'y a que quelques utilisateurs ou quelques mises à jour, ou s'il n'est pas réellement critique que des modifications soient remplacées, le coût de la programmation nécessaire à la gestion des accès concurrentiels peut être supérieur au bénéfice qu'elle apporte. Dans ce cas, vous ne devez pas configurer l'application pour gérer les conflits d'accès concurrentiel.

### Accès concurrentiel pessimiste (verrouillage)

Si votre application doit éviter la perte accidentelle de données dans des scénarios d'accès concurrentiel, une manière de le faire consiste à utiliser des verrous de base de données. Ceci est appelé « accès concurrentiel pessimiste ». Par exemple, avant de lire une ligne d'une base de données, vous demandez un verrou pour lecture seule ou pour accès avec

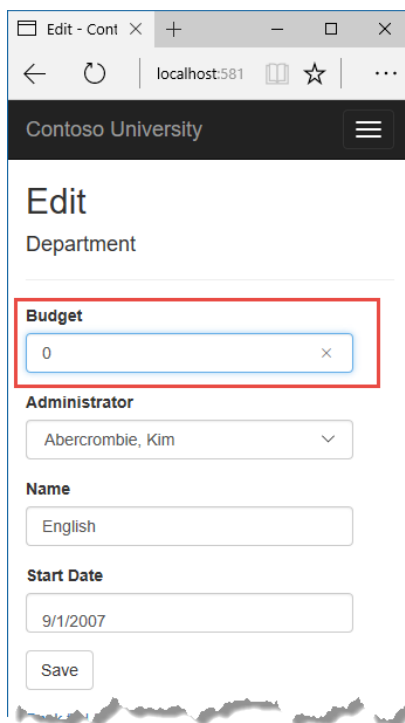


mise à jour. Si vous verrouillez une ligne pour accès avec mise à jour, aucun autre utilisateur n'est autorisé à verrouiller la ligne pour lecture seule ou pour accès avec mise à jour, car ils obtiendraient ainsi une copie de données qui sont en cours de modification. Si vous verrouillez une ligne pour accès en lecture seule, d'autres utilisateurs peuvent également la verrouiller pour accès en lecture seule, mais pas pour accès avec mise à jour.

La gestion des verrous présente des inconvénients. Elle peut être complexe à programmer. Elle nécessite des ressources de gestion de base de données importantes, et peut provoquer des problèmes de performances au fil de l'augmentation du nombre d'utilisateurs d'une application. Pour ces raisons, certains systèmes de gestion de base de données ne prennent pas en charge l'accès concurrentiel pessimiste. Entity Framework Core n'en fournit pas de prise en charge intégrée et ce didacticiel ne vous montre comment l'implémenter.

## Accès concurrentiel optimiste

La solution alternative à l'accès concurrentiel pessimiste est l'accès concurrentiel optimiste. L'accès concurrentiel optimiste signifie autoriser la survenance des conflits d'accès concurrentiel, puis de réagir correctement quand ils surviennent. Par exemple, Jane consulte la page Department Edit et change le montant de « Budget » pour le département « English » en le passant de \$350 000,00 à \$0,00.



The screenshot shows a web browser window with the address bar displaying 'localhost:581'. The page title is 'Contoso University'. The main heading is 'Edit Department'. Below the heading, there are four input fields: 'Budget' (containing '0'), 'Administrator' (a dropdown menu showing 'Abercrombie, Kim'), 'Name' (containing 'English'), and 'Start Date' (containing '9/1/2007'). A 'Save' button is located at the bottom of the form. The 'Budget' field is highlighted with a red rectangular box.

Avant que Jane clique sur **Save**, John consulte la même page et change le champ Start Date de 01/09/2007 en 01/09/2013.

Contoso University

## Edit

Department

**Budget**

350000.00

**Administrator**

Abercrombie, Kim

**Name**

English

**Start Date**

9/1/2013

Save

Jane clique la première sur **Save** et voit sa modification quand le navigateur revient à la page Index.

Contoso University

## Departments

[Create New](#)

Name	Budget	Administrator	Start Date	
English	\$0.00	Abercrombie, Kim	2007-09-01	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
matric	\$1000000.00	Eakhruri, Radj	2007-09-01	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

John clique à son tour sur **Save** sur une page Edit qui affiche toujours un budget de \$350 000,00. Ce qui se passe ensuite est déterminé par la façon dont vous gérez les conflits d'accès concurrentiel.

Voici quelques-unes des options :

- Vous pouvez effectuer le suivi des propriétés modifiées par un utilisateur et mettre à jour seulement les colonnes correspondantes dans la base de données.

Dans l'exemple de scénario, aucune donnée ne serait perdue, car des propriétés différentes ont été mises à jour par chacun des deux utilisateurs. La prochaine fois que quelqu'un examine le département « English », il voit à la fois les modifications de Jane et de John : une date de début au 01/09/2013 et un budget de zéro dollars. Cette méthode de mise à jour peut réduire le nombre de conflits qui peuvent entraîner des pertes de données, mais elle ne peut pas éviter la perte de données si des modifications concurrentes sont apportées à la même propriété d'une entité. Un tel fonctionnement d'Entity Framework dépend de la façon dont vous implémentez votre code de mise à jour. Il n'est pas souvent pratique dans une application web, car il peut nécessiter la gestion de grandes quantités d'états pour effectuer le suivi de toutes les valeurs de propriété d'origine d'une entité, ainsi que des nouvelles valeurs. La gestion de grandes quantités d'états peut affecter les performances de l'application, car elle nécessite des ressources serveur, ou doit être incluse dans la page web elle-même (par exemple dans des champs masqués) ou dans un cookie.

- Vous pouvez laisser les modifications de John remplacer les modifications de Jane.

La prochaine fois que quelqu'un consultera le département « English », il verra la date du 01/09/2013 et la valeur \$350 000,00 restaurée. Ceci s'appelle un scénario *Priorité au client* ou *Priorité au dernier entré* (Last in Wins). (Toutes les valeurs provenant du client sont prioritaires par rapport à ce qui se trouve dans le magasin de données.) Comme indiqué dans l'introduction de cette section, si vous ne codez rien pour la gestion des accès concurrentiels, ceci se produit automatiquement.

- Vous pouvez empêcher les modifications de John de faire l'objet d'une mise à jour dans la base de données.

En règle générale, vous affichez un message d'erreur, vous lui montrez l'état actuel des données et vous lui permettez de réappliquer ses modifications s'il veut toujours les faire. Il s'agit alors d'un scénario *Priorité au magasin*. (Les valeurs du magasin de données sont prioritaires par rapport à celles soumises par le client.) Dans ce didacticiel, vous allez implémenter le scénario *Priorité au magasin*. Cette méthode garantit qu'aucune modification n'est remplacée sans qu'un utilisateur soit averti de ce qui se passe.

## Détection des conflits d'accès concurrentiel

Vous pouvez résoudre les conflits en gérant les exceptions `DbConcurrencyException` levées par Entity Framework. Pour savoir quand lever ces exceptions, Entity Framework doit être en mesure de détecter les conflits. Par conséquent, vous devez configurer de façon appropriée la base de données et le modèle de données. Voici quelques options pour l'activation de la détection des conflits :

- Dans la table de base de données, incluez une colonne de suivi qui peut être utilisée pour déterminer quand une ligne a été modifiée. Vous pouvez ensuite configurer Entity Framework pour inclure cette colonne dans la clause WHERE des commandes SQL UPDATE et DELETE.

Le type de données de la colonne de suivi est généralement `rowversion`. La valeur de `rowversion` est un nombre séquentiel qui est incrémenté chaque fois que la ligne est mise à jour. Dans une commande UPDATE ou DELETE, la clause WHERE inclut la valeur d'origine de la colonne de suivi (la version d'origine de la ligne). Si la ligne à mettre à jour a été changée par un autre utilisateur, la valeur de la colonne `rowversion` est différente de la valeur d'origine : l'instruction UPDATE ou DELETE ne peut donc pas trouver la ligne à mettre à jour en raison de la clause WHERE. Quand Entity Framework trouve qu'aucune ligne n'a été mise à jour par la commande UPDATE ou DELETE (c'est-à-dire quand le nombre de lignes affectées est égal à zéro), il interprète ceci comme étant un conflit d'accès concurrentiel.

- Configurez Entity Framework de façon à inclure les valeurs d'origine de chaque colonne dans la table de la clause WHERE des commandes UPDATE et DELETE.

Comme dans la première option, si quelque chose dans la ligne a changé depuis la première lecture de la ligne, la clause WHERE ne retourne pas de ligne à mettre à jour, ce qui est interprété par Entity Framework comme un conflit d'accès concurrentiel. Pour les tables de base de données qui ont beaucoup de colonnes, cette approche peut aboutir à des clauses WHERE de très grande taille et nécessiter la gestion de grandes quantités d'états. Comme indiqué précédemment, la gestion de grandes quantités d'états peut affecter les performances de l'application. Par conséquent, cette approche n'est généralement pas recommandée et n'est pas la méthode utilisée dans ce didacticiel.

Si vous ne voulez pas implémenter cette approche de l'accès concurrentiel, vous devez marquer toutes les propriétés qui ne sont pas des clés primaires de l'entité dont vous voulez suivre les accès concurrentiels en leur ajoutant l'attribut

ConcurrencyCheck. Cette modification permet à Entity Framework d'inclure toutes les colonnes dans la clause SQL WHERE des instructions UPDATE et DELETE.

Dans le reste de ce didacticiel, vous ajoutez une propriété de suivi `rowversion` à l'entité `Department`, vous créez un contrôleur et des vues, et vous testez pour vérifier que tout fonctionne correctement.

## Ajouter une propriété de suivi

Dans *Models/Department.cs*, ajoutez une propriété de suivi nommée `RowVersion` :

```
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; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

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

L'attribut `Timestamp` spécifie que cette colonne sera incluse dans la clause WHERE des commandes UPDATE et DELETE envoyées à la base de données. L'attribut est nommé `Timestamp`, car les versions précédentes de SQL Server utilisaient un type de données SQL

`timestamp` avant son remplacement par le type SQL `rowversion`. Le type .NET pour `rowversion` est un tableau d'octets.

Si vous préférez utiliser l'API actuelle, vous pouvez utiliser la méthode `IsConcurrencyToken` (dans `Data/SchoolContext.cs`) pour spécifier la propriété de suivi, comme indiqué dans l'exemple suivant :

```
modelBuilder.Entity<Department>()  
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

En ajoutant une propriété, vous avez changé le modèle de base de données et vous devez donc effectuer une autre migration.

Enregistrez vos modifications et générez le projet, puis entrez les commandes suivantes dans la fenêtre Commande :

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

## Créer un contrôleur Departments et des vues

Générez automatiquement un modèle de contrôleur Departments et des vues, comme vous l'avez fait précédemment pour les étudiants, les cours et les enseignants.

The screenshot shows the 'Add Controller' dialog box with the following configuration:

- Model class:** Department (ContosoUniversity.Models)
- Data context class:** SchoolContext (ContosoUniversity.Data)
- Views:**
  - ☒ Generate views
  - ☒ Reference script libraries
  - ☒ Use a layout page:
- Controller name:** DepartmentsController

The 'Add' button is highlighted with a red box.

Dans le fichier *DepartmentsController.cs*, changez les quatre occurrences de « FirstMidName » en « FullName », de façon que les listes déroulantes de l'administrateur du département contiennent le nom complet de l'enseignant et non pas simplement son nom de famille.

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

## Mettre à jour la vue Index

Le moteur de génération de modèles automatique a créé une colonne RowVersion pour la vue Index, mais ce champ ne doit pas être affiché.

Remplacez le code dans *Students/Index.cshtml* par le code suivant.

```

@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```



Ceci change l'en-tête en « Departments », supprime la colonne RowVersion et montre à l'administrateur le nom complet au lieu du prénom.

## Mettre à jour les méthodes de modification

Dans la méthode `HttpGet Edit` et la méthode `Details`, ajoutez `AsNoTracking`. Dans la méthode `HttpGet Edit`, ajoutez un chargement hâtif pour l'administrateur.

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);
```

Remplacez le code existant pour la méthode `HttpPost Edit` méthode par le code suivant :

```

[HttpPost]
[ValidateAntiForgeryToken]
//References | 0 requests | 0 exceptions
public async Task<ActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).SingleOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
                    ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID != clientValues.InstructorID)
                {
                    Instructor databaseInstructor = await _context.Instructors.SingleOrDefaultAsync(i => i.ID == databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current value: {databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you attempted to edit "
                    + "was modified by another user after you got the original value. The "
                    + "edit operation was canceled and the current values in the database "
                    + "have been displayed. If you still want to edit this record, click "
                    + "the Save button again. Otherwise click the Back to List hyperlink.");
                departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }

    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}

```

Le code commence par essayer de lire le département à mettre à jour. Si la méthode `SingleOrDefaultAsync` retourne null, c'est que le département a été supprimé par un autre utilisateur. Dans ce cas, le code utilise les valeurs du formulaire envoyé pour créer une entité `Department` de façon que la page `Edit` puisse être réaffichée avec un message

d'erreur. Vous pouvez aussi ne pas recréer l'entité `Department` si vous affichez seulement un message d'erreur sans réafficher les champs du département.

La vue stocke la valeur d'origine de `RowVersion` dans un champ masqué, et cette méthode reçoit cette valeur dans le paramètre `rowVersion`. Avant d'appeler `SaveChanges`, vous devez placer la valeur d'origine de la propriété `RowVersion` dans la collection `OriginalValues` pour l'entité.

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

Ensuite, quand Entity Framework crée une commande SQL UPDATE, cette commande inclut une clause WHERE qui recherche une ligne contenant la valeur d'origine de `RowVersion`. Si aucune ligne n'est affectée par la commande UPDATE (aucune ligne ne contient la valeur `RowVersion` d'origine), Entity Framework lève une exception `DbUpdateConcurrencyException`.

Le code du bloc catch pour cette exception obtient l'entité `Department` affectée qui a les valeurs mises à jour de la propriété `Entries` sur l'objet d'exception.

```
var exceptionEntry = ex.Entries.Single();
```

La collection `Entries` n'a qu'un objet `EntityEntry`. Vous pouvez utiliser cet objet pour obtenir les nouvelles valeurs entrées par l'utilisateur et les valeurs actuelles de la base de données.

```
var clientValues = (Department)exceptionEntry.Entity;  
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

Le code ajoute un message d'erreur personnalisé pour chaque colonne dont les valeurs dans la base de données diffèrent de ce que l'utilisateur a entré dans la page Edit (un seul champ est montré ici par souci de concision).

```
var databaseValues = (Department)databaseEntry.ToObject();  
  
if (databaseValues.Name != clientValues.Name)  
{  
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");  
}
```

Enfin, le code affecte la nouvelle valeur récupérée auprès de la base de données à `RowVersion` pour `departmentToUpdate`. Cette nouvelle valeur de `RowVersion` est stockée dans le champ masqué quand la page Edit est réaffichée et, la prochaine fois que l'utilisateur clique sur **Save**, seules les erreurs d'accès concurrentiel qui se produisent depuis le réaffichage de la page Edit sont interceptées.

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

L'instruction `ModelState.Remove` est nécessaire, car `ModelState` contient l'ancienne valeur de `RowVersion`. Dans la vue, la valeur `ModelState` d'un champ est prioritaire par rapport aux valeurs de propriétés du modèle quand les deux sont présentes.

## Mettre à jour la vue Edit

Dans *Views/Departments/Edit.cshtml*, faites les modifications suivantes :

- Ajoutez un champ masqué pour enregistrer la valeur de la propriété `RowVersion`, immédiatement après le champ masqué pour la propriété `DepartmentID`.
- Ajoutez une option « Select Administrator » à la liste déroulante.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
        </form>
    </div>

```

```

        <div class="form-group">
            <label asp-for="InstructorID" class="control-label"></label>
            <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID"
                <option value="">-- Select Administrator --</option>
            </select>
            <span asp-validation-for="InstructorID" class="text-danger"></span>
        </div>
        <div class="form-group">
            <input type="submit" value="Save" class="btn btn-default" />
        </div>
    </form>
</div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

## Tester les conflits d'accès concurrentiel

Exécutez l'application et accédez à la page Index des départements. Cliquez avec le bouton droit sur le lien hypertexte **Edit** pour le département « English », sélectionnez **Ouvrir dans un nouvel onglet**, puis cliquez sur le lien hypertexte **Edit** pour le département « English ». Les deux onglets du navigateur affichent maintenant les mêmes informations.

Changez un champ sous le premier onglet du navigateur, puis cliquez sur **Save**.

The screenshot shows a web browser window with two tabs. The active tab is 'Edit - Contoso'. The address bar shows 'localhost:5813/Departments'. The page title is 'Contoso University'. The main content area is titled 'Edit Department'. It contains several form fields: 'Name' with the value 'English', 'Budget' with the value '50000.00' (highlighted with a red box), 'Start Date' with the value '9/1/2007', and 'InstructorID' with a dropdown menu showing 'Abercrombie, Kim'. At the bottom is a 'Save' button.

Le navigateur affiche la page Index avec la valeur modifiée.

Changez un champ sous le deuxième onglet du navigateur.

Departments - Edit - Cont

localhost:5813/Departr

Contoso University

## Edit

### Department

**Name**

**Budget**

**Start Date**

**InstructorID**

Save

Cliquez sur **Enregistrer**. Vous voyez un message d'erreur :

Departmen Edit - Cont

localhost:5813/Departr

Contoso University

## Edit

### Department

- The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.

**Name**

**Budget**

Current value: \$50,000.00

**Start Date**

**InstructorID**

Save

Cliquez à nouveau sur **Save**. La valeur que vous avez entrée sous le deuxième onglet du navigateur est enregistrée. Vous voyez les valeurs enregistrées quand la page Index apparaît.

## Mettre à jour la page Delete

Pour la page Delete, Entity Framework détecte les conflits d'accès concurrentiel provoqués par un autre utilisateur qui modifie le service de façon similaire. Quand la méthode `HttpGet Delete` affiche la vue de confirmation, la vue inclut la version d'origine de `RowVersion` dans un champ masqué. Cette valeur est ensuite disponible pour la méthode `HttpPost Delete` qui est appelée quand l'utilisateur confirme la suppression. Quand Entity Framework crée la commande SQL DELETE, il inclut une clause WHERE avec la valeur d'origine de `RowVersion`. Si la commande a pour résultat qu'aucune ligne n'est affectée (ce qui signifie que la ligne a été modifiée après l'affichage de la page de confirmation de la suppression), une exception d'accès concurrentiel est levée et la méthode `HttpGet Delete` est appelée avec un indicateur d'erreur défini sur true pour réafficher la page de confirmation avec un message d'erreur. Il est également possible qu'aucune ligne ne soit affectée en raison du fait que la ligne a été supprimée par un autre utilisateur : dans ce cas, aucun message d'erreur n'est donc affiché.

## Mettre à jour les méthodes Delete dans le contrôleur Departments

Dans *DepartmentsController.cs*, remplacez la méthode `HttpGet Delete` par le code suivant :

```

public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}

```

La méthode accepte un paramètre facultatif qui indique si la page est réaffichée après une erreur d'accès concurrentiel. Si cet indicateur a la valeur true et que le département spécifié n'existe plus, c'est qu'il a été supprimé par un autre utilisateur. Dans ce cas, le code redirige vers la page Index. Si cet indicateur a la valeur true et que le département existe, c'est qu'il a été modifié par un autre utilisateur. Dans ce cas, le code envoie un message d'erreur à la vue en utilisant `ViewData`.

Remplacez le code de la méthode `HttpPost Delete` (nommée `DeleteConfirmed`) par le code suivant :



```

[HttpPost]
[ValidateAntiForgeryToken]
1 reference | 0 requests | 0 exceptions
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID });
    }
}

```

Dans le code du modèle généré automatiquement que vous venez de remplacer, cette méthode n'acceptait qu'un seul ID d'enregistrement :

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

Vous avez changé ce paramètre en une instance d'entité `Department` créée par le classeur de modèles. Ceci permet à Entity Framework d'accéder à la valeur de la propriété `RowVersion` en plus de la clé d'enregistrement.

```
public async Task<IActionResult> Delete(Department department)
```

Vous avez également changé le nom de la méthode d'action de `DeleteConfirmed` en `Delete`. Le code du modèle généré automatiquement utilisait le nom `DeleteConfirmed` pour donner à la méthode `HttpPost` une signature unique. (Pour le CLR, les méthodes surchargées doivent avoir des paramètres de méthode différents.) Maintenant que les signatures sont uniques, vous pouvez appliquer la convention MVC et utiliser le même nom pour les méthodes delete `HttpPost` et `HttpGet`.

Si le département est déjà supprimé, la méthode `AnyAsync` retourne la valeur `false` et l'application revient simplement à la méthode `Index`.

Si une erreur d'accès concurrentiel est interceptée, le code réaffiche la page de confirmation de suppression et fournit un indicateur indiquant qu'elle doit afficher un message d'erreur d'accès concurrentiel.

## Mettre à jour la vue Delete

Dans `Views/Departments/Delete.cshtml`, remplacez le code du modèle généré automatiquement par le code suivant, qui ajoute un champ de message d'erreur et des champs masqués pour les propriétés `DepartmentID` et `RowVersion`. Les modifications apparaissent en surbrillance.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

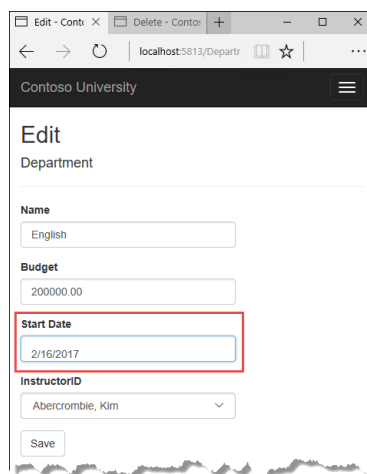
    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

Ceci apporte les modifications suivantes :

- Ajoute un message d'erreur entre les titres h2 et h3.
- Remplace FirstMidName par FullName dans le champ **Administrator**.
- Supprime le champ RowVersion.
- Ajoute un champ masqué pour la propriété RowVersion.

Exécutez l'application et accédez à la page Index des départements. Cliquez avec le bouton droit sur le lien hypertexte **Delete** pour le département « English », sélectionnez **Ouvrir dans un nouvel onglet** puis, sous le premier onglet, cliquez sur le lien hypertexte **Edit** pour le département « English ».

Dans la première fenêtre, changez une des valeurs, puis cliquez sur **Save** :



Contoso University

## Edit

Department

Name  
English

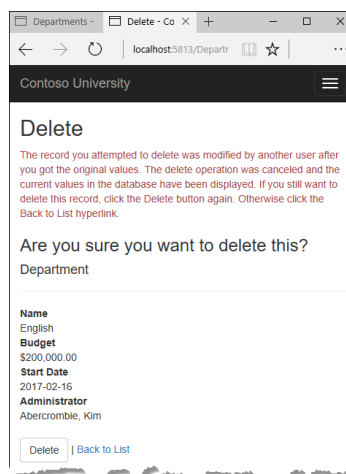
Budget  
200000.00

Start Date  
2/16/2017

InstructorID  
Abercrombie, Kim

Save

Sous le deuxième onglet, cliquez sur **Delete**. Vous voyez le message d'erreur d'accès concurrentiel et les valeurs du département sont actualisées avec ce qui est actuellement dans la base de données.



Contoso University

## Delete

The record you attempted to delete was modified by another user after you got the original values. The delete operation was canceled and the current values in the database have been displayed. If you still want to delete this record, click the Delete button again. Otherwise click the Back to List hyperlink.

Are you sure you want to delete this?

Department

Name  
English

Budget  
\$200,000.00

Start Date  
2017-02-16

Administrator  
Abercrombie, Kim

Delete | Back to List

Si vous recliquez sur **Delete**, vous êtes redirigé vers la page Index, qui montre que le département a été supprimé.

## Mettre à jour les vues Details et Create

Vous pouvez éventuellement nettoyer le code du modèle généré automatiquement dans les vues Details et Create.

Remplacez le code de *Views/Departments/Details.cshtml* pour supprimer la colonne RowVersion et afficher le nom complet de l'administrateur.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Remplacez le code de *Views/Departments/Create.cshtml* pour ajouter une option de sélection à la liste déroulante.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

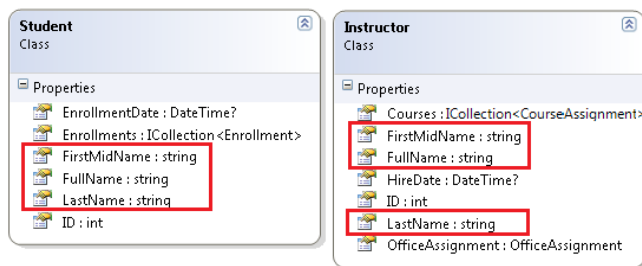
## Implémenter l'héritage

Dans le didacticiel précédent, vous avez géré les exceptions d'accès concurrentiel. Ce didacticiel vous indiquera comment implémenter l'héritage dans le modèle de données.

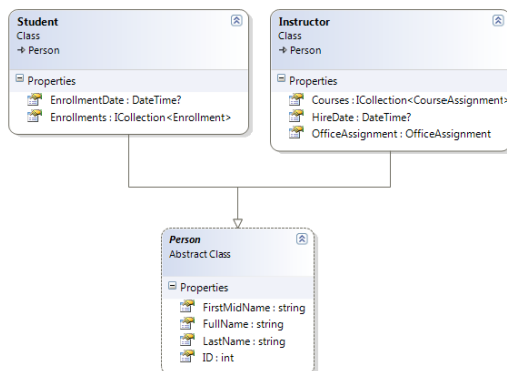
En programmation orientée objet, vous pouvez utiliser l'héritage pour faciliter la réutilisation du code. Dans ce didacticiel, vous allez modifier les classes `Instructor` et `Student` afin qu'elles dérivent d'une classe de base `Person` qui contient des propriétés telles que `LastName`, communes aux formateurs et aux étudiants. Vous n'ajouterez ni ne modifierez aucune page web, mais vous modifierez une partie du code et ces modifications seront automatiquement répercutées dans la base de données.

## Mapper l'héritage à la base de données

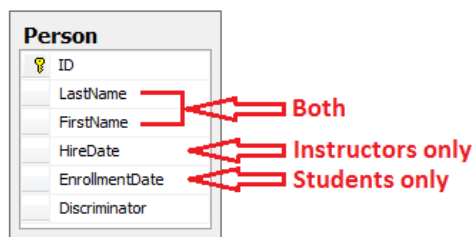
Les classes `Instructor` et `Student` du modèle de données `School` ont plusieurs propriétés identiques :



Supposons que vous souhaitez éliminer le code redondant pour les propriétés partagées par les entités `Instructor` et `Student`. Ou vous souhaitez écrire un service capable de mettre en forme les noms sans se soucier du fait que le nom provienne d'un formateur ou d'un étudiant. Vous pouvez créer une classe de base `Person` qui contient uniquement les propriétés partagées, puis paramétrer les classes `Instructor` et `Student` pour qu'elles héritent de cette classe de base, comme indiqué dans l'illustration suivante :

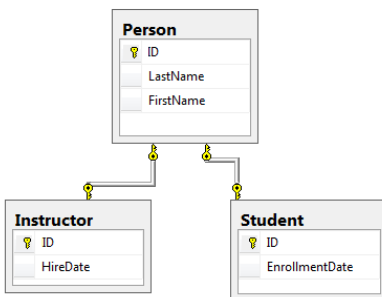


Il existe plusieurs façons de représenter cette structure d'héritage dans la base de données. Vous pouvez avoir une table de personnes qui inclut des informations sur les étudiants et les formateurs dans une table unique. Certaines des colonnes pourraient s'appliquer uniquement aux formateurs (HireDate), certaines uniquement aux étudiants (EnrollmentDate) et certaines aux deux (LastName, FirstName). En règle générale, vous pouvez avoir une colonne de discriminateur pour indiquer le type que chaque ligne représente. Par exemple, la colonne de discriminateur peut avoir « Instructor » pour les formateurs et « Student » pour les étudiants.



Ce modèle de génération d'une structure d'héritage d'entité à partir d'une table de base de données unique porte le nom d'héritage TPH (table par hiérarchie).

Une alternative consiste à faire en sorte que la base de données ressemble plus à la structure d'héritage. Par exemple, vous pouvez avoir uniquement les champs de nom dans la table Person, et des tables Instructor et Student distinctes avec les champs de date.



Ce modèle consistant à créer une table de base de données pour chaque classe d'entité est appelé héritage TPT (table par type).

Une autre option encore consiste à mapper tous les types non abstraits à des tables individuelles. Toutes les propriétés d'une classe, y compris les propriétés héritées, sont mappées aux colonnes de la table correspondante. Ce modèle porte le nom d'héritage TPC (table par classe concrète). Si vous avez implémenté l'héritage TPC pour les classes Person, Student et Instructor comme indiqué précédemment, les tables Student et Instructor ne seraient pas différents avant et après l'implémentation de l'héritage.

Les modèles d'héritage TPC et TPH fournissent généralement de meilleures performances que les modèles d'héritage TPT, car les modèles TPT peuvent entraîner des requêtes de jointure complexes.

Ce didacticiel montre comment implémenter l'héritage TPH. TPH est le seul modèle d'héritage pris en charge par Entity Framework Core. Vous allez créer une classe `Person`, modifier les classes `Instructor` et `Student` à dériver de `Person`, ajouter la nouvelle classe à `DbContext` et créer une migration.

**Conseil :** Pensez à enregistrer une copie du projet avant d'apporter les modifications suivantes. Ensuite, si vous rencontrez des problèmes et devez recommencer, il sera plus facile de démarrer à partir du projet enregistré que d'annuler les étapes effectuées pour ce didacticiel ou de retourner au début de la série entière.

## Créer la classe Person

Dans le dossier `Models`, créez `Person.cs` et remplacez le code du modèle par le code suivant :

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

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        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; }

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



## Mettre à jour Student et Instructor

Dans *Instructor.cs*, dérivez la classe Instructor de la classe Person et supprimez les champs de clé et de nom. Le code ressemblera à l'exemple suivant :

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

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

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

Apportez les mêmes modifications dans *Student.cs*.

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

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

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

## Ajouter la classe Person au modèle

Ajoutez le type d'entité Person à *SchoolContext.cs*. Les nouvelles lignes apparaissent en surbrillance.

```
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; }
        public DbSet<Person> People { 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<Person>().ToTable("Person");

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

C'est là tout ce dont Entity Framework a besoin pour configurer l'héritage TPH (table par hiérarchie). Comme vous le verrez, lorsque la base de données sera mise à jour, elle aura une table Person à la place des tables Student et Instructor.

## Créer et mettre à jour des migrations

Enregistrez vos modifications et générez le projet. Ensuite, ouvrez la fenêtre de commande dans le dossier du projet et entrez la commande suivante :

dotnet ef migrations add Inheritance

N'exécutez pas encore la commande `database update`. Cette commande entraîne une perte de données, car elle supprime la table `Instructor` et renomme la table `Student` en `Person`. Vous devez fournir un code personnalisé pour préserver les données existantes.

Ouvrez `Migrations/<horodatage>_Inheritance.cs` et remplacez la méthode `Up` par le code suivant :

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator, OldId) " +
        "SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId FROM dbo.Student");

    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldId", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
        name: "IX_Enrollment_StudentID",
        table: "Enrollment",
        column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

Ce code prend en charge les tâches de mise à jour de base de données suivantes :

- Supprime les contraintes de clé étrangère et les index qui pointent vers la table `Student`.
- Renomme la table `Instructor` en `Person` et apporte les modifications nécessaires pour qu'elle stocke les données des étudiants :
- Ajoute une `EnrollmentDate` nullable pour les étudiants.
- Ajoute la colonne `Discriminator` pour indiquer si une ligne est pour un étudiant ou un formateur.
- Rend `HireDate` nullable étant donné que les lignes d'étudiant n'ont pas de dates d'embauche.
- Ajoute un champ temporaire qui sera utilisé pour mettre à jour les clés étrangères qui pointent vers les étudiants. Lorsque vous copiez des étudiants dans la table `Person`, ils obtiennent de nouvelles valeurs de clés primaires.

- Copie des données à partir de la table Student dans la table Person. Cela entraîne l'affectation de nouvelles valeurs de clés primaires aux étudiants.
- Corrige les valeurs de clés étrangères qui pointent vers les étudiants.
- Crée de nouveau les index et les contraintes de clé étrangère, désormais pointées vers la table Person.

(Si vous aviez utilisé un GUID à la place d'un entier comme type de clé primaire, les valeurs des clés primaires des étudiants n'auraient pas changé, et plusieurs de ces étapes auraient pu être omises.)

Exécutez la commande `database update` :

```
dotnet ef database update
```

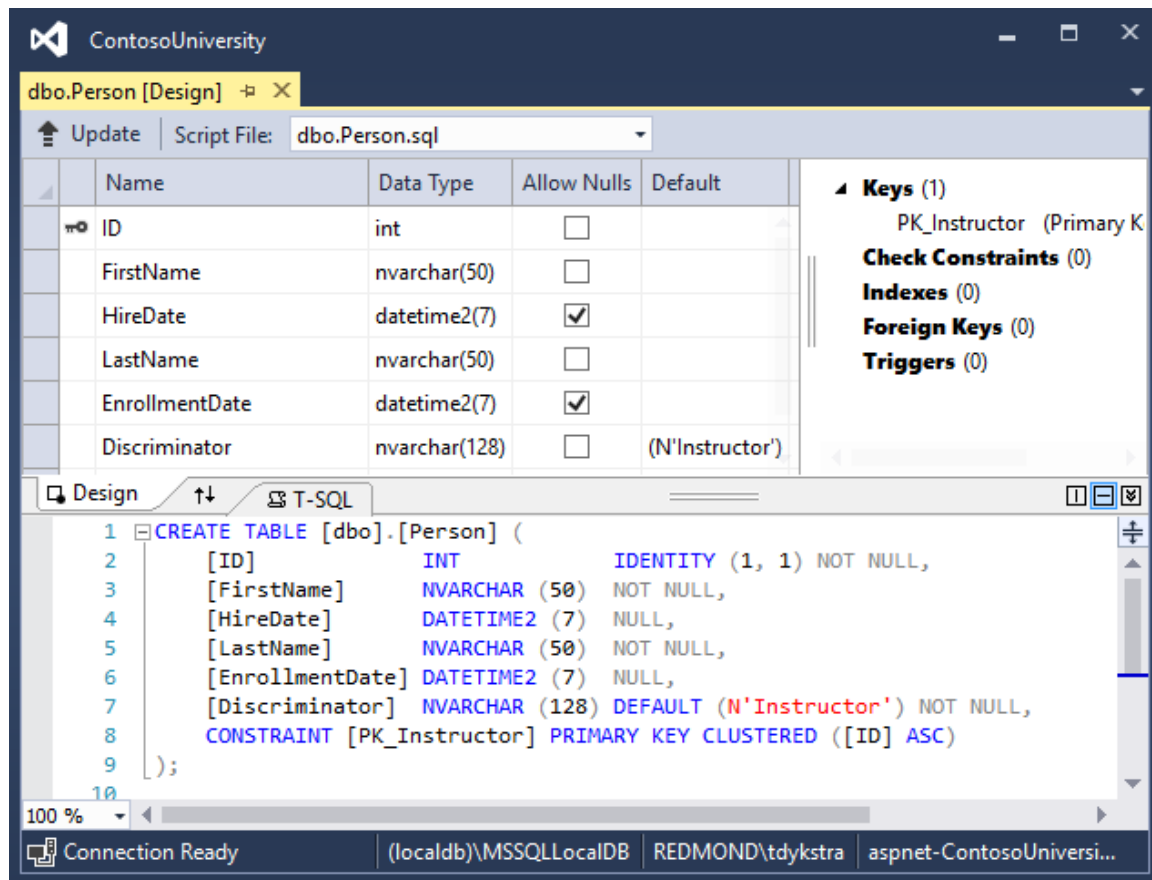
(Dans un système de production, vous apporteriez les modifications correspondantes à la méthode `Down` au cas où vous auriez à l'utiliser pour revenir à la version précédente de la base de données. Pour ce didacticiel, vous n'utiliserez pas la méthode `Down`.)

**Notes :** Vous pouvez obtenir d'autres erreurs en apportant des modifications au schéma dans une base de données qui comporte déjà des données. Si vous obtenez des erreurs de migration que vous ne pouvez pas résoudre, vous pouvez changer le nom de la base de données dans la chaîne de connexion ou supprimer la base de données. Avec une nouvelle base de données, il n'y a pas de données à migrer et la commande de mise à jour de base de données a plus de chances de s'exécuter sans erreur. Pour supprimer la base de données, utilisez SSIX ou exécutez la commande CLI `database drop`.

## Tester l'implémentation

Exécutez l'application et essayez différentes pages. Tout fonctionne comme avant.

Dans l'**Explorateur d'objets SQL Server**, développez **Data Connections/SchoolContext** puis **Tables**, et vous constatez que les tables Student et Instructor ont été remplacées par une table Person. Ouvrez le concepteur de la table Person et vous constatez qu'elle possède toutes les colonnes qui existaient dans les tables Student et Instructor.



Cliquez avec le bouton droit sur la table Person, puis cliquez sur **Afficher les données de la table** pour voir la colonne de discriminateur.

ContosoUniversity

dbo.Person [Data]

Max Rows: 1000

	ID	FirstName	HireDate	LastName	Enrollme...	Discriminator
1	1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
2	2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
3	3	Roger	7/1/1998 ...	Harui	NULL	Instructor
4	4	Candace	1/15/2001...	Kapoor	NULL	Instructor
5	5	Roger	2/12/2004...	Zheng	NULL	Instructor
7	7	Nancy	8/17/2016...	Davolio	NULL	Instructor
8	8	Carson	NULL	Alexander	9/1/2010 ...	Student
9	9	Meredith	NULL	Alonso	9/1/2012 ...	Student
10	10	Arturo	NULL	Anand	9/1/2013 ...	Student
11	11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
12	12	Yan	NULL	Li	9/1/2012 ...	Student