

Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Step By Step, Guide



Getting Started with Entity Framework 6 Code First using MVC 5

Tom Dykstra, Rick Anderson

Summary: The Contoso University sample web application demonstrates how to create ASP.NET MVC 5 applications using the Entity Framework 6, Code First workflow. This tutorial shows how to build the application using Visual Studio 2013.

Category: Step-by-Step, Guide

Applies to: Entity Framework 6, MVC 5, Visual Studio 2013

Source: ASP.NET (<http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application>)

E-book publication date: April, 2014

For more titles, visit the [E-Book Gallery for Microsoft Technologies](#).

Copyright © 2014 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Table of Contents

Contents

Creating an Entity Framework Data Model	9
Introduction	9
Software versions used in the tutorial	9
Tutorial versions	9
Questions and comments	9
The Contoso University Web Application	10
Prerequisites	11
Create an MVC Web Application	12
Set Up the Site Style	13
Install Entity Framework 6	17
Create the Data Model	17
The Student Entity	18
The Enrollment Entity	19
The Course Entity	20
Create the Database Context	21
Specifying entity sets	22
Specifying the connection string	22
Specifying singular table names	22
Set up EF to initialize the database with test data	23
Set up EF to use a SQL Server Express LocalDB database	26
Creating a Student Controller and Views	26
View the Database	31
Conventions	33
Summary	33
Implementing Basic CRUD Functionality with the Entity Framework in ASP.NET MVC Application	34
Create a Details Page	37
Route data	38
Update the Create Page	41
Update the Edit HttpPost Page	46
Entity States and the Attach and SaveChanges Methods	47

Updating the Delete Page	50
Ensuring that Database Connections Are Not Left Open	53
Handling Transactions	54
Summary.....	54
Sorting, Filtering, and Paging with the Entity Framework in an ASP.NET MVC Application	55
Add Column Sort Links to the Students Index Page	56
Add Sorting Functionality to the Index Method	56
Add Column Heading Hyperlinks to the Student Index View.....	58
Add a Search Box to the Students Index Page	60
Add Filtering Functionality to the Index Method	60
Add a Search Box to the Student Index View.....	62
Add Paging to the Students Index Page.....	63
Install the PagedList.Mvc NuGet Package	64
Add Paging Functionality to the Index Method.....	65
Add Paging Links to the Student Index View	67
Create an About Page That Shows Student Statistics	71
Create the View Model.....	71
Modify the Home Controller.....	72
Modify the About View.....	73
Summary.....	74
Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application.....	75
Enable connection resiliency	75
Enable Command Interception.....	77
Create a logging interface and class	77
Create interceptor classes.....	80
Test logging and connection resiliency.....	86
Summary.....	91
Code First Migrations and Deployment with the Entity Framework in an ASP.NET MVC Application... 92	92
Enable Code First Migrations.....	92
Set up the Seed Method	96
Execute the First Migration	101
Deploy to Windows Azure.....	103

Using Code First Migrations to Deploy the Database	103
Get a Windows Azure account	103
Create a web site and a SQL database in Windows Azure	103
Deploy the application to Windows Azure	107
Advanced Migrations Scenarios	122
Code First Initializers	122
Summary	123
Creating a More Complex Data Model for an ASP.NET MVC Application	124
Customize the Data Model by Using Attributes	125
The DataType Attribute	126
The StringLengthAttribute	128
The Column Attribute	130
Complete Changes to the Student Entity	132
The Required Attribute	133
The Display Attribute	134
The FullName Calculated Property	134
Create the Instructor Entity	134
The Courses and OfficeAssignment Navigation Properties	135
Create the OfficeAssignment Entity	136
The Key Attribute	137
The ForeignKey Attribute	137
The Instructor Navigation Property	137
Modify the Course Entity	138
The DatabaseGenerated Attribute	139
Foreign Key and Navigation Properties	139
Create the Department Entity	139
The Column Attribute	140
Foreign Key and Navigation Properties	141
Modify the Enrollment Entity	141
Foreign Key and Navigation Properties	142
Many-to-Many Relationships	142
Entity Diagram Showing Relationships	145
Customize the Data Model by adding Code to the Database Context	147

Seed the Database with Test Data	148
Add a Migration and Update the Database	154
Summary	157
Reading Related Data with the Entity Framework in an ASP.NET MVC Application	158
Lazy, Eager, and Explicit Loading of Related Data	160
Performance considerations.....	161
Disable lazy loading before serialization.....	161
Create a Courses Page That Displays Department Name	162
Create an Instructors Page That Shows Courses and Enrollments	165
Create a View Model for the Instructor Index View	168
Create the Instructor Controller and Views.....	168
Modify the Instructor Index View	171
Adding Explicit Loading	178
Summary	179
Updating Related Data with the Entity Framework in an ASP.NET MVC Application	180
Customize the Create and Edit Pages for Courses	183
Adding an Edit Page for Instructors	191
Adding Course Assignments to the Instructor Edit Page	195
Update the DeleteConfirmed Method	205
Add office location and courses to the Create page	205
Handling Transactions	209
Summary	209
Async and Stored Procedures with the Entity Framework in an ASP.NET MVC Application	210
Why bother with asynchronous code	212
Create the Department controller	213
Use stored procedures for inserting, updating, and deleting	217
Deploy to Windows Azure	221
Summary	222
Handling Concurrency with the Entity Framework 6 in an ASP.NET MVC 5 Application (10 of 12)	223
Concurrency Conflicts	224
Pessimistic Concurrency (Locking)	225
Optimistic Concurrency	225
Detecting Concurrency Conflicts.....	228

Add an Optimistic Concurrency Property to the Department Entity	229
Modify the Department Controller	230
Testing Optimistic Concurrency Handling	233
Updating the Delete Page	240
Summary	248
Implementing Inheritance with the Entity Framework 6 in an ASP.NET MVC 5 Application (11 of 12)	249
Options for mapping inheritance to database tables	249
Create the Person class	251
Make Student and Instructor classes inherit from Person	252
Add the Person Entity Type to the Model	253
Create and Update a Migrations File	253
Testing	255
Deploy to Windows Azure	258
Summary	260
Advanced Entity Framework 6 Scenarios for an MVC 5 Web Application (12 of 12)	261
Performing Raw SQL Queries	263
Calling a Query that Returns Entities	264
Calling a Query that Returns Other Types of Objects	265
Calling an Update Query	267
No-Tracking Queries	273
Examining SQL sent to the database	278
Repository and unit of work patterns	283
Proxy classes	284
Automatic change detection	286
Automatic validation	286
Entity Framework Power Tools	286
Entity Framework source code	289
Summary	289
Acknowledgments	289
VB	289
Common errors, and solutions or workarounds for them	289
Cannot create/shadow copy	290
Update-Database not recognized	290

Validation failed	290
HTTP 500.19 error	290
Error locating SQL Server instance	291

Creating an Entity Framework Data Model

[Download Completed Project](#)

Introduction

The Contoso University sample web application demonstrates how to create ASP.NET MVC 5 applications using the Entity Framework 6 and Visual Studio 2013. This tutorial uses the Code First workflow. For information about how to choose between Code First, Database First, and Model First, see [Entity Framework Development Workflows](#).

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This tutorial series explains how to build the Contoso University sample application. You can [download the completed application](#).

Software versions used in the tutorial

- [Visual Studio 2013](#)
- .NET 4.5
- Entity Framework 6 (EntityFramework 6.1.0 NuGet package)
- [Windows Azure SDK 2.2](#) (or later, for the optional Azure deployment steps)

The tutorial should also work with [Visual Studio 2013 Express for Web](#) or Visual Studio 2012. The [VS 2012 version of the Windows Azure SDK](#) is required for Windows Azure deployment with Visual Studio 2012.

Tutorial versions

For previous versions of this tutorial, see [the EF 4.1 / MVC 3 e-book](#) and [Getting Started with EF 5 using MVC 4](#).

Questions and comments

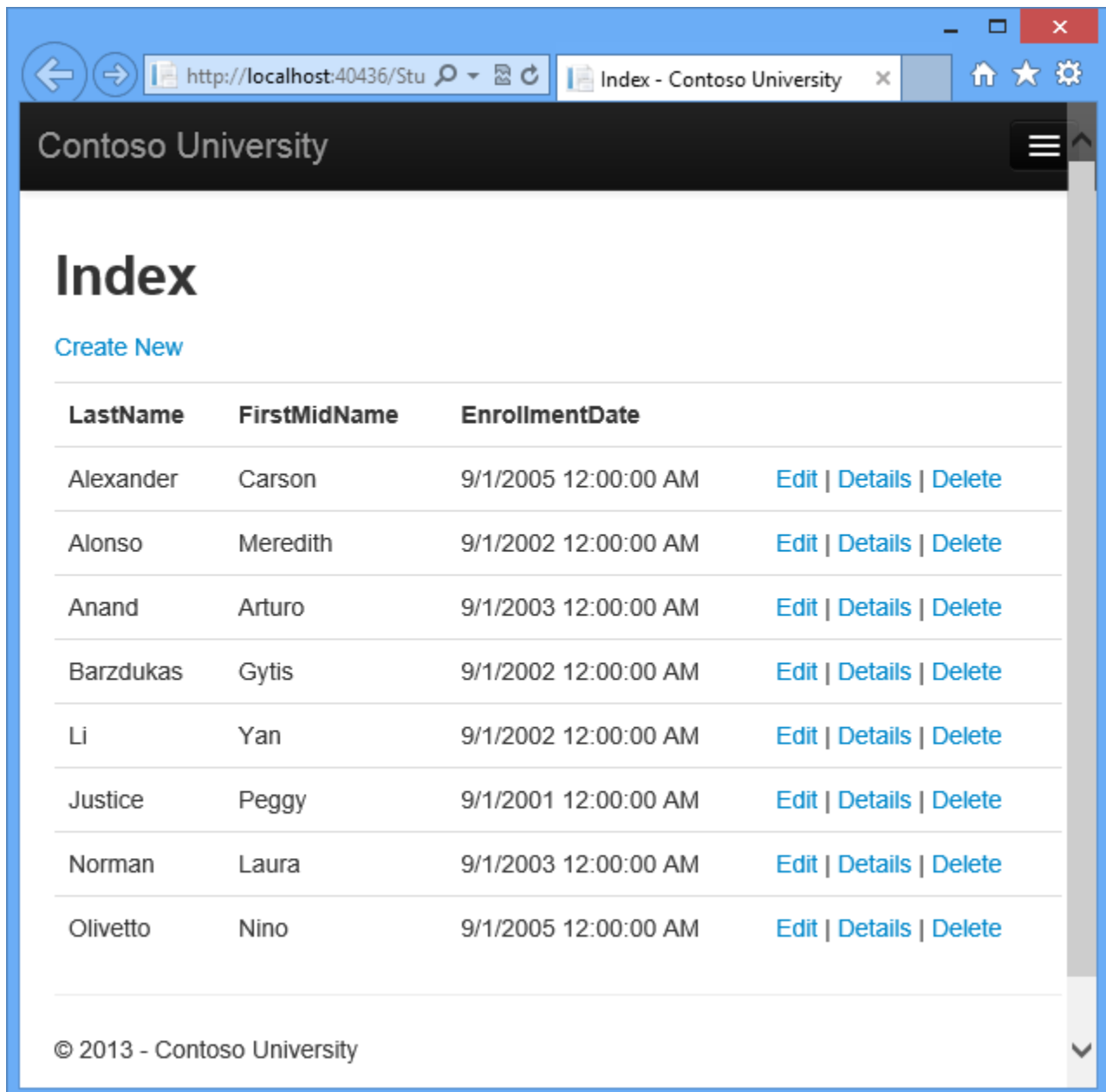
Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the pages in the version of this tutorial on the ASP.NET site. If you have questions that are not directly related to the tutorial, you can post them to the [ASP.NET Entity Framework forum](#), the [Entity Framework and LINQ to Entities forum](#), or [StackOverflow.com](#).

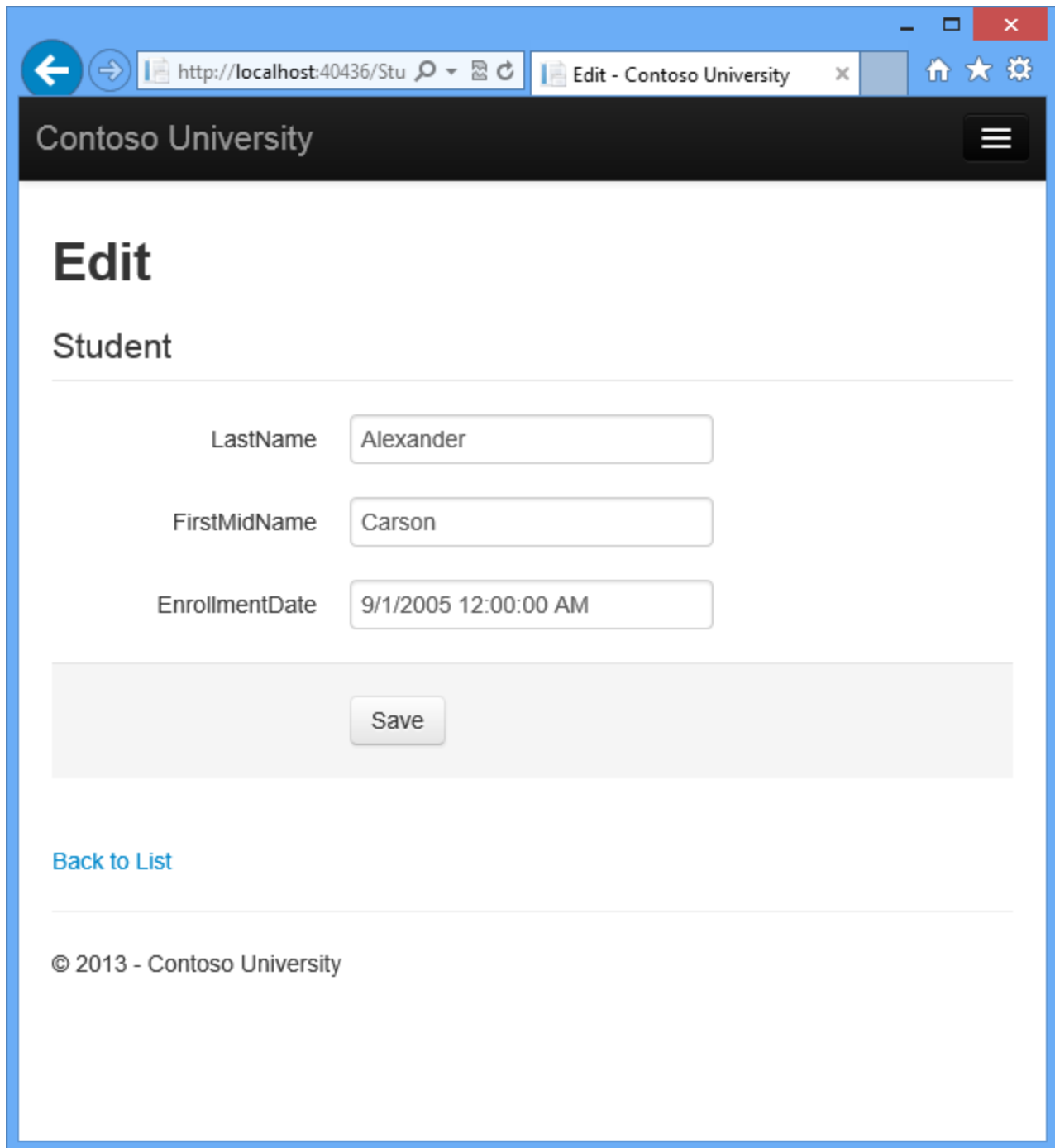
If you run into a problem you can't resolve, you can generally find the solution to the problem by comparing your code to the completed project that you can download. For some common errors and how to solve them, see [Common errors, and solutions or workarounds for them](#).

The Contoso University Web Application

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

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





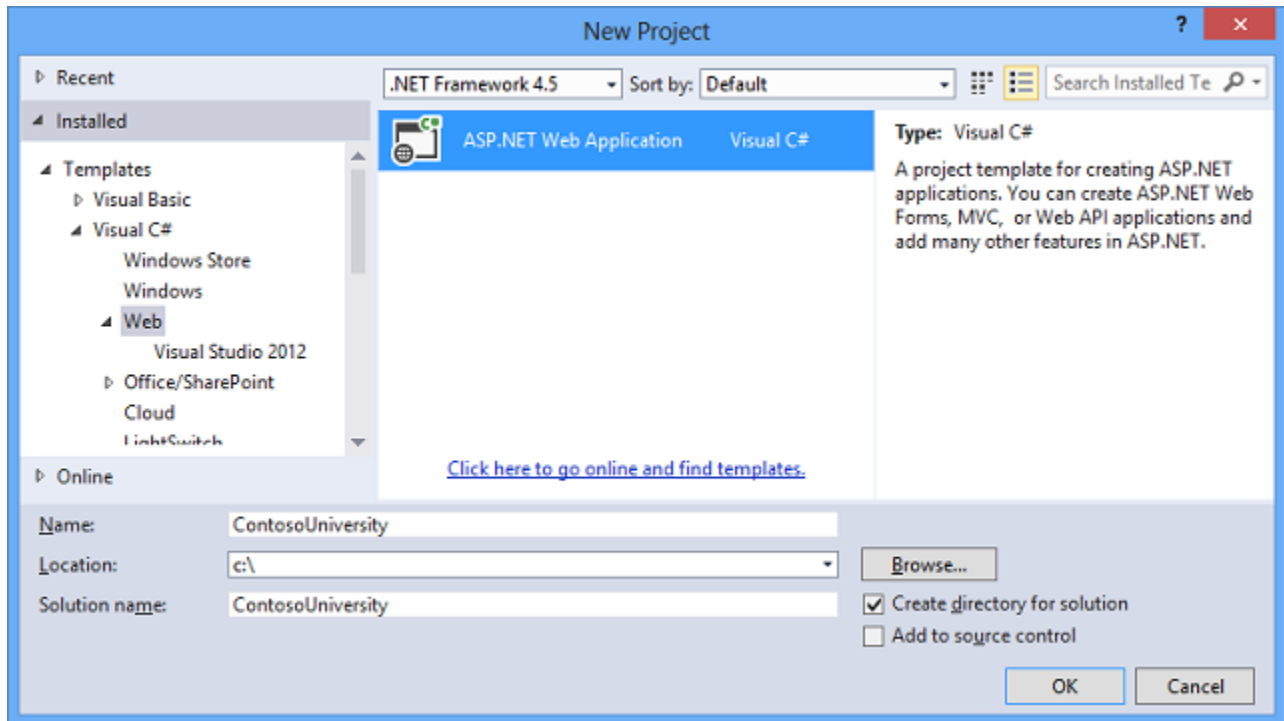
The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Prerequisites

See **Software Versions** at the top of the chapter. Entity Framework 6 is not a prerequisite because you install the EF NuGet package is part of the tutorial.

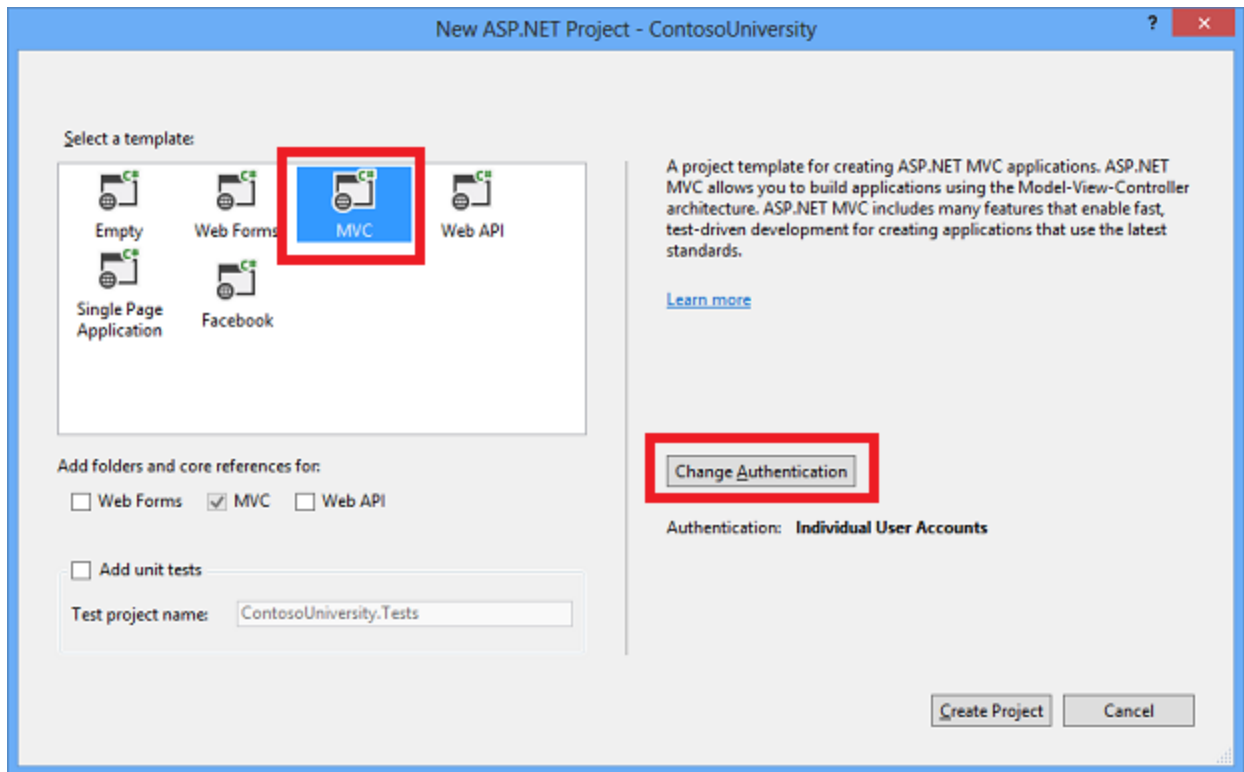
Create an MVC Web Application

Open Visual Studio and create a new C# Web project named "ContosoUniversity".

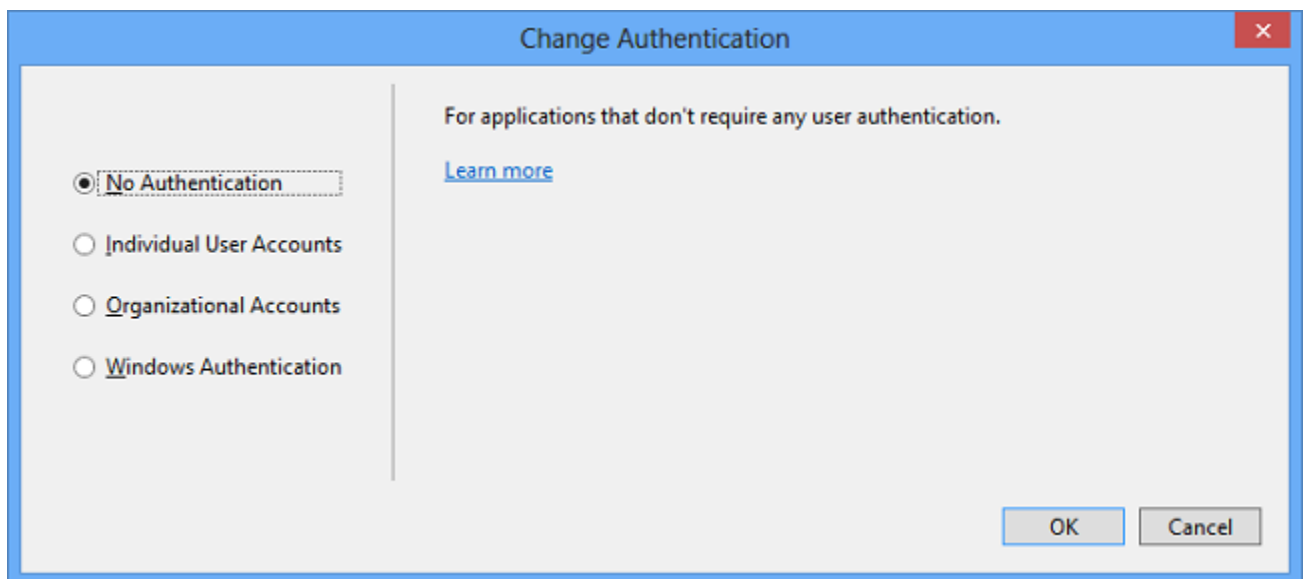


In the **New ASP.NET Project** dialog box select the **MVC** template.

Click **Change Authentication**.



In the **Change Authentication** dialog box, select **No Authentication**, and then click **OK**. For this tutorial you won't be requiring users to log on or restricting access based on who's logged on.



Back in the New ASP.NET Project dialog box, click **OK** to create the project.

Set Up the Site Style

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

Open `Views\Shared_Layout.cshtml`, and make the following changes:

- Change each occurrence of "My ASP.NET Application" and "Application name" to "Contoso University".
- Add menu entries for Students, Courses, Instructors, and Departments.

The changes are highlighted.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - Contoso University</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <button type="button" class="btn btn-navbar" data-
toggle="collapse" data-target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Contoso University", "Index", "Home", null,
new { @class = "brand" })
        <div class="nav-collapse collapse">
          <ul class="nav">
            <li>@Html.ActionLink("Home", "Index", "Home")</li>
            <li>@Html.ActionLink("About", "About", "Home")</li>
            <li>@Html.ActionLink("Students", "Index",
"Student")</li>
            <li>@Html.ActionLink("Courses", "Index",
"Course")</li>
            <li>@Html.ActionLink("Instructors", "Index",
"Instructor")</li>
            <li>@Html.ActionLink("Departments", "Index",
"Department")</li>
          </ul>
        </div>
      </div>
    </div>
  </div>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; @DateTime.Now.Year - Contoso University</p>
    </div>
  </div>
</body>
</html>
```

```

        </footer>
    </div>

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

```

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

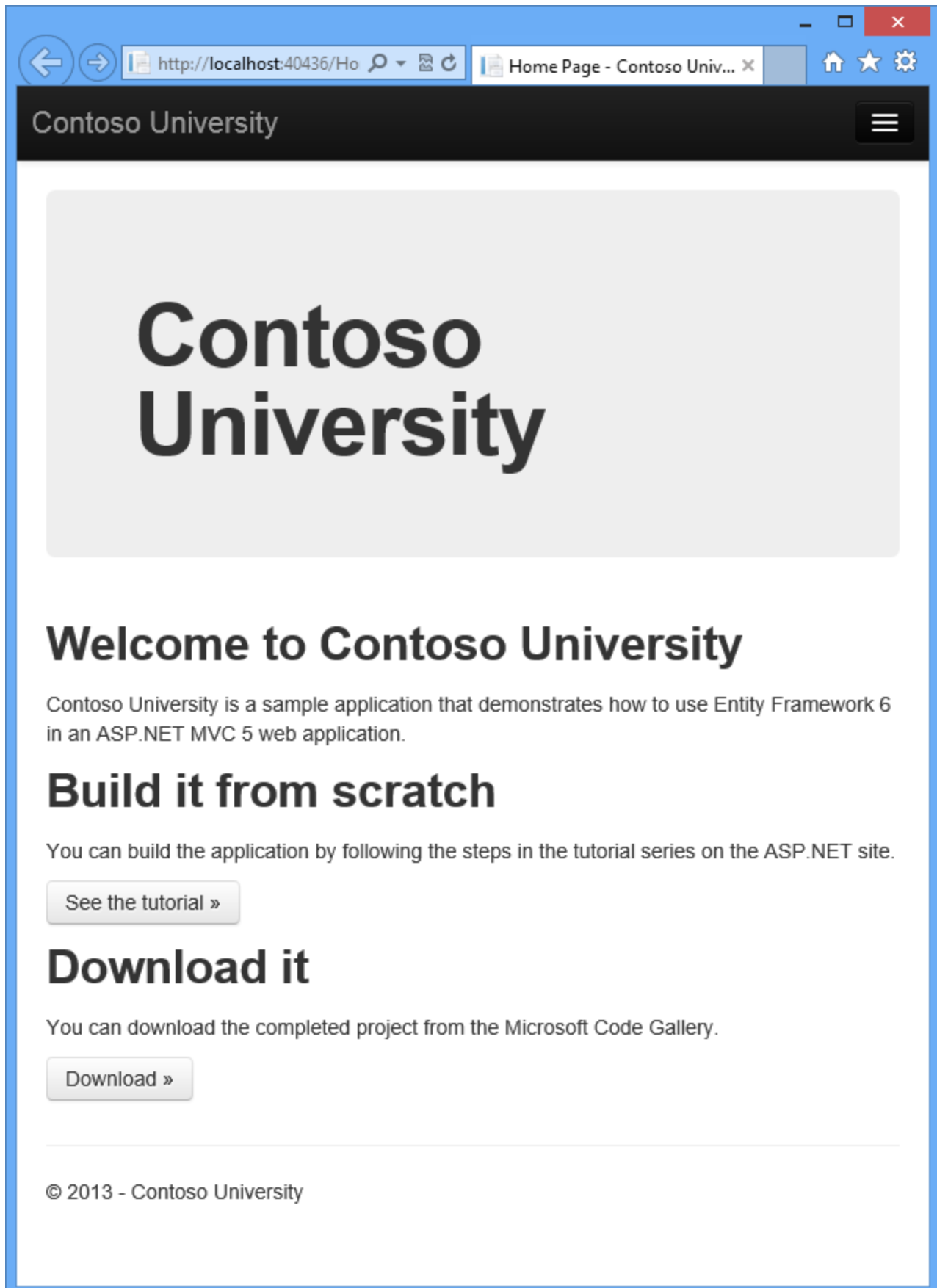
```

@{
    ViewBag.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 6 in an ASP.NET MVC 5 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 the tutorial series on the ASP.NET site.</p>
        <p><a class="btn btn-default" href="http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/">See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from the Microsoft Code Gallery.</p>
        <p><a class="btn btn-default" href="http://code.msdn.microsoft.com/ASPNET-MVC-Application-b01a9fe8">Download &raquo;</a></p>
    </div>
</div>

```

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

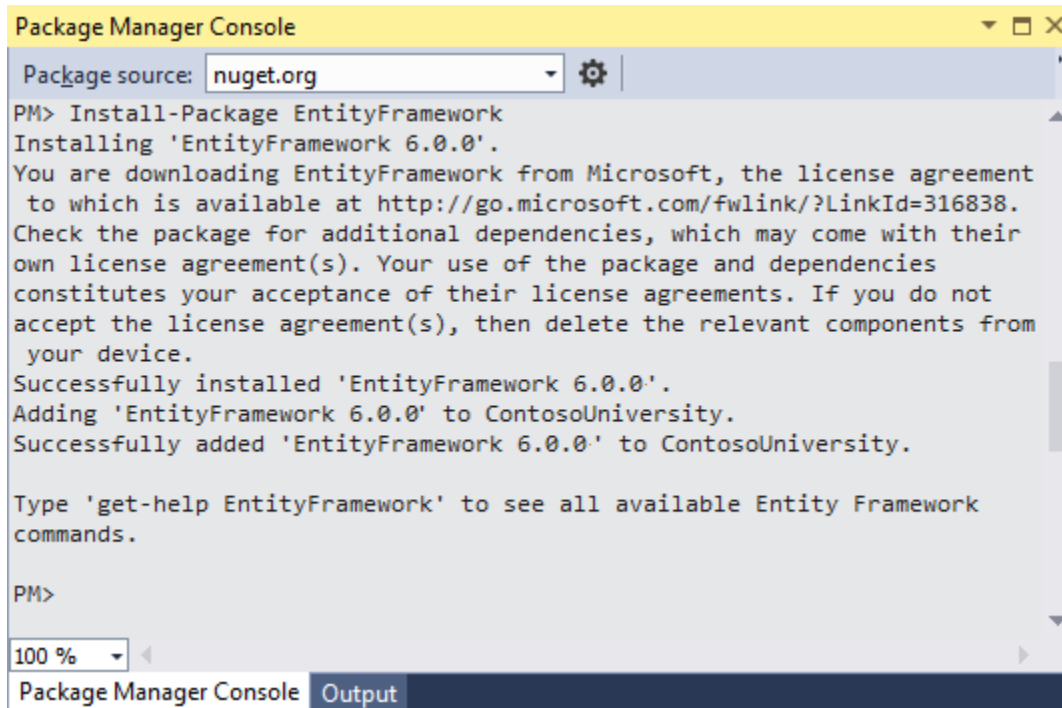


Install Entity Framework 6

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

In the **Package Manager Console** window enter the following command:

```
Install-Package EntityFramework
```

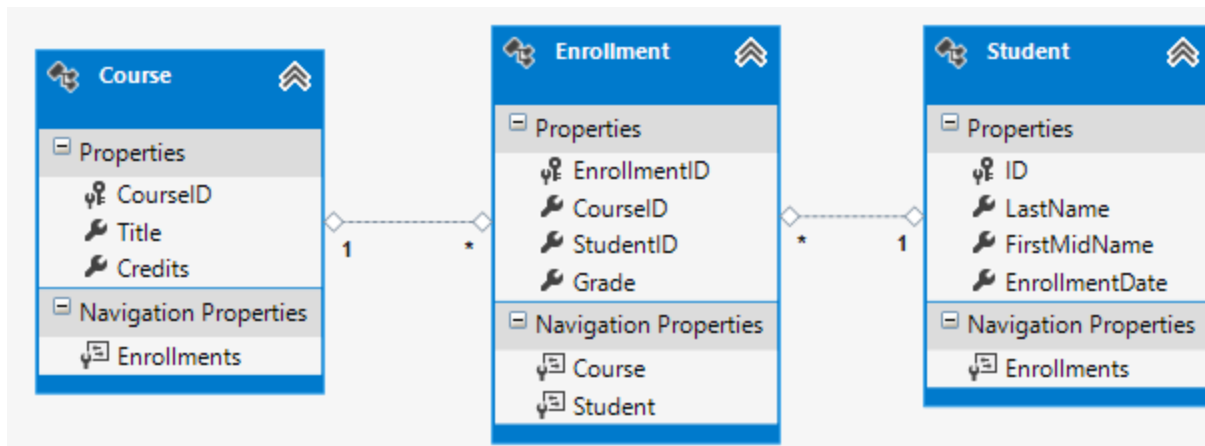


The image shows 6.0.0 being installed, but NuGet will install the latest version of Entity Framework (excluding pre-release versions), which as of the most recent update to the tutorial is 6.1.0.

This step is one of a few steps that this tutorial has you do manually, but which could have been done automatically by the ASP.NET MVC scaffolding feature. You're doing them manually so that you can see the steps required to use the Entity Framework. You'll use scaffolding later to create the MVC controller and views. An alternative is to let scaffolding automatically install the EF NuGet package, create the database context class, and create the connection string. When you're ready to do it that way, all you have to do is skip those steps and scaffold your MVC controller after you create your entity classes.

Create the Data Model

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

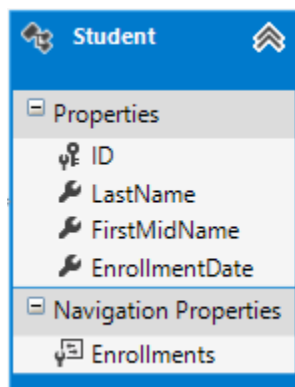


There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

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

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

The Student Entity



In the *Models* folder, create a class file named *Student.cs* and replace the template code with the following code:

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

```

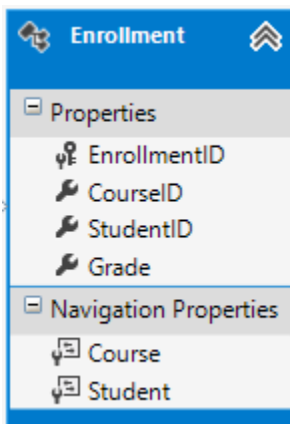
The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

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

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

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

The Enrollment Entity



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

```

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

```

```

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

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

```

The `EnrollmentID` property will be the primary key; this entity uses the *classnameID* pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using `ID` without `classname` makes it easier to implement inheritance in the data model.

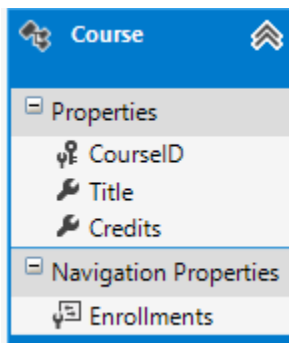
The `Grade` property is an [enum](#). The question mark after the `Grade` type declaration indicates that the `Grade` property is [nullable](#). A grade that's null is different from a zero grade — null means a grade isn't known or hasn't been assigned yet.

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

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

Entity Framework interprets a property as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named the same simply *<primary key property name>* (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course Entity



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

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

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

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

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

We'll say more about the [DatabaseGenerated](#) attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

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

To create a folder in the ContosoUniversity project, right-click the project in **Solution Explorer** and click **Add**, and then click **New Folder**. Name the new folder *DAL* (for Data Access Layer). In that folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

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

namespace ContosoUniversity.DAL
{
    public class SchoolContext : DbContext
    {
        public SchoolContext() : base("SchoolContext")
        {
        }
    }
}
```

```

public DbSet<Student> Students { get; set; }
public DbSet<Enrollment> Enrollments { get; set; }
public DbSet<Course> Courses { get; set; }

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

```

Specifying entity sets

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

You could have omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

Specifying the connection string

The name of the connection string (which you'll add to the `Web.config` file later) is passed in to the constructor.

```

public SchoolContext() : base("SchoolContext")
{
}

```

You could also pass in the connection string itself instead of the name of one that is stored in the `Web.config` file. For more information about options for specifying the database to use, see [Entity Framework - Connections and Models](#).

If you don't specify a connection string or the name of one explicitly, Entity Framework assumes that the connection string name is the same as the class name. The default connection string name in this example would then be `SchoolContext`, the same as what you're specifying explicitly.

Specifying singular table names

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

Set up EF to initialize the database with test data

The Entity Framework can automatically create (or drop and re-create) a database for you when the application runs. You can specify that this should be done every time your application runs or only when the model is out of sync with the existing database. You can also write a `Seed` method that the Entity Framework automatically calls after creating the database in order to populate it with test data.

The default behavior is to create a database only if it doesn't exist (and throw an exception if the model has changed and the database already exists). In this section you'll specify that the database should be dropped and re-created whenever the model changes. Dropping the database causes the loss of all your data. This is generally OK during development, because the `Seed` method will run when the database is re-created and will re-create your test data. But in production you generally don't want to lose all your data every time you need to change the database schema. Later you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the DAL folder, create a new class file named *SchoolInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class SchoolInitializer : System.Data.Entity.
DropCreateDatabaseIfModelChanges<SchoolContext>
    {
        protected override void Seed(SchoolContext context)
        {
            var students = new List<Student>
            {
                new
Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Pa
rse("2005-09-01")},
                new
Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Par
se("2002-09-01")},
                new
Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Parse(
"2003-09-01")},
                new
Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Par
se("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")}
    };

    students.ForEach(s => context.Students.Add(s));
    context.SaveChanges();
    var courses = new List<Course>
    {
        new Course{CourseID=1050,Title="Chemistry",Credits=3},
        new Course{CourseID=4022,Title="Microeconomics",Credits=3},
        new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
        new Course{CourseID=1045,Title="Calculus",Credits=4},
        new Course{CourseID=3141,Title="Trigonometry",Credits=4},
        new Course{CourseID=2021,Title="Composition",Credits=3},
        new Course{CourseID=2042,Title="Literature",Credits=4}
    };
    courses.ForEach(s => context.Courses.Add(s));
    context.SaveChanges();
    var enrollments = new List<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},
    };
    enrollments.ForEach(s => context.Enrollments.Add(s));
    context.SaveChanges();
    }
}
}

```

The `Seed` method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate `DbSet` property, and then saves the changes to the database. It isn't

necessary to call the `SaveChanges` method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

To tell Entity Framework to use your initializer class, add an element to the `entityFramework` element in the application *Web.config* file (the one in the root project folder), as shown in the following example:

```
<entityFramework>
  <contexts>
    <context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity">
      <databaseInitializer type="ContosoUniversity.DAL.SchoolInitializer,
ContosoUniversity" />
    </context>
  </contexts>
  <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
    <parameters>
      <parameter value="v11.0" />
    </parameters>
  </defaultConnectionFactory>
  <providers>
    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
  </providers>
</entityFramework>
```

The `context type` specifies the fully qualified context class name and the assembly it's in, and the `databaseinitializer type` specifies the fully qualified name of the initializer class and the assembly it's in. (When you don't want EF to use the initializer, you can set an attribute on the `context` element: `disableDatabaseInitialization="true"`.) For more information, see [Entity Framework - Config File Settings](#).

As an alternative to setting the initializer in the *Web.config* file is to do it in code by adding a `Database.SetInitializer` statement to the `Application_Start` method in in the *Global.asax.cs* file. For more information, see [Understanding Database Initializers in Entity Framework Code First](#).

The application is now set up so that when you access the database for the first time in a given run of the application, the Entity Framework compares the database to the model (your `SchoolContext` and entity classes). If there's a difference, the application drops and re-creates the database.

Note: When you deploy an application to a production web server, you must remove or disable code that drops and re-creates the database. You'll do that in a later tutorial in this series.

Set up EF to use a SQL Server Express LocalDB database

[LocalDB](#) is a lightweight version of the SQL Server Express Database Engine. It's easy to install and configure, starts on demand, and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. You can put LocalDB database files in the *App_Data* folder of a web project if you want to be able to copy the database with the project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files. In Visual Studio 2012 and later versions, LocalDB is installed by default with Visual Studio.

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

In this tutorial you'll work with LocalDB. Open the application *Web.config* file and add a `connectionStrings` element preceding the `appSettings` element, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file in the *Views* subfolder that you don't need to update.)

```
<connectionStrings>
  <add name="SchoolContext" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=ContosoUniversity1;Integrated
Security=SSPI;" providerName="System.Data.SqlClient"/>
</connectionStrings>
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

The connection string you've added specifies that Entity Framework will use a LocalDB database named *ContosoUniversity1.mdf*. (The database doesn't exist yet; EF will create it.) If you wanted the database to be created in your *App_Data* folder, you could add

`AttachDBFilename=|DataDirectory|\ContosoUniversity1.mdf` to the connection string.

For more information about connection strings, see [SQL Server Connection Strings for ASP.NET Web Applications](#).

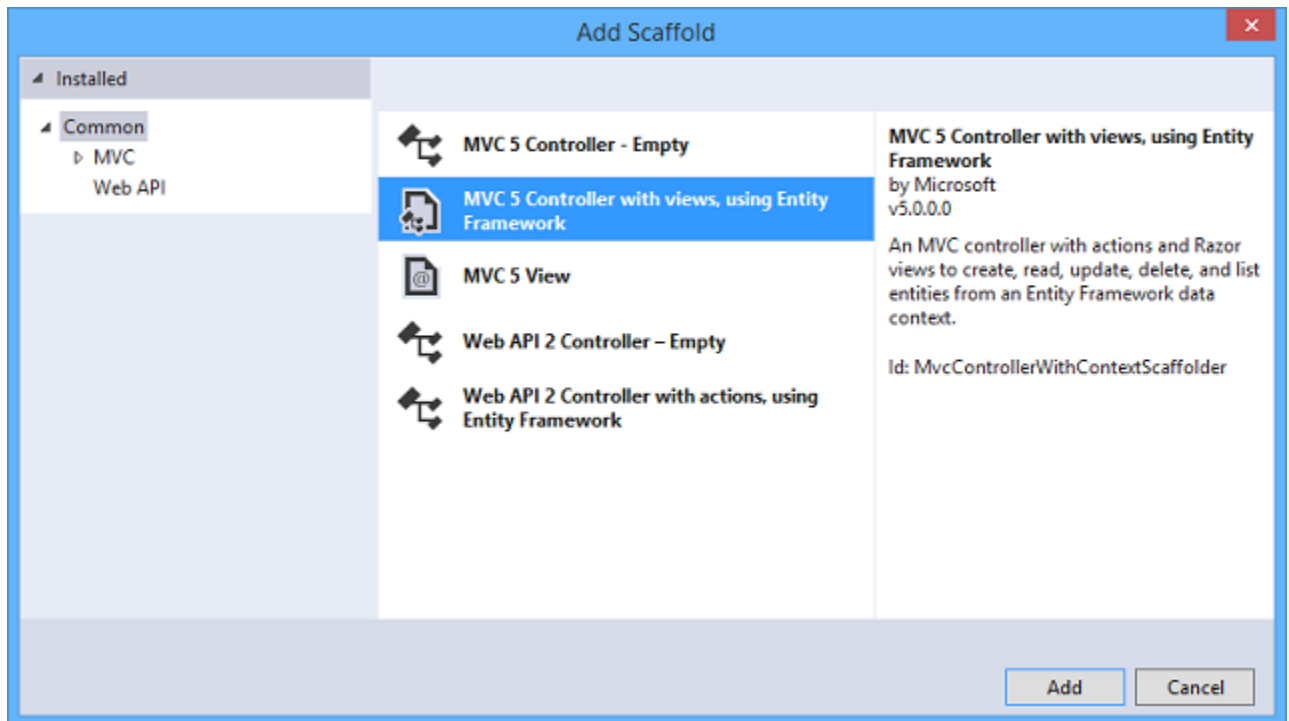
You don't actually have to have a connection string in the *Web.config* file. If you don't supply a connection string, Entity Framework will use a default one based on your context class. For more information, see [Code First to a New Database](#).

Creating a Student Controller and Views

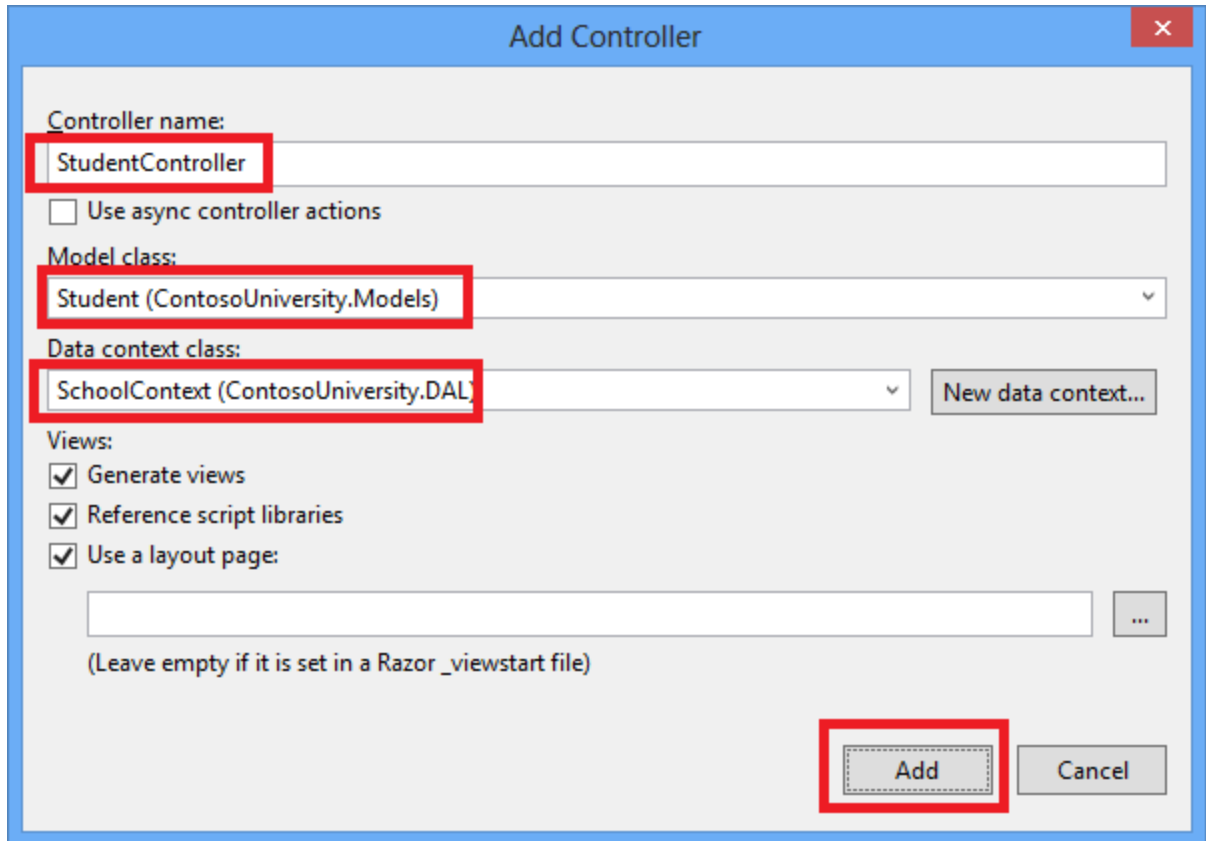
Now you'll create a web page to display data, and the process of requesting the data will automatically trigger

the creation of the database. You'll begin by creating a new controller. But before you do that, build the project to make the model and context classes available to MVC controller scaffolding.

1. Right-click the **Controllers** folder in **Solution Explorer**, select **Add**, and then click **New Scaffolded Item**.
2. In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using Entity Framework**.



3. In the Add Controller dialog box, make the following selections and then click **Add**:
 - o Controller name: **StudentController**.
 - o Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)
 - o Data context class: **SchoolContext (ContosoUniversity.DAL)**.
 - o Leave the default values for the other fields.



When you click **Add**, the scaffolder creates a `StudentController.cs` file and a set of views (.cshtml files) that work with the controller. In the future when you create projects that use Entity Framework you can also take advantage of some additional functionality of the scaffolder: just create your first model class, don't create a connection string, and then in the **Add Controller** box specify new context class. The scaffolder will create your `DbContext` class and your connection string as well as the controller and views.

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

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

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

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

The `Student/Index.cshtml` view displays this list in a table:

```
<table>
  <tr>
```

```

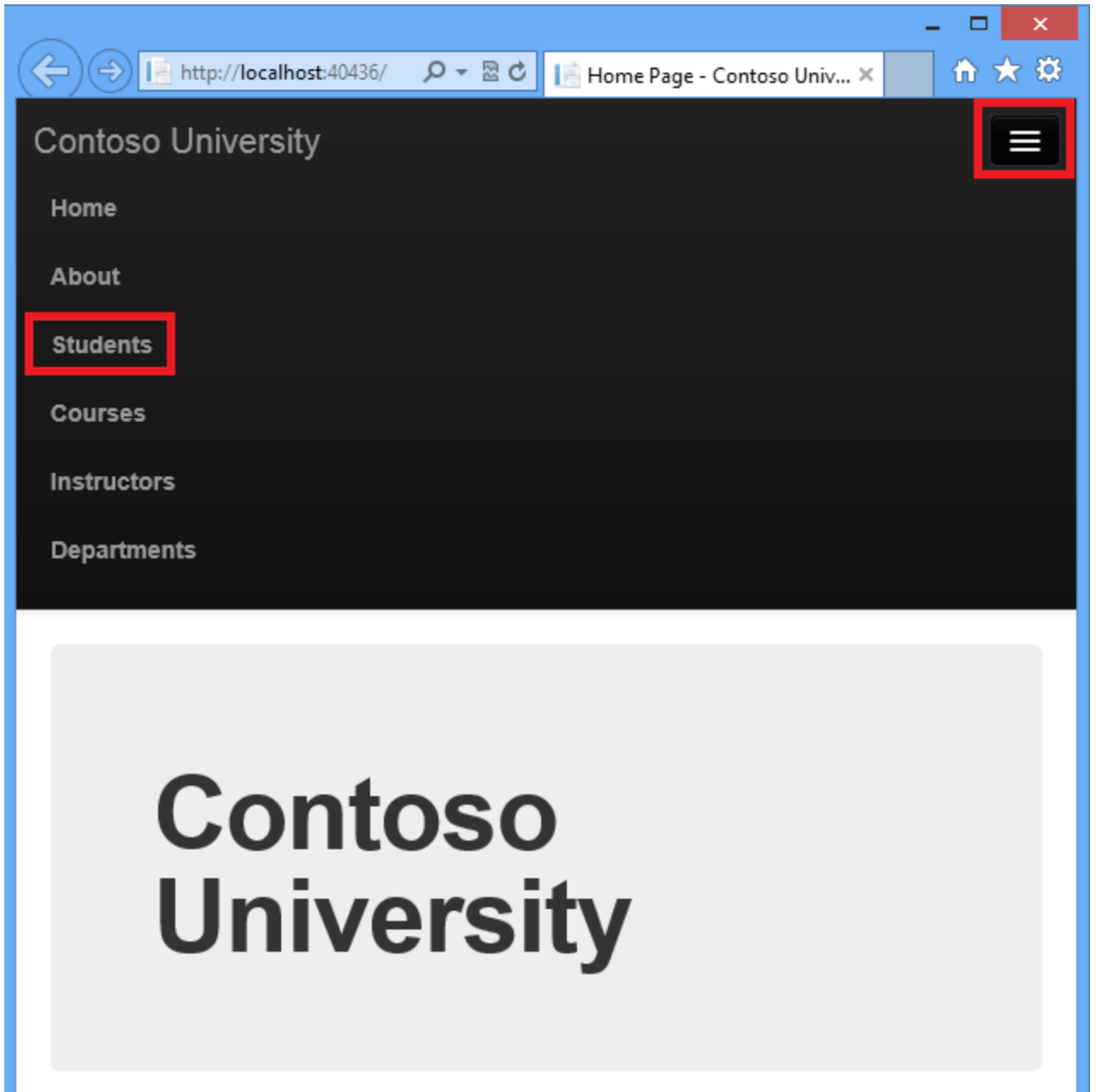
        <th>
            @Html.DisplayNameFor(model => model.LastName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.FirstMidName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </th>
        <th></th>
    </tr>

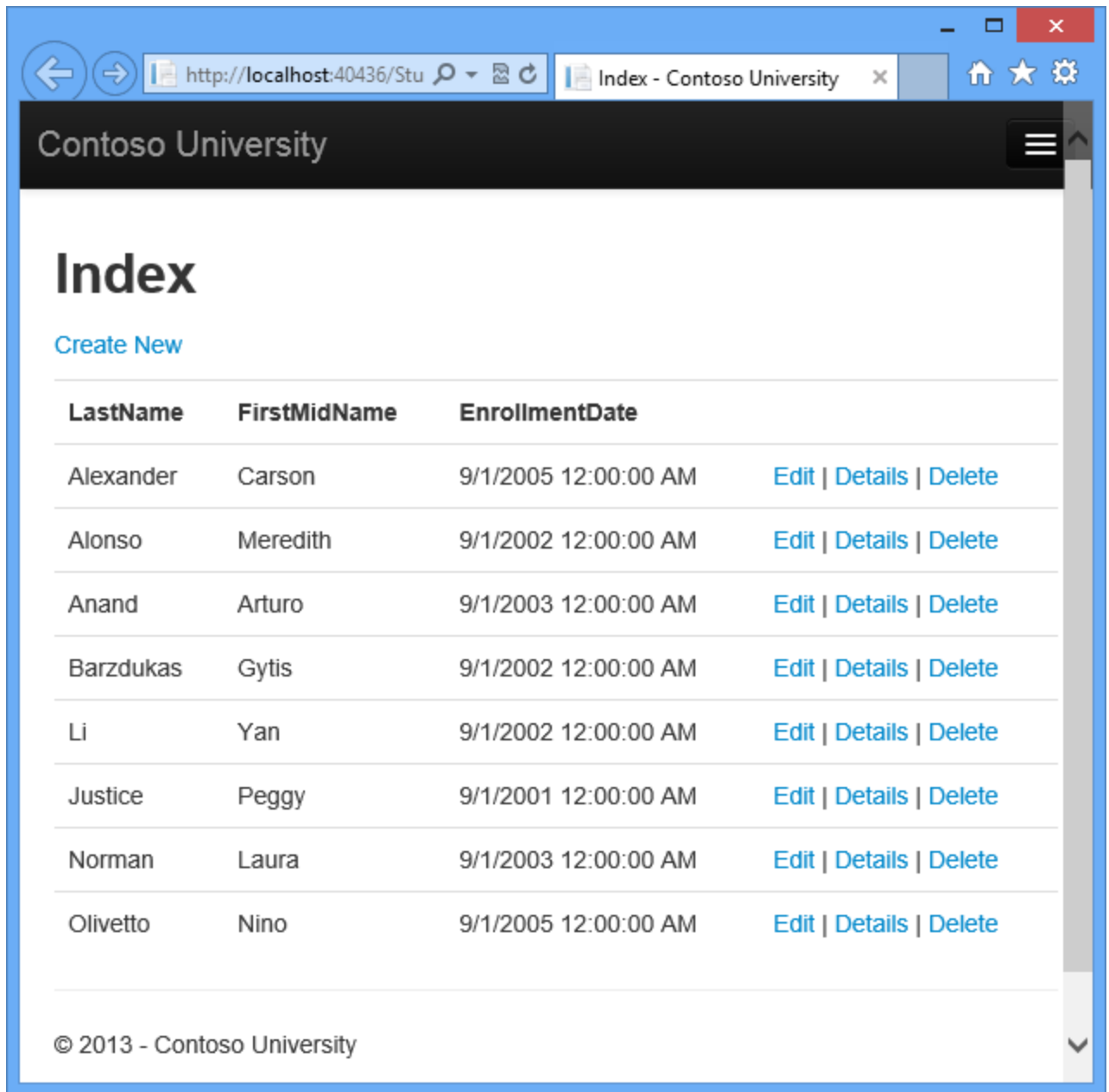
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem =>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

```

5. Press CTRL+F5 to run the project. (If you get a "Cannot create Shadow Copy" error, close the browser and try again.)

Click the **Students** tab to see the test data that the `Seed` method inserted. Depending on how narrow your browser window is, you'll see the Student tab link in the top address bar or you'll have to click the upper right corner to see the link.





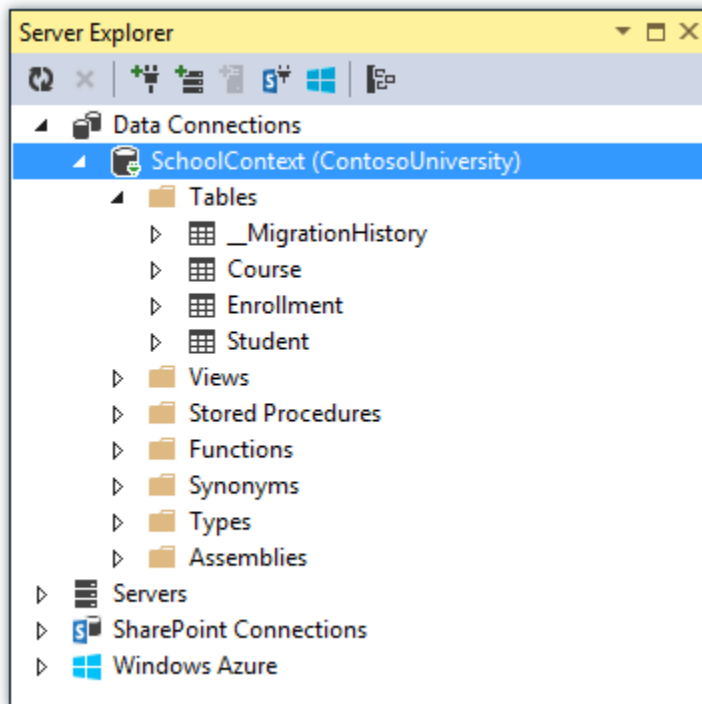
View the Database

When you ran the Students page and the application tried to access the database, EF saw that there was no database and so it created one, then it ran the seed method to populate the database with data.

You can use either **Server Explorer** or **SQL Server Object Explorer (SSOX)** to view the database in Visual Studio. For this tutorial you'll use **Server Explorer**. (In Visual Studio Express editions earlier than 2013, **Server Explorer** is called **Database Explorer**.)

1. Close the browser.

- In **Server Explorer**, expand **Data Connections**, expand **School Context (ContosoUniversity)**, and then expand **Tables** to see the tables in your new database.



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

	StudentID	LastName	FirstMidName	EnrollmentDate
▶	1	Alexander	Carson	9/1/2005 12:00:00 AM
	2	Alonso	Meredith	9/1/2002 12:00:00 AM
	3	Anand	Arturo	9/1/2003 12:00:00 AM
	4	Barzdukas	Gytis	9/1/2002 12:00:00 AM
	5	Li	Yan	9/1/2002 12:00:00 AM
	6	Justice	Peggy	9/1/2001 12:00:00 AM
	7	Norman	Laura	9/1/2003 12:00:00 AM
	8	Olivetto	Nino	9/1/2005 12:00:00 AM
*	NULL	NULL	NULL	NULL

Connection Ready | (localdb)\v11.0 | REDMOND\tdykstra | ContosoUniversity.Mode...

4. Close the **Server Explorer** connection.

The *ContosoUniversity1.mdf* and *.ldf* database files are in the `C:\Users\ folder.`

Because you're using the `DropCreateDatabaseIfModelChanges` initializer, you could now make a change to the `Student` class, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, run the Students page again, and then look at the table again, you will see a new `EmailAddress` column.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of *conventions*, or assumptions that the Entity Framework makes. Some of them have already been noted or were used without your being aware of them:

- The pluralized forms of entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classnameID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named the same simply `<primary key property name>` (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

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

Summary

You've now created a simple application that uses the Entity Framework and SQL Server Express LocalDB to store and display data. In the following tutorial you'll learn how to perform basic CRUD (create, read, update, delete) operations.

Implementing Basic CRUD Functionality with the Entity Framework in ASP.NET MVC Application

In the previous tutorial you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

Note It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about how to implement repositories, see the [ASP.NET Data Access Content Map](#).

In this tutorial, you'll create the following web pages:

Contoso University

Details

Student

LastName
Alexander

FirstMidName
Carson

EnrollmentDate
9/1/2005 12:00:00 AM

Enrollments

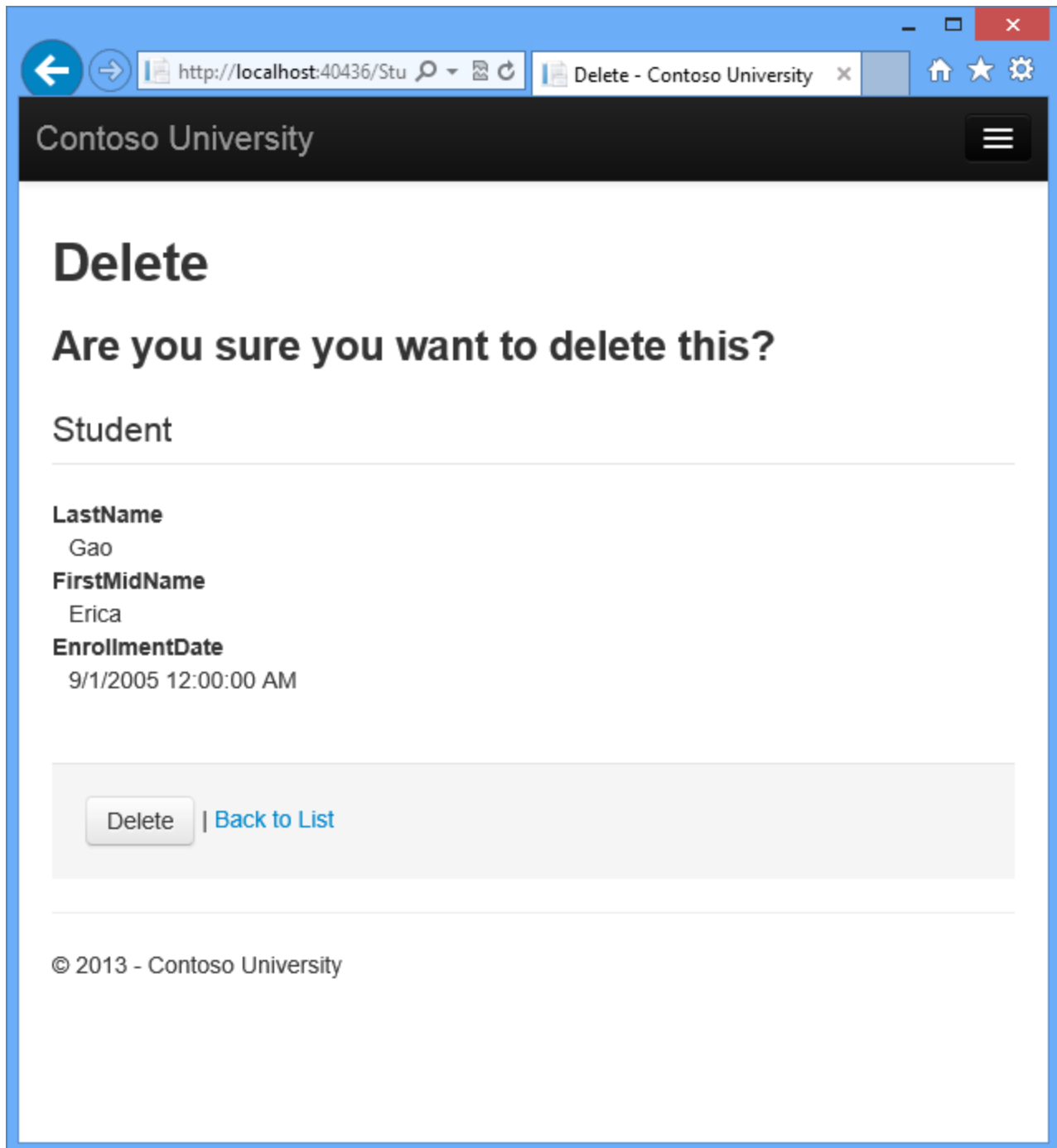
Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

[Edit](#) | [Back to List](#)

© 2013 - Contoso University

The screenshot shows a web browser window with the following elements:

- Browser Address Bar:** `http://localhost:54112/Stu` and `Create - Contoso Univer...`
- Page Header:** **Contoso University** with a hamburger menu icon.
- Section Header:** **Create Student**
- Form Fields:**
 - LastName:** Input field containing `Gao`.
 - FirstMidName:** Input field containing `Erica`.
 - EnrollmentDate:** Input field containing `9/31/2005`. A red border highlights the field, and a red error message is displayed to its right: `The value '9/31/2005' is not valid for EnrollmentDate.`
- Buttons:** A `Create` button and a `Back to List` link.
- Footer:** `© 2013 - Contoso University`



Create a Details Page

The scaffolded code for the `Students Index` page left out the `Enrollments` property, because that property holds a collection. In the `Details` page you'll display the contents of the collection in an HTML table.

In `Controllers\StudentController.cs`, the action method for the `Details` view uses the [Find](#) method to retrieve a single `Student` entity.

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

The key value is passed to the method as the `id` parameter and comes from *route data* in the **Details** hyperlink on the Index page.

Route data

Route data is data that the model binder found in a URL segment specified in the routing table. For example, the default route specifies `controller`, `action`, and `id` segments:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

In the following URL, the default route maps `Instructor` as the `controller`, `Index` as the `action` and `1` as the `id`; these are route data values.

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

"`?courseID=2021`" is a query string value. The model binder will also work if you pass the `id` as a query string value:

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

The URLs are created by `ActionLink` statements in the Razor view. In the following code, the `id` parameter matches the default route, so `id` is added to the route data.

```
@Html.ActionLink("Select", "Index", new { id = item.PersonID })
```

In the following code, `courseID` doesn't match a parameter in the default route, so it's added as a query string.

```
@Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
```

1. Open `Views\Student\Details.cshtml`. Each field is displayed using a `DisplayFor` helper, as shown in the following example:

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

2. After the `EnrollmentDate` field and immediately before the closing `</dl>` tag, add the highlighted code to display a list of enrollments, as shown in the following example:

```
<dt>
    @Html.DisplayNameFor(model => model.EnrollmentDate)
</dt>
<dd>
    @Html.DisplayFor(model => model.EnrollmentDate)
</dd>
<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>
</dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

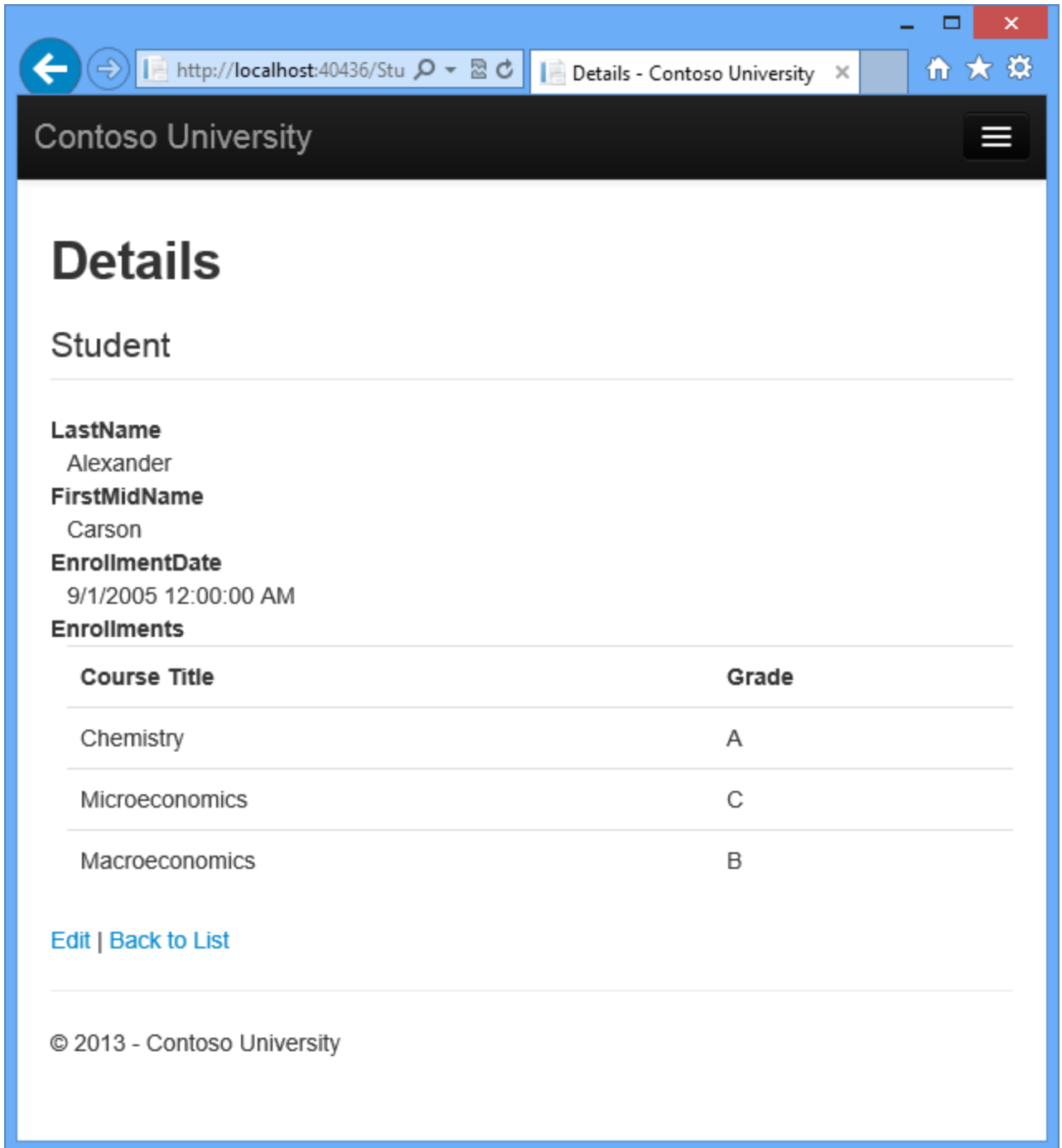
If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each `Enrollment` entity in the property, it displays the course title and the grade. The course

title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the `Enrollments` entity. All of this data is retrieved from the database automatically when it's needed. (In other words, you are using lazy loading here. You did not specify *eager loading* for the `Courses` navigation property, so the enrollments were not retrieved in the same query that got the students. Instead, the first time you try to access the `Enrollments` navigation property, a new query is sent to the database to retrieve the data. You can read more about lazy loading and eager loading in the [Reading Related Data](#) tutorial later in this series.)

3. Run the page by selecting the **Students** tab and clicking a **Details** link for Alexander Carson. (If you press CTRL+F5 while the `Details.cshtml` file is open, you'll get an HTTP 400 error because Visual Studio tries to run the Details page but it wasn't reached from a link that specifies the student to display. In that case, just remove "Student/Details" from the URL and try again, or close the browser, right-click the project, and click **View**, and then click **View in Browser**.)

You see the list of courses and grades for the selected student:



Update the Create Page

1. In *Controllers\StudentController.cs*, replace the `HttpPost Create` action method with the following code to add a `try-catch` block and remove `ID` from the [Bind attribute](#) for the scaffolded method:

```
[HttpPost]
```

```

[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "LastName, FirstMidName,
EnrollmentDate")]Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Students.Add(student);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line
here to write a log.
        ModelState.AddModelError("", "Unable to save changes. Try
again, and if the problem persists see your system administrator.");
    }
    return View(student);
}

```

This code adds the `Student` entity created by the ASP.NET MVC model binder to the `Students` entity set and then saves the changes to the database. (*Model binder* refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a `Student` entity for you using property values from the `Form` collection.)

You removed `ID` from the `Bind` attribute because `ID` is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the `ID` value.

Security Note: The `ValidateAntiForgeryToken` attribute helps prevent [cross-site request forgery](#) attacks. It requires a corresponding `Html.AntiForgeryToken()` statement in the view, which you'll see later.

The `Bind` attribute protects against *over-posting*. For example, suppose the `Student` entity includes a `Secret` property that you don't want this web page to update.

```

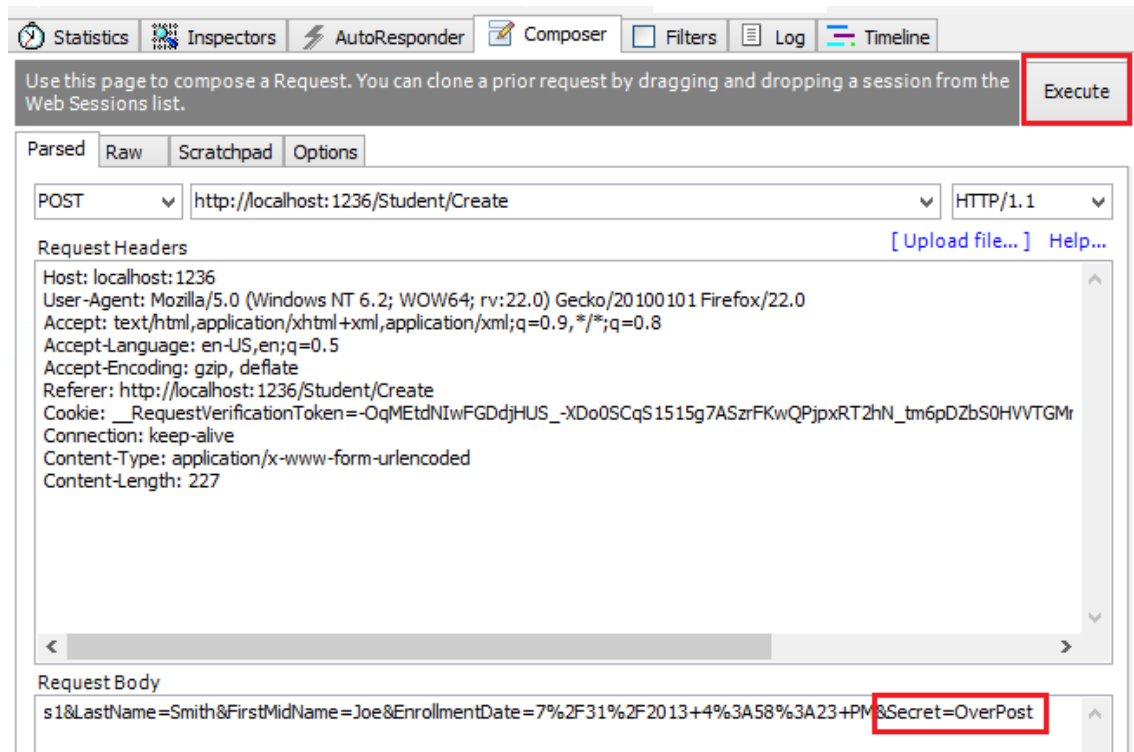
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }

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

```

}

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as [fiddler](#), or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a `Student` instance, the model binder would pick up that `Secret` form value and use it to update the `Student` entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to update that property.

It's a security best practice to use the `Include` parameter with the `Bind` attribute to *whitelist* fields. It's also possible to use the `Exclude` parameter to *blacklist* fields you want to exclude. The reason `Include` is more secure is that when you add a new property to the entity, the new field is not automatically protected by an `Exclude` list.

Another alternative approach, and one preferred by many, is to use only view models with model binding. The view model contains only the properties you want to bind. Once the MVC model binder has finished, you copy the view model properties to the entity instance.

Other than the `Bind` attribute, the `try-catch` block is the only change you've made to the scaffolded code. If an exception that derives from [DataException](#) is caught while the changes are being saved, a generic error message is displayed. [DataException](#) exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in [Monitoring and Telemetry \(Building Real-World Cloud Apps with Windows Azure\)](#).

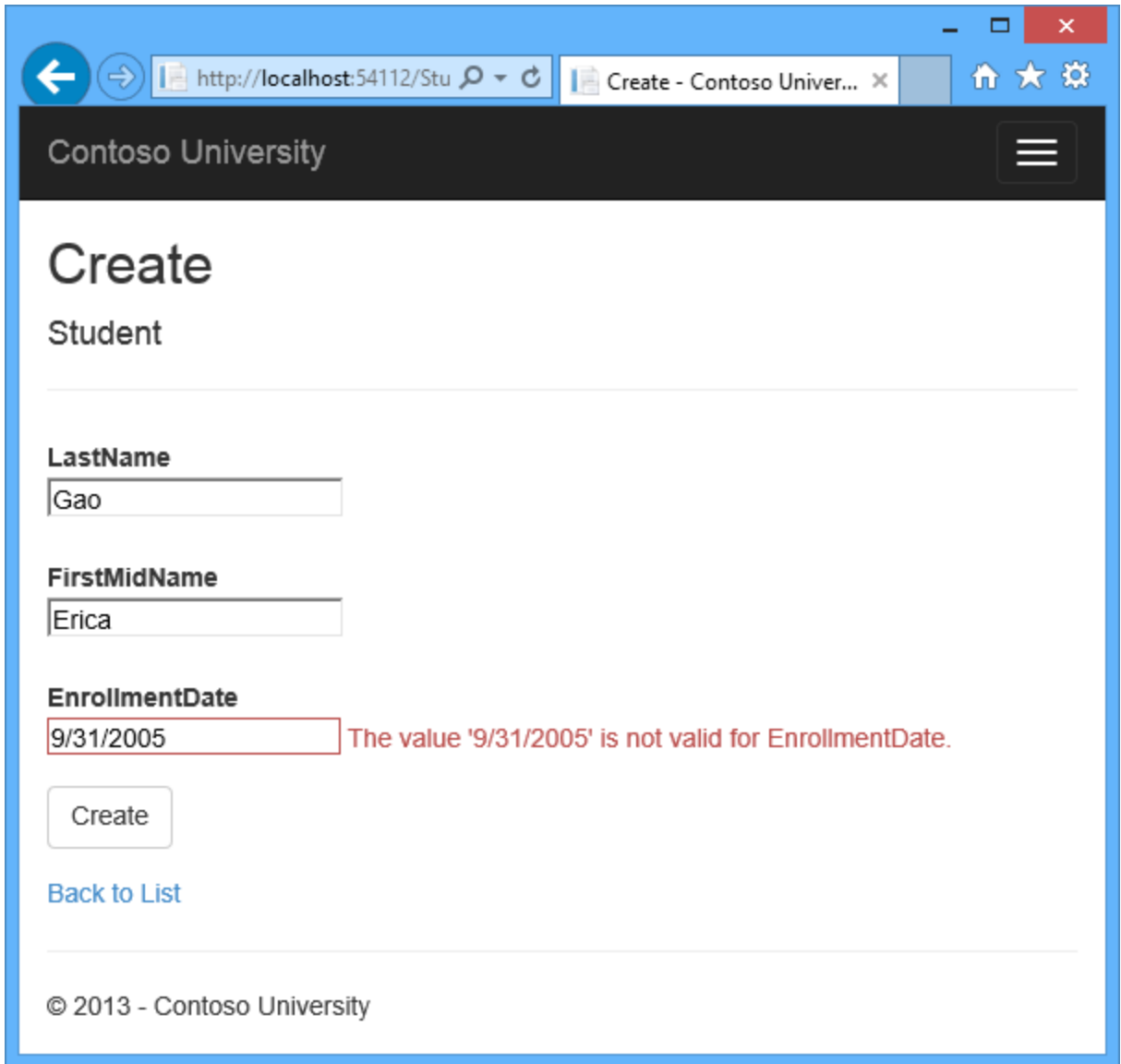
The code in `Views\Student\Create.cshtml` is similar to what you saw in `Details.cshtml`, except that `EditorFor` and `ValidationMessageFor` helpers are used for each field instead of `DisplayFor`. Here is the relevant code:

```
<div class="form-group">
  @Html.LabelFor(model => model.LastName, new { @class = "control-label col-md-2" })
  <div class="col-md-10">
    @Html.EditorFor(model => model.LastName)
    @Html.ValidationMessageFor(model => model.LastName)
  </div>
</div>
```

`Create.cshtml` also includes `@Html.AntiForgeryToken()`, which works with the `ValidateAntiForgeryToken` attribute in the controller to help prevent [cross-site request forgery](#) attacks.

No changes are required in `Create.cshtml`.

2. Run the page by selecting the **Students** tab and clicking **Create New**.
3. Enter names and an invalid date and click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the **Create** method.

```
if (ModelState.IsValid)
{
    db.Students.Add(student);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

4. Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Contoso University

Index

[Create New](#)

LastName	FirstMidName	EnrollmentDate	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit Details Delete
Justice	Peggy	9/1/2001 12:00:00 AM	Edit Details Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit Details Delete
Olivetto	Nino	9/1/2005 12:00:00 AM	Edit Details Delete
Gao	Erica	9/1/2005 12:00:00 AM	Edit Details Delete

© 2013 - Contoso University

Update the Edit HttpPost Page

In *Controllers\StudentController.cs*, the `HttpGet Edit` method (the one without the `HttpPost` attribute) uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` method. You don't need to change this method.

However, replace the `HttpPost Edit` action method with the following code to add a try-catch block:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include = "ID, LastName, FirstMidName, EnrollmentDate")]Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Entry(student).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to write a log.
        ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see your system administrator.");
    }
    return View(student);
}
```

This code is similar to what you saw in the `HttpPost Create` method. However, instead of adding the entity created by the model binder to the entity set, this code sets a flag on the entity indicating it has been changed. When the [SaveChanges](#) method is called, the [Modified](#) flag causes the Entity Framework to create SQL statements to update the database row. All columns of the database row will be updated, including those that the user didn't change, and [concurrency conflicts](#) are ignored.

Entity States and the Attach and SaveChanges Methods

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the [Add](#) method, that entity's state is set to `Added`. Then when you call the [SaveChanges](#) method, the database context issues a SQL `INSERT` command.

An entity may be in one of the [following states](#):

- **Added.** The entity does not yet exist in the database. The `SaveChanges` method must issue an `INSERT` statement.
- **Unchanged.** Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- **Modified.** Some or all of the entity's property values have been modified. The `SaveChanges` method must issue an `UPDATE` statement.

- Deleted. The entity has been marked for deletion. The `SaveChanges` method must issue a `DELETE` statement.
- Detached. The entity isn't being tracked by the database context.

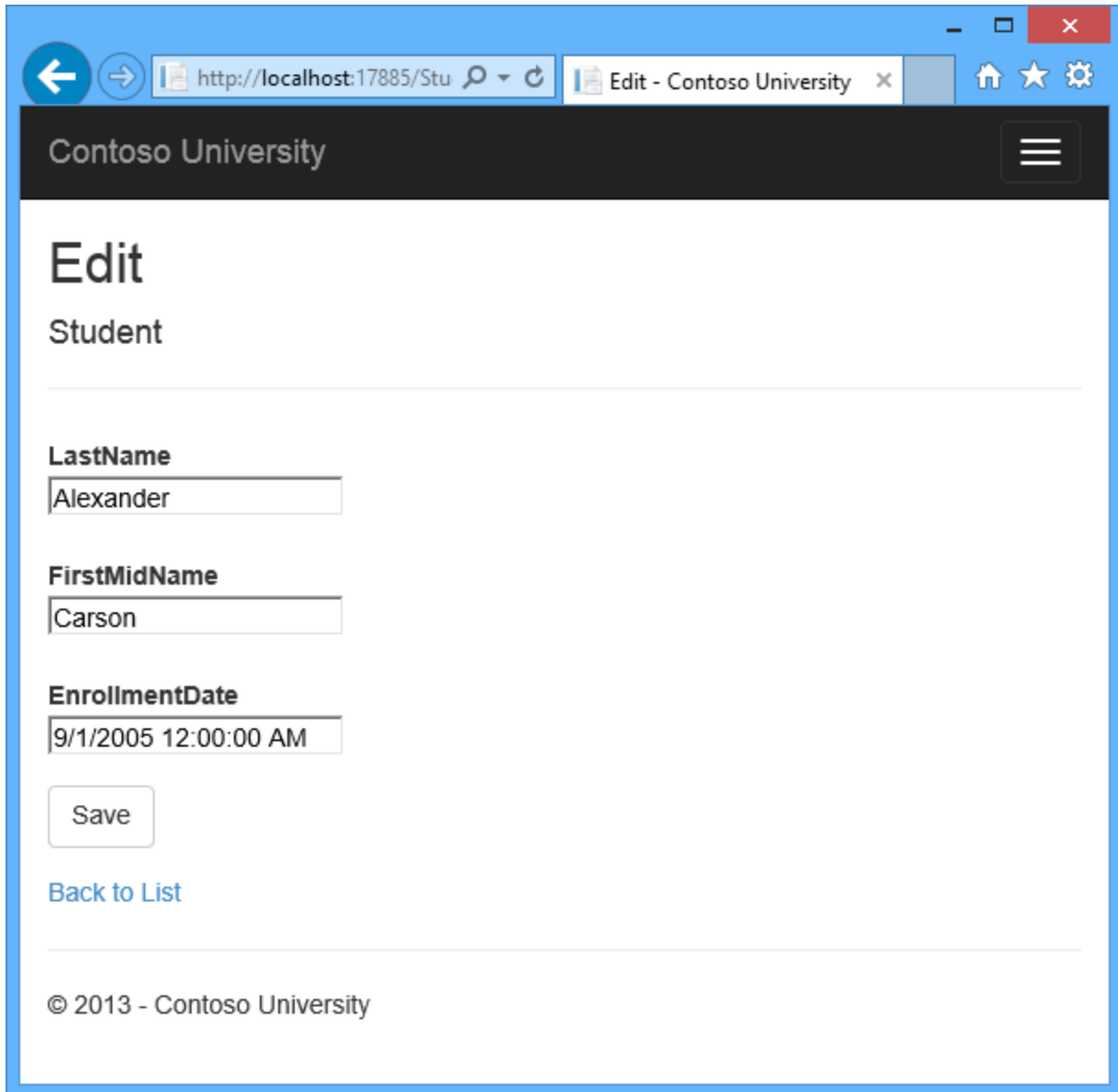
In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL `UPDATE` statement that updates only the actual properties that you changed.

The disconnected nature of web apps doesn't allow for this continuous sequence. The [DbContext](#) that reads an entity is disposed after a page is rendered. When the `HttpPost Edit` action method is called, a new request is made and you have a new instance of the [DbContext](#), so you have to manually set the entity state to `Modified`. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want the SQL `Update` statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the `HttpPost Edit` method is called. Then you can create a `Student` entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`. For more information, see [Entity states and SaveChanges](#) and [Local Data](#) in the MSDN Data Developer Center.

The HTML and Razor code in `Views\Student\Edit.cshtml` is similar to what you saw in `Create.cshtml`, and no changes are required.

Run the page by selecting the **Students** tab and then clicking an **Edit** hyperlink.



Change some of the data and click **Save**. You see the changed data in the Index page.

Contoso University

Index

[Create New](#)

LastName	FirstMidName	EnrollmentDate	
Alexander	Carson	9/1/2011 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit Details Delete
Justice	Peggy	9/1/2001 12:00:00 AM	Edit Details Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit Details Delete
Olivetto	Nino	9/1/2005 12:00:00 AM	Edit Details Delete
Gao	Erica	9/1/2005 12:00:00 AM	Edit Details Delete

© 2013 - Contoso University

Updating the Delete Page

In *Controllers\StudentController.cs*, the template code for the `HttpGet Delete` method uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` and `Edit` methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost Delete` method is called and then that method actually performs the delete operation.

You'll add a `try-catch` block to the `HttpPost Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the `HttpPost Delete` method calls the `HttpGet Delete` method, passing it a parameter that indicates that an error has occurred. The `HttpGet Delete` method then redisplay the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

1. Replace the `HttpGet Delete` action method with the following code, which manages error reporting:

```
public ActionResult Delete(int? id, bool? saveChangesError=false)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    if (saveChangesError.GetValueOrDefault())
    {
        ViewBag.ErrorMessage = "Delete failed. Try again, and if the
problem persists see your system administrator.";
    }
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

This code accepts an [optional parameter](#) that indicates whether the method was called after a failure to save changes. This parameter is `false` when the `HttpGet Delete` method is called without a previous failure. When it is called by the `HttpPost Delete` method in response to a database update error, the parameter is `true` and an error message is passed to the view.

2. Replace the `HttpPost Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id)
{
    try
    {
        Student student = db.Students.Find(id);
```

```

        db.Students.Remove(student);
        db.SaveChanges();
    }
    catch (DataException/* dex */)
    {
        //Log the error (uncomment dex variable name and add a line
        here to write a log.
        return RedirectToAction("Delete", new { id = id,
        saveChangesError = true });
    }
    return RedirectToAction("Index");
}

```

This code retrieves the selected entity, then calls the [Remove](#) method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet delete` methods.

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query to retrieve the row by replacing the lines of code that call the `Find` and `Remove` methods with the following code:

```

Student studentToDelete = new Student() { ID = id };
db.Entry(studentToDelete).State = EntityState.Deleted;

```

This code instantiates a `Student` entity using only the primary key value and then sets the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity.

As noted, the `HttpGet Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing any edit operation, create operation, or any other operation that changes data) creates a security risk. For more information, see [ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes](#) on Stephen Walther's blog.

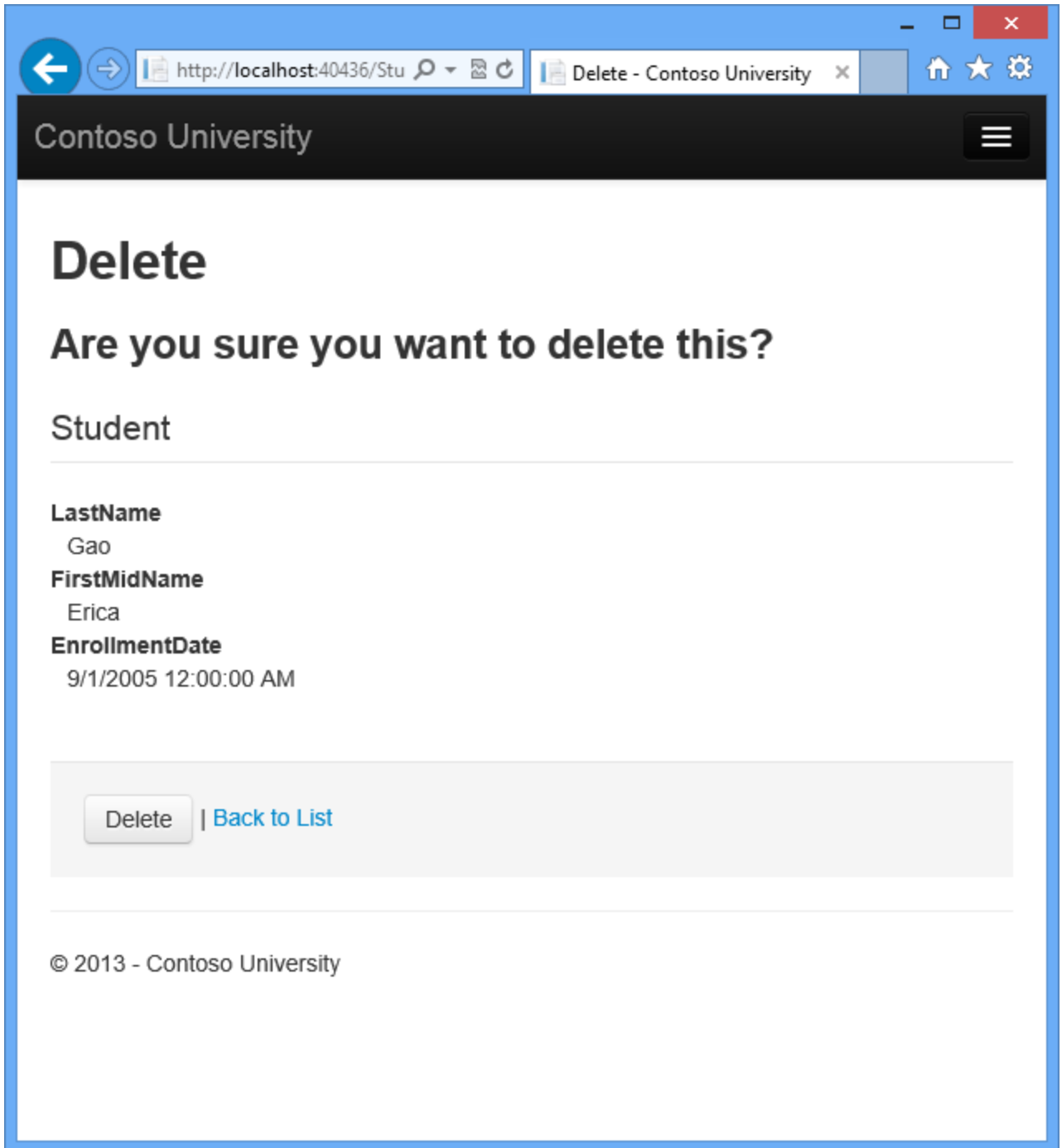
3. In `Views\Student\Delete.cshtml`, add an error message between the `h2` heading and the `h3` heading, as shown in the following example:

```

<h2>Delete</h2>
<p class="error">@ViewBag.ErrorMessage</p>
<h3>Are you sure you want to delete this?</h3>

```

Run the page by selecting the **Students** tab and clicking a **Delete** hyperlink:



4. Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the [concurrency tutorial](#).)

Ensuring that Database Connections Are Not Left Open

To make sure that database connections are properly closed and the resources they hold freed up, you have to dispose the context instance when you are done with it. That is why the scaffolded

code provides a [Dispose](#) method at the end of the `StudentController` class in *StudentController.cs*, as shown in the following example:

```
protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}
```

The base `Controller` class already implements the `IDisposable` interface, so this code simply adds an override to the `Dispose(bool)` method to explicitly dispose the context instance.

Handling Transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Working with Transactions](#) on MSDN.

Summary

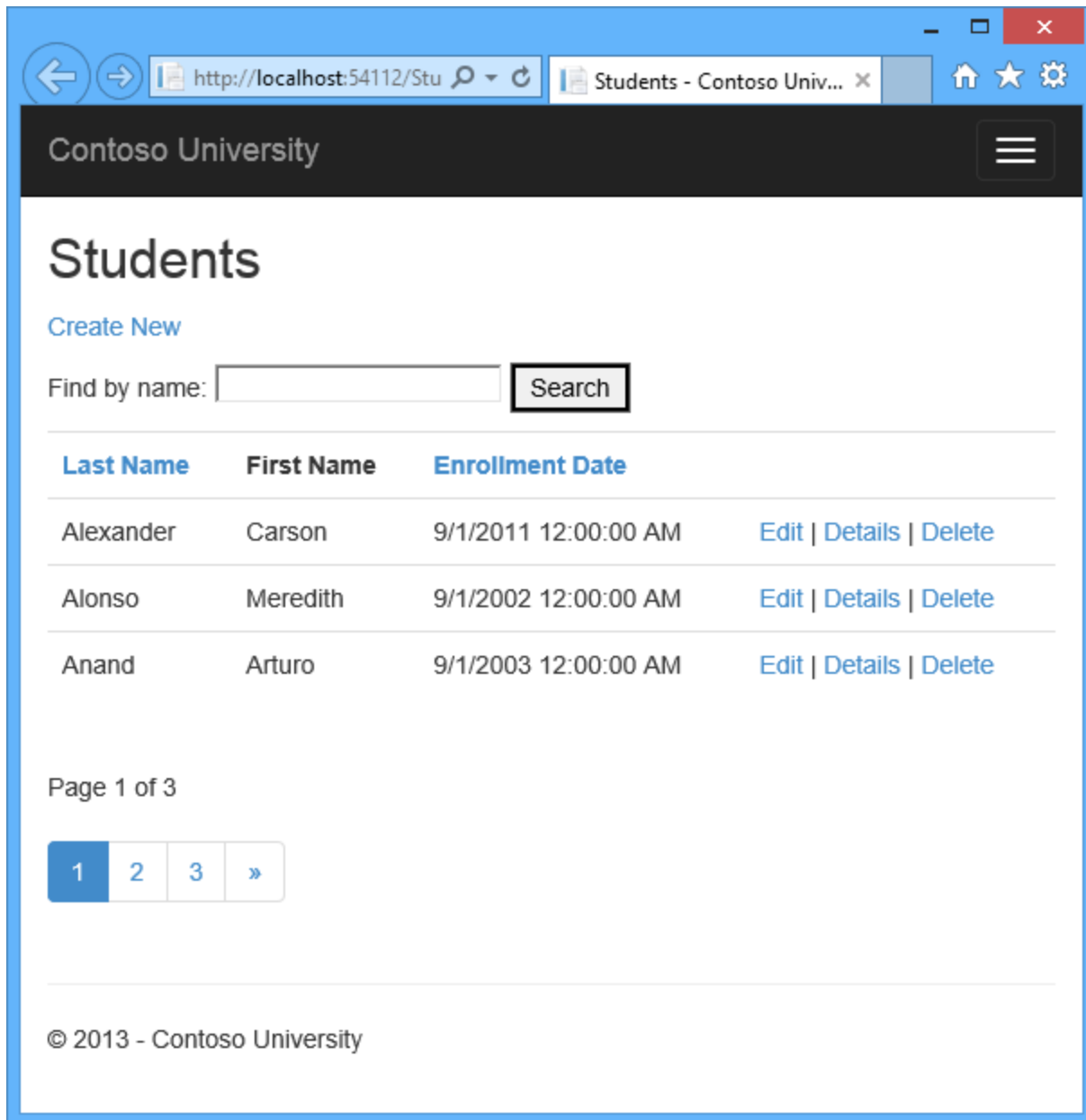
You now have a complete set of pages that perform simple CRUD operations for `Student` entities. You used MVC helpers to generate UI elements for data fields. For more information about MVC helpers, see [Rendering a Form Using HTML Helpers](#) (the page is for MVC 3 but is still relevant for MVC 5).

In the next tutorial you'll expand the functionality of the Index page by adding sorting and paging.

Sorting, Filtering, and Paging with the Entity Framework in an ASP.NET MVC Application

In the previous tutorial you implemented a set of web pages for basic CRUD operations for `Student` entities. In this tutorial you'll add sorting, filtering, and paging functionality to the **Students** Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



Add Column Sort Links to the Students Index Page

To add sorting to the Student Index page, you'll change the `Index` method of the `Student` controller and add code to the `Student Index` view.

Add Sorting Functionality to the Index Method

In `Controllers\StudentController.cs`, replace the `Index` method with the following code:

```
public ActionResult Index(string sortOrder)
{
```

```

ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
var students = from s in db.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(students.ToList());
}

```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by `LastName`, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewBag` variables are used so that the view can configure the column heading hyperlinks with the appropriate query string values:

```

ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";

```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, `ViewBag.NameSortParm` should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses [LINQ to Entities](#) to specify the column to sort by. The code creates an [IQueryable](#) variable before the `switch` statement, modifies it in the `switch` statement, and calls the `ToList` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToList`. Therefore, this code results in a single query that is not executed until the `return View` statement.

As an alternative to writing different LINQ statements for each sort order, you can dynamically create a LINQ statement. For information about dynamic LINQ, see [Dynamic LINQ](#).

Add Column Heading Hyperlinks to the Student Index View

In `Views\Student\Index.cshtml`, replace the `<tr>` and `<th>` elements for the heading row with the highlighted code:

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder =
ViewBag.NameSortParm })
        </th>
        <th>First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder =
ViewBag.DateSortParm })
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
```

This code uses the information in the `ViewBag` properties to set up hyperlinks with the appropriate query string values.

Run the page and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.

Contoso University

Index

[Create New](#)

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2011 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete
Justice	Peggy	9/1/2001 12:00:00 AM	Edit Details Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit Details Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit Details Delete
Olivetto	Nino	9/1/2005 12:00:00 AM	Edit Details Delete

© 2013 - Contoso University

After you click the **Last Name** heading, students are displayed in descending last name order.

Contoso University

Index

[Create New](#)

Last Name	First Name	Enrollment Date	
Olivetto	Nino	9/1/2005 12:00:00 AM	Edit Details Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit Details Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit Details Delete
Justice	Peggy	9/1/2001 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Alexander	Carson	9/1/2011 12:00:00 AM	Edit Details Delete

© 2013 - Contoso University

Add a Search Box to the Students Index Page

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add Filtering Functionality to the Index Method

In *Controllers\StudentController.cs*, replace the `Index` method with the following code (the changes are highlighted):

```
public ActionResult Index(string sortOrder, string searchString)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" :
    "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s =>
s.LastName.ToUpper().Contains(searchString.ToUpper())
||
s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
    }
    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(students.ToList());
}
```

You've added a `searchString` parameter to the `Index` method. You've also added to the LINQ statement a `where` clause that selects only students whose first name or last name contains the search string. The search string value is received from a text box that you'll add to the `Index` view. The statement that adds the [where](#) clause is executed only if there's a value to search for.

Note In many cases you can call the same method either on an Entity Framework entity set or as an extension method on an in-memory collection. The results are normally the same but in some cases may be different.

For example, the .NET Framework implementation of the `Contains` method returns all rows when you pass an empty string to it, but the Entity Framework provider for SQL Server Compact 4.0 returns zero rows for empty strings. Therefore the code in the example (putting the `where` statement inside an `if` statement) makes sure that you get the same results for all versions of SQL Server. Also, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but Entity Framework SQL Server providers perform case-insensitive comparisons by default. Therefore, calling the `ToUpper` method to make the test

explicitly case-insensitive ensures that results do not change when you change the code later to use a repository, which will return an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.)

Null handling may also be different for different database providers or when you use an `IQueryable` object compared to when you use an `IEnumerable` collection. For example, in some scenarios a `Where` condition such as `table.Column != 0` may not return columns that have null as the value. For more information, see [Incorrect handling of null variables in 'where' clause](#).

Add a Search Box to the Student Index View

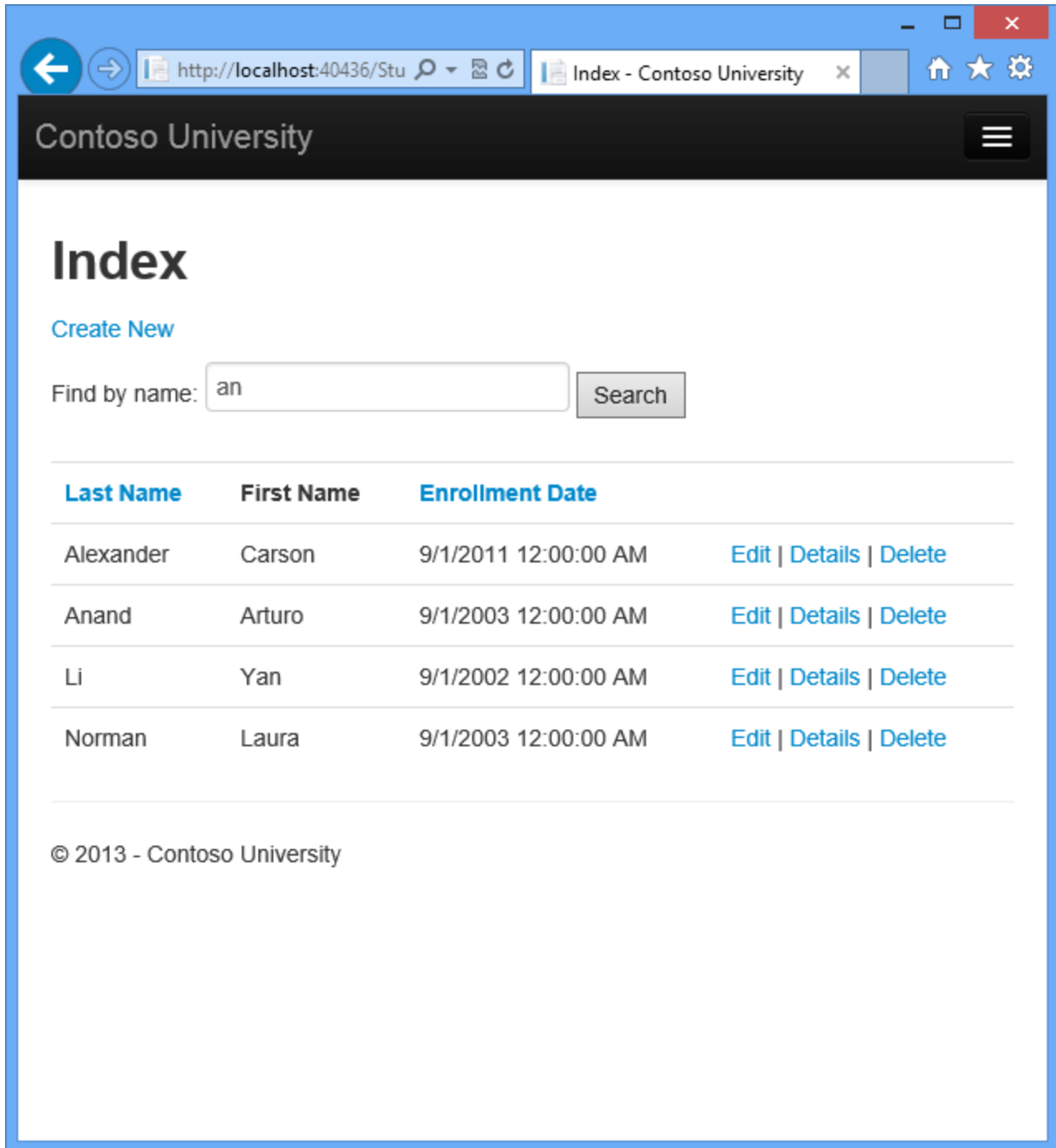
In `Views\Student\Index.cshtml`, add the highlighted code immediately before the opening `table` tag in order to create a caption, a text box, and a **Search** button.

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>

@using (Html.BeginForm())
{
    <p>
        Find by name: @Html.TextBox("SearchString")
        <input type="submit" value="Search" /></p>
}

<table>
    <tr>
```

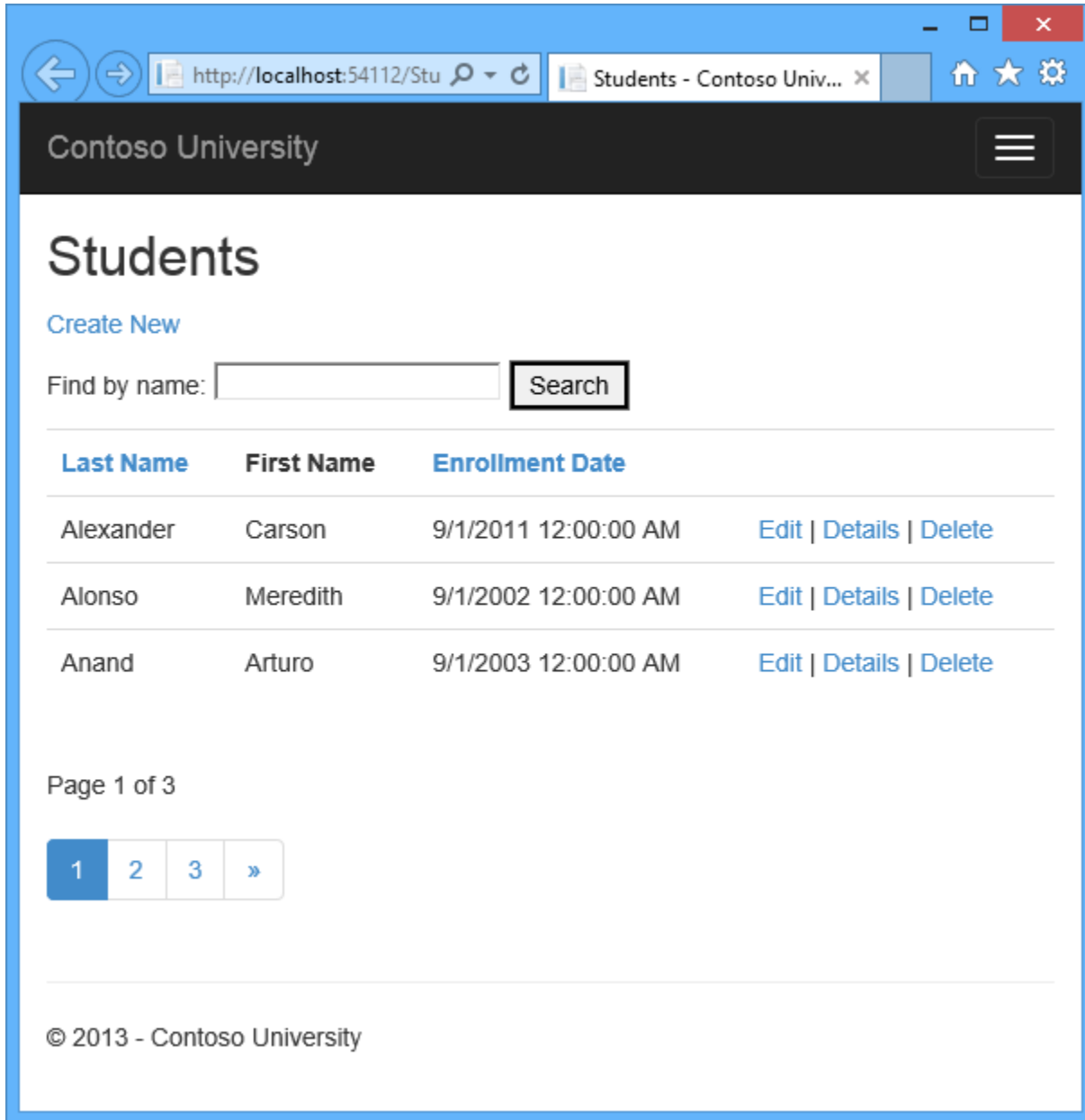
Run the page, enter a search string, and click **Search** to verify that filtering is working.



Notice the URL doesn't contain the "an" search string, which means that if you bookmark this page, you won't get the filtered list when you use the bookmark. You'll change the **Search** button to use query strings for filter criteria later in the tutorial.

Add Paging to the Students Index Page

To add paging to the Students Index page, you'll start by installing the **PagedList.Mvc** NuGet package. Then you'll make additional changes in the `Index` method and add paging links to the `Index` view. **PagedList.Mvc** is one of many good paging and sorting packages for ASP.NET MVC, and its use here is intended only as an example, not as a recommendation for it over other options. The following illustration shows the paging links.



Install the PagedList.MVC NuGet Package

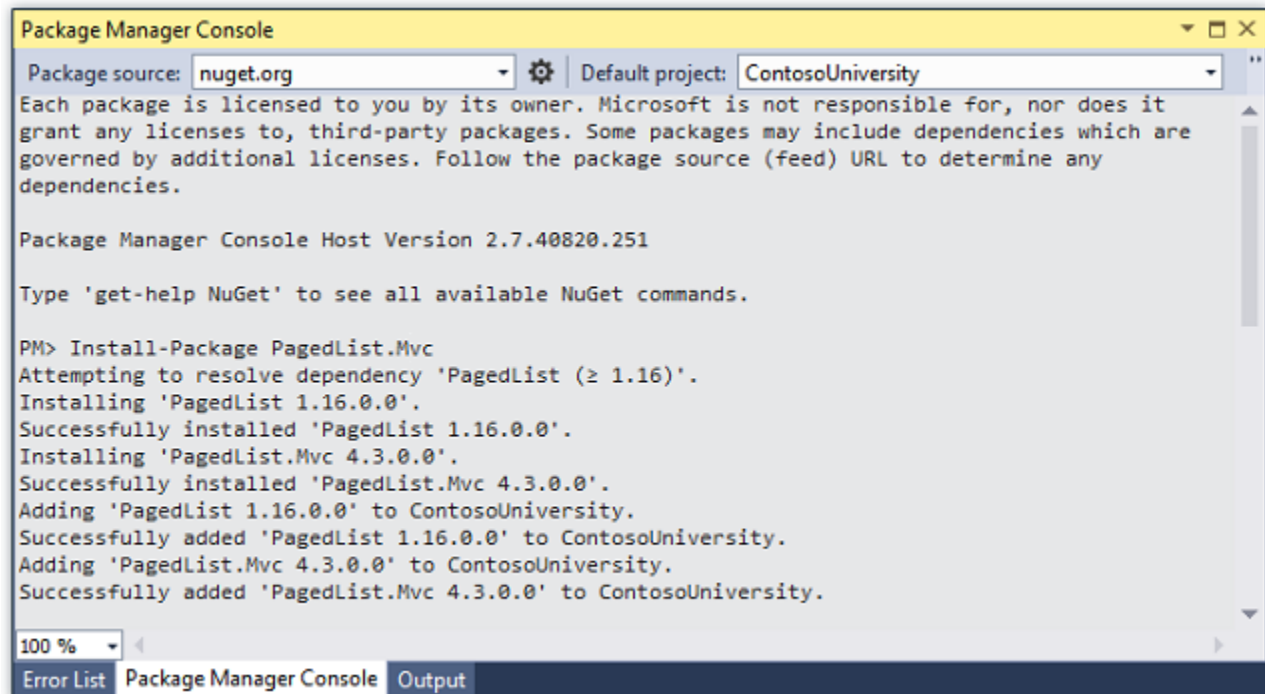
The NuGet **PagedList.Mvc** package automatically installs the **PagedList** package as a dependency. The **PagedList** package installs a `PagedList` collection type and extension methods for `IQueryable` and `IEnumerable` collections. The extension methods create a single page of

data in a `PagedList` collection out of your `IQueryable` or `IEnumerable`, and the `PagedList` collection provides several properties and methods that facilitate paging. The `PagedList.Mvc` package installs a paging helper that displays the paging buttons.

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

In the **Package Manager Console** window, make sure the **Package source** is **nuget.org** and the **Default project** is **ContosoUniversity**, and then enter the following command:

```
Install-Package PagedList.Mvc
```



Add Paging Functionality to the Index Method

In `Controllers\StudentController.cs`, add a `using` statement for the `PagedList` namespace:

```
using PagedList;
```

Replace the `Index` method with the following code:

```
public ActionResult Index(string sortOrder, string currentFilter, string
searchString, int? page)
{
    ViewBag.CurrentSort = sortOrder;
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
```

```

        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewBag.CurrentFilter = searchString;

    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s =>
s.LastName.ToUpper().Contains(searchString.ToUpper())
||
s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
    }
    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: // Name ascending
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    int pageNumber = (page ?? 1);
    return View(students.ToPagedList(pageNumber, pageSize));
}

```

This code adds a `page` parameter, a current sort order parameter, and a current filter parameter to the method signature:

```
public ActionResult Index(string sortOrder, string currentFilter, string
searchString, int? page)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the `page` variable will contain the page number to display.

A `ViewBag` property provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging:

```
ViewBag.CurrentSort = sortOrder;
```

Another property, `ViewBag.CurrentFilter`, provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed. If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the submit button is pressed. In that case, the `searchString` parameter is not null.

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

At the end of the method, the `ToPagedList` extension method on the students `IQueryable` object converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view:

```
int pageSize = 3;
int pageNumber = (page ?? 1);
return View(students.ToPagedList(pageNumber, pageSize));
```

The `ToPagedList` method takes a page number. The two question marks represent the [null-coalescing operator](#). The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add Paging Links to the Student Index View

In `Views\Student\Index.cshtml`, replace the existing code with the following code. The changes are highlighted.

```
@model PagedList.IPagedList<ContosoUniversity.Models.Student>
@using PagedList.Mvc;
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />

@{
    ViewBag.Title = "Students";
}

<h2>Students</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
```

```

        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as
string)
        <input type="submit" value="Search" />
    </p>
}
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder =
ViewBag.NameSortParm, currentFilter=ViewBag.CurrentFilter })
        </th>
        <th>
            First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder =
ViewBag.DateSortParm, currentFilter=ViewBag.CurrentFilter })
        </th>
        <th></th>
    </tr>

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

</table>
<br />
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of
@Model.PageCount

@Html.PagedListPager(Model, page => Url.Action("Index",
    new { page, sortOrder = ViewBag.CurrentSort, currentFilter =
ViewBag.CurrentFilter }))

```

The `@model` statement at the top of the page specifies that the view now gets a `PagedList` object instead of a `List` object.

The `using` statement for `PagedList.Mvc` gives access to the MVC helper for the paging buttons.

The code uses an overload of [BeginForm](#) that allows it to specify [FormMethod.Get](#).

```
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as
string)
        <input type="submit" value="Search" />
    </p>
}
```

The default [BeginForm](#) submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The [W3C guidelines for the use of HTTP GET](#) recommend that you should use GET when the action does not result in an update.

The text box is initialized with the current search string so when you click a new page you can see the current search string.

```
Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
```

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
@Html.ActionLink("Last Name", "Index", new { sortOrder=ViewBag.NameSortParm,
currentFilter=ViewBag.CurrentFilter })
```

The current page and total number of pages are displayed.

```
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of
@Model.PageCount
```

If there are no pages to display, "Page 0 of 0" is shown. (In that case the page number is greater than the page count because `Model.PageNumber` is 1, and `Model.PageCount` is 0.)

The paging buttons are displayed by the `PagedListPager` helper:

```
@Html.PagedListPager( Model, page => Url.Action("Index", new { page }) )
```

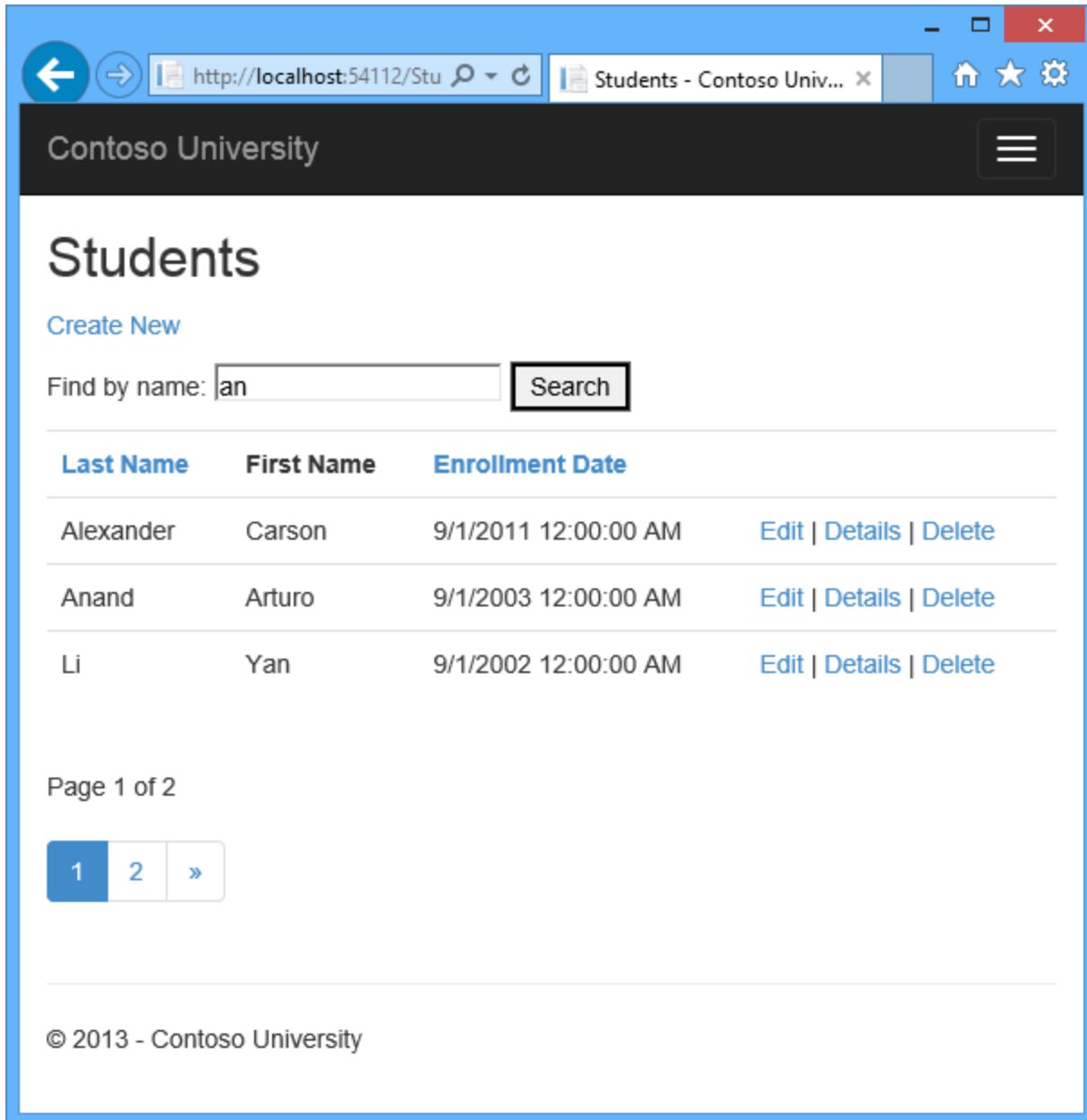
The `PagedListPager` helper provides a number of options that you can customize, including URLs and styling. For more information, see [TroyGoode / PagedList](#) on the GitHub site.

Run the page.

The screenshot shows a web browser window with the URL `http://localhost:54112/Stu`. The page title is "Students - Contoso Univ...". The page content includes a header for "Contoso University", a "Students" title, a "Create New" link, a search bar with the text "Find by name:" and a "Search" button, and a table of student records. The table has columns for "Last Name", "First Name", and "Enrollment Date". Each row includes links for "Edit", "Details", and "Delete". Below the table is a pagination control showing "Page 1 of 3" and buttons for "1", "2", "3", and a right arrow. The footer contains the text "© 2013 - Contoso University".

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2011 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.



Create an About Page That Shows Student Statistics

For the Contoso University website's About page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Modify the `About` method in the `Home` controller.
- Modify the `About` view.

Create the View Model

Create a *ViewModels* folder in the project folder. In that folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

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

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

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

Modify the Home Controller

In *HomeController.cs*, add the following `using` statements at the top of the file:

```
using ContosoUniversity.DAL;
using ContosoUniversity.ViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class:

```
public class HomeController : Controller
{
    private SchoolContext db = new SchoolContext();
```

Replace the `About` method with the following code:

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

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Add a `Dispose` method:

```
protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}
```

Modify the About View

Replace the code in the *Views\Home>About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.ViewModels.EnrollmentDateGroup>

@{
    ViewBag.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>
```

Run the app and click the **About** link. The count of students for each enrollment date is displayed in a table.

The screenshot shows a web browser window with the following content:

- Address bar: `http://localhost:40436/Ho`
- Page title: `Student Body Statistics - Co...`
- Header: `Contoso University`
- Main heading: `Student Body Statistics`
- Table:

Enrollment Date	Students
9/1/2001	1
9/1/2002	3
9/1/2003	2
9/1/2005	1
9/1/2011	1

© 2013 - Contoso University

Summary

In this tutorial you've seen how to create a data model and implement basic CRUD, sorting, filtering, paging, and grouping functionality. In the next tutorial you'll begin looking at more advanced topics by expanding the data model.

Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application

So far the application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider, and you have to deploy the database to a database server.

In this tutorial you'll learn how to use two features of Entity Framework 6 that are especially valuable when you are deploying to the cloud environment: connection resiliency (automatic retries for transient errors) and command interception (catch all SQL queries sent to the database in order to log or change them).

This connection resiliency and command interception tutorial is optional. If you skip this tutorial, a few minor adjustments will have to be made in subsequent tutorials.

Enable connection resiliency

When you deploy the application to Windows Azure, you'll deploy the database to Windows Azure SQL Database, a cloud database service. Transient connection errors are typically more frequent when you connect to a cloud database service than when your web server and your database server are directly connected together in the same data center. Even if a cloud web server and a cloud database service are hosted in the same data center, there are more network connections between them that can have problems, such as load balancers.

Also a cloud service is typically shared by other users, which means its responsiveness can be affected by them. And your access to the database might be subject to throttling. Throttling means the database service throws exceptions when you try to access it more frequently than is allowed in your Service Level Agreement (SLA).

Many or most connection problems when you're accessing a cloud service are transient, that is, they resolve themselves in a short period of time. So when you try a database operation and get a type of error that is typically transient, you could try the operation again after a short wait, and the operation might be successful. You can provide a much better experience for your users if you handle transient errors by automatically trying again, making most of them invisible to the customer. The connection resiliency feature in Entity Framework 6 automates that process of retrying failed SQL queries.

The connection resiliency feature must be configured appropriately for a particular database service:

- It has to know which exceptions are likely to be transient. You want to retry errors caused by a temporary loss in network connectivity, not errors caused by program bugs, for example.
- It has to wait an appropriate amount of time between retries of a failed operation. You can wait longer between retries for a batch process than you can for an online web page where a user is waiting for a response.
- It has to retry an appropriate number of times before it gives up. You might want to retry more times in a batch process than you would in an online application.

You can configure these settings manually for any database environment supported by an Entity Framework provider, but default values that typically work well for an online application that uses Windows Azure SQL Database have already been configured for you, and those are the settings you'll implement for the Contoso University application.

All you have to do to enable connection resiliency is create a class in your assembly that derives from the [DbConfiguration](#) class, and in that class set the SQL Database *execution strategy*, which in EF is another term for *retry policy*.

1. In the DAL folder, add a class file named *SchoolConfiguration.cs*.
2. Replace the template code with the following code:

```
using System.Data.Entity;
using System.Data.Entity.SqlServer;

namespace ContosoUniversity.DAL
{
    public class SchoolConfiguration : DbConfiguration
    {
        public SchoolConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new
                SqlAzureExecutionStrategy());
        }
    }
}
```

The Entity Framework automatically runs the code it finds in a class that derives from `DbConfiguration`. You can use the `DbConfiguration` class to do configuration tasks in code that you would otherwise do in the *Web.config* file. For more information, see [EntityFramework Code-Based Configuration](#).

3. In *StudentController.cs*, add a `using` statement for `System.Data.Entity.Infrastructure`.

```
using System.Data.Entity.Infrastructure;
```

4. Change all of the `catch` blocks that catch `DataException` exceptions so that they catch `RetryLimitExceededException` exceptions instead. For example:

```

catch (RetryLimitExceededException /* dex */)
{
    //Log the error (uncomment dex variable name and add a line here to
write a log.
    ModelState.AddModelError("", "Unable to save changes. Try again,
and if the problem persists see your system administrator.");
}

```

You were using `DataException` to try to identify errors that might be transient in order to give a friendly "try again" message. But now that you've turned on a retry policy, the only errors likely to be transient will already have been tried and failed several times and the actual exception returned will be wrapped in the `RetryLimitExceededException` exception.

For more information, see [Entity Framework Connection Resiliency / Retry Logic](#).

Enable Command Interception

Now that you've turned on a retry policy, how do you test to verify that it is working as expected? It's not so easy to force a transient error to happen, especially when you're running locally, and it would be especially difficult to integrate actual transient errors into an automated unit test. To test the connection resiliency feature, you need a way to intercept queries that Entity Framework sends to SQL Server and replace the SQL Server response with an exception type that is typically transient.

You can also use query interception in order to implement a best practice for cloud applications: [log the latency and success or failure of all calls to external services](#) such as database services. EF6 provides a [dedicated logging API](#) that can make it easier to do logging, but in this section of the tutorial you'll learn how to use the Entity Framework's [interception feature](#) directly, both for logging and for simulating transient errors.

Create a logging interface and class

A [best practice for logging](#) is to do it by using an interface rather than hard-coding calls to `System.Diagnostics.Trace` or a logging class. That makes it easier to change your logging mechanism later if you ever need to do that. So in this section you'll create the logging interface and a class to implement it./p>

1. Create a folder in the project and name it *Logging*.
2. In the *Logging* folder, create a class file named *ILogger.cs*, and replace the template code with the following code:

```

using System;

namespace ContosoUniversity.Logging
{
    public interface ILogger
    {

```

```

        void Information(string message);
        void Information(string fmt, params object[] vars);
        void Information(Exception exception, string fmt, params
object[] vars);

        void Warning(string message);
        void Warning(string fmt, params object[] vars);
        void Warning(Exception exception, string fmt, params object[]
vars);

        void Error(string message);
        void Error(string fmt, params object[] vars);
        void Error(Exception exception, string fmt, params object[]
vars);

        void TraceApi(string componentName, string method, TimeSpan
timespan);
        void TraceApi(string componentName, string method, TimeSpan
timespan, string properties);
        void TraceApi(string componentName, string method, TimeSpan
timespan, string fmt, params object[] vars);
    }
}

```

The interface provides three tracing levels to indicate the relative importance of logs, and one designed to provide latency information for external service calls such as database queries. The logging methods have overloads that let you pass in an exception. This is so that exception information including stack trace and inner exceptions is reliably logged by the class that implements the interface, instead of relying on that being done in each logging method call throughout the application.

The TraceApi methods enable you to track the latency of each call to an external service such as SQL Database.

3. In the *Logging* folder, create a class file named *Logger.cs*, and replace the template code with the following code:

```

using System;
using System.Diagnostics;
using System.Text;

namespace ContosoUniversity.Logging
{
    public class Logger : ILogger
    {
        public void Information(string message)
        {
            Trace.TraceInformation(message);
        }

        public void Information(string fmt, params object[] vars)
        {

```

```

        Trace.TraceInformation(fmt, vars);
    }

    public void Information(Exception exception, string fmt, params
object[] vars)
    {
        Trace.TraceInformation(FormatExceptionMessage(exception,
fmt, vars));
    }

    public void Warning(string message)
    {
        Trace.TraceWarning(message);
    }

    public void Warning(string fmt, params object[] vars)
    {
        Trace.TraceWarning(fmt, vars);
    }

    public void Warning(Exception exception, string fmt, params
object[] vars)
    {
        Trace.TraceWarning(FormatExceptionMessage(exception, fmt,
vars));
    }

    public void Error(string message)
    {
        Trace.TraceError(message);
    }

    public void Error(string fmt, params object[] vars)
    {
        Trace.TraceError(fmt, vars);
    }

    public void Error(Exception exception, string fmt, params
object[] vars)
    {
        Trace.TraceError(FormatExceptionMessage(exception, fmt,
vars));
    }

    public void TraceApi(string componentName, string method,
TimeSpan timespan)
    {
        TraceApi(componentName, method, timespan, "");
    }

    public void TraceApi(string componentName, string method,
TimeSpan timespan, string fmt, params object[] vars)
    {
        TraceApi(componentName, method, timespan,
string.Format(fmt, vars));
    }

```



```

        public void TraceApi(string componentName, string method,
        TimeSpan timespan, string properties)
        {
            string message = String.Concat("Component:", componentName,
            ";Method:", method, ";Timespan:", timespan.ToString(), ";Properties:",
            properties);
            Trace.TraceInformation(message);
        }

        private static string FormatExceptionMessage(Exception
        exception, string fmt, object[] vars)
        {
            // Simple exception formatting: for a more comprehensive
            version see
            // http://code.msdn.microsoft.com/windowsazure/Fix-It-app-
            for-Building-cdd80df4
            var sb = new StringBuilder();
            sb.Append(string.Format(fmt, vars));
            sb.Append(" Exception: ");
            sb.Append(exception.ToString());
            return sb.ToString();
        }
    }
}

```

The implementation uses `System.Diagnostics` to do the tracing. This is a built-in feature of .NET which makes it easy to generate and use tracing information. There are many "listeners" you can use with `System.Diagnostics` tracing, to write logs to files, for example, or to write them to blob storage in Windows Azure. See some of the options, and links to other resources for more information, in [Troubleshooting Windows Azure Web Sites in Visual Studio](#). For this tutorial you'll only look at logs in the Visual Studio **Output** window.

In a production application you might want to consider tracing packages other than `System.Diagnostics`, and the `ILogger` interface makes it relatively easy to switch to a different tracing mechanism if you decide to do that.

Create interceptor classes

Next you'll create the classes that the Entity Framework will call into every time it is going to send a query to the database, one to simulate transient errors and one to do logging. These interceptor classes must derive from the `DbCommandInterceptor` class. In them you write method overrides that are automatically called when query is about to be executed. In these methods you can examine or log the query that is being sent to the database, and you can change the query before it's sent to the database or return something to Entity Framework yourself without even passing the query to the database.

1. To create the interceptor class that will log every SQL query that is sent to the database, create a class file named *SchoolInterceptorLogging.cs* in the *DAL* folder, and replace the template code with the following code:

```

using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorLogging : DbCommandInterceptor
    {
        private ILogger _logger = new Logger();
        private readonly Stopwatch _stopwatch = new Stopwatch();

        public override void ScalarExecuting(DbCommand command,
            DbCommandInterceptionContext<object> interceptionContext)
        {
            base.ScalarExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void ScalarExecuted(DbCommand command,
            DbCommandInterceptionContext<object> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database",
"SchoolInterceptor.ScalarExecuted", _stopwatch.Elapsed, "Command: {0}:
", command.CommandText);
            }
            base.ScalarExecuted(command, interceptionContext);
        }

        public override void NonQueryExecuting(DbCommand command,
            DbCommandInterceptionContext<int> interceptionContext)
        {
            base.NonQueryExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void NonQueryExecuted(DbCommand command,
            DbCommandInterceptionContext<int> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {

```

```

        _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
    }
    else
    {
        _logger.TraceApi("SQL Database",
"SchoolInterceptor.NonQueryExecuted", _stopwatch.Elapsed, "Command:
{0}: ", command.CommandText);
    }
    base.NonQueryExecuted(command, interceptionContext);
}

public override void ReaderExecuting(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
{
    base.ReaderExecuting(command, interceptionContext);
    _stopwatch.Restart();
}

public override void ReaderExecuted(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
{
    _stopwatch.Stop();
    if (interceptionContext.Exception != null)
    {
        _logger.Error(interceptionContext.Exception, "Error
executing command: {0}", command.CommandText);
    }
    else
    {
        _logger.TraceApi("SQL Database",
"SchoolInterceptor.ReaderExecuted", _stopwatch.Elapsed, "Command: {0}:
", command.CommandText);
    }
    base.ReaderExecuted(command, interceptionContext);
}
}
}
}

```

For successful queries or commands, this code writes an Information log with latency information. For exceptions, it creates an Error log.

2. To create the interceptor class that will generate dummy transient errors when you enter "Throw" in the **Search** box, create a class file named *SchoolInterceptorTransientErrors.cs* in the *DAL* folder, and replace the template code with the following code:

```

using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;

```

```

using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorTransientErrors :
    DbCommandInterceptor
    {
        private int _counter = 0;
        private ILogger _logger = new Logger();

        public override void ReaderExecuting(DbCommand command,
        DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            bool throwTransientErrors = false;
            if (command.Parameters.Count > 0 &&
        command.Parameters[0].Value.ToString() == "Throw")
            {
                throwTransientErrors = true;
                command.Parameters[0].Value = "an";
                command.Parameters[1].Value = "an";
            }

            if (throwTransientErrors && _counter < 4)
            {
                _logger.Information("Returning transient error for
        command: {0}", command.CommandText);
                _counter++;
                interceptionContext.Exception =
        CreateDummySQLException();
            }

            private SQLException CreateDummySQLException()
            {
                // The instance of SQL Server you attempted to connect to
        does not support encryption
                var sqlErrorNumber = 20;

                var sqlErrorCtor =
        typeof(SQLException).GetConstructors(BindingFlags.Instance |
        BindingFlags.NonPublic).Where(c => c.GetParameters().Count() ==
        7).Single();
                var sqlError = sqlErrorCtor.Invoke(new object[] {
        sqlErrorNumber, (byte)0, (byte)0, "", "", "", 1 });

                var errorCollection =
        Activator.CreateInstance(typeof(SQLExceptionCollection), true);
                var addMethod = typeof(SQLExceptionCollection).GetMethod("Add",
        BindingFlags.Instance | BindingFlags.NonPublic);
                addMethod.Invoke(errorCollection, new[] { sqlError });

                var sqlExceptionCtor =
        typeof(SQLException).GetConstructors(BindingFlags.Instance |
        BindingFlags.NonPublic).Where(c => c.GetParameters().Count() ==
        4).Single();

```

```

        var sqlException =
            (SqlException)sqlExceptionCtor.Invoke(new object[] { "Dummy",
            errorCollection, null, Guid.NewGuid() });

        return sqlException;
    }
}

```

This code only overrides the `ReaderExecuting` method, which is called for queries that can return multiple rows of data. If you wanted to check connection resiliency for other types of queries, you could also override the `NonQueryExecuting` and `ScalarExecuting` methods, as the logging interceptor does.

When you run the Student page and enter "Throw" as the search string, this code creates a dummy SQL Database exception for error number 20, a type known to be typically transient. Other error numbers currently recognized as transient are 64, 233, 10053, 10054, 10060, 10928, 10929, 40197, 40501, and 40613, but these are subject to change in new versions of SQL Database.

The code returns the exception to Entity Framework instead of running the query and passing back query results. The transient exception is returned four times, and then the code reverts to the normal procedure of passing the query to the database.

Because everything is logged, you'll be able to see that Entity Framework tries to execute the query four times before finally succeeding, and the only difference in the application is that it takes longer to render a page with query results.

The number of times the Entity Framework will retry is configurable; the code specifies four times because that's the default value for the SQL Database execution policy. If you change the execution policy, you'd also change the code here that specifies how many times transient errors are generated. You could also change the code to generate more exceptions so that Entity Framework will throw the `RetryLimitExceededException` exception.

The value you enter in the Search box will be in `command.Parameters[0]` and `command.Parameters[1]` (one is used for the first name and one for the last name). When the value "Throw" is found, it is replaced in those parameters by "an" so that some students will be found and returned.

This is just a convenient way to test connection resiliency based on changing some input to the application UI. You can also write code that generates transient errors for all queries or updates, as explained later in the comments about the `DbInterception.Add` method.

3. In *Global.asax*, add the following `using` statements:

```
using ContosoUniversity.DAL;
```

```
using System.Data.Entity.Infrastructure.Interception;
```

4. Add the highlighted line to the `Application_Start` method:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    DbInterception.Add(new SchoolInterceptorTransientErrors());
    DbInterception.Add(new SchoolInterceptorLogging());
}
```

These lines of code are what causes your interceptor code to be run when Entity Framework sends queries to the database. Notice that because you created separate interceptor classes for transient error simulation and logging, you can independently enable and disable them.

You can add interceptors using the `DbInterception.Add` method anywhere in your code; it doesn't have to be in the `Application_Start` method. Another option is to put this code in the `DbConfiguration` class that you created earlier to configure the execution policy.

```
public class SchoolConfiguration : DbConfiguration
{
    public SchoolConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new
        SqlAzureExecutionStrategy());
        DbInterception.Add(new SchoolInterceptorTransientErrors());
        DbInterception.Add(new SchoolInterceptorLogging());
    }
}
```

Wherever you put this code, be careful not to execute `DbInterception.Add` for the same interceptor more than once, or you'll get additional interceptor instances. For example, if you add the logging interceptor twice, you'll see two logs for every SQL query.

Interceptors are executed in the order of registration (the order in which the `DbInterception.Add` method is called). The order might matter depending on what you're doing in the interceptor. For example, an interceptor might change the SQL command that it gets in the `CommandText` property. If it does change the SQL command, the next interceptor will get the changed SQL command, not the original SQL command.

You've written the transient error simulation code in a way that lets you cause transient errors by entering a different value in the UI. As an alternative, you could write the interceptor code to always generate the sequence of transient exceptions without checking for a particular parameter value. You could then add the interceptor only when you want to generate transient errors. If you do this, however, don't add the interceptor

until after database initialization has completed. In other words, do at least one database operation such as a query on one of your entity sets before you start generating transient errors. The Entity Framework executes several queries during database initialization, and they aren't executed in a transaction, so errors during initialization could cause the context to get into an inconsistent state.

Test logging and connection resiliency

1. Press F5 to run the application in debug mode, and then click the **Students** tab.
2. Look at the Visual Studio **Output** window to see the tracing output. You might have to scroll up past some JavaScript errors to get to the logs written by your logger.

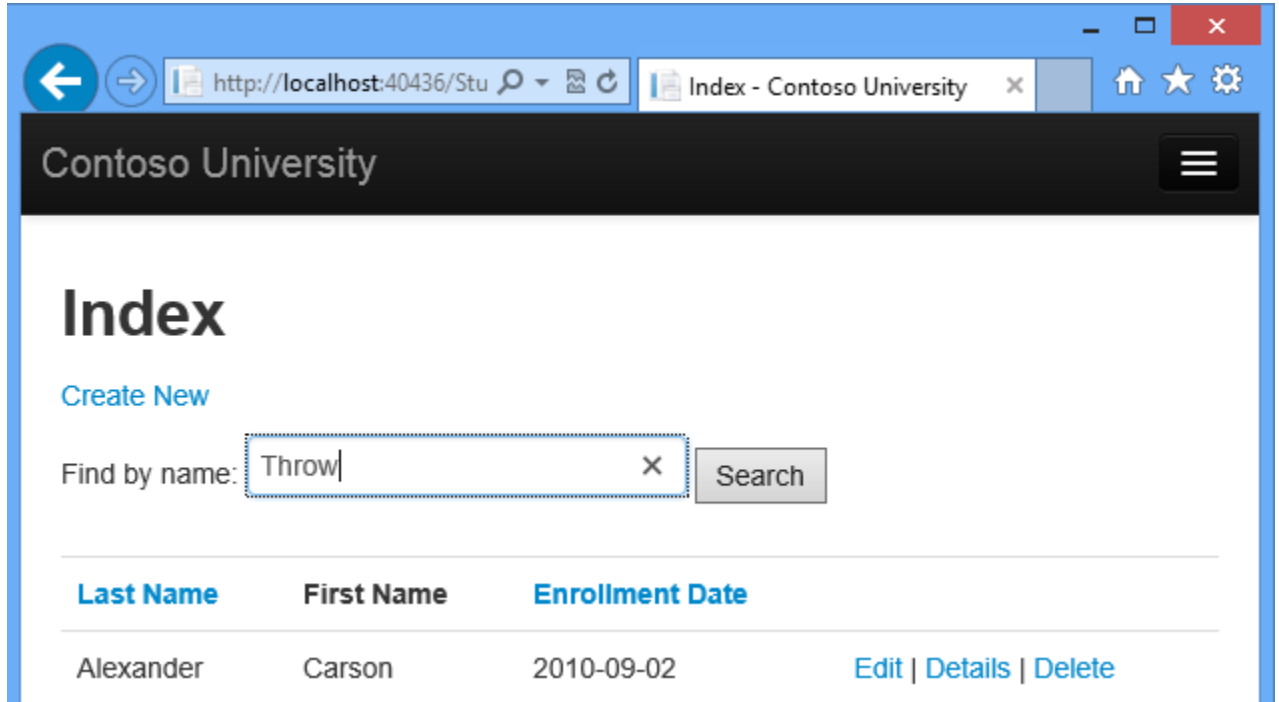
Notice that you can see the actual SQL queries sent to the database. You see some initial queries and commands that Entity Framework does to get started, checking the database version and migration history table (you'll learn about migrations in the next tutorial). And you see a query for paging, to find out how many students there are, and finally you see the query that gets the student data.

Output

Show output from: Debug

```
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0276705;Properties
:Command: select serverproperty('EngineEdition'):
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0424968;Properties
:Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0450531;Properties
:Command:
SELECT Count(*)
FROM INFORMATION_SCHEMA.TABLES AS t
WHERE t.TABLE_TYPE = 'BASE TABLE'
AND (t.TABLE_SCHEMA + '.' + t.TABLE_NAME IN
('dbo.Course', 'dbo.Department', 'dbo.Instructor', 'dbo.OfficeAssignment', 'dbo.Enrol
lment', 'dbo.Student', 'dbo.CourseInstructor')
OR t.TABLE_NAME = 'EdmMetadata'):
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0262633;Properties
:Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0373571;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM ( SELECT
COUNT(1) AS [A1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
) AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0227459;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM ( SELECT
COUNT(1) AS [A1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
) AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0298657;Properties
:Command: SELECT TOP (1)
[Project1].[C1] AS [C1],
[Project1].[MigrationId] AS [MigrationId],
[Project1].[Model] AS [Model]
FROM ( SELECT
[Extent1].[MigrationId] AS [MigrationId],
[Extent1].[Model] AS [Model],
1 AS [C1]
FROM [dbo].[__MigrationHistory] AS [Extent1]
WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
) AS [Project1]
ORDER BY [Project1].[MigrationId] DESC:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0246607;Properties
:Command: SELECT
[GroupBy1].[A1] AS [C1]
FROM ( SELECT
COUNT(1) AS [A1]
FROM [dbo].[Student] AS [Extent1]
) AS [GroupBy1]:
'iisexpress.exe' (CLR v4.0.30319: /LM/W3SVC/12/ROOT-1-130329870303366864): Loaded
'EntityFrameworkDynamicProxies-ContosoUniversity'.
..
iisexpress.exe Information: 0 : Component:SQL
```


3. In the **Students** page, enter "Throw" as the search string, and click **Search**.



You'll notice that the browser seems to hang for several seconds while Entity Framework is retrying the query several times. The first retry happens very quickly, then the wait before increases before each additional retry. This process of waiting longer before each retry is called *exponential backoff*.

When the page displays, showing students who have "an" in their names, look at the output window, and you'll see that the same query was attempted five times, the first four times returning transient exceptions. For each transient error you'll see the log that you write when generating the transient error in the `SchoolInterceptorTransientErrors` class ("Returning transient error for command...") and you'll see the log written when `SchoolInterceptorLogging` gets the exception.

```

Output
Show output from: Debug
iisexpress.exe Information: 0 : Returning transient error for command: SELECT
  [GroupBy1].[A1] AS [C1]
  FROM ( SELECT
    COUNT(1) AS [A1]
    FROM [dbo].[Student] AS [Extent1]
    WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
      int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
        [FirstName])) AS int)) > 0)
    ) AS [GroupBy1]
iisexpress.exe Error: 0 : Error executing command: SELECT
  [GroupBy1].[A1] AS [C1]
  FROM ( SELECT
    COUNT(1) AS [A1]
    FROM [dbo].[Student] AS [Extent1]
    WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
      int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
        [FirstName])) AS int)) > 0)
    ) AS [GroupBy1] Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:5430f70e-b790-4798-ac6d-d28f91ec3529
iisexpress.exe Information: 0 : Returning transient error for command: SELECT
  [GroupBy1].[A1] AS [C1]
  FROM ( SELECT
    COUNT(1) AS [A1]
    FROM [dbo].[Student] AS [Extent1]
    WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
      int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
        [FirstName])) AS int)) > 0)
    ) AS [GroupBy1]
iisexpress.exe Error: 0 : Error executing command: SELECT
  [GroupBy1].[A1] AS [C1]
  FROM ( SELECT
    COUNT(1) AS [A1]
    FROM [dbo].[Student] AS [Extent1]
    WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
      int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
        [FirstName])) AS int)) > 0)
    ) AS [GroupBy1] Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:bf3d3750-18e6-4e20-9ce7-a31ccd41d74b

```

Since you entered a search string, the query that returns student data is parameterized:

```

SELECT TOP (3)
  [Project1].[ID] AS [ID],
  [Project1].[LastName] AS [LastName],
  [Project1].[FirstMidName] AS [FirstMidName],
  [Project1].[EnrollmentDate] AS [EnrollmentDate]
  FROM ( SELECT [Project1].[ID] AS [ID], [Project1].[LastName] AS
[LastName], [Project1].[FirstMidName] AS [FirstMidName],
[Project1].[EnrollmentDate] AS [EnrollmentDate], row_number() OVER
(ORDER BY [Project1].[LastName] ASC) AS [row_number]
  FROM ( SELECT
    [Extent1].[ID] AS [ID],

```

```

        [Extent1].[LastName] AS [LastName],
        [Extent1].[FirstMidName] AS [FirstMidName],
        [Extent1].[EnrollmentDate] AS [EnrollmentDate]
FROM [dbo].[Student] AS [Extent1]
WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0),
UPPER([Extent1].[LastName])) AS int)) > 0) OR ((
CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].[FirstMidName])) AS
int)) > 0)
    ) AS [Project1]
) AS [Project1]
WHERE [Project1].[row_number] > 0
ORDER BY [Project1].[LastName] ASC

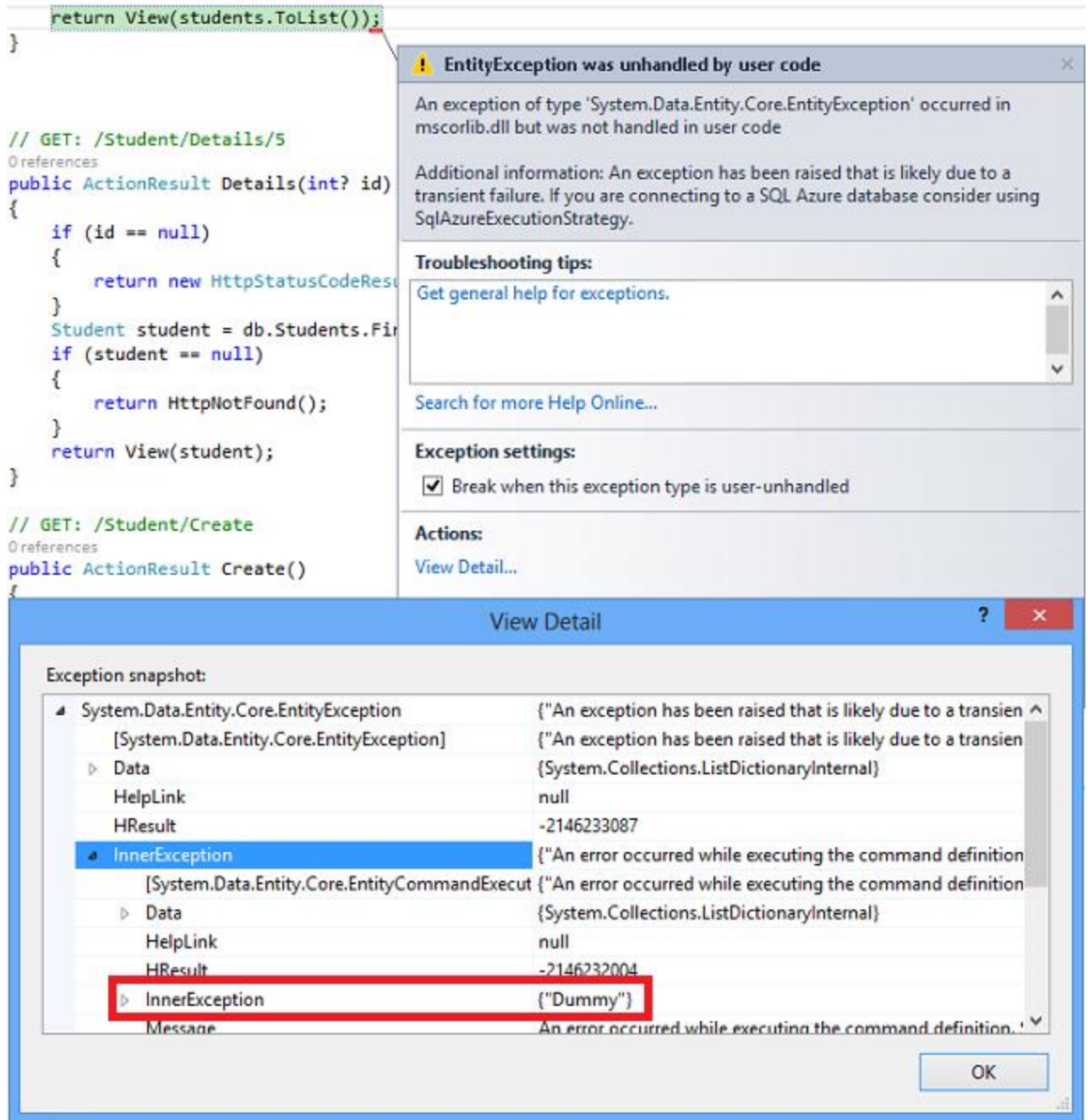
```

You're not logging the value of the parameters, but you could do that. If you want to see the parameter values, you can write logging code to get parameter values from the `Parameters` property of the `DbCommand` object that you get in the interceptor methods.

Note that you can't repeat this test unless you stop the application and restart it. If you wanted to be able to test connection resiliency multiple times in a single run of the application, you could write code to reset the error counter in `SchoolInterceptorTransientErrors`.

4. To see the difference the execution strategy (retry policy) makes, comment out the `SetExecutionStrategy` line in `SchoolConfiguration.cs`, run the `Students` page in debug mode again, and search for "Throw" again.

This time the debugger stops on the first generated exception immediately when it tries to execute the query the first time.



5. Uncomment the `SetExecutionStrategy` line in `SchoolConfiguration.cs`.

Summary

In this tutorial you've seen how to enable connection resiliency and log SQL commands that Entity Framework composes and sends to the database. In the next tutorial you'll deploy the application to the Internet, using Code First Migrations to deploy the database.

Code First Migrations and Deployment with the Entity Framework in an ASP.NET MVC Application

So far the application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this tutorial you'll deploy the Contoso University application to the cloud in a Windows Azure Web Site.

The tutorial contains the following sections:

- Enable Code First Migrations. The Migrations feature enables you to change the data model and deploy your changes to production by updating the database schema without having to drop and re-create the database.
- Deploy to Windows Azure. This step is optional; you can continue with the remaining tutorials without having deployed the project.

Enable Code First Migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You have configured the Entity Framework to automatically drop and re-create the database each time you change the data model. When you add, remove, or change entity classes or change your `DbContext` class, the next time you run the application it automatically deletes your existing database, creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The [Code First Migrations](#) feature solves this problem by enabling Code First to update the database schema instead of dropping and re-creating the database. In this tutorial, you'll deploy the application, and to prepare for that you'll enable Migrations.

1. Disable the initializer that you set up earlier by commenting out or deleting the `contexts` element that you added to the application `Web.config` file.

```
<entityFramework>
  <!--<contexts>
    <context type="ContosoUniversity.DAL.SchoolContext,
ContosoUniversity">
      <databaseInitializer
type="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity" />
    </context>
  </contexts>-->
```

```

    <defaultConnectionFactory
type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
    <parameters>
        <parameter value="v11.0" />
    </parameters>
</defaultConnectionFactory>
<providers>
    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
</providers>
</entityFramework>

```

2. Also in the application *Web.config* file, change the name of the database in the connection string to ContosoUniversity2.

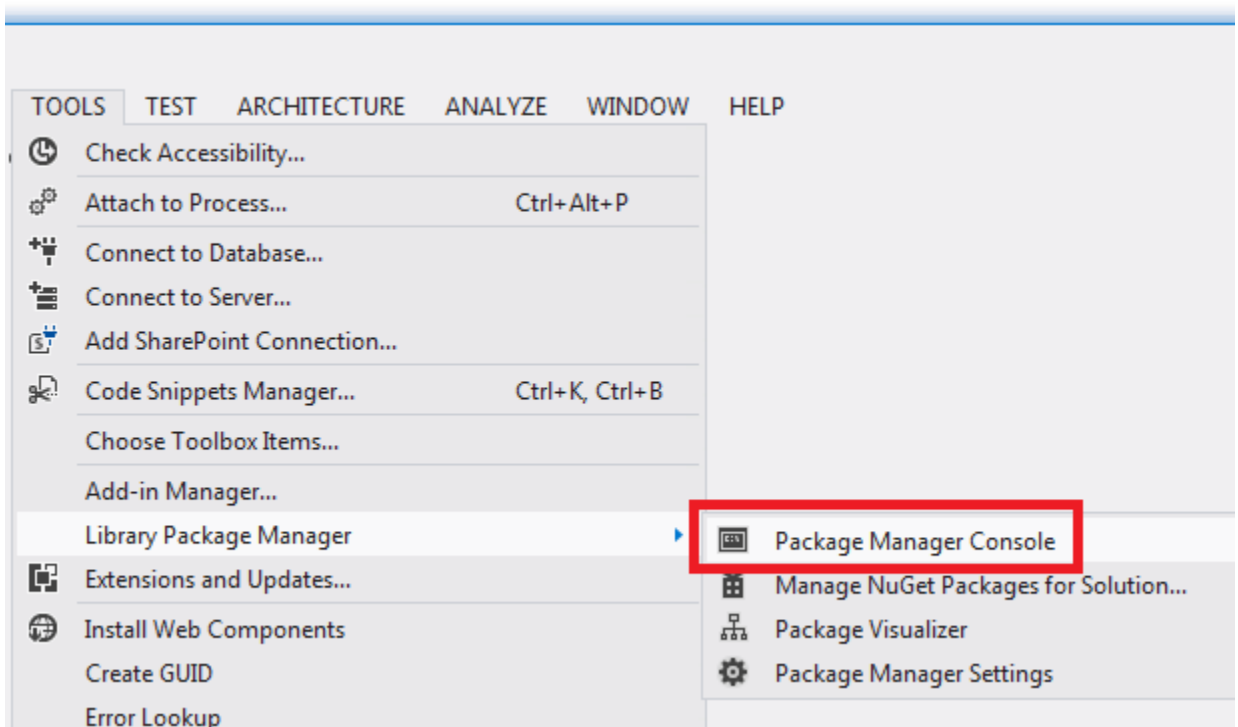
```

<connectionStrings>
    <add name="SchoolContext" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=ContosoUniversity2;Integrated
Security=SSPI;" providerName="System.Data.SqlClient" />
</connectionStrings>

```

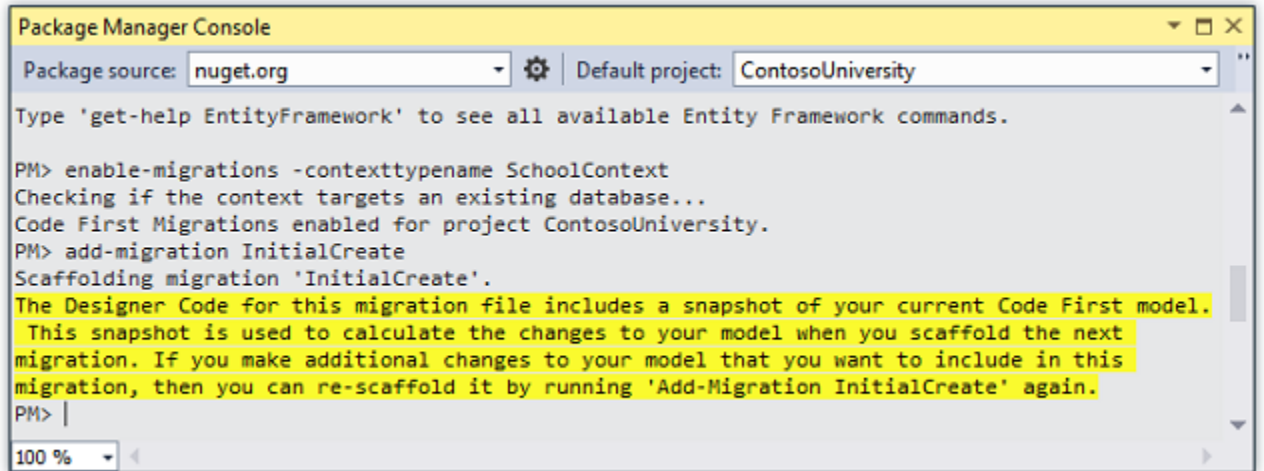
This change sets up the project so that the first migration will create a new database. This isn't required but you'll see later why it's a good idea.

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



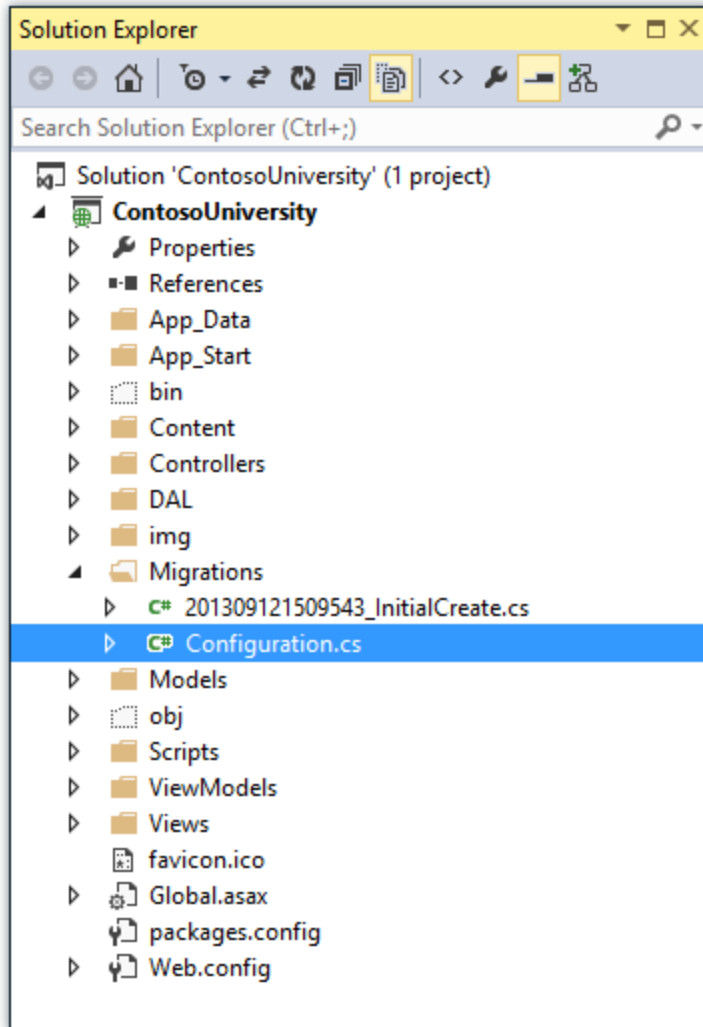
4. At the `PM>` prompt enter the following commands:

```
enable-migrations
add-migration InitialCreate
```



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

(If you missed the step above that directs you to change the database name, Migrations will find the existing database and automatically do the `add-migration` command. That's OK, it just means you won't run a test of the migrations code before you deploy the database. Later when you run the `update-database` command nothing will happen because the database will already exist.)



Like the initializer class that you saw earlier, the `Configuration` class includes a `Seed` method.

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

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

        // You can use the DbSet<T>.AddOrUpdate() helper extension
        method
        // to avoid creating duplicate seed data. E.g.
```



```

        //
        // context.People.AddOrUpdate(
        //     p => p.FullName,
        //     new Person { FullName = "Andrew Peters" },
        //     new Person { FullName = "Brice Lambson" },
        //     new Person { FullName = "Rowan Miller" }
        // );
    }
}

```

The purpose of the [Seed](#) method is to enable you to insert or update test data after Code First creates or updates the database. The method is called when the database is created and every time the database schema is updated after a data model change.

Set up the Seed Method

When you are dropping and re-creating the database for every data model change, you use the initializer class's `Seed` method to insert test data, because after every model change the database is dropped and all the test data is lost. With Code First Migrations, test data is retained after database changes, so including test data in the [Seed](#) method is typically not necessary. In fact, you don't want the `Seed` method to insert test data if you'll be using Migrations to deploy the database to production, because the `Seed` method will run in production. In that case you want the `Seed` method to insert into the database only the data that you need in production. For example, you might want the database to include actual department names in the `Department` table when the application becomes available in production.

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

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

```

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

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

```

```

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

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

        var enrollments = new List<Enrollment>
        {

```

```

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

```

```

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

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

```

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

Some of the statements that insert data use the [AddOrUpdate](#) method to perform an "upsert" operation. Because the `Seed` method runs every time you execute the `update-database` command, typically after each migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it *overrides* any changes to data that you may have made while testing the application. With test data in some tables you might not want that to

happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist. The Seed method uses both approaches.

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

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

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

Sequence contains more than one element

For information about how to handle redundant data such as two students named "Alexander Carson", see [Seeding and Debugging Entity Framework \(EF\) DBs](#) on Rick Anderson's blog. For more information about the `AddOrUpdate` method, see [Take care with EF 4.3 AddOrUpdate Method](#) on Julie Lerman's blog.

The code that creates `Enrollment` entities assumes you have the `ID` value in the entities in the `students` collection, although you didn't set that property in the code that creates the collection.

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

You can use the `ID` property here because the `ID` value is set when you call `SaveChanges` for the `students` collection. EF automatically gets the primary key value when it inserts an entity into the database, and it updates the `ID` property of the entity in memory.

The code that adds each `Enrollment` entity to the `Enrollments` entity set doesn't use the `AddOrUpdate` method. It checks if an entity already exists and inserts the entity if it doesn't exist. This approach will preserve changes you make to an enrollment grade by using the application UI. The code loops through each member of the `Enrollment` [List](#) and if the enrollment is not found in the database, it adds the enrollment to the database. The first time you update the database, the database will be empty, so it will add each enrollment.

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

```

        if (enrollmentInDataBase == null)
        {
            context.Enrollments.Add(e);
        }
    }
}

```

2. Build the project.

Execute the First Migration

When you executed the `add-migration` command, Migrations generated the code that would create the database from scratch. This code is also in the *Migrations* folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them.

```

public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Course",
            c => new
            {
                CourseID = c.Int(nullable: false),
                Title = c.String(),
                Credits = c.Int(nullable: false),
            })
            .PrimaryKey(t => t.CourseID);

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

        CreateTable(
            "dbo.Student",
            c => new
            {
                ID = c.Int(nullable: false, identity: true),
                LastName = c.String(),
                FirstMidName = c.String(),
                EnrollmentDate = c.DateTime(nullable: false),
            })
            .PrimaryKey(t => t.ID);
    }
}

```

```

    }

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

```

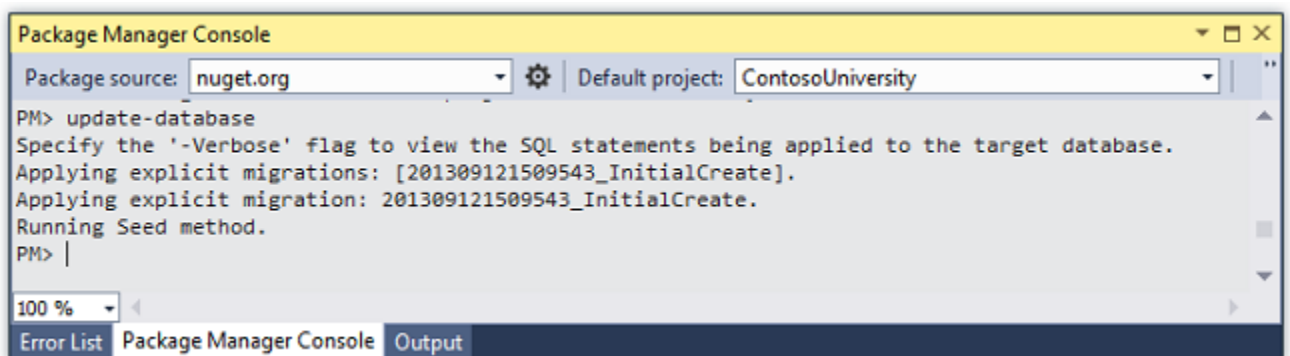
Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This is the initial migration that was created when you entered the `add-migration InitialCreate` command. The parameter (`InitialCreate` in the example) is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

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

```
update-database
```



The `update-database` command runs the `Up` method to create the database and then it runs the `Seed` method to populate the database. The same process will run automatically in production after you deploy the application, as you'll see in the following section.

2. Use **Server Explorer** to inspect the database as you did in the first tutorial, and run the application to verify that everything still works the same as before.

Deploy to Windows Azure

So far the application has been running locally in IIS Express on your development computer. To make it available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this section of the tutorial you'll deploy it to a Windows Azure Web Site. This section is optional; you can skip this and continue with the following tutorial, or you can adapt the instructions in this section for a different hosting provider of your choice.

Using Code First Migrations to Deploy the Database

To deploy the database you'll use Code First Migrations. When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box labeled **Execute Code First Migrations (runs on application start)**. This setting causes the deployment process to automatically configure the application *Web.config* file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

Visual Studio doesn't do anything with the database during the deployment process while it is copying your project to the destination server. When you run the deployed application and it accesses the database for the first time after deployment, Code First checks if the database matches the data model. If there's a mismatch, Code First automatically creates the database (if it doesn't exist yet) or updates the database schema to the latest version (if a database exists but doesn't match the model). If the application implements a Migrations `Seed` method, the method runs after the database is created or the schema is updated.

Your Migrations `Seed` method inserts test data. If you were deploying to a production environment, you would have to change the `Seed` method so that it only inserts data that you want to be inserted into your production database. For example, in your current data model you might want to have real courses but fictional students in the development database. You can write a `Seed` method to load both in development, and then comment out the fictional students before you deploy to production. Or you can write a `Seed` method to load only courses, and enter the fictional students in the test database manually by using the application's UI.

Get a Windows Azure account

You'll need a Windows Azure account. If you don't already have one, but you do have an MSDN subscription, you can [activate your MSDN subscription benefits](#). Otherwise, you can create a free trial account in just a couple of minutes. For details, see [Windows Azure Free Trial](#).

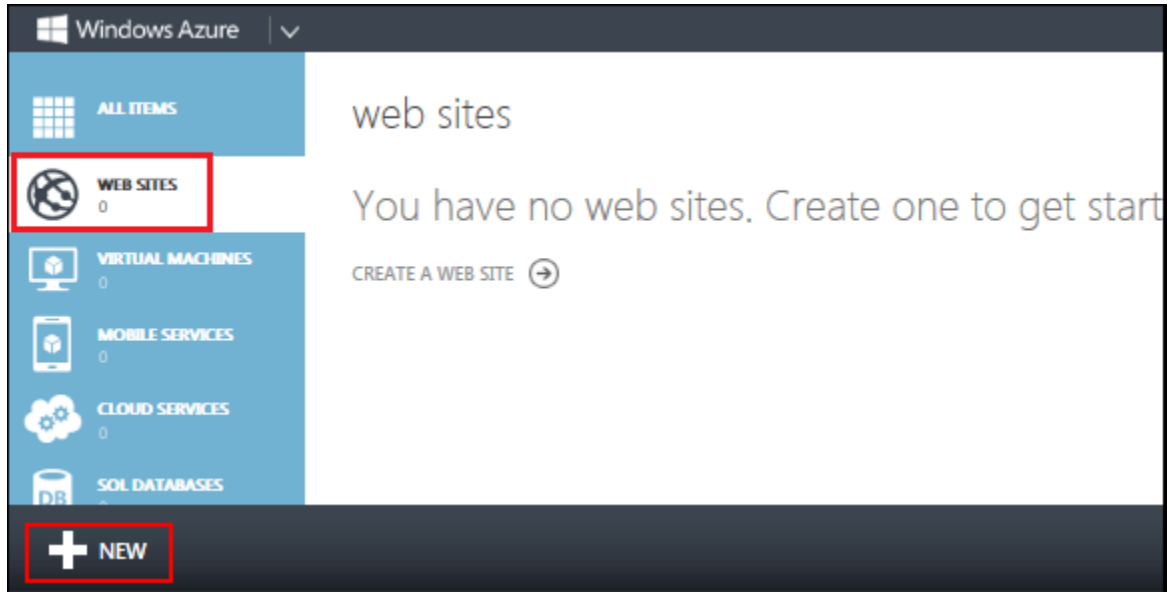
Create a web site and a SQL database in Windows Azure

Your Windows Azure Web Site will run in a shared hosting environment, which means it runs on virtual machines (VMs) that are shared with other Windows Azure clients. A shared hosting

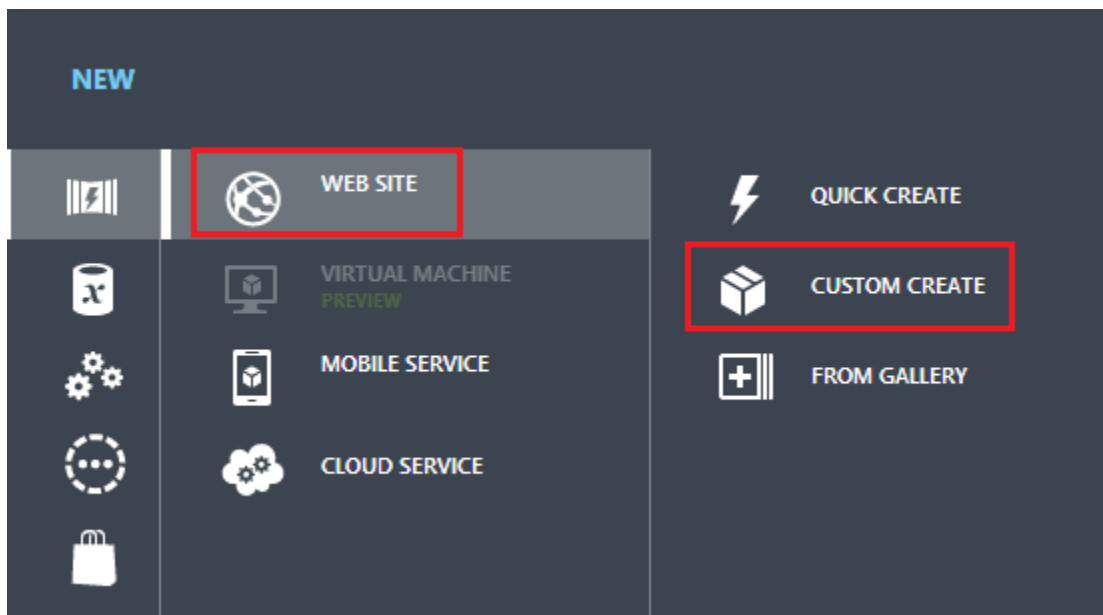
environment is a low-cost way to get started in the cloud. Later, if your web traffic increases, the application can scale to meet the need by running on dedicated VMs.

You'll deploy the database to Windows Azure SQL Database. SQL Database is a cloud-based relational database service that is built on SQL Server technologies. Tools and applications that work with SQL Server also work with SQL Database.

1. In the [Windows Azure Management Portal](#), click **Web Sites** in the left tab, and then click **New**.

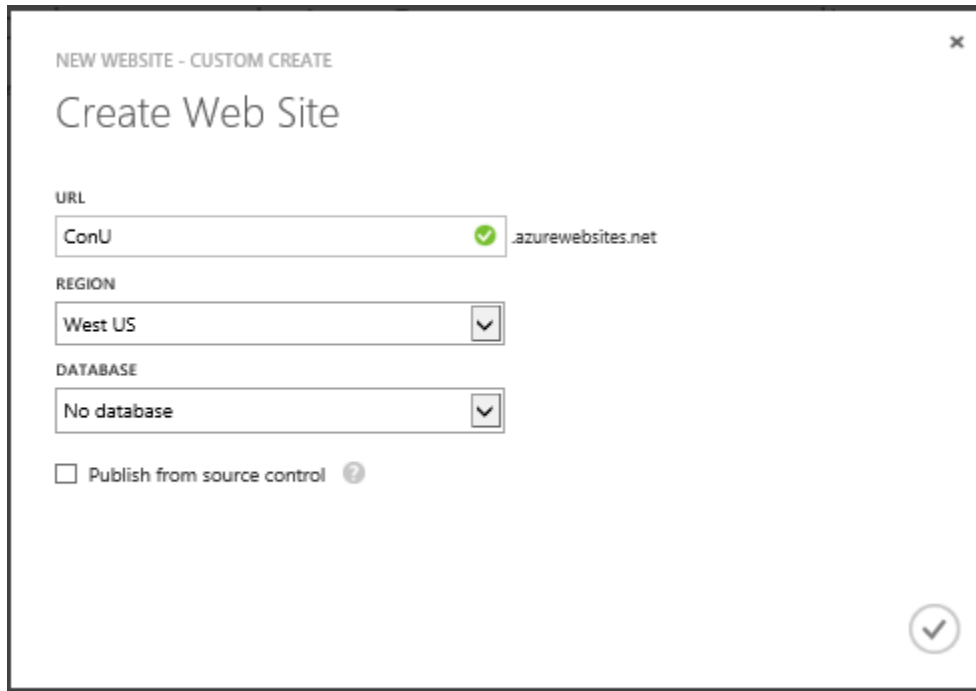


2. Click **CUSTOM CREATE**.



The **New Web Site - Custom Create** wizard opens.

3. In the **New Web Site** step of the wizard, enter a string in the **URL** box to use as the unique URL for your application. The complete URL will consist of what you enter here plus the suffix that you see next to the text box. The illustration shows "ConU", but that URL is probably taken so you will have to choose a different one.



NEW WEBSITE - CUSTOM CREATE

Create Web Site

URL

ConU ✓ .azurewebsites.net

REGION

West US ▼

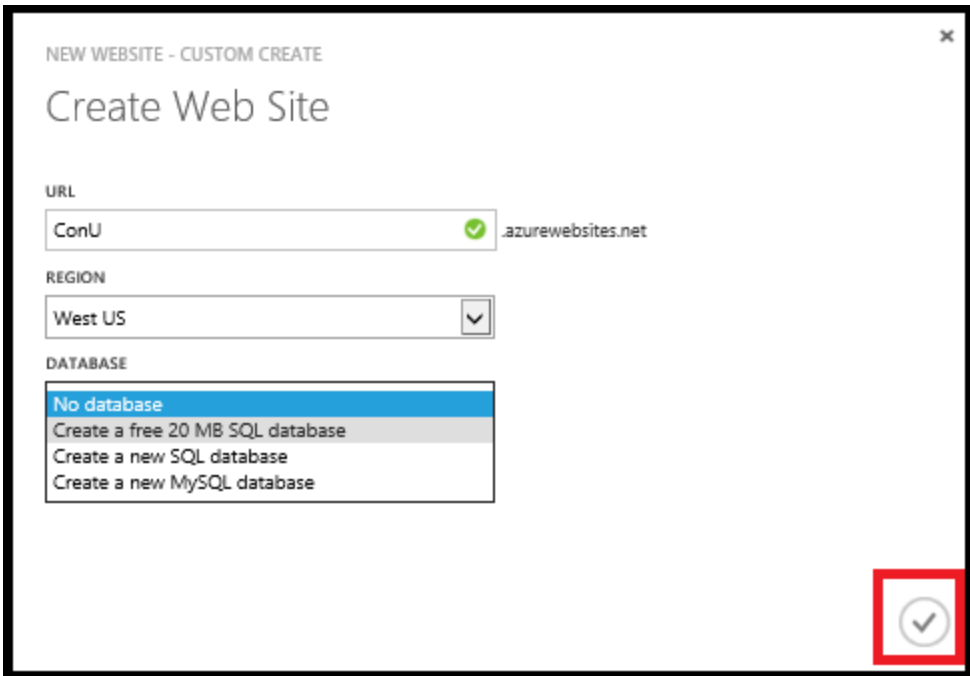
DATABASE

No database ▼

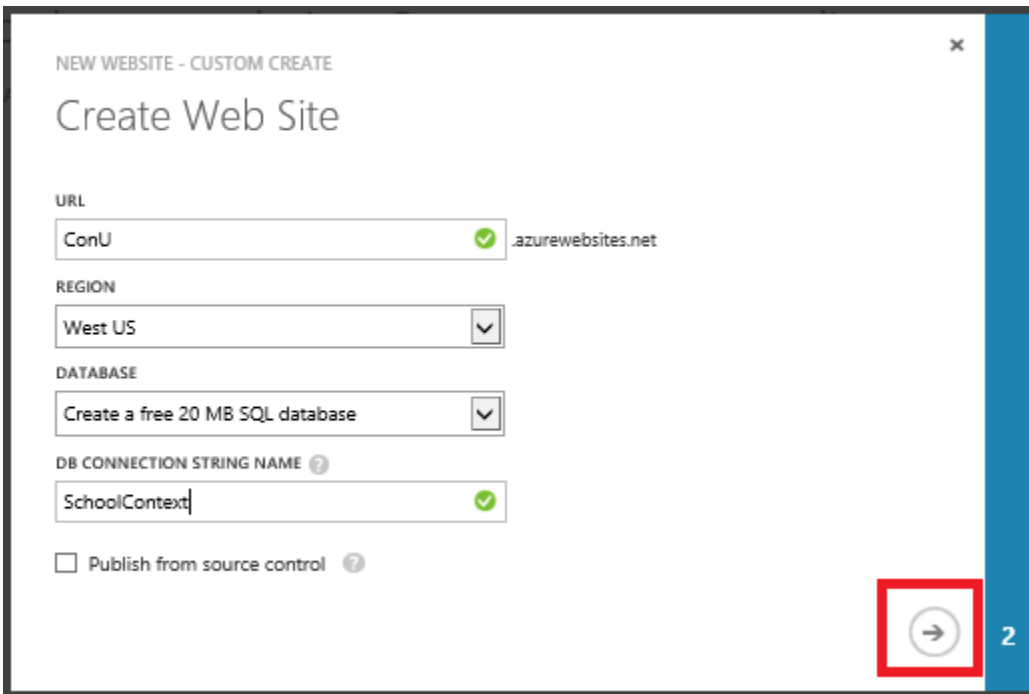
Publish from source control ?

✓

4. In the **Region** drop-down list, choose a region close to you. This setting specifies which data center your web site will run in.
5. In the **Database** drop-down list, choose **Create a free 20 MB SQL database**.



6. In the **DB CONNECTION STRING NAME**, enter *SchoolContext*.



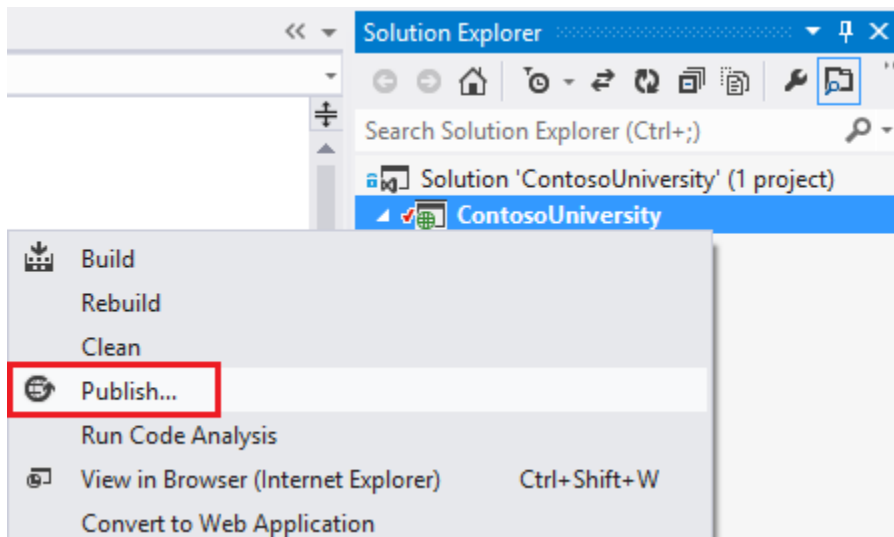
7. Click the arrow that points to the right at the bottom of the box. The wizard advances to the **Database Settings** step.
8. In the **Name** box, enter *ContosoUniversityDB*.

9. In the **Server** box, select **New SQL Database server**. Alternatively, if you previously created a server, you can select that server from the drop-down list.
10. Enter an administrator **LOGIN NAME** and **PASSWORD**. If you selected **New SQL Database server** you aren't entering an existing name and password here, you're entering a new name and password that you're defining now to use later when you access the database. If you selected a server that you created previously, you'll enter credentials for that server. For this tutorial, you won't select the **Advanced** check box. The **Advanced** options enable you to set the database [collation](#).
11. Choose the same **Region** that you chose for the web site.
12. Click the check mark at the bottom right of the box to indicate that you're finished.

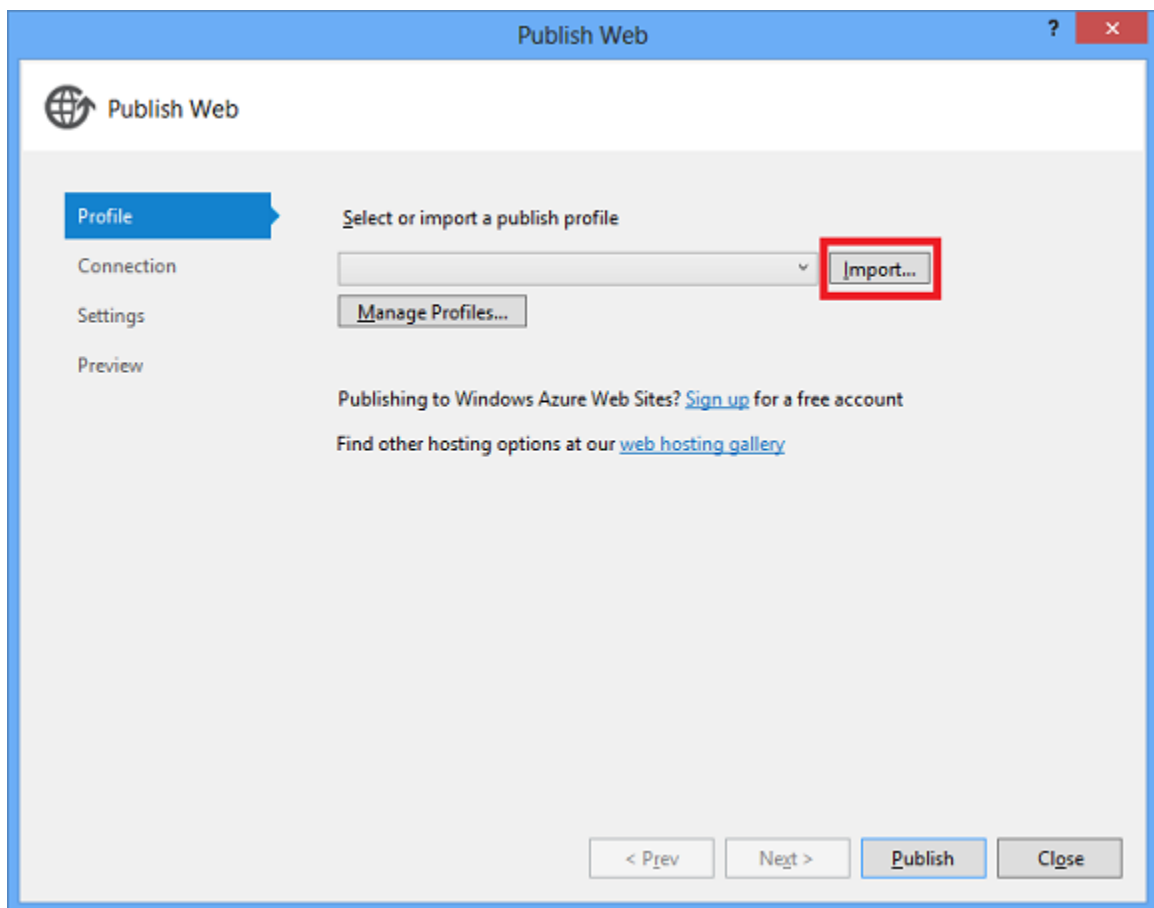
The Management Portal returns to the Web Sites page, and the **Status** column shows that the site is being created. After a while (typically less than a minute), the **Status** column shows that the site was successfully created. In the navigation bar at the left, the number of sites you have in your account appears next to the **Web Sites** icon, and the number of databases appears next to the **SQL Databases** icon.

Deploy the application to Windows Azure

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.



2. In the **Profile** tab of the **Publish Web** wizard, click **Import**.

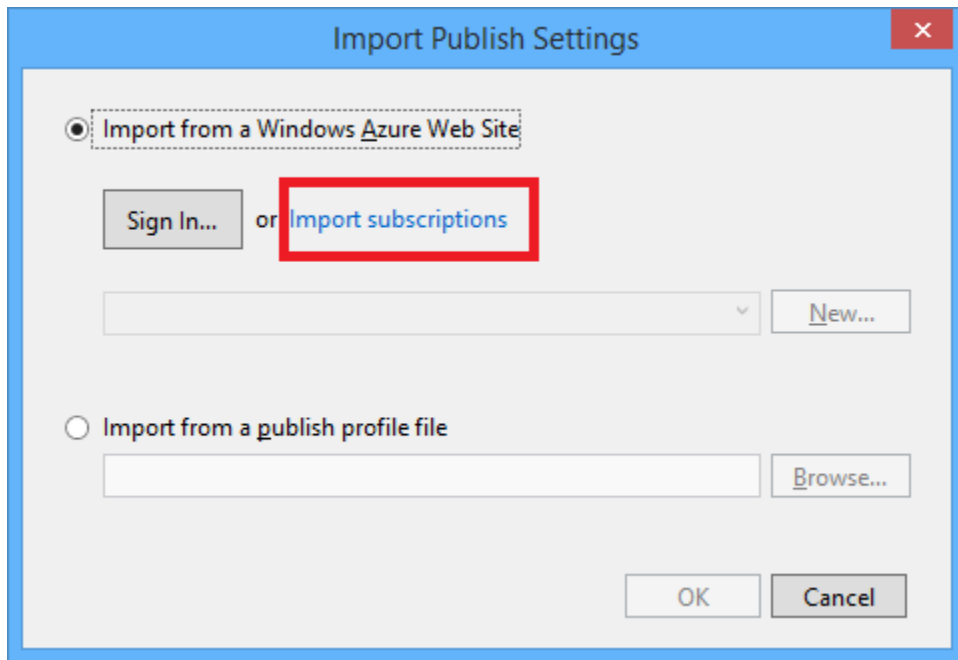


3. If you have not previously added your Windows Azure subscription in Visual Studio, perform the following steps. These steps enable Visual Studio to connect to your

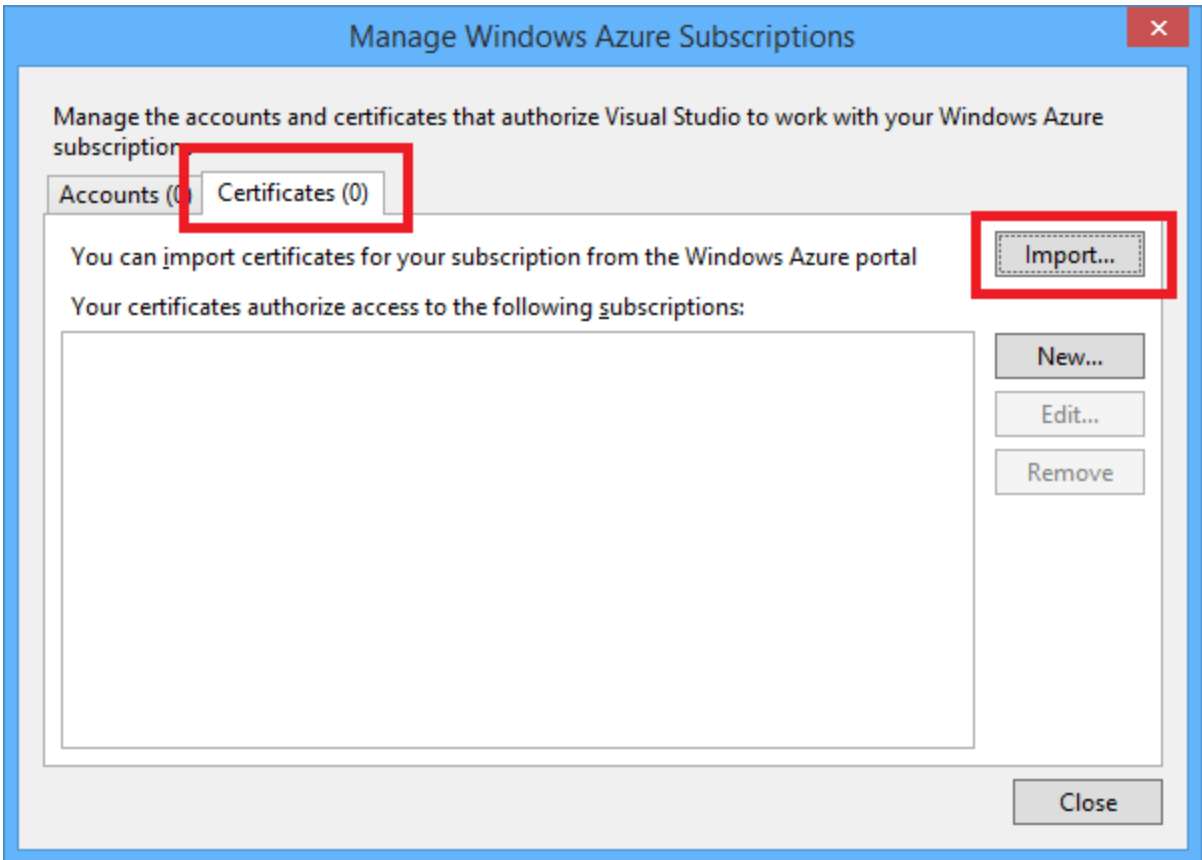
Windows Azure subscription so that the drop-down list under **Import from a Windows Azure web site** will include your web site.

As an alternative, you can sign in directly to your Windows Azure account without downloading a subscription file. To use this method, click **Sign In** instead of **Manage subscriptions** in the next step. This alternative is simpler, but as this tutorial is being written in November, 2013, only the subscription file download enables **Server Explorer** to connect to Windows Azure SQL Database.

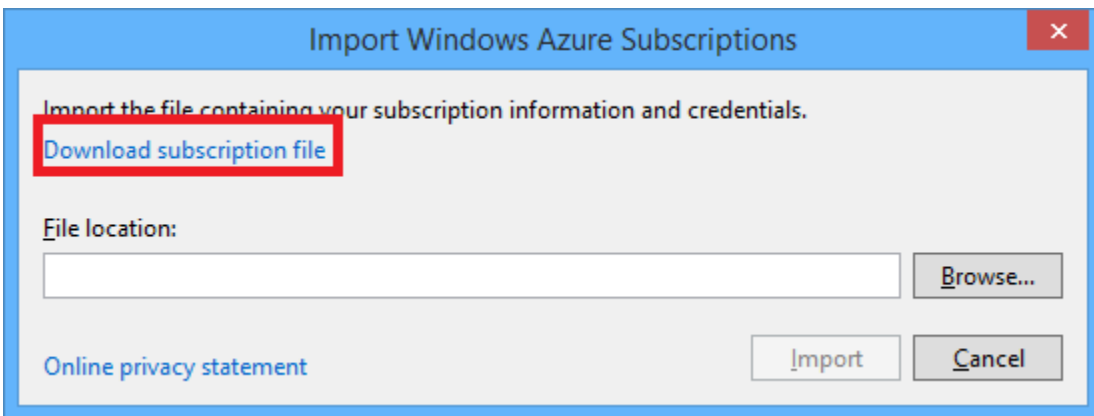
a. In the **Import Publish Profile** dialog box, click **Manage subscriptions**.



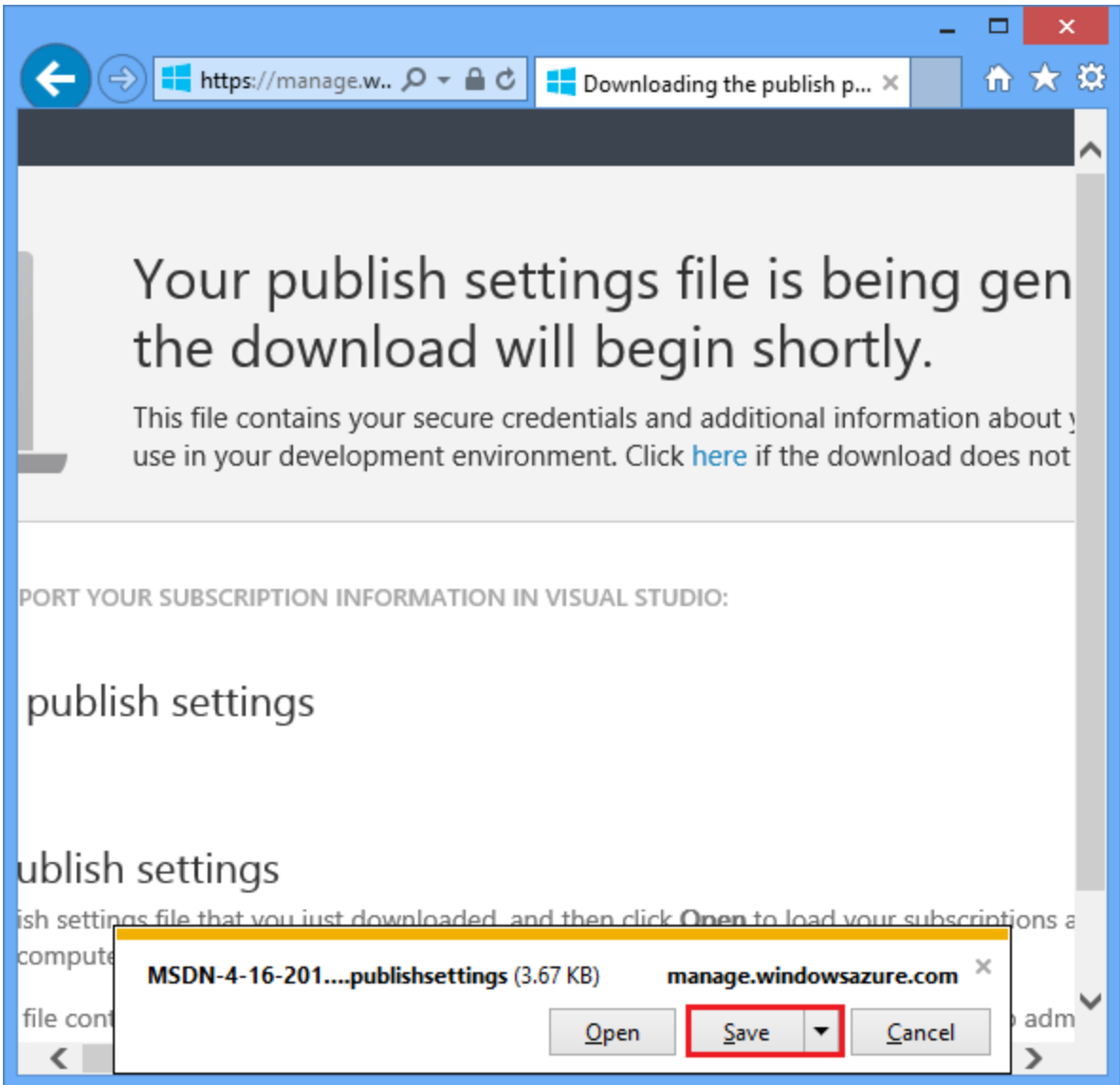
b. In the **Manage Windows Azure Subscriptions** dialog box, click the **Certificates** tab, and then click **Import**.



c. In the **Import Windows Azure Subscriptions** dialog box, click **Download subscription file** .

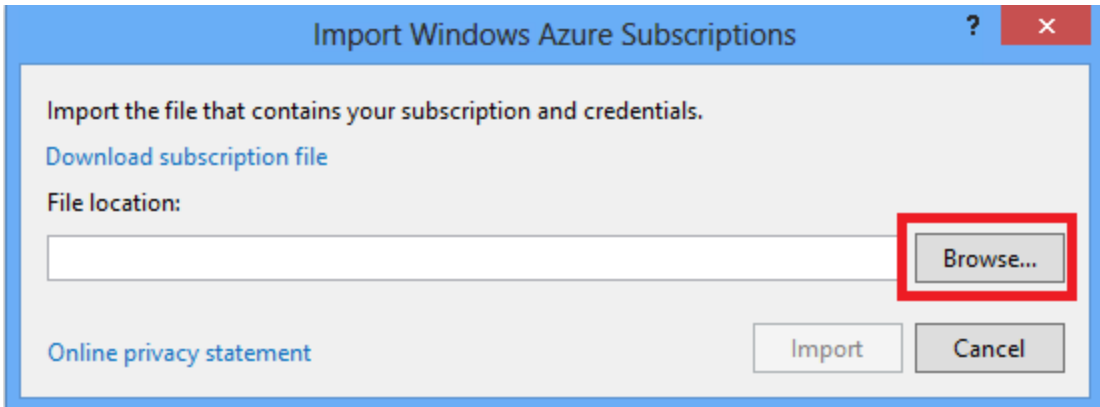


d. In your browser window, save the *.publishsettings* file.

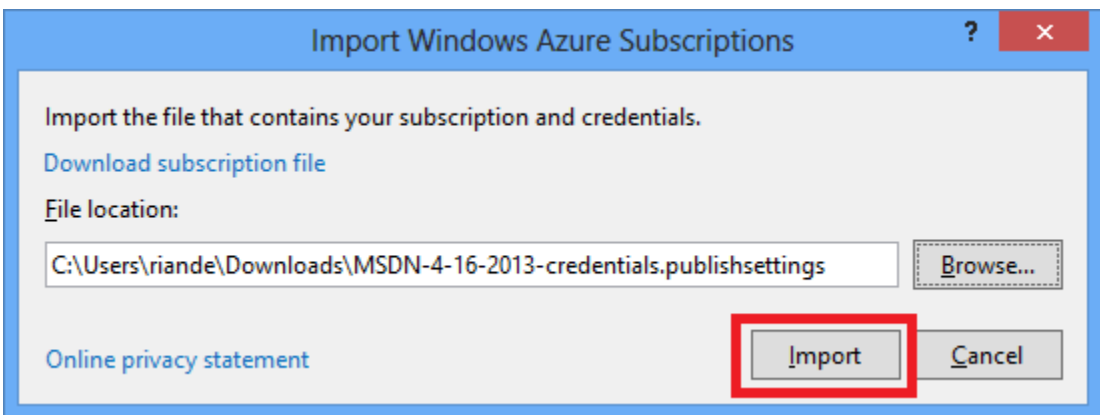


Security Note: The *publishsettings* file contains your credentials (unencoded) that are used to administer your Windows Azure subscriptions and services. The security best practice for this file is to store it temporarily outside your source directories (for example in the *Downloads* folder), and then delete it once the import has completed. A malicious user who gains access to the *.publishsettings* file can edit, create, and delete your Windows Azure services.

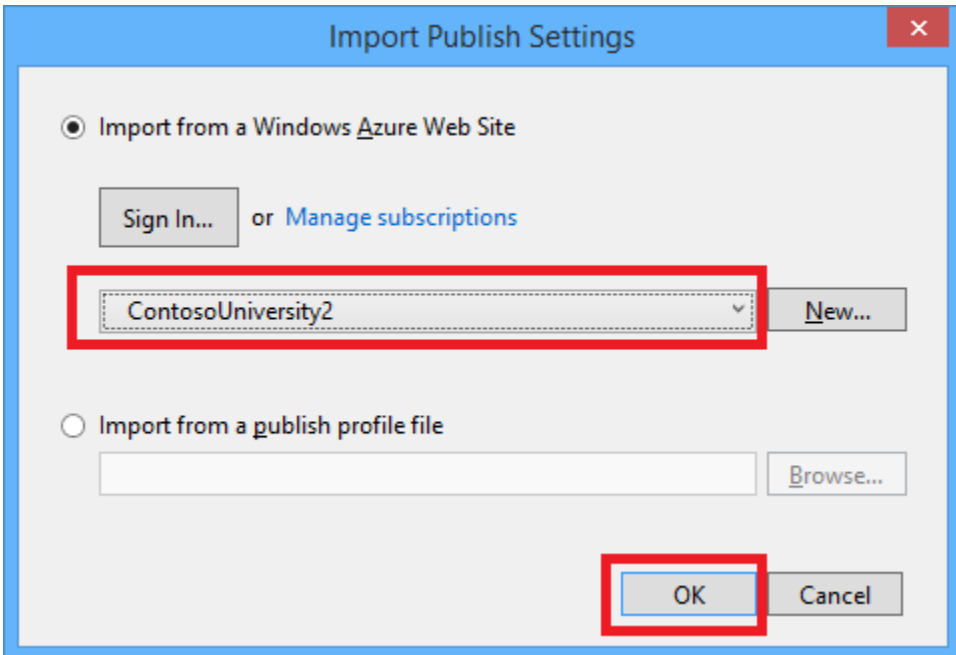
e. In the **Import Windows Azure Subscriptions** dialog box, click **Browse** and navigate to the *.publishsettings* file.



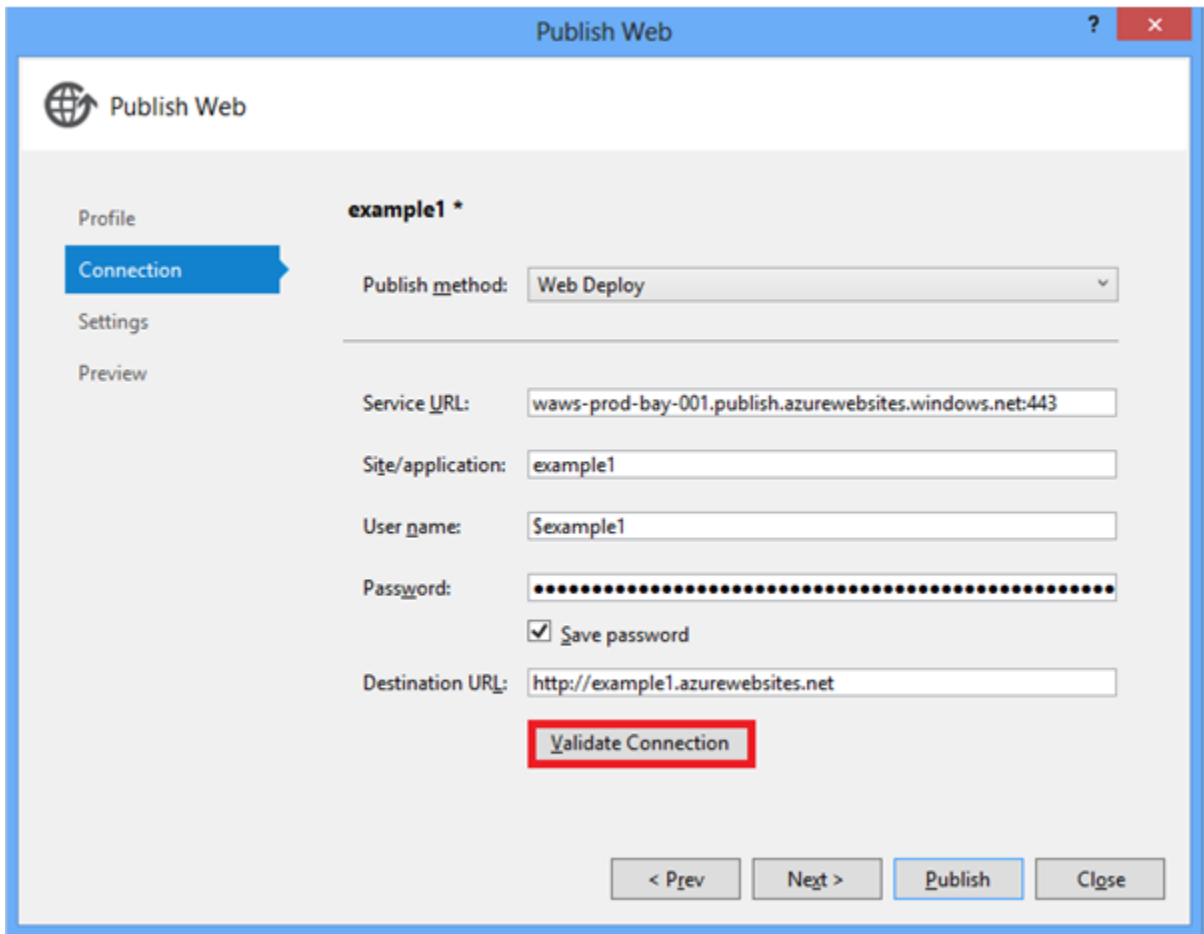
e. Click **Import**.



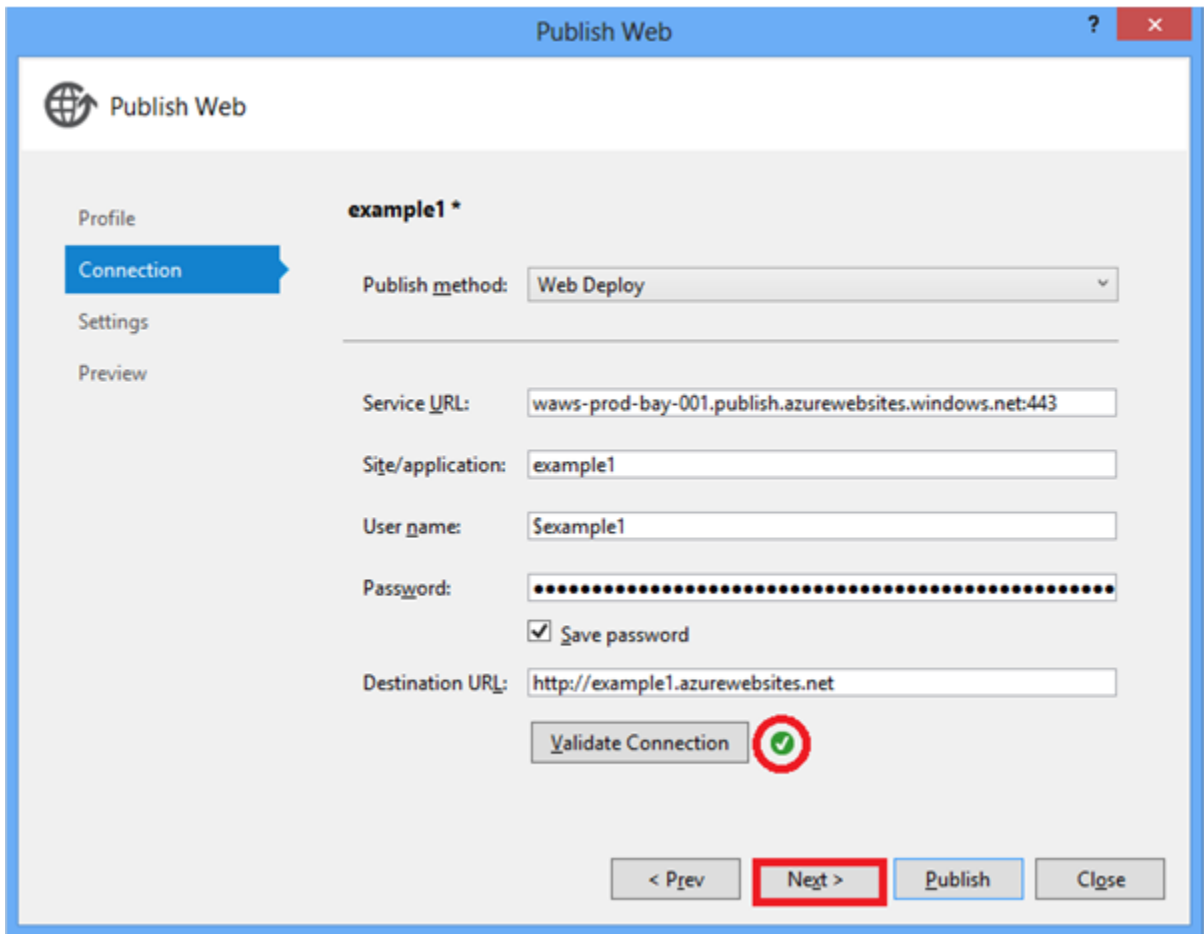
4. Close the **Manage Windows Azure Subscriptions** box.
5. In the **Import Publish Profile** dialog box, select **Import from a Windows Azure web site**, select your web site from the drop-down list, and then click **OK**.



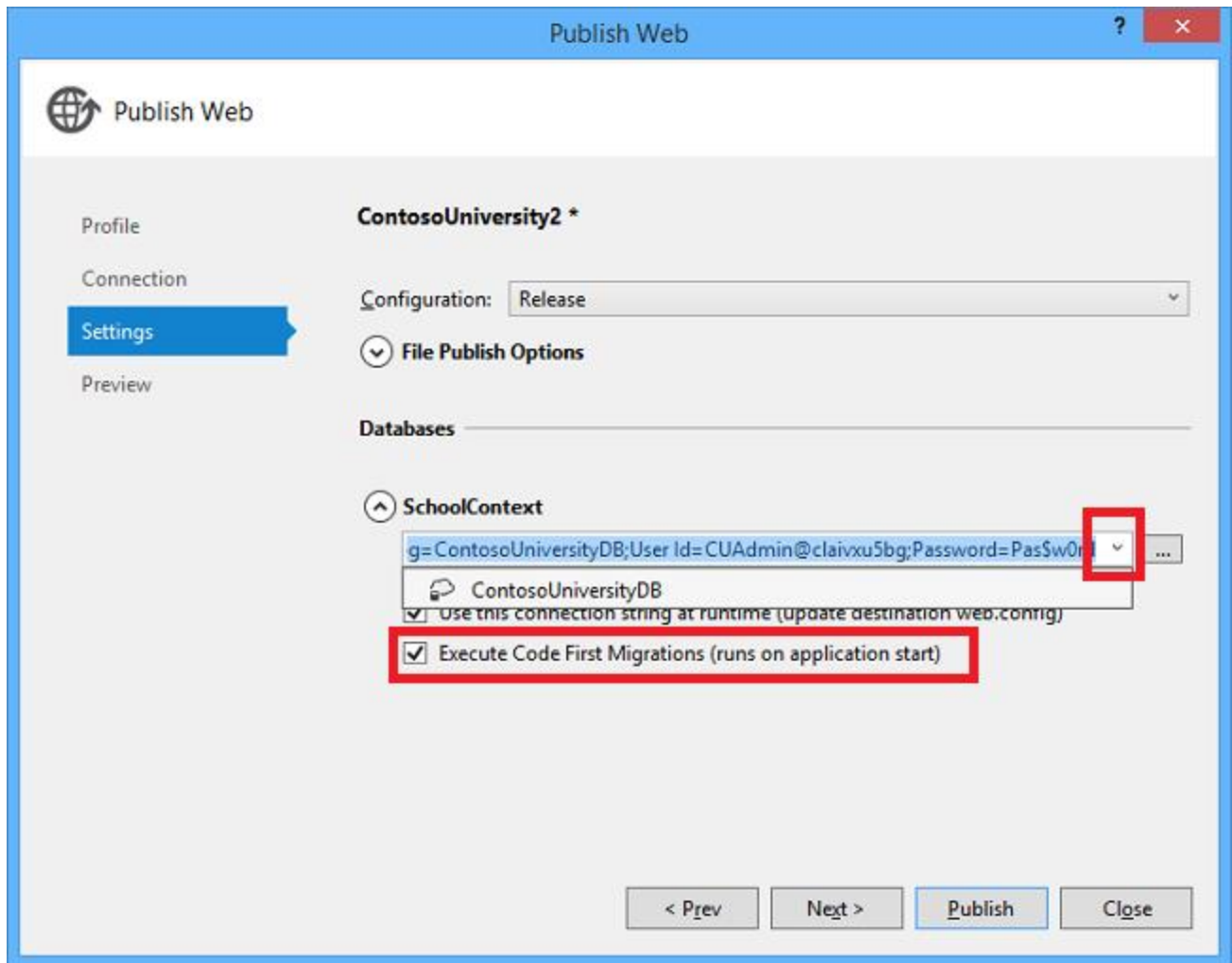
6. In the **Connection** tab, click **Validate Connection** to make sure that the settings are correct.



7. When the connection has been validated, a green check mark is shown next to the **Validate Connection** button. Click **Next**.



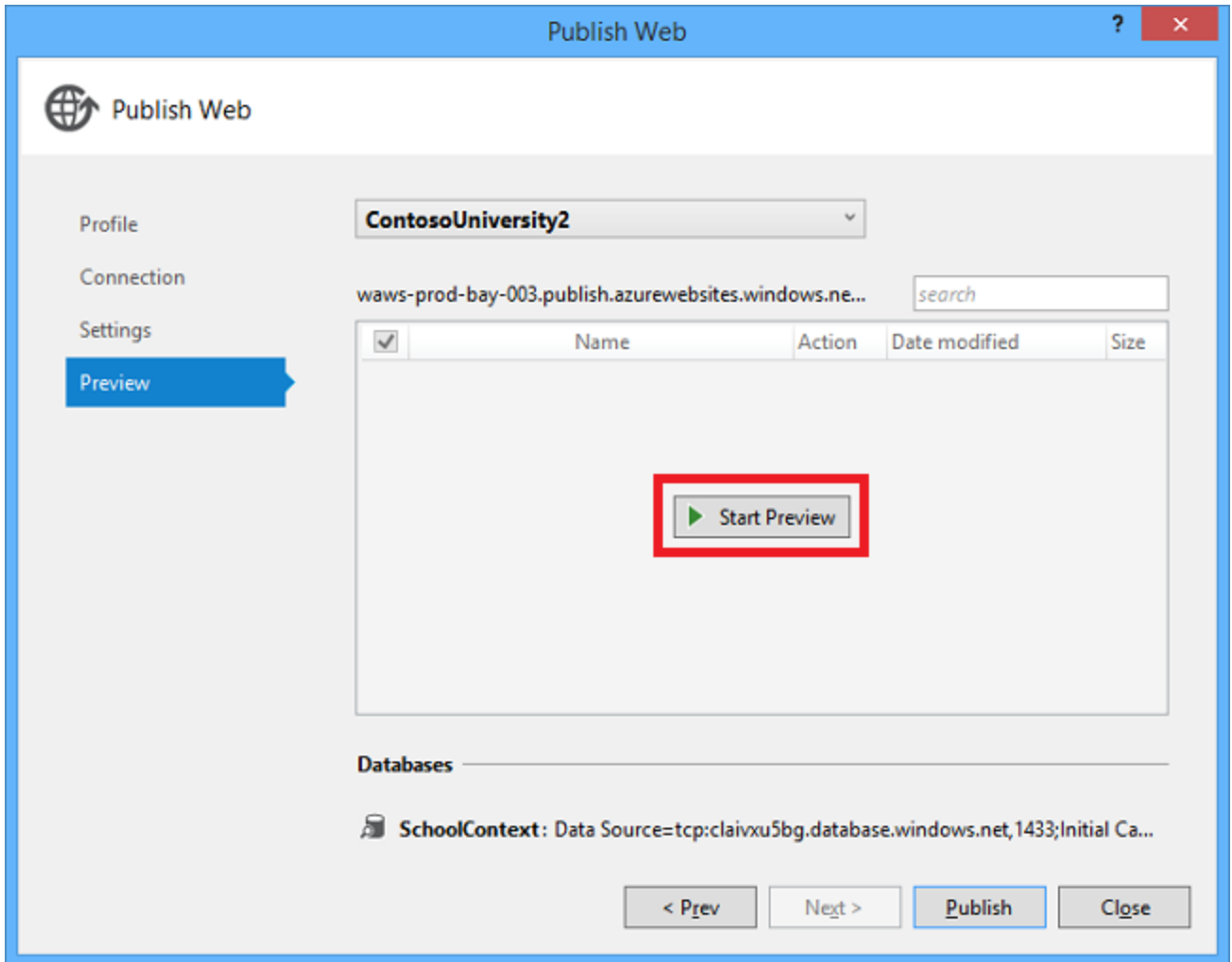
8. Open the **Remote connection string** drop-down list under **SchoolContext** and select the connection string for the database you created.
9. Select **Execute Code First Migrations (runs on application start)**.



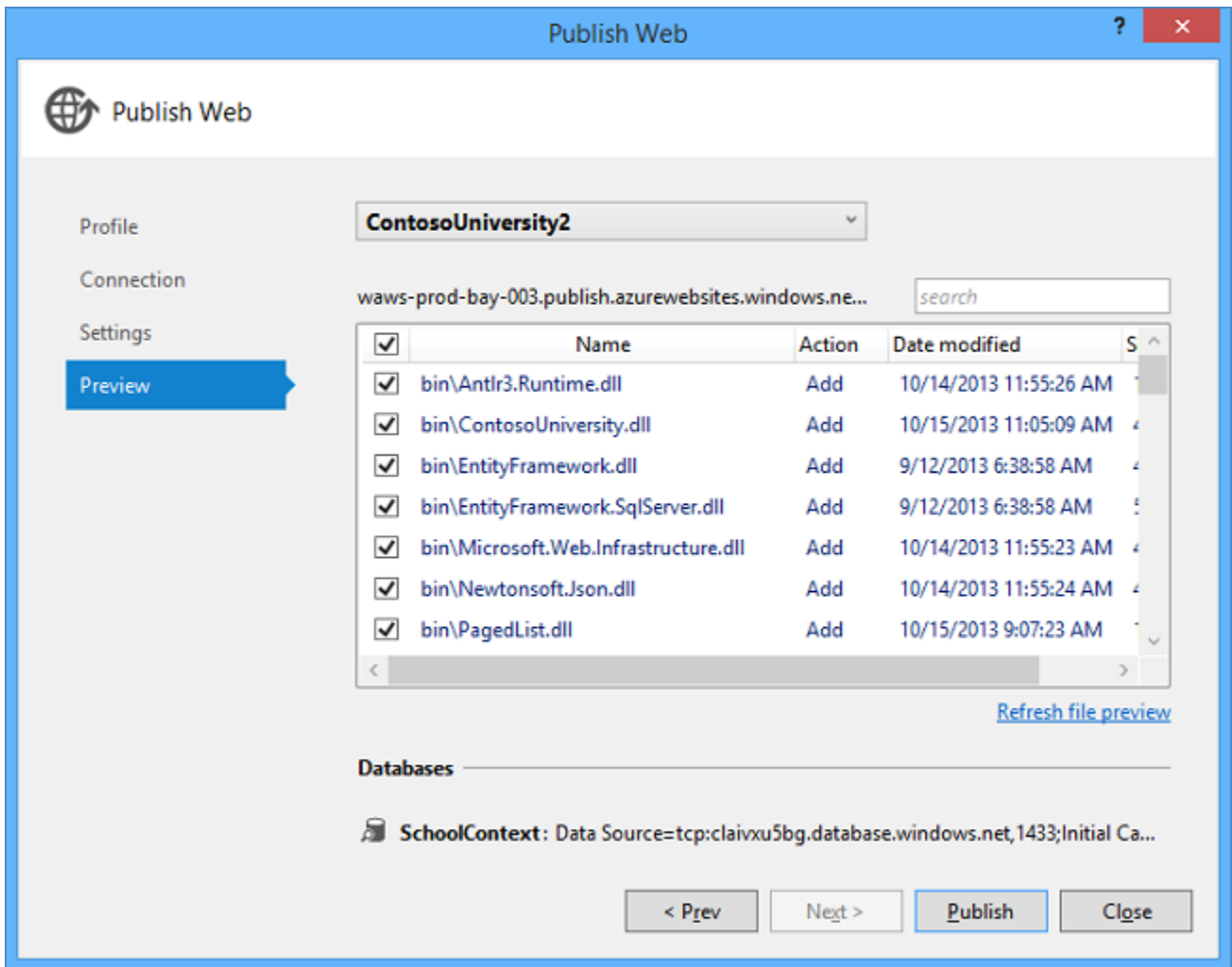
This setting causes the deployment process to automatically configure the application *Web.config* file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

10. Click **Next**.

11. In the **Preview** tab, click **Start Preview**.



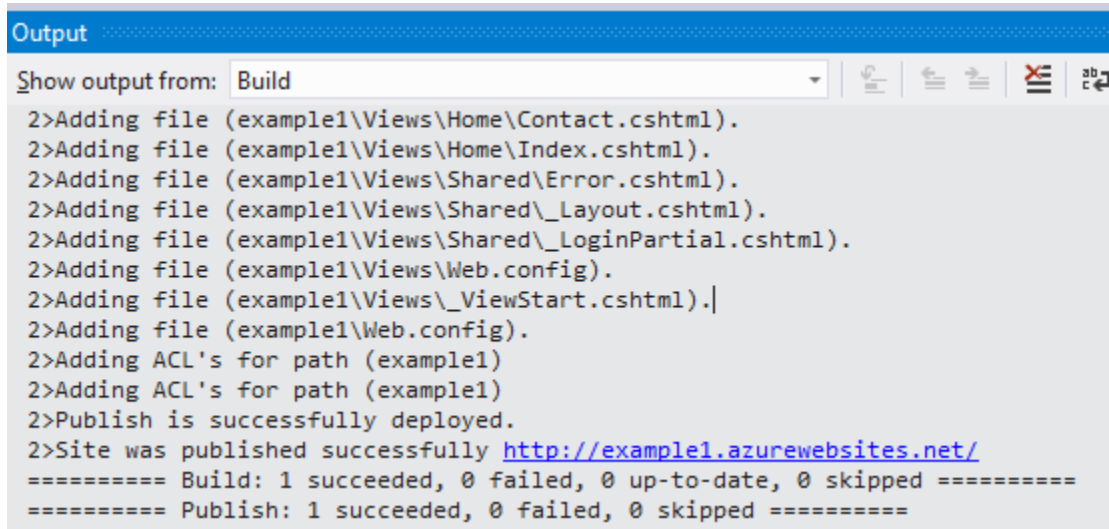
The tab displays a list of the files that will be copied to the server. Displaying the preview isn't required to publish the application but is a useful function to be aware of. In this case, you don't need to do anything with the list of files that is displayed. The next time you deploy this application, only the files that have changed will be in this list.



12. Click **Publish**.

Visual Studio begins the process of copying the files to the Windows Azure server.

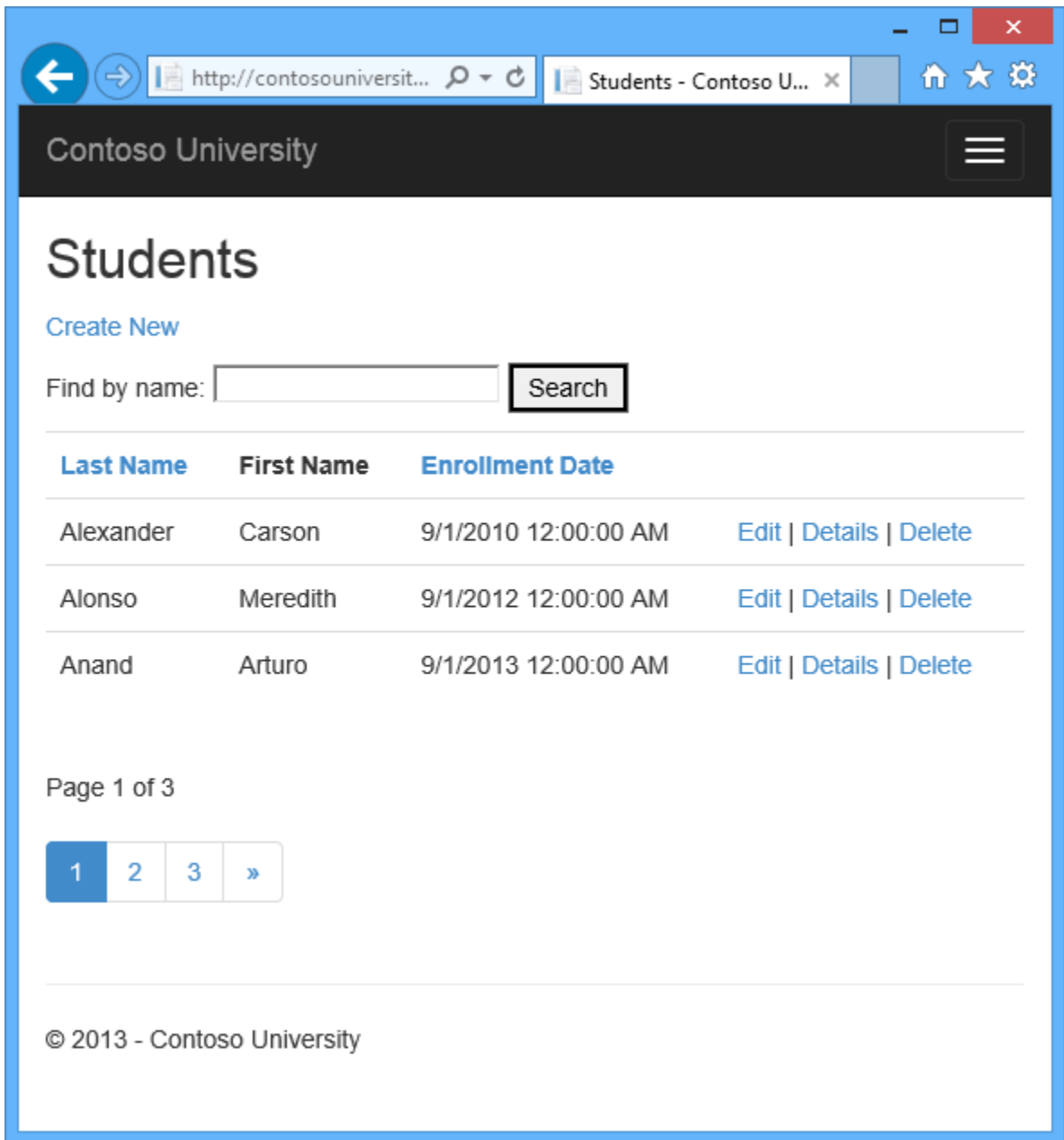
13. The **Output** window shows what deployment actions were taken and reports successful completion of the deployment.



The screenshot shows an 'Output' window with a blue header. Below the header is a dropdown menu set to 'Build' and a toolbar with icons for refresh, back, forward, and close. The main area contains the following text:

```
2>Adding file (example1\Views\Home\Contact.cshtml).
2>Adding file (example1\Views\Home\Index.cshtml).
2>Adding file (example1\Views\Shared\Error.cshtml).
2>Adding file (example1\Views\Shared\_Layout.cshtml).
2>Adding file (example1\Views\Shared\_LoginPartial.cshtml).
2>Adding file (example1\Views\Web.config).
2>Adding file (example1\Views\_ViewStart.cshtml).|
2>Adding file (example1\Web.config).
2>Adding ACL's for path (example1)
2>Adding ACL's for path (example1)
2>Publish is successfully deployed.
2>Site was published successfully http://example1.azurewebsites.net/
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Publish: 1 succeeded, 0 failed, 0 skipped =====
```

14. Upon successful deployment, the default browser automatically opens to the URL of the deployed web site.
The application you created is now running in the cloud. Click the Students tab.



At this point your *SchoolContext* database has been created in the Windows Azure SQL Database because you selected **Execute Code First Migrations (runs on app start)**. The *Web.config* file in the deployed web site has been changed so that the [MigrateDatabaseToLatestVersion](#) initializer runs the first time your code reads or writes data in the database (which happened when you selected the **Students** tab):

```

</runtime>
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFa
  <parameters>
    <parameter value="v11.0" />
  </parameters>
</defaultConnectionFactory>
<contexts>
  <context type="ContosoUniversity.Models.SchoolContext, ContosoUniversity">
    <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
      [ContosoUniversity.Models.SchoolContext, ContosoUniversity],
      [ContosoUniversity.Migrations.Configuration, ContosoUniversi
      EntityFramework, PublicKeyToken=b77a5c561934e089]">
      <parameters>
        <parameter value="SchoolContext_DatabasePublish"/>
      </parameters>
    </databaseInitializer>
  </context>
</contexts>
</entityFramework>
</configuration>

```

The deployment process also created a new connection string (*SchoolContext_DatabasePublish*) for Code First Migrations to use for updating the database schema and seeding the database.

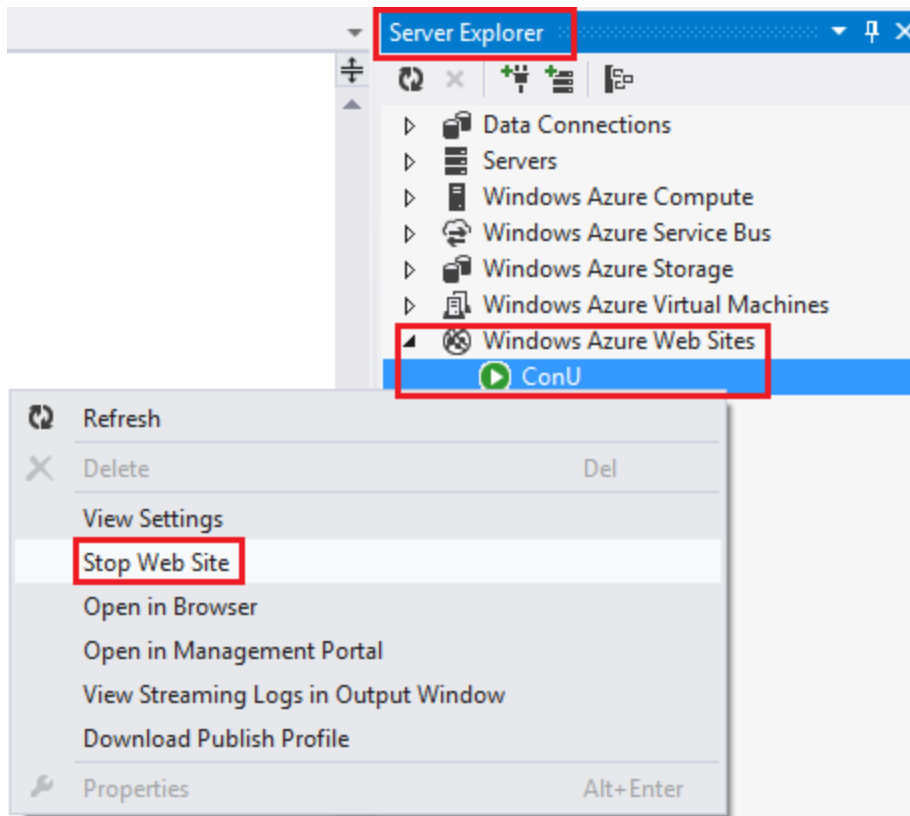
```

<connectionStrings>
  <add name="SchoolContext" connectionString="Data Source=tcp:d015leqjqx.database.winc
  <add name="SchoolContext_DatabasePublish" connectionString="Data Source=tcp:d015leqj
</connectionStrings>

```

You can find the deployed version of the Web.config file on your own computer in *ContosoUniversity\obj\Release\Package\PackageTmp\Web.config*. You can access the deployed *Web.config* file itself by using FTP. For instructions, see [ASP.NET Web Deployment using Visual Studio: Deploying a Code Update](#). Follow the instructions that start with "To use an FTP tool, you need three things: the FTP URL, the user name, and the password."

Note: The web app doesn't implement security, so anyone who finds the URL can change the data. For instructions on how to secure the web site, see [Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site](#). You can prevent other people from using the site by using the Windows Azure Management Portal or **Server Explorer** in Visual Studio to stop the site.



Advanced Migrations Scenarios

If you deploy a database by running migrations automatically as shown in this tutorial, and you are deploying to a web site that runs on multiple servers, you could get multiple servers trying to run migrations at the same time. Migrations are atomic, so if two servers try to run the same migration, one will succeed and the other will fail (assuming the operations can't be done twice). In that scenario if you want to avoid those issues, you can call migrations manually and set up your own code so that it only happens once. For more information, see [Running and Scripting Migrations from Code](#) on Rowan Miller's blog and [Migrate.exe](#) (for executing migrations from the command line) on MSDN.

For information about other migrations scenarios, see [Migrations Screencast Series](#).

Code First Initializers

In the deployment section you saw the [MigrateDatabaseToLatestVersion](#) initializer being used. Code First also provides other initializers, including [CreateDatabaseIfNotExists](#) (the default), [DropCreateDatabaseIfModelChanges](#) (which you used earlier) and [DropCreateDatabaseAlways](#). The `DropCreateAlways` initializer can be useful for setting up conditions for unit tests. You can also write your own initializers, and you can call an initializer explicitly if you don't want to wait until the application reads from or writes to the database. At the time this tutorial is being written in November, 2013, you can only use the Create and

DropCreate initializers before you enable migrations. The Entity Framework team is working on making these initializers usable with migrations as well.

For more information about initializers, see [Understanding Database Initializers in Entity Framework Code First](#) and chapter 6 of the book [Programming Entity Framework: Code First](#) by Julie Lerman and Rowan Miller.

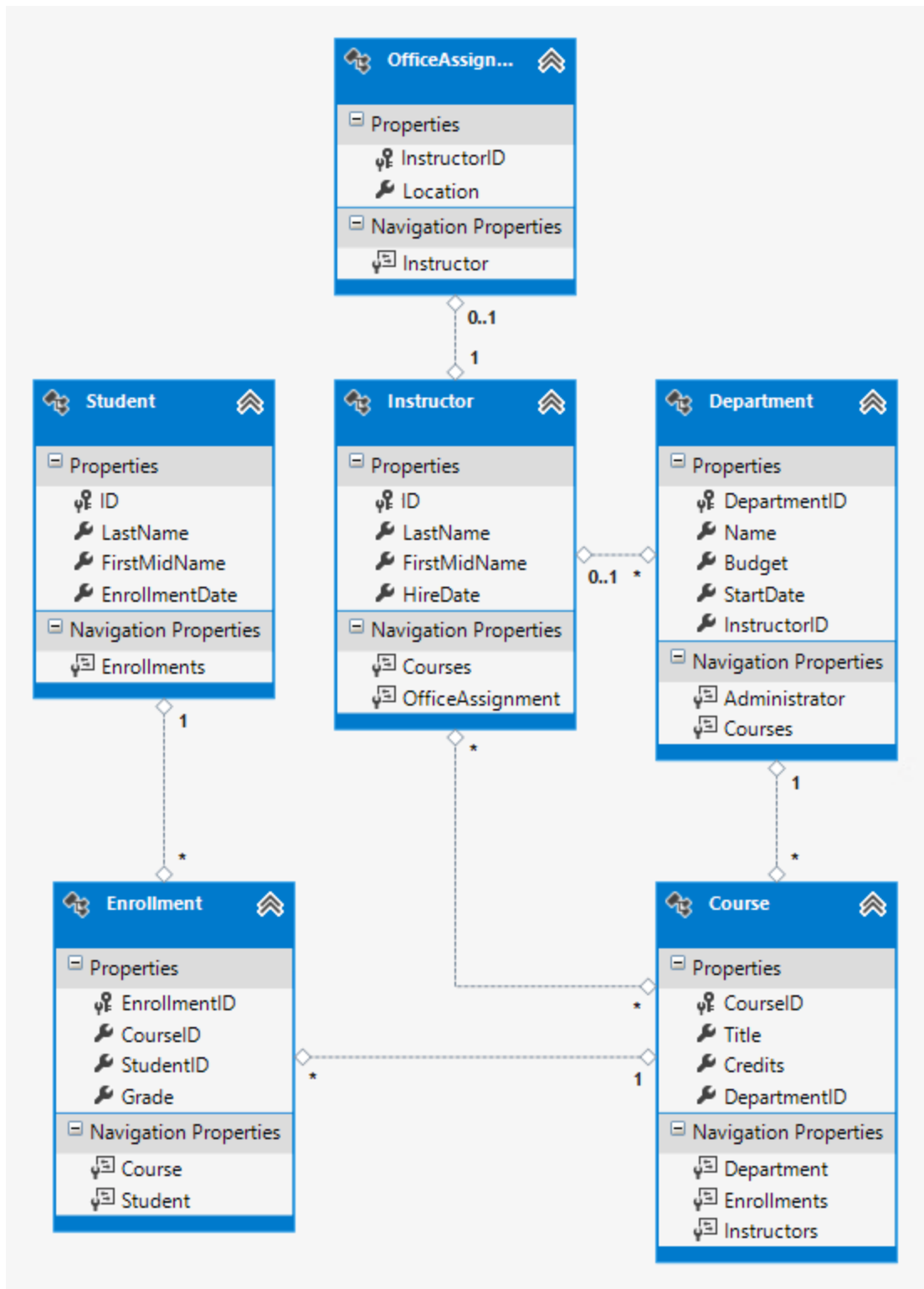
Summary

In this tutorial you've seen how to enable migrations and deploy the application. In the next tutorial you'll begin looking at more advanced topics by expanding the data model.

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

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

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



Customize the Data Model by Using Attributes

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

The `DataType` Attribute

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

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

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

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

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

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

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

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

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

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

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

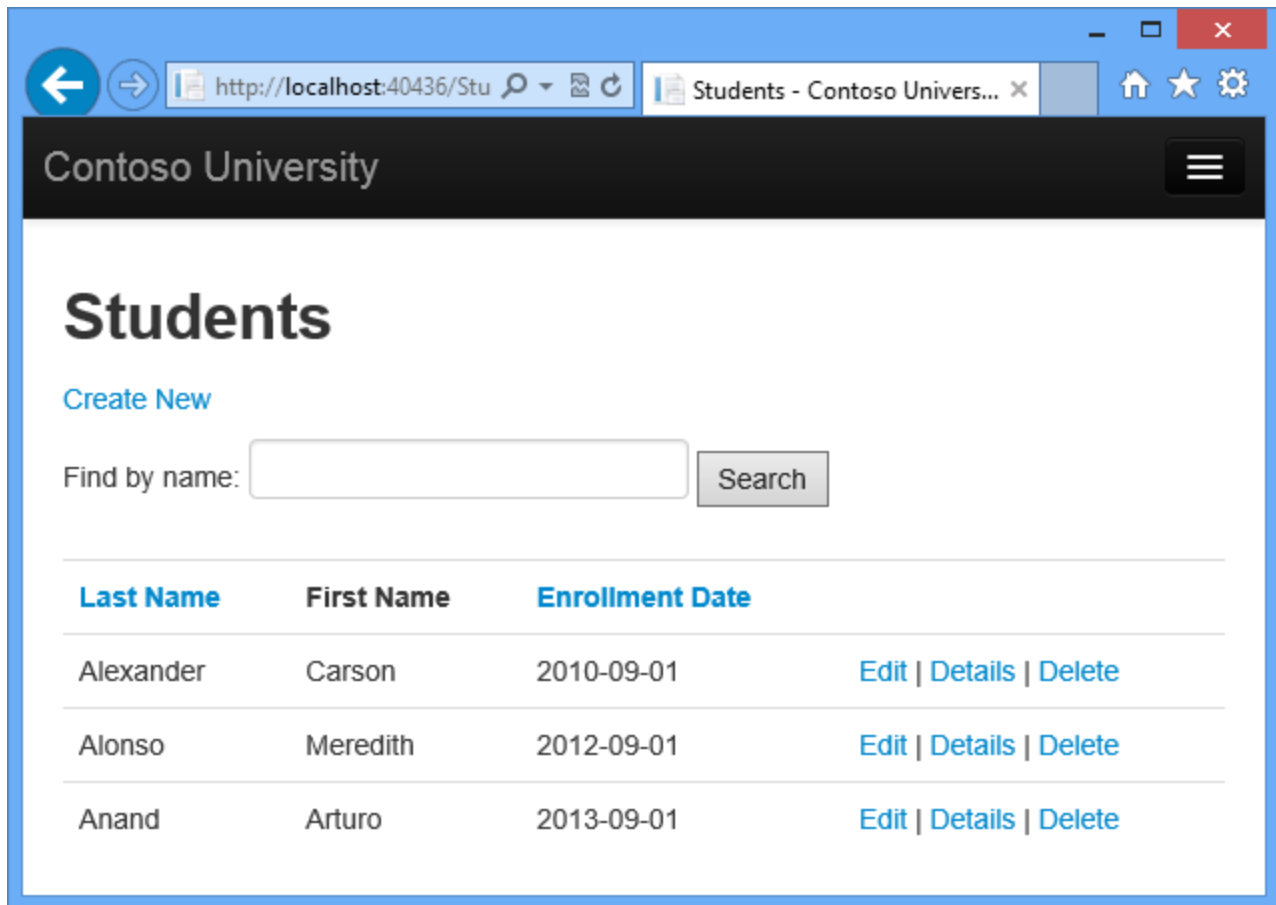
`DisplayFormat`:

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

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

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

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



The `StringLengthAttribute`

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

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

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

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

```

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

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

```

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

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

The [MaxLength](#) attribute provides similar functionality to the [StringLength](#) attribute but doesn't provide client side validation.

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

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

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

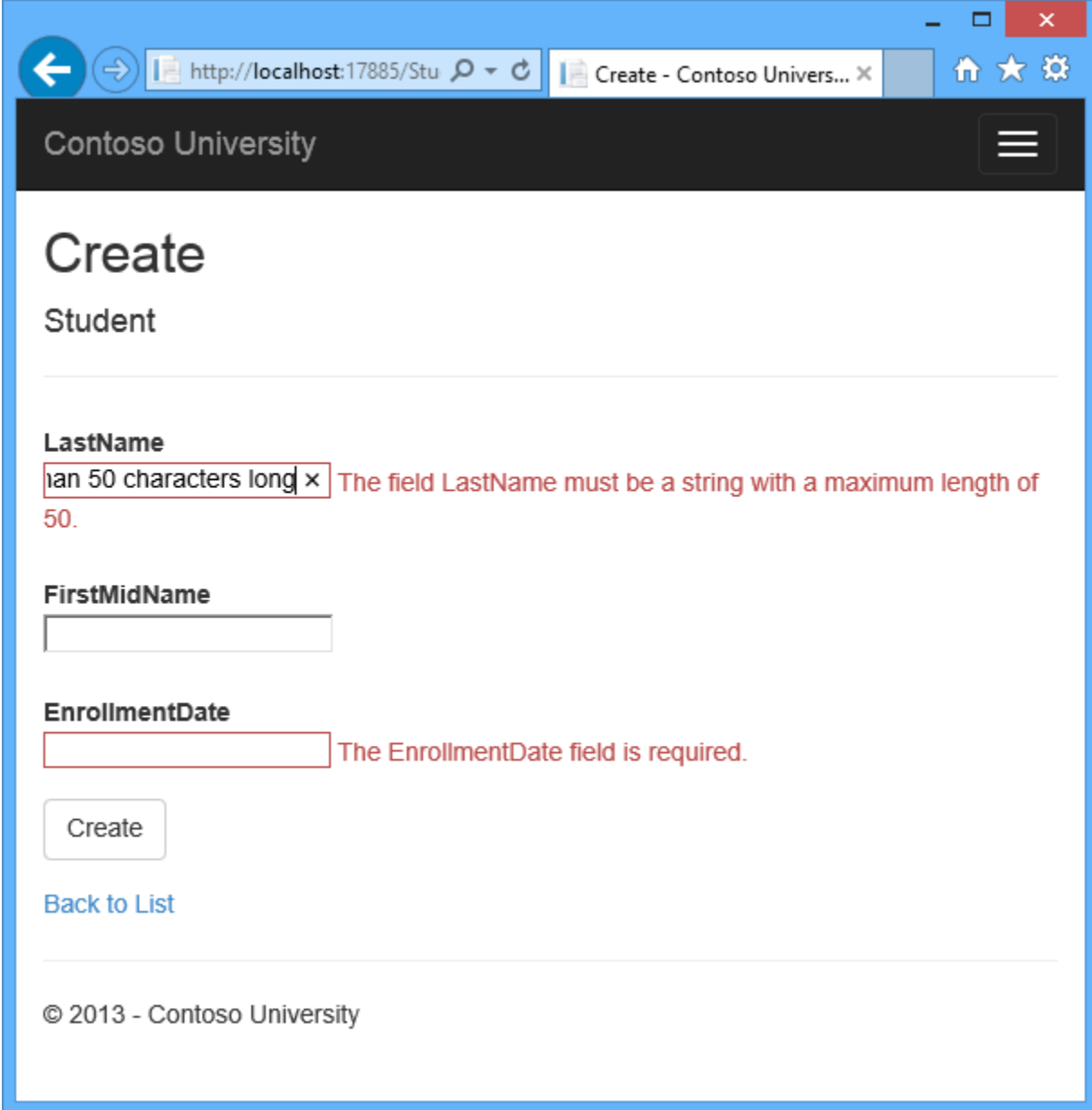
In the Package Manager Console (PMC), enter the following commands:

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

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

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

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



The screenshot shows a web browser window with the URL `http://localhost:17885/Stu`. The page title is "Create - Contoso Univers...". The page content includes a header for "Contoso University" and a main heading "Create Student". There are three input fields: "LastName", "FirstMidName", and "EnrollmentDate". The "LastName" field contains the text "ian 50 characters long" and has a red border with an error message: "The field LastName must be a string with a maximum length of 50." The "EnrollmentDate" field is empty and has a red border with an error message: "The EnrollmentDate field is required." Below the fields is a "Create" button and a "Back to List" link. The footer of the page reads "© 2013 - Contoso University".

The Column Attribute

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

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

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

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

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

        public DateTime EnrollmentDate { get; set; }

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

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

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

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

Name	Data Type	Allow Nulls	Default
ID	int	<input type="checkbox"/>	
LastName	nvarchar(50)	<input checked="" type="checkbox"/>	
FirstName	nvarchar(50)	<input checked="" type="checkbox"/>	
EnrollmentDate	datetime	<input type="checkbox"/>	

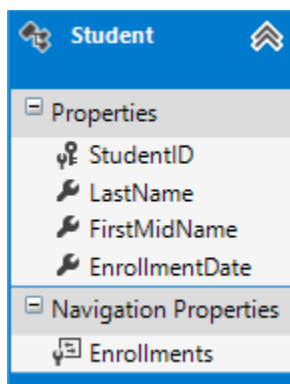
The following image shows the original column name as it was before you applied the first two migrations. In addition to the column name changing from `FirstMidName` to `FirstName`, the two name columns have changed from `MAX` length to 50 characters.

Name	Data Type	Allow Nulls	Default
ID	int	<input type="checkbox"/>	
LastName	nvarchar(MAX)	<input checked="" type="checkbox"/>	
FirstMidName	nvarchar(MAX)	<input checked="" type="checkbox"/>	
EnrollmentDate	datetime	<input type="checkbox"/>	

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

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

Complete Changes to the Student Entity



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

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

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

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

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

The Required Attribute

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

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

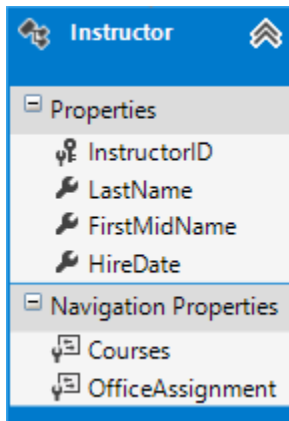
The Display Attribute

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

The FullName Calculated Property

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

Create the Instructor Entity



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

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

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

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

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

```

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

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

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

```

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

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

```

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

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

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

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

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

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

```

The Courses and OfficeAssignment Navigation Properties

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

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

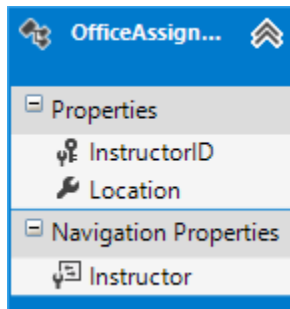
An instructor can teach any number of courses, so `Courses` is defined as a collection of `Course` entities.

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

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

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

Create the OfficeAssignment Entity



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

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

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

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

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

The Key Attribute

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

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

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

The ForeignKey Attribute

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

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

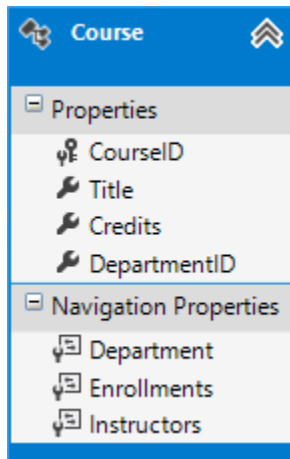
Later in the tutorial you'll see how to configure this relationship with the fluent API.

The Instructor Navigation Property

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

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

Modify the Course Entity



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

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

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

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

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

        public int DepartmentID { get; set; }

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

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

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

The DatabaseGenerated Attribute

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

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

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

Foreign Key and Navigation Properties

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

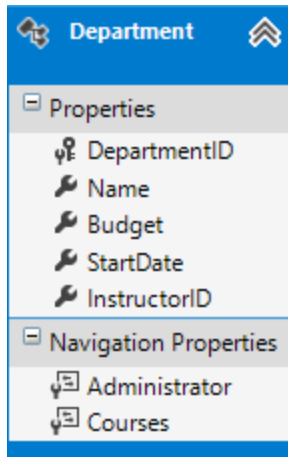
- A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.
- ```
public int DepartmentID { get; set; }
public virtual Department Department { get; set; }
```
- A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

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

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

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

## Create the Department Entity



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

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

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

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

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

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

 public int? InstructorID { get; set; }

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

## The Column Attribute

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

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

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

## Foreign Key and Navigation Properties

The foreign key and navigation properties reflect the following relationships:

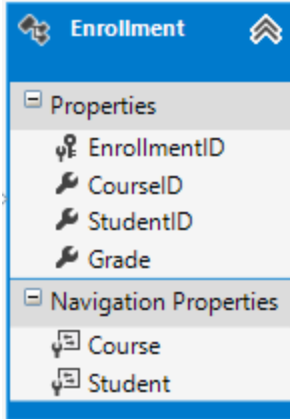
- A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the `Instructor` entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation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

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

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

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

Modify the Enrollment Entity



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

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

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

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

Foreign Key and Navigation Properties

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

- An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

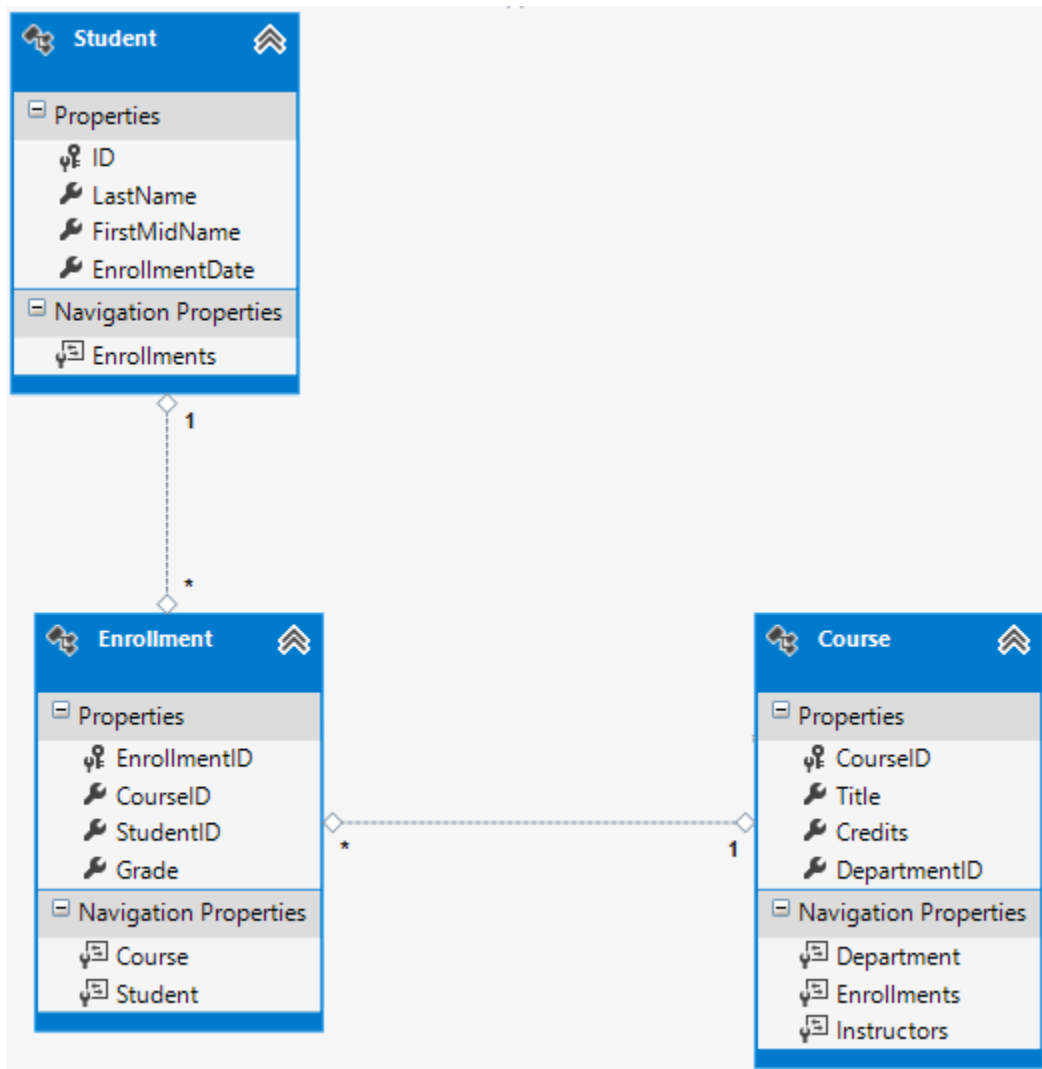

```
public int CourseID { get; set; }
public virtual Course Course { get; set; }
```
- An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:


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

Many-to-Many Relationships

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

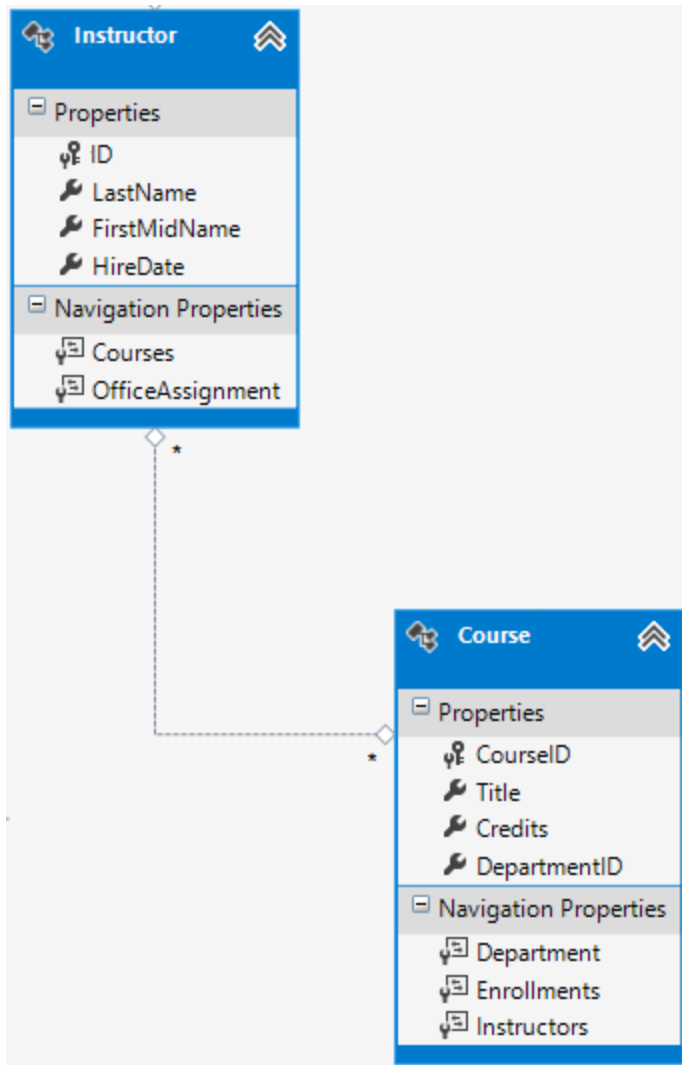
The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the [Entity Framework Power Tools](#); creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)



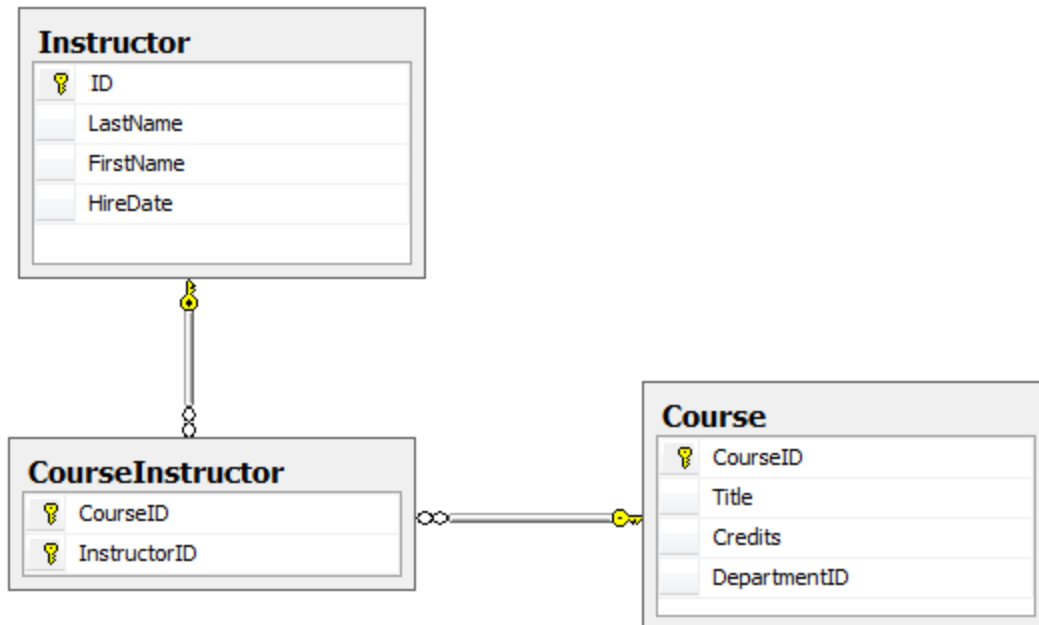
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

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

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



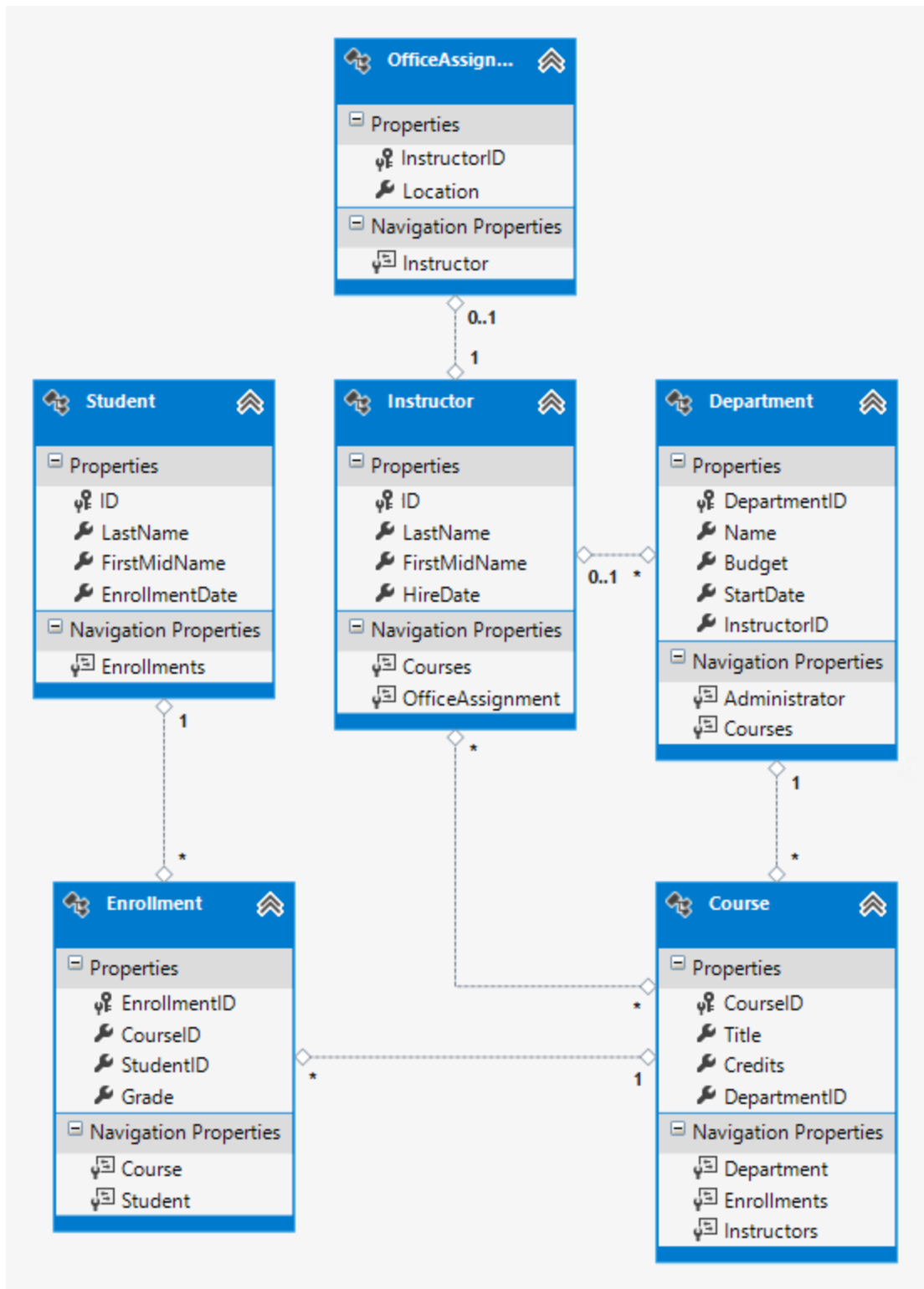
A join table is required in the database, however, as shown in the following database diagram:



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

Entity Diagram Showing Relationships

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



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

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

Customize the Data Model by adding Code to the Database Context

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

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

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

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

To add the new entities to the data model and perform database mapping that you didn't do by using attributes, replace the code in `DAL\SchoolContext.cs` with the following code:

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

namespace ContosoUniversity.DAL
{
    public class SchoolContext : DbContext
    {
        public DbSet<Course> Courses { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

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

            modelBuilder.Entity<Course>()
```

```

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

```

The new statement in the [OnModelCreating](#) method configures the many-to-many join table:

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

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

```

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

```

For information about what "fluent API" statements are doing behind the scenes, see the [Fluent API](#) blog post.

Seed the Database with Test Data

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

```

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

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

```

```

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

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

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

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

```

```

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

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

```

```

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

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

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

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

    context.SaveChanges();

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

```



```

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

```

```

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

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

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

```

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

```

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

```

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

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

Add a Migration and Update the Database

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

```
add-Migration ComplexDataModel
```

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

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

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

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

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

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

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

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

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

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

```
update-database
```

Note: It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. The simplest approach is to rename the database in `Web.config` file. The following example shows the name changed to `CU_Test`:

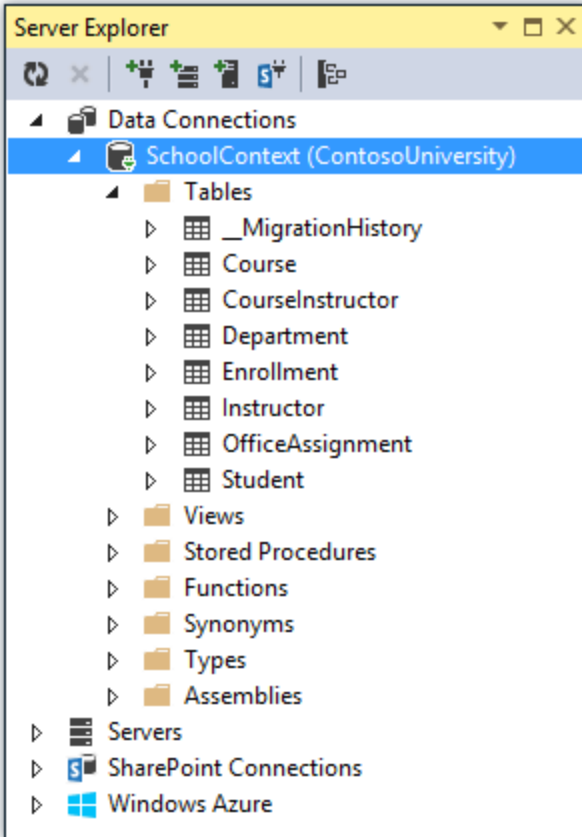
```
<add name="SchoolContext" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=CU_Test;Integrated
Security=SSPI;"
providerName="System.Data.SqlClient" />
```

With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see [How to Drop a Database from Visual Studio 2012](#).

If that fails, another thing you can try is re-initialize the database by entering the following command in the PMC:

```
update-database -TargetMigration:0
```

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



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

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

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

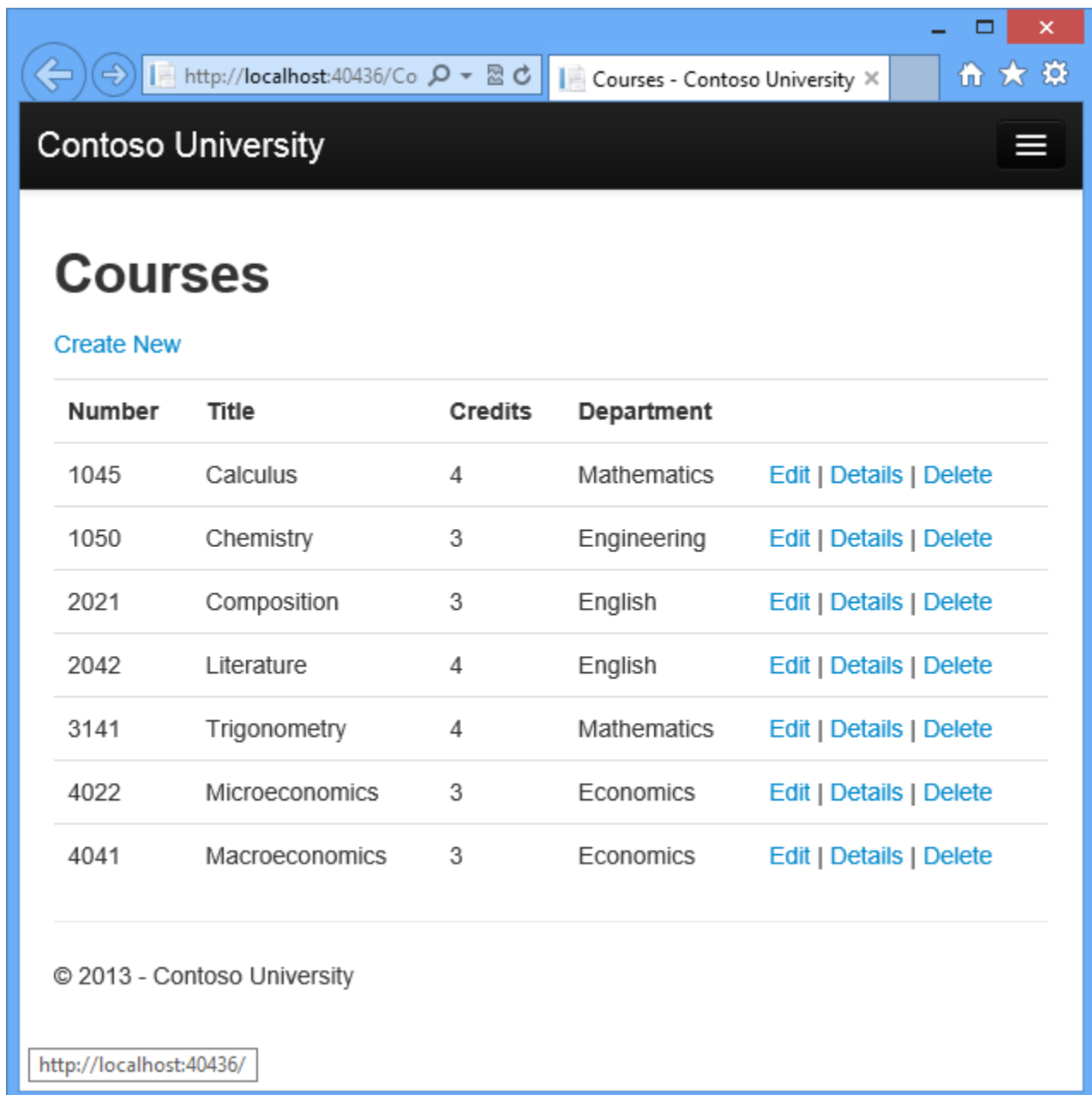
Summary

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

Reading Related Data with the Entity Framework in an ASP.NET MVC Application

In the previous tutorial you completed the School data model. In this tutorial you'll read and display related data — that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





http://localhost:40436/Ins

Instructors - Contoso Unive... x



Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	
Abercrombie	Kim	3/11/1995		Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	Select Edit Details Delete
Zheng	Roger	2/12/2004		Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
------	-------

Lazy, Eager, and Explicit Loading of Related Data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved. The `DbContext` class enables lazy loading by default.

```
departments = context.Departments
foreach (Department d in departments) ← Query: all Department rows
{
    foreach (Course c in d.Courses) ← Query: Course rows related to
    {                                     Department d
        courseList.Add(d.Name + c.Title);
    }
}
```

- *Eager loading*. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the `Include` method.

```
departments = context.Departments.Include(x => x.Courses)
foreach (Department d in departments) ← Query: all Department
{                                     rows and related
    foreach (Course c in d.Courses)    Course rows
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

- *Explicit loading*. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually by getting the object state manager entry for an entity and calling the [Collection.Load](#) method for collections or the [Reference.Load](#) method for properties that hold a single entity. (In the following example, if you wanted to load the `Administrator` navigation property, you'd replace `Collection(x => x.Courses)` with `Reference(x => x.Administrator)`.) Typically you'd use explicit loading only when you've turned lazy loading off.

```

departments = context.Departments.ToList();
foreach (Department d in departments) ← Query: all Department rows
{
    context.Entry(d).Collection(x => x.Courses).Load(); ← Query: Course rows
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

← Query: Course rows related to Department d

Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as *deferred loading*.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, in the above examples, suppose that each department has ten related courses. The eager loading example would result in just a single (join) query and a single round trip to the database. The lazy loading and explicit loading examples would both result in eleven queries and eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios lazy loading is more efficient. Eager loading might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, lazy loading might perform better because eager loading would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Lazy loading can mask code that causes performance problems. For example, code that doesn't specify eager or explicit loading but processes a high volume of entities and uses several navigation properties in each iteration might be very inefficient (because of many round trips to the database). An application that performs well in development using an on premise SQL server might have performance problems when moved to Windows Azure SQL Database due to the increased latency and lazy loading. Profiling the database queries with a realistic test load will help you determine if lazy loading is appropriate. For more information see [Demystifying Entity Framework Strategies: Loading Related Data](#) and [Using the Entity Framework to Reduce Network Latency to SQL Azure](#).

Disable lazy loading before serialization

If you leave lazy loading enabled during serialization, you can end up querying significantly more data than you intended. Serialization generally works by accessing each property on an instance of a type. Property access triggers lazy loading, and those lazy loaded entities are serialized. The serialization process then accesses each property of the lazy-loaded entities, potentially causing even more lazy loading and serialization. To prevent this run-away chain reaction, turn lazy loading off before you serialize an entity.

Serialization can also be complicated by the proxy classes that the Entity Framework uses, as explained in the [Advanced Scenarios tutorial](#).

One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the [Using Web API with Entity Framework](#) tutorial.

If you don't use DTOs, you can disable lazy loading and avoid proxy issues by [disabling proxy creation](#).

Here are some other [ways to disable lazy loading](#):

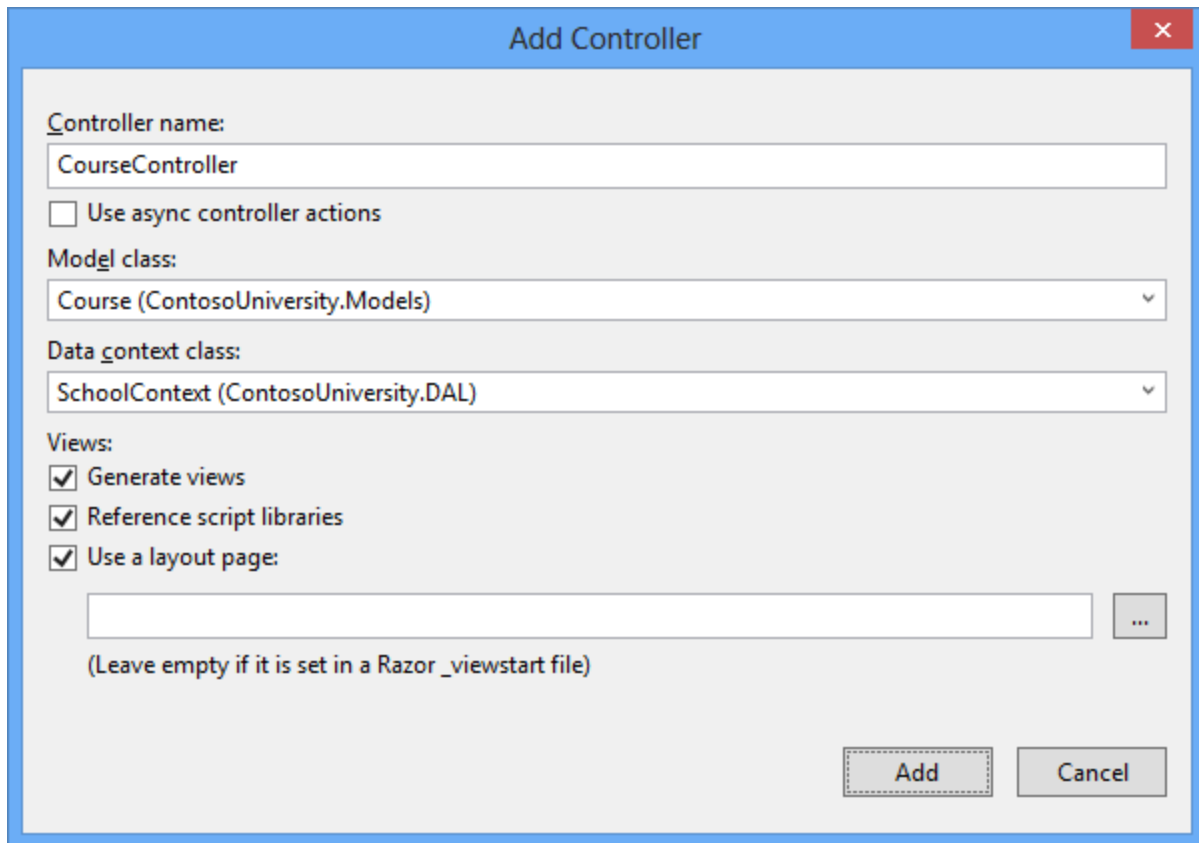
- For specific navigation properties, omit the `virtual` keyword when you declare the property.
- For all navigation properties, set `LazyLoadingEnabled` to `false`, put the following code in the constructor of your context class:

```
this.Configuration.LazyLoadingEnabled = false;
```

Create a Courses Page That Displays Department Name

The `Course` entity includes a navigation property that contains the `Department` entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the `Name` property from the `Department` entity that is in the `Course.Department` navigation property.

Create a controller named `CourseController` for the `Course` entity type, using the same options for the **MVC 5 Controller with views, using Entity Framework** scaffolder that you did earlier for the `Student` controller, as shown in the following illustration:



Open *Controllers\CourseController.cs* and look at the `Index` method:

```
public ActionResult Index()
{
    var courses = db.Courses.Include(c => c.Department);
    return View(courses.ToList());
}
```

The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Open *Views\Course\Index.cshtml* and replace the template code with the following code. The changes are highlighted:

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

@{
    ViewBag.Title = "Courses";
}

<h2>Courses</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
```

```

<table class="table">
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.CourseID)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Credits)
    </th>
    <th>
      Department
    </th>
  </tr>
  <tr>
    <td>
      @Html.DisplayFor(modelItem => item.CourseID)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Title)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Credits)
    </td>
    <td>
      @Html.DisplayFor(modelItem => item.Department.Name)
    </td>
    <td>
      @Html.ActionLink("Edit", "Edit", new { id=item.CourseID }) |
      @Html.ActionLink("Details", "Details", new { id=item.CourseID })
      |
      @Html.ActionLink("Delete", "Delete", new { id=item.CourseID })
    </td>
  </tr>
</table>

```

You've made the following changes to the scaffolded code:

- Changed the heading from **Index** to **Courses**.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Moved the **Department** column to the right side and changed its heading. The scaffolder correctly chose to display the `Name` property from the `Department` entity, but here in the `Course` page the column heading should be **Department** rather than **Name**.

Notice that for the `Department` column, the scaffolded code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

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

Run the page (select the **Courses** tab on the Contoso University home page) to see the list with department names.

The screenshot shows a web browser window with the URL `http://localhost:40436/Co`. The page title is "Courses - Contoso University". The main content area features a "Contoso University" header, a "Courses" title, and a "Create New" link. Below is a table of courses:

Number	Title	Credits	Department	
1045	Calculus	4	Mathematics	Edit Details Delete
1050	Chemistry	3	Engineering	Edit Details Delete
2021	Composition	3	English	Edit Details Delete
2042	Literature	4	English	Edit Details Delete
3141	Trigonometry	4	Mathematics	Edit Details Delete
4022	Microeconomics	3	Economics	Edit Details Delete
4041	Macroeconomics	3	Economics	Edit Details Delete

© 2013 - Contoso University

`http://localhost:40436/`

Create an Instructors Page That Shows Courses and Enrollments

In this section you'll create a controller and view for the `Instructor` entity in order to display the Instructors page:



http://localhost:40436/Ins

Instructors - Contoso Unive... x



Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	
Abercrombie	Kim	3/11/1995		Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	Select Edit Details Delete
Zheng	Roger	2/12/2004		Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
------	-------

This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, lazy loading might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll add explicit loading for `Enrollment` entities and their related `Student` entities. (Explicit loading isn't necessary because lazy loading is enabled, but this shows how to do explicit loading.)

Create a View Model for the Instructor Index View

The Instructors page shows three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

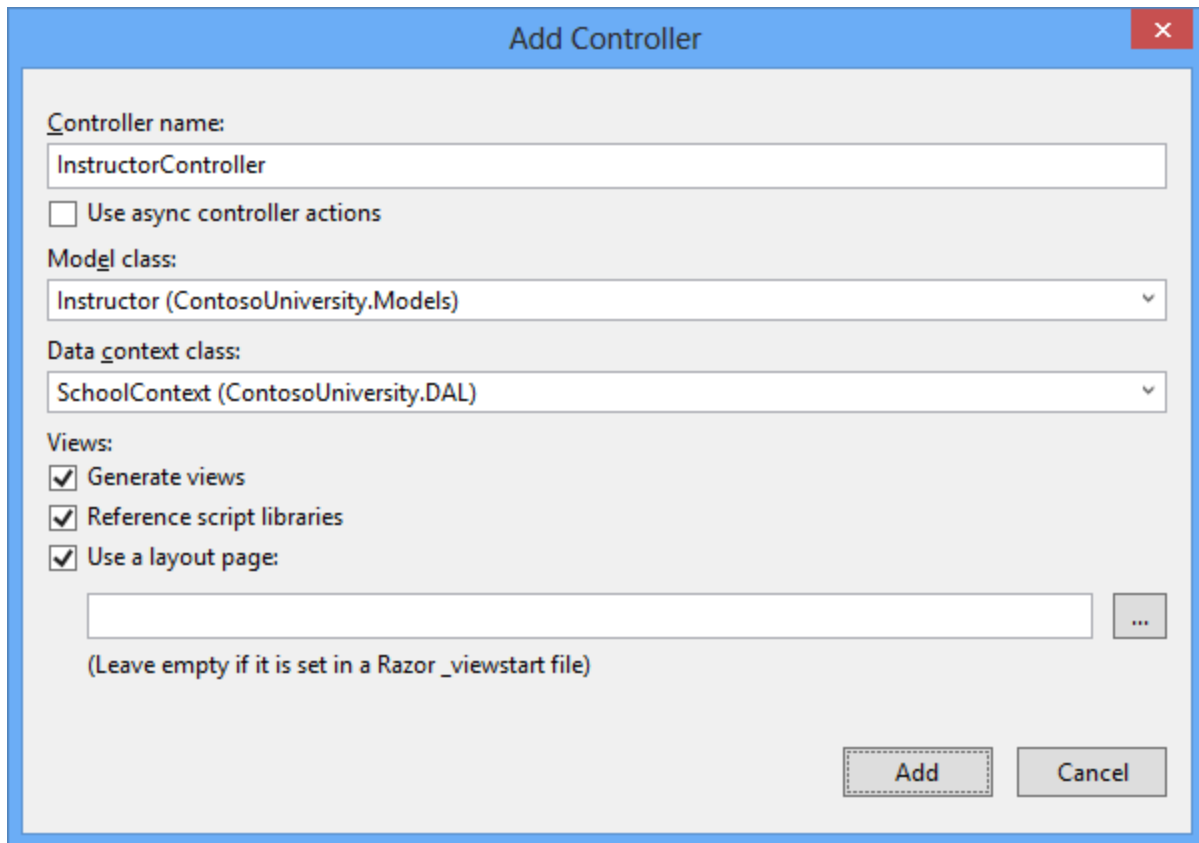
In the `ViewModels` folder, create `InstructorIndexData.cs` and replace the existing code with the following code:

```
using System.Collections.Generic;
using ContosoUniversity.Models;

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

Create the Instructor Controller and Views

Create an `InstructorController` controller with EF read/write actions as shown in the following illustration:



Open *Controllers\InstructorController.cs* and add a `using` statement for the `ViewModels` namespace:

```
using ContosoUniversity.ViewModels;
```

The scaffolded code in the `Index` method specifies eager loading only for the `OfficeAssignment` navigation property:

```
public ActionResult Index()
{
    var instructors = db.Instructors.Include(i => i.OfficeAssignment);
    return View(instructors.ToList());
}
```

Replace the `Index` method with the following code to load additional related data and put it in the view model:

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);
}
```

```

if (id != null)
{
    ViewBag.InstructorID = id.Value;
    viewModel.Courses = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single().Courses;
}

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

return View(viewModel);
}

```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course, and passes all of the required data to the view. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.Courses` navigation property.

```

var viewModel = new InstructorIndexData();
viewModel.Instructors = db.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses.Select(c => c.Department))
    .OrderBy(i => i.LastName);

```

The second `Include` method loads `Courses`, and for each `Course` that is loaded it does eager loading for the `Course.Department` navigation property.

```

.Include(i => i.Courses.Select(c => c.Department))

```

As mentioned previously, eager loading is not required but is done to improve performance. Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. `Course` entities are required when an instructor is selected in the web page, so eager loading is better than lazy loading only if the page is displayed more often with a course selected than without.

If an instructor ID was selected, the selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `Courses` navigation property.

```

if (id != null)
{
    ViewBag.InstructorID = id.Value;
    viewModel.Courses = viewModel.Instructors.Where(i => i.ID ==
id.Value).Single().Courses;
}

```

```
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single `Instructor` entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `Courses` property.

You use the [Single](#) method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is [SingleOrDefault](#), which returns a default value (`null` in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a `null` reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the `Where` condition instead of calling the `Where` method separately:

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

Instead of:

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

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the `Enrollment` entities from that course's `Enrollments` navigation property.

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

Modify the Instructor Index View

In `Views\Instructor\Index.cshtml`, replace the template code with the following code. The changes are highlighted:

```
@model ContosoUniversity.ViewModels.InstructorIndexData
```

```
@{
    ViewBag.Title = "Instructors";
}

<h2>Instructors</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>Last Name</th>
```

```

        <th>First Name</th>
        <th>Hire Date</th>
        <th>Office</th>
        <th></th>
    </tr>

    @foreach (var item in Model.Instructors)
    {
        string selectedRow = "";
        if (item.ID == ViewBag.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>
                @Html.ActionLink("Select", "Index", new { id = item.ID }) |
                @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
                @Html.ActionLink("Details", "Details", new { id = item.ID })
                |
                @Html.ActionLink("Delete", "Delete", new { id = item.ID })
            </td>
        </tr>
    }
</table>

```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related `OfficeAssignment` entity.)

```

<td>
    @if (item.OfficeAssignment != null)
    {
        @item.OfficeAssignment.Location
    }
</td>

```

- Added code that will dynamically add `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a [Bootstrap](#) class.

```
string selectedRow = "";
if (item.InstructorID == ViewBag.InstructorID)
{
    selectedRow = "success";
}
<tr class="@selectedRow" valign="top">
```

- Added a new `ActionLink` labeled **Select** immediately before the other links in each row, which causes the selected instructor ID to be sent to the `Index` method.

Run the application and select the **Instructors** tab. The page displays the `Location` property of related `OfficeAssignment` entities and an empty table cell when there's no related `OfficeAssignment` entity.

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	
Abercrombie	Kim	3/11/1995		Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	Select Edit Details Delete
Zheng	Roger	2/12/2004		Select Edit Details Delete

© 2013 - Contoso University

In the `Views\Instructor\Index.cshtml` file, after the closing `table` element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

```
@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 == ViewBag.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>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a `Select` hyperlink that sends the ID of the selected course to the `Index` action method.

Run the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



http://localhost:40436/Ins

Instructors - Contoso Unive... x



Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	
Abercrombie	Kim	3/11/1995		Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	Select Edit Details Delete
Zheng	Roger	2/12/2004		Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@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>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.



http://localhost:40436/Ins

Instructors - Contoso Unive... x



Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	
Abercrombie	Kim	3/11/1995		Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	Select Edit Details Delete
Zheng	Roger	2/12/2004		Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
------	-------

Adding Explicit Loading

Open *InstructorController.cs* and look at how the `Index` method gets the list of enrollments for a selected course:

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

When you retrieved the list of instructors, you specified eager loading for the `Courses` navigation property and for the `Department` property of each course. Then you put the `Courses` collection in the view model, and now you're accessing the `Enrollments` navigation property from one entity in that collection. Because you didn't specify eager loading for the `Course.Enrollments` navigation property, the data from that property is appearing in the page as a result of lazy loading.

If you disabled lazy loading without changing the code in any other way, the `Enrollments` property would be null regardless of how many enrollments the course actually had. In that case, to load the `Enrollments` property, you'd have to specify either eager loading or explicit loading. You've already seen how to do eager loading. In order to see an example of explicit loading, replace the `Index` method with the following code, which explicitly loads the `Enrollments` property. The code changed are highlighted.

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();

    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
        ViewBag.CourseID = courseID.Value;
        // Lazy loading
        //viewModel.Enrollments = viewModel.Courses.Where(
        //    x => x.CourseID == courseID).Single().Enrollments;
        // Explicit loading
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID ==
courseID).Single();
        db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
    }
}
```

```

        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            db.Entry(enrollment).Reference(x => x.Student).Load();
        }

        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}

```

After getting the selected `Course` entity, the new code explicitly loads that course's `Enrollments` navigation property:

```
db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
```

Then it explicitly loads each `Enrollment` entity's related `Student` entity:

```
db.Entry(enrollment).Reference(x => x.Student).Load();
```

Notice that you use the `Collection` method to load a collection property, but for a property that holds just one entity, you use the `Reference` method.

Run the Instructor Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Summary

You've now used all three ways (lazy, eager, and explicit) to load related data into navigation properties. In the next tutorial you'll learn how to update related data.

Updating Related Data with the Entity Framework in an ASP.NET MVC Application

In the previous tutorial you displayed related data; in this tutorial you'll update related data. For most relationships, this can be done by updating either foreign key fields or navigation properties. For many-to-many relationships, the Entity Framework doesn't expose the join table directly, so you add and remove entities to and from the appropriate navigation properties.

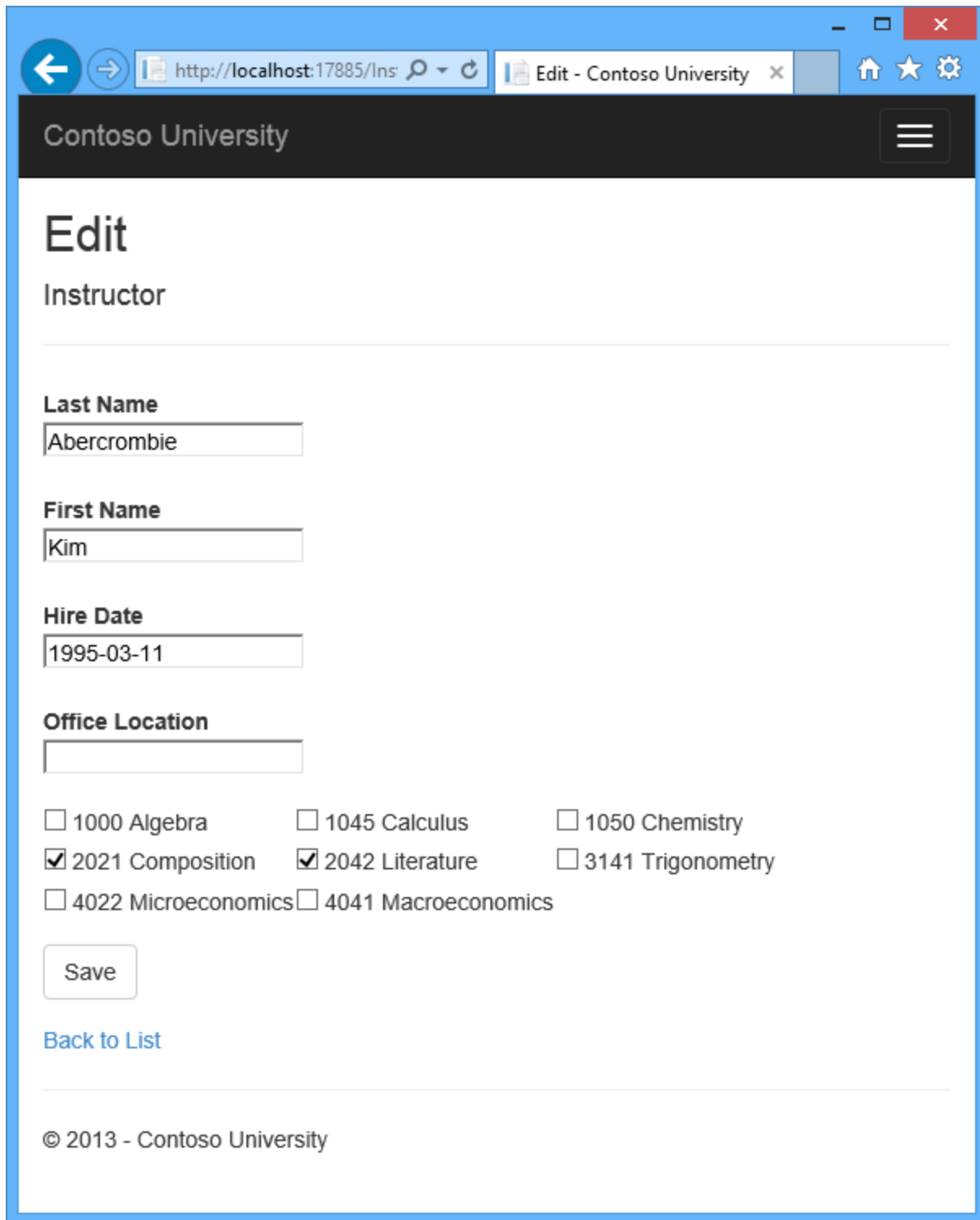
The following illustrations show some of the pages that you'll work with.

The image shows a web browser window with the following elements:

- Browser Address Bar:** `http://localhost:17885/Co` and `Create - Contoso Univer...`
- Page Header:** **Contoso University** with a hamburger menu icon.
- Section Header:** **Create**
- Form Fields:**
 - Course** (empty text input)
 - Number:**
 - Title:**
 - Credits:**
 - Department:** (dropdown menu)
- Buttons:**
- Links:** [Back to List](#)
- Footer:** © 2013 - Contoso University

The screenshot shows a web browser window with the following elements:

- Browser Address Bar:** Contains the URL `http://localhost:17885/Co` and the page title `Edit - Contoso University`.
- Page Header:** A dark bar with the text `Contoso University` on the left and a hamburger menu icon on the right.
- Main Content Area:**
 - Section Header:** `Edit Course`
 - Form Fields:**
 - Number:** A text input field containing the value `1000`.
 - Title:** A text input field containing the value `Algebra`.
 - Credits:** A text input field containing the value `4`.
 - Department:** A dropdown menu with the selected value `Mathematics`.
 - Buttons:** A `Save` button and a `Back to List` link.
- Footer:** A horizontal line followed by the text `© 2013 - Contoso University`.



Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that

include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CourseController.cs*, delete the four `Create` and `Edit` methods and replace them with the following code:

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

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include =
"CourseID,Title,Credits,DepartmentID")]Course course)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Courses.Add(course);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to
        write a log.)
        ModelState.AddModelError("", "Unable to save changes. Try again, and if
        the problem persists, see your system administrator.");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

public ActionResult Edit(int?id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Course course = db.Courses.Find(id);
    if (course == null)
    {
        return HttpNotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

[HttpPost]
```

```

[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include =
"CourseID,Title,Credits,DepartmentID")]Course course)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Entry(course).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to
write a log.)
        ModelState.AddModelError("", "Unable to save changes. Try again, and if
the problem persists, see your system administrator.");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

private void PopulateDepartmentsDropDownList(object selectedDepartment =
null)
{
    var departmentsQuery = from d in db.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery, "DepartmentID",
"Name", selectedDepartment);
}

```

Add the following using statement at the beginning of the file:

```
using System.Data.Entity.Infrastructure;
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in a `ViewBag` property. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name `DepartmentID` to the [DropDownList](#) helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named `DepartmentID`.

The `HttpGet Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

```

public ActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}

```

The `HttpGet Edit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Course course = db.Courses.Find(id);
    if (course == null)
    {
        return HttpNotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The `HttpPost` methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error:

```
        catch (RetryLimitExceededException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to
            write a log.)
            ModelState.AddModelError("", "Unable to save changes. Try again, and if
            the problem persists, see your system administrator.");
        }
        PopulateDepartmentsDropDownList(course.DepartmentID);
        return View(course);
```

This code ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

The `Course` views are already scaffolded with drop-down lists for the department field, but you don't want the `DepartmentID` caption for this field, so make the following highlighted change to the `Views\Course\Create.cshtml` file to change the caption.

```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Course</h4>
```

```

<hr />
@Html.ValidationSummary(true)

<div class="form-group">
    @Html.LabelFor(model => model.CourseID, new { @class = "control-
label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.CourseID)
        @Html.ValidationMessageFor(model => model.CourseID)
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Title, new { @class = "control-
label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title)
    </div>
</div>

<div class="form-group">
    @Html.LabelFor(model => model.Credits, new { @class = "control-
label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.Credits)
        @Html.ValidationMessageFor(model => model.Credits)
    </div>
</div>

<div class="form-group">
    <label class="control-label col-md-2"
for="DepartmentID">Department</label>
    <div class="col-md-10">
        @Html.DropDownList("DepartmentID", String.Empty)
        @Html.ValidationMessageFor(model => model.DepartmentID)
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Create" class="btn btn-default"
/>
    </div>
</div>
</div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

Make the same change in *Views\Course\Edit.cshtml*.

Normally the scaffolder doesn't scaffold a primary key because the key value is generated by the database and can't be changed and isn't a meaningful value to be displayed to users. For Course entities the scaffolder does include a text box for the `CourseID` field because it understands that the `DatabaseGeneratedOption.None` attribute means the user should be able to enter the primary key value. But it doesn't understand that because the number is meaningful you want to see it in the other views, so you need to add it manually.

In *Views\Course\Edit.cshtml*, add a course number field before the **Title** field. Because it's the primary key, it's displayed, but it can't be changed.

```
<div class="form-group">
  @Html.LabelFor(model => model.CourseID, new { @class = "control-label
col-md-2" })
  <div class="col-md-10">
    @Html.DisplayFor(model => model.CourseID)
  </div>
</div>
```

There's already a hidden field (`Html.HiddenFor` helper) for the course number in the Edit view. Adding an *Html.LabelFor* helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the Edit page.

In *Views\Course\Delete.cshtml* and *Views\Course\Details.cshtml*, change the department name caption from "Name" to "Department" and add a course number field before the **Title** field.

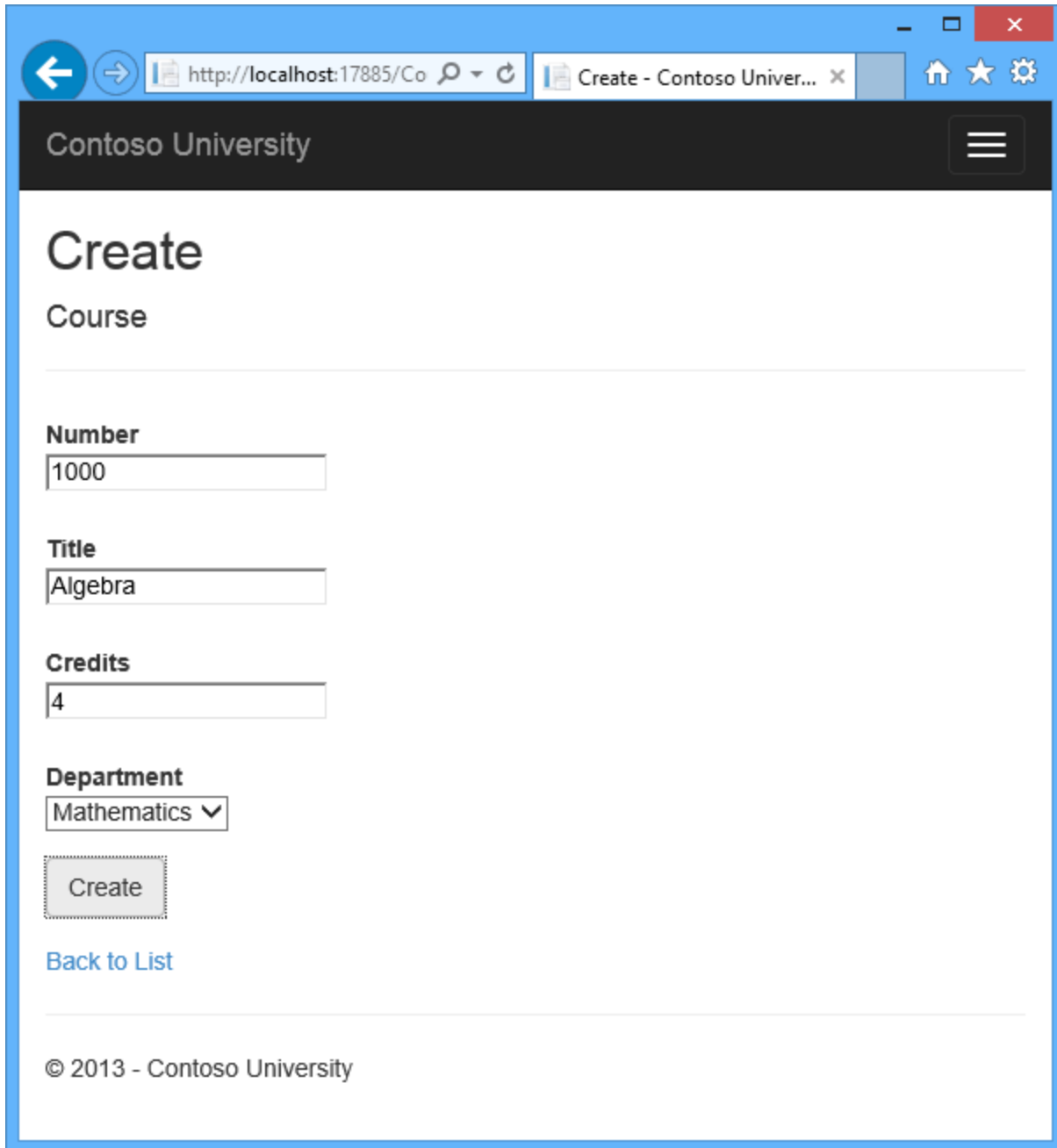
```
<dt>
  Department
</dt>

<dd>
  @Html.DisplayFor(model => model.Department.Name)
</dd>

<dt>
  @Html.DisplayNameFor(model => model.CourseID)
</dt>

<dd>
  @Html.DisplayFor(model => model.CourseID)
</dd>
```

Run the **Create** page (display the Course Index page and click **Create New**) and enter data for a new course:



Click **Create**. The Course Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Contoso University

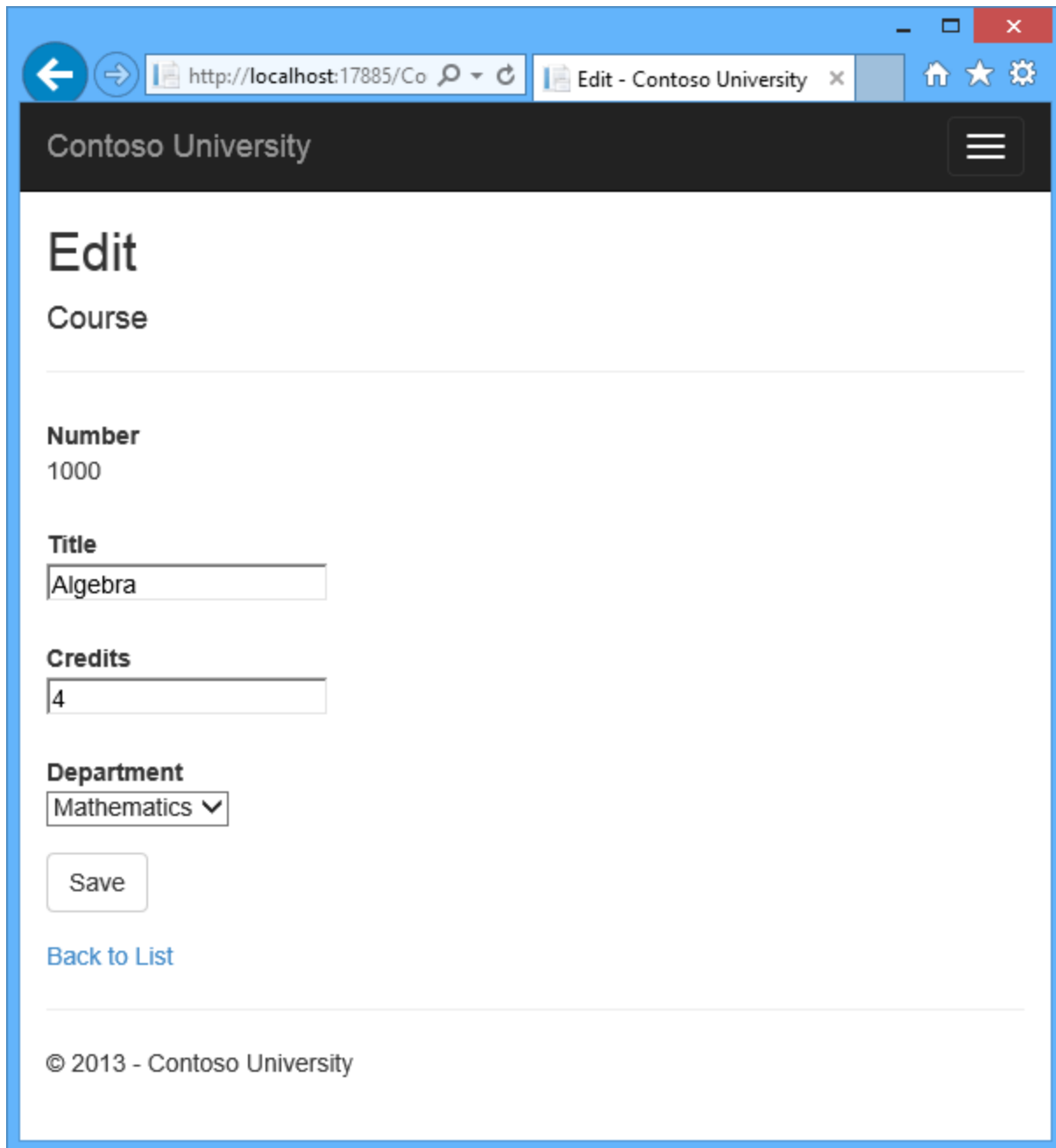
Courses

[Create New](#)

Number	Title	Credits	Department	
1000	Algebra	4	Mathematics	Edit Details Delete
1045	Calculus	4	Mathematics	Edit Details Delete
1050	Chemistry	3	Engineering	Edit Details Delete
2021	Composition	3	English	Edit Details Delete
2042	Literature	4	English	Edit Details Delete
3141	Trigonometry	4	Mathematics	Edit Details Delete
4022	Microeconomics	3	Economics	Edit Details Delete
4041	Macroeconomics	3	Economics	Edit Details Delete

© 2013 - Contoso University

Run the **Edit** page (display the Course Index page and click **Edit** on a course).



Change data on the page and click **Save**. The Course Index page is displayed with the updated course data.

Adding an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means you must handle the following situations:

- If the user clears the office assignment and it originally had a value, you must remove and delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, you must create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, you must change the value in an existing `OfficeAssignment` entity.

Open `InstructorController.cs` and look at the `HttpGet Edit` method:

```
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors.Find(id);
    if (instructor == null)
    {
        return HttpNotFound();
    }
    ViewBag.ID = new SelectList(db.OfficeAssignments, "InstructorID",
"Location", instructor.ID);
    return View(instructor);
}
```

The scaffolded code here isn't what you want. It's setting up data for a drop-down list, but you what you need is a text box. Replace this method with the following code:

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Where(i => i.ID == id)
        .Single();
    if (instructor == null)
    {
        return HttpNotFound();
    }
    return View(instructor);
}
```

This code drops the `ViewBag` statement and adds eager loading for the associated `OfficeAssignment` entity. You can't perform eager loading with the `Find` method, so the `Where` and `Single` methods are used instead to select the instructor.

Replace the `HttpPost Edit` method with the following code. which handles office assignment updates:

```
[HttpPost, ActionName("Edit")]
```

```

[ValidateAntiForgeryToken]
public ActionResult EditPost(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var instructorToUpdate = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Where(i => i.ID == id)
        .Single();

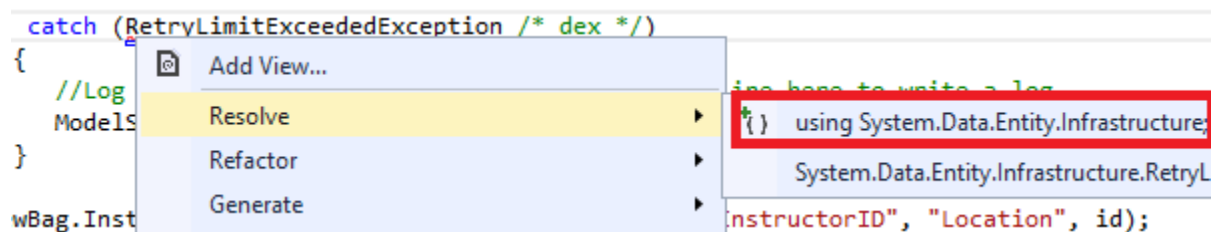
    if (TryUpdateModel(instructorToUpdate, "",
        new string[] { "LastName", "FirstMidName", "HireDate",
"OfficeAssignment" }))
    {
        try
        {
            if
(
                String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment.Location)
            )
            {
                instructorToUpdate.OfficeAssignment = null;
            }

            db.Entry(instructorToUpdate).State = EntityState.Modified;
            db.SaveChanges();

            return RedirectToAction("Index");
        }
        catch (RetryLimitExceededException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to
write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and
if the problem persists, see your system administrator.");
        }
    }
    return View(instructorToUpdate);
}

```

The reference to `RetryLimitExceededException` requires a using statement; to add it, right-click `RetryLimitExceededException`, and then click **Resolve - using System.Data.Entity.Infrastructure**.



The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the `HttpGet` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the `HttpGet Edit` method.
- Updates the retrieved `Instructor` entity with values from the model binder. The [TryUpdateModel](#) overload used enables you to *whitelist* the properties you want to include. This prevents over-posting, as explained in [the second tutorial](#).

```
if (TryUpdateModel(instructorToUpdate, "",
    new string[] { "LastName", "FirstMidName", "HireDate",
        "OfficeAssignment" }))
```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

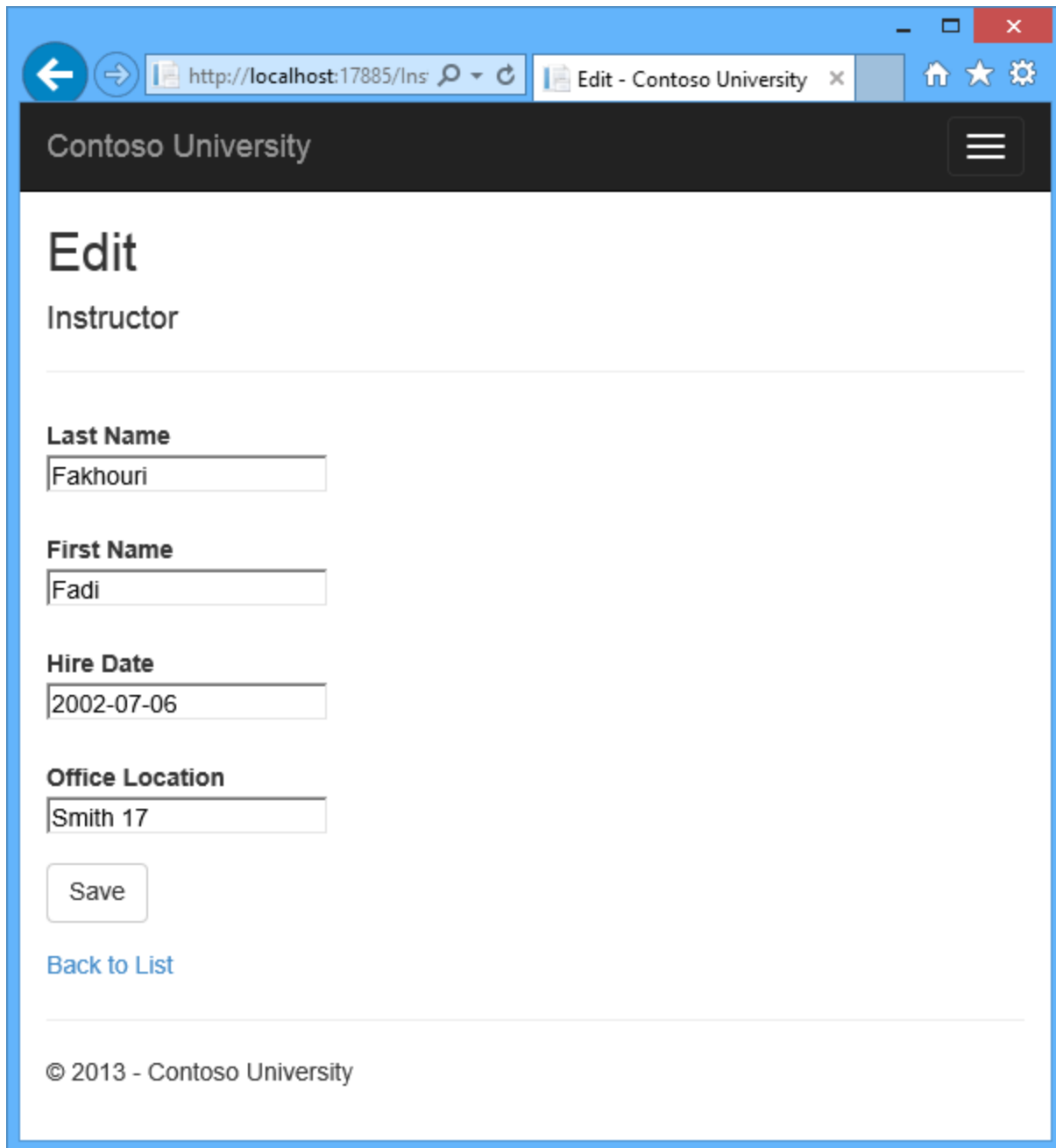
```
if
(String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment.Location
))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

In `Views\Instructor\Edit.cshtml`, after the `div` elements for the **Hire Date** field, add a new field for editing the office location:

```
<div class="form-group">
    @Html.LabelFor(model => model.OfficeAssignment.Location, new { @class =
"control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.OfficeAssignment.Location)
        @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
    </div>
</div>
```

Run the page (select the **Instructors** tab and then click **Edit** on an instructor). Change the **Office Location** and click **Save**.



Adding Course Assignments to the Instructor Edit Page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:

Contoso University

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

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

[Back to List](#)

© 2013 - Contoso University

The relationship between the `Course` and `Instructor` entities is many-to-many, which means you do not have direct access to the foreign key properties which are in the join table. Instead, you add and remove entities to and from the `Instructor.Courses` navigation property.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating navigation properties in order to create or delete relationships.

To provide data to the view for the list of check boxes, you'll use a view model class. Create *AssignedCourseData.cs* in the *ViewModels* folder and replace the existing code with the following code:

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

In *InstructorController.cs*, replace the `HttpGet Edit` method with the following code. The changes are highlighted.

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses)
        .Where(i => i.ID == id)
        .Single();
    PopulateAssignedCourseData(instructor);
    if (instructor == null)
    {
        return HttpNotFound();
    }
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = db.Courses;
    var instructorCourses = new HashSet<int>(instructor.Courses.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)
    });
}
ViewBag.Courses = viewModel;
}

```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a [HashSet](#) collection. The `Assigned` property is set to `true` for courses the instructor is assigned. The view will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in a `ViewBag` property.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, which calls a new method that updates the `Courses` navigation property of the `Instructor` entity. The changes are highlighted.

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var instructorToUpdate = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses)
        .Where(i => i.ID == id)
        .Single();

    if (TryUpdateModel(instructorToUpdate, "",
        new string[] { "LastName", "FirstMidName", "HireDate",
        "OfficeAssignment" }))
    {
        try
        {
            if
            (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }

            UpdateInstructorCourses(selectedCourses, instructorToUpdate);

            db.Entry(instructorToUpdate).State = EntityState.Modified;

```

```

        db.SaveChanges();

        return RedirectToAction("Index");
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here
to write a log.
        ModelState.AddModelError("", "Unable to save changes. Try again,
and if the problem persists, see your system administrator.");
    }
}
PopulateAssignedCourseData(instructorToUpdate);
return View(instructorToUpdate);
}
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.Courses = new List<Course>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.Courses.Select(c => c.CourseID));
    foreach (var course in db.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.Courses.Add(course);
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.Courses.Remove(course);
            }
        }
    }
}
}

```

The method signature is now different from the `HttpGet Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of `Course` entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the `Courses` navigation property, you'll do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require

any change to the code that calls [TryUpdateModel](#) because you're using the *whitelisting* overload and `Courses` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `Courses` navigation property with an empty collection:

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Remove(course);
    }
}
```

In `Views\Instructor\Edit.cshtml`, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the `OfficeAssignment` field and before the `div` element for the **Save** button:

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                <td>
                    @{
                        int cnt = 0;
                        List<ContosoUniversity.ViewModels.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>

```

After you paste the code, if line breaks and indentation don't look like they do here, manually fix everything so that it looks like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error.

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The `value` attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have `checked` attributes, which selects them (displays them checked).

After changing course assignments, you'll want to be able to verify the changes when the site returns to the `Index` page. Therefore, you need to add a column to the table in that page. In this case you don't need to use the `ViewBag` object, because the information you want to display is already in the `Courses` navigation property of the `Instructor` entity that you're passing to the page as the model.

In `Views\Instructor\Index.cshtml`, add a **Courses** heading immediately following the **Office** heading, as shown in the following example:

```

<tr>
    <th>Last Name</th>
    <th>First Name</th>
    <th>Hire Date</th>
    <th>Office</th>
    <th>Courses</th>

```

```
<th></th>
</tr>
```

Then add a new detail cell immediately following the office location detail cell:

```
<td>
  @if (item.OfficeAssignment != null)
  {
    @item.OfficeAssignment.Location
  }
</td>
<td>
  @{
    foreach (var course in item.Courses)
    {
      @course.CourseID @: @course.Title <br />
    }
  }
</td>
<td>
  @Html.ActionLink("Select", "Index", new { id = item.ID }) |
  @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
  @Html.ActionLink("Details", "Details", new { id = item.ID }) |
  @Html.ActionLink("Delete", "Delete", new { id = item.ID })
</td>
```

Run the **Instructor Index** page to see the courses assigned to each instructor:

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	3/11/1995		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	7/6/2002	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	7/1/1998	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	1/15/2001	Thompson 304	1050 Chemistry	Select Edit Details Delete
test	test	1/1/2011		1000 Algebra	Select Edit Details Delete
Zheng	Roger	2/12/2004		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

© 2013 - Contoso University

Click **Edit** on an instructor to see the Edit page.

Contoso University

Edit Instructor

Last Name
Abercrombie

First Name
Kim

Hire Date
1995-03-11

Office Location

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

[Back to List](#)

© 2013 - Contoso University

Change some course assignments and click **Save**. The changes you make are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the DeleteConfirmed Method

In *InstructorController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Where(i => i.ID == id)
        .Single();

    instructor.OfficeAssignment = null;
    db.Instructors.Remove(instructor);

    var department = db.Departments
        .Where(d => d.InstructorID == id)
        .SingleOrDefault();
    if (department != null)
    {
        department.InstructorID = null;
    }

    db.SaveChanges();
    return RedirectToAction("Index");
}
```

This code makes two changes:

- Deletes the office assignment record (if any) when the instructor is deleted.
- If the instructor is assigned as administrator of any department, removes the instructor assignment from that department. Without this code, you would get a referential integrity error if you tried to delete an instructor who was assigned as administrator for a department.

Add office location and courses to the Create page

In *InstructorController.cs*, delete the `HttpGet` and `HttpPost Create` methods, and then add the following code in their place:

```
public ActionResult Create()
{
    var instructor = new Instructor();
```

```

        instructor.Courses = new List<Course>();
        PopulateAssignedCourseData(instructor);
        return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public ActionResult Create([Bind(Include =
        "LastName,FirstMidName,HireDate,OfficeAssignment" )]Instructor instructor,
        string[] selectedCourses)
    {
        if (selectedCourses != null)
        {
            instructor.Courses = new List<Course>();
            foreach (var course in selectedCourses)
            {
                var courseToAdd = db.Courses.Find(int.Parse(course));
                instructor.Courses.Add(courseToAdd);
            }
        }
        if (ModelState.IsValid)
        {
            db.Instructors.Add(instructor);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
        PopulateAssignedCourseData(instructor);
        return View(instructor);
    }
}

```

This code is similar to what you saw for the Edit methods except that initially no courses are selected. The `HttpGet Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The `HttpPost Create` method adds each selected course to the `Courses` navigation property before the template code that checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date) so that when the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `Courses` navigation property you have to initialize the property as an empty collection:

```
instructor.Courses = new List<Course>();
```

As an alternative to doing this in controller code, you could do it in the `Course` model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<Course> _courses;
public virtual ICollection<Course> Courses
```

```

{
    get
    {
        return _courses ?? (_courses = new List<Course>());
    }
    set
    {
        _courses = value;
    }
}

```

If you modify the `Courses` property in this way, you can remove the explicit property initialization code in the controller.

In `Views\Instructor\Create.cshtml`, add an office location text box and course check boxes after the hire date field and before the **Submit** button.

```

<div class="form-group">
    @Html.LabelFor(model => model.OfficeAssignment.Location, new { @class =
"control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.OfficeAssignment.Location)
        @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                <td>
                    @{
                        int cnt = 0;
                        List<ContosoUniversity.ViewModels.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>
</div>

```


After you paste the code, fix line breaks and indentation as you did earlier for the Edit page.

Run the Create page and add an instructor.

Contoso University

Create

Instructor

Last Name

First Name

Hire Date

Office Location

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

[Back to List](#)

© 2013 - Contoso University

Handling Transactions

As explained in the [Basic CRUD Functionality tutorial](#), by default the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Working with Transactions](#) on MSDN.

Summary

You have now completed this introduction to working with related data. So far in these tutorials you've worked with code that does synchronous I/O. You can make the application use web server resources more efficiently by implementing asynchronous code, and that's what you'll do in the next tutorial.

Async and Stored Procedures with the Entity Framework in an ASP.NET MVC Application

In earlier tutorials you learned how to read and update data using the synchronous programming model. In this tutorial you see how to implement the asynchronous programming model. Asynchronous code can help an application perform better because it makes better use of server resources.

In this tutorial you'll also see how to use stored procedures for insert, update, and delete operations on an entity.

Finally, you'll redeploy the application to Windows Azure, along with all of the database changes that you've implemented since the first time you deployed.

The following illustrations show some of the pages that you'll work with.

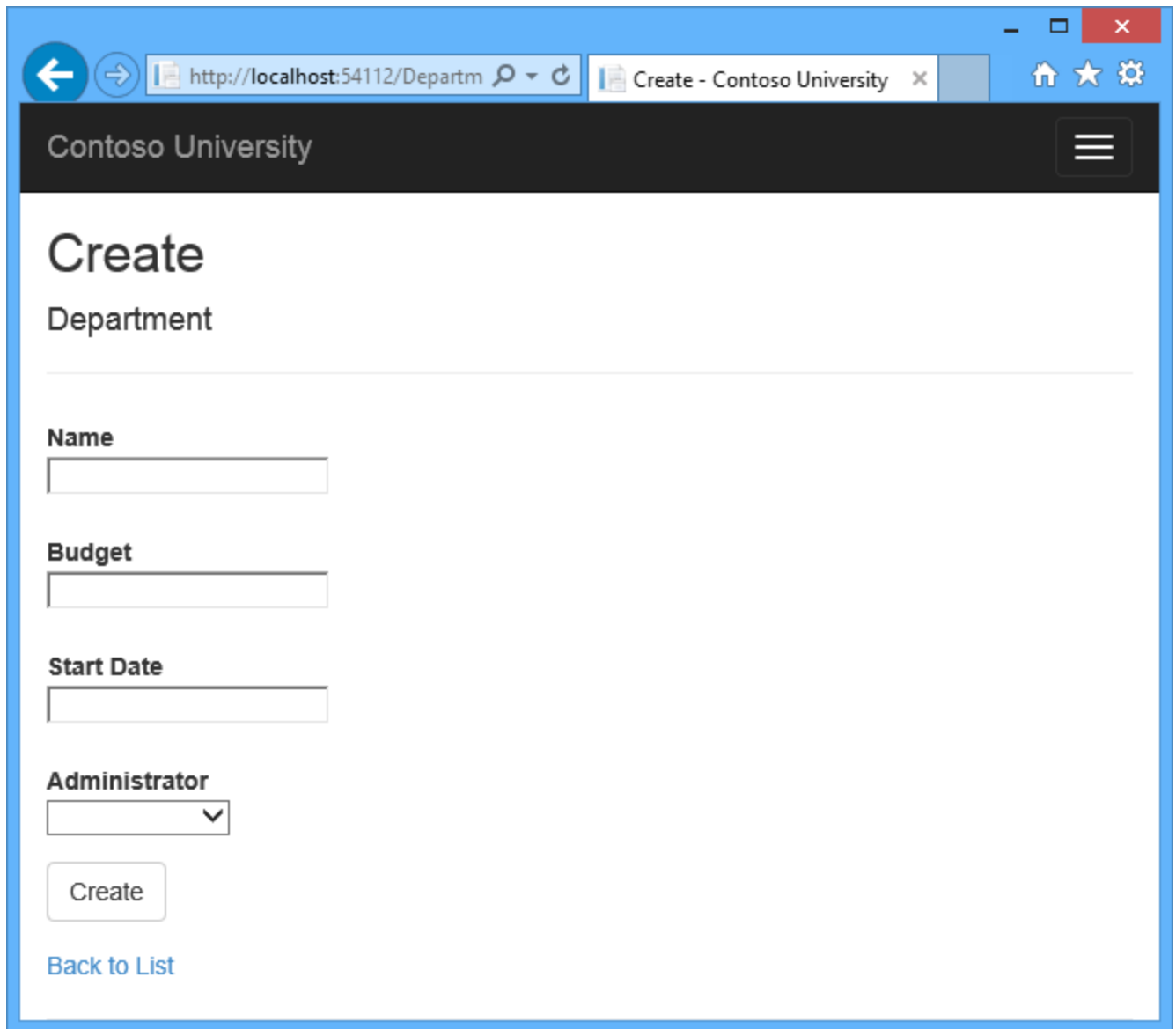
Contoso University

Departments

[Create New](#)

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2013-10-29		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete
New	\$3.00	2013-01-01	Kapoor, Candace	Edit Details Delete
Another	\$1.00	2012-01-01	Harui, Roger	Edit Details Delete

© 2013 - Contoso University



Why bother with asynchronous code

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

In earlier versions of .NET, writing and testing asynchronous code was complex, error prone, and hard to debug. In .NET 4.5, writing, testing, and debugging asynchronous code is so much

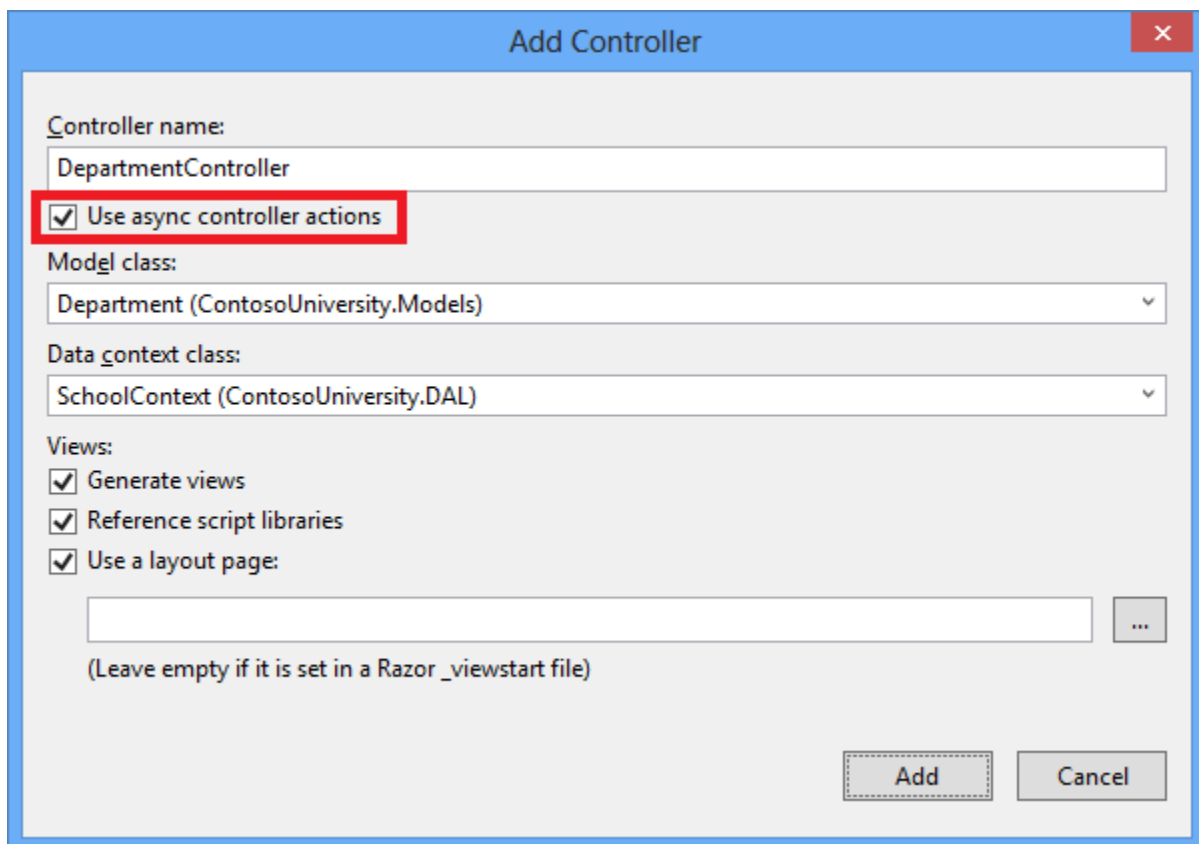
easier that you should generally write asynchronous code unless you have a reason not to. Asynchronous code does introduce a small amount of overhead, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

For more information about asynchronous programming, see the following resources:

- [Entity Framework Async Query and Save](#)
- [Using Asynchronous Methods in ASP.NET MVC 4](#)
- [How to Build ASP./NET Web Applications Using Async](#) (Video)

Create the Department controller

Create a Department controller the same way you did the earlier controllers, except this time select the **Use async controller actions** actions check box.



The screenshot shows the 'Add Controller' dialog box. The 'Controller name' field contains 'DepartmentController'. The 'Use async controller actions' checkbox is checked and highlighted with a red box. The 'Model class' dropdown is set to 'Department (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.DAL)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. The 'Add' button is highlighted with a dashed border.

The following highlights show how what was added to the synchronous code for the `Index` method to make it asynchronous:

```
public async Task<ActionResult> Index()  
{  
    var departments = db.Departments.Include(d => d.Administrator);  
    return View(await departments.ToListAsync());  
}
```

```
}
```

Four changes were applied to enable the Entity Framework database query to execute asynchronously:

- The method is marked with the `async` keyword, which tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<ActionResult>` object that is returned.
- The return type was changed from `ActionResult` to `Task<ActionResult>`. The `Task<T>` type represents ongoing work with a result of type `T`.
- The `await` keyword was applied to the web service call. When the compiler sees this keyword, behind the scenes it splits the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- The asynchronous version of the `ToList` extension method was called.

Why is the `departments.ToList` statement modified but not the `departments = db.Departments` statement? The reason is that only statements that cause queries or commands to be sent to the database are executed asynchronously. The `departments = db.Departments` statement sets up a query but the query is not executed until the `ToList` method is called. Therefore, only the `ToList` method is executed asynchronously.

In the `Details` method and the `HttpGet Edit` and `Delete` methods, the `Find` method is the one that causes a query to be sent to the database, so that's the method that gets executed asynchronously:

```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = await db.Departments.FindAsync(id);
    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

In the `Create`, `HttpPost Edit`, and `DeleteConfirmed` methods, it is the `SaveChanges` method call that causes a command to be executed, not statements such as `db.Departments.Add(department)` which only cause entities in memory to be modified.

```
public async Task<ActionResult> Create(Department department)
{
    if (ModelState.IsValid)
    {
        db.Departments.Add(department);
        await db.SaveChangesAsync();
    }
}
```

```

        return RedirectToAction("Index");
    }

```

Open *Views\Department\Index.cshtml*, and replace the template code with the following code:

```

@model IEnumerable<ContosoUniversity.Models.Department>
@{
    ViewBag.Title = "Departments";
}
<h2>Departments</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Budget)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.StartDate)
        </th>
        <th>
            Administrator
        </th>
    </tr>
    @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>
                @Html.ActionLink("Edit", "Edit", new { id=item.DepartmentID }) |
                @Html.ActionLink("Details", "Details", new { id=item.DepartmentID
            </td>
            </tr>
        }
        @Html.ActionLink("Delete", "Delete", new { id=item.DepartmentID
    </td>
    </tr>
}
</table>

```


This code changes the title from Index to Departments, moves the Administrator name to the right, and provides the full name of the administrator.

In the Create, Delete, Details, and Edit views, change the caption for the `InstructorID` field to "Administrator" the same way you changed the department name field to "Department" in the Course views.

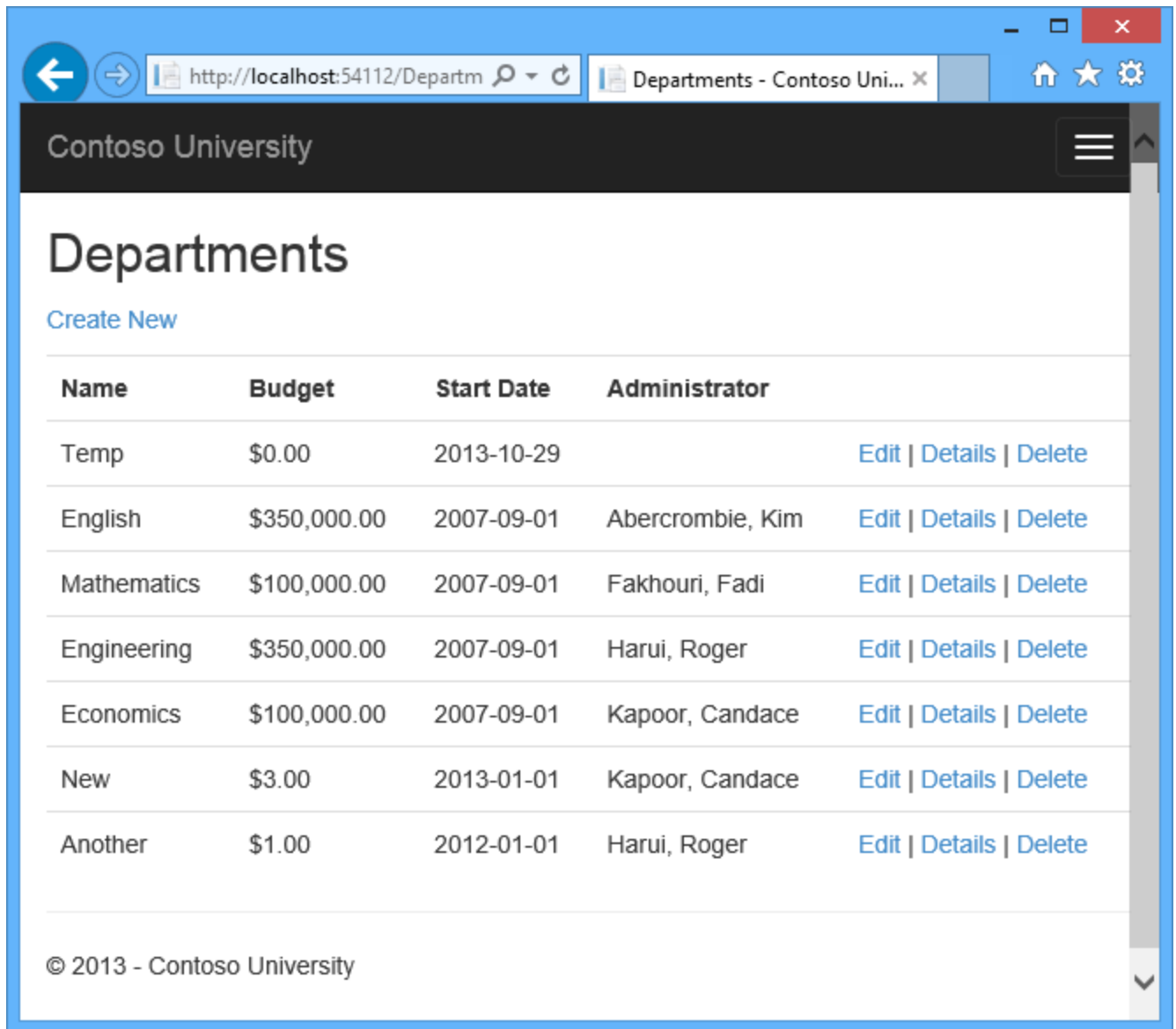
In the Create and Edit views use the following code:

```
<label class="control-label col-md-2"
for="InstructorID">Administrator</label>
```

In the Delete and Details views use the following code:

```
<dt>
    Administrator
</dt>
```

Run the application, and click the **Departments** tab.



Everything works the same as in the other controllers, but in this controller all of the SQL queries are executing asynchronously.

Some things to be aware of when you are using asynchronous programming with the Entity Framework:

- The async code is not thread safe. In other words, in other words, don't try to do multiple operations in parallel using the same context instance.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

Use stored procedures for inserting, updating, and deleting

Some developers and DBAs prefer to use stored procedures for database access. In earlier versions of Entity Framework you can retrieve data using a stored procedure by [executing a raw SQL query](#), but you can't instruct EF to use stored procedures for update operations. In EF 6 it's easy to configure Code First to use stored procedures.

1. In `DAL\SchoolContext.cs`, add the highlighted code to the `OnModelCreating` method.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    modelBuilder.Entity<Course>()
        .HasMany(c => c.Instructors).WithMany(i => i.Courses)
        .Map(t => t.MapLeftKey("CourseID")
            .MapRightKey("InstructorID")
            ..ToTable("CourseInstructor"));
    modelBuilder.Entity<Department>().MapToStoredProcedures();
}
```

This code instructs Entity Framework to use stored procedures for insert, update, and delete operations on the `Department` entity.

2. In Package Manage Console, enter the following command:

```
add-migration DepartmentSP
```

Open `Migrations\<timestamp>_DepartmentSP.cs` to see the code in the `Up` method that creates Insert, Update, and Delete stored procedures:

```
public override void Up()
{
    CreateStoredProcedure(
        "dbo.Department_Insert",
        p => new
        {
            Name = p.String(maxLength: 50),
            Budget = p.Decimal(precision: 19, scale: 4, storeType:
"money"),
            StartDate = p.DateTime(),
            InstructorID = p.Int(),
        },
        body:
        @"INSERT [dbo].[Department] ([Name], [Budget], [StartDate],
[InstructorID])
VALUES (@Name, @Budget, @StartDate, @InstructorID)

DECLARE @DepartmentID int
SELECT @DepartmentID = [DepartmentID]
FROM [dbo].[Department]
WHERE @@ROWCOUNT > 0 AND [DepartmentID] =
scope_identity()

SELECT t0.[DepartmentID]
FROM [dbo].[Department] AS t0
```

```

        WHERE @@ROWCOUNT > 0 AND t0.[DepartmentID] =
@DepartmentID"
    );

    CreateStoredProcedure(
        "dbo.Department_Update",
        p => new
        {
            DepartmentID = p.Int(),
            Name = p.String(maxLength: 50),
            Budget = p.Decimal(precision: 19, scale: 4, storeType:
"money"),
            StartDate = p.DateTime(),
            InstructorID = p.Int(),
        },
        body:
        @"UPDATE [dbo].[Department]
        SET [Name] = @Name, [Budget] = @Budget, [StartDate] =
@StartDate, [InstructorID] = @InstructorID
        WHERE ([DepartmentID] = @DepartmentID)"
    );

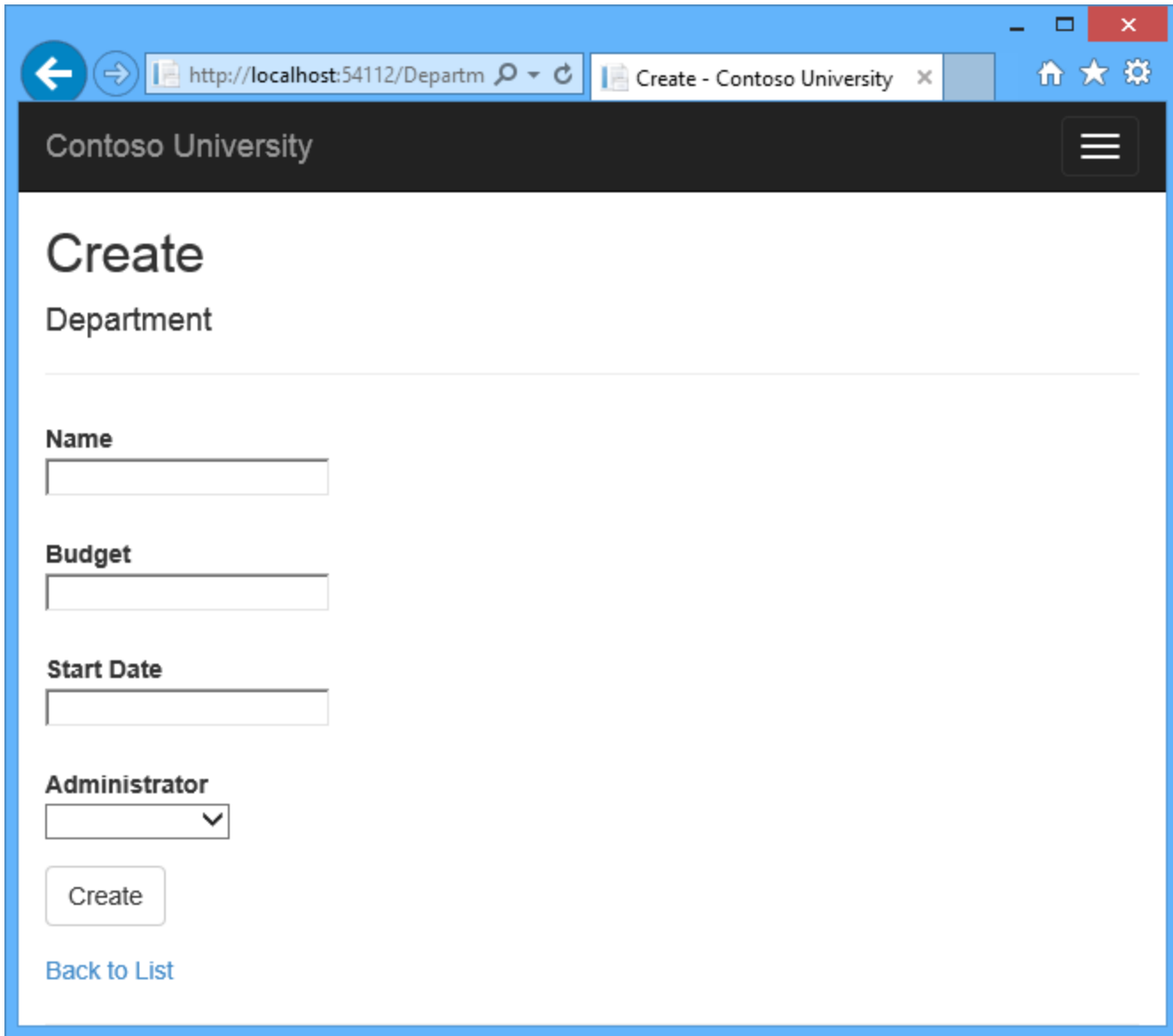
    CreateStoredProcedure(
        "dbo.Department_Delete",
        p => new
        {
            DepartmentID = p.Int(),
        },
        body:
        @"DELETE [dbo].[Department]
        WHERE ([DepartmentID] = @DepartmentID)"
    );
}

```

3. In Package Manage Console, enter the following command:

```
update-database
```

4. Run the application in debug mode, click the **Departments** tab, and then click **Create New**.
5. Enter data for a new department, and then click **Create**.



6. In Visual Studio, look at the logs in the **Output** window to see that a stored procedure was used to insert the new Department row.

```
Output
Show output from: Debug
FROM [dbo].[Instructor] AS [Extent1]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
000047;Properties:Command: [dbo].[Department_Insert]:
iisexpress.exe Information: 0 : Component:SQL
Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
000041;Properties:Command: SELECT
[Extent1].[DepartmentID] AS [DepartmentID],
[Extent1].[Name] AS [Name],
[Extent1].[Budget] AS [Budget],
[Extent1].[StartDate] AS [StartDate],
[Extent1].[InstructorID] AS [InstructorID],
[Extent2].[ID] AS [ID],
[Extent2].[LastName] AS [LastName],
[Extent2].[FirstName] AS [FirstName],
[Extent2].[HireDate] AS [HireDate]
FROM [dbo].[Department] AS [Extent1]
LEFT OUTER JOIN [dbo].[Instructor] AS [Extent2] ON [Extent1].
[InstructorID] = [Extent2].[ID]:
```

Code First creates default stored procedure names. If you are using an existing database, you might need to customize the stored procedure names in order to use stored procedures already defined in the database. For information about how to do that, see [Entity Framework Code First Insert/Update/Delete Stored Procedures](#).

If you want to customize what generated stored procedures do, you can edit the scaffolded code for the migrations `Up` method that creates the stored procedure. That way your changes are reflected whenever that migration is run and will be applied to your production database when migrations runs automatically in production after deployment.

If you want to change an existing stored procedure that was created in a previous migration, you can use the Add-Migration command to generate a blank migration, and then manually write code that calls the [AlterStoredProcedure](#) method.

Deploy to Windows Azure

This section requires you to have completed the optional **Deploying the app to Windows Azure** section in the [Migrations and Deployment](#) tutorial of this series. If you had migrations errors that you resolved by deleting the database in your local project, skip this section.

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.
2. Click **Publish**.

Visual Studio deploys the application to Windows Azure, and the application opens in your default browser, running in Windows Azure.

3. Test the application to verify it's working.

The first time you run a page that accesses the database, the Entity Framework runs all of the migrations `Up` methods required to bring the database up to date with the current data model. You can now use all of the web pages that you added since the last time you deployed, including the Department pages that you added in this tutorial.

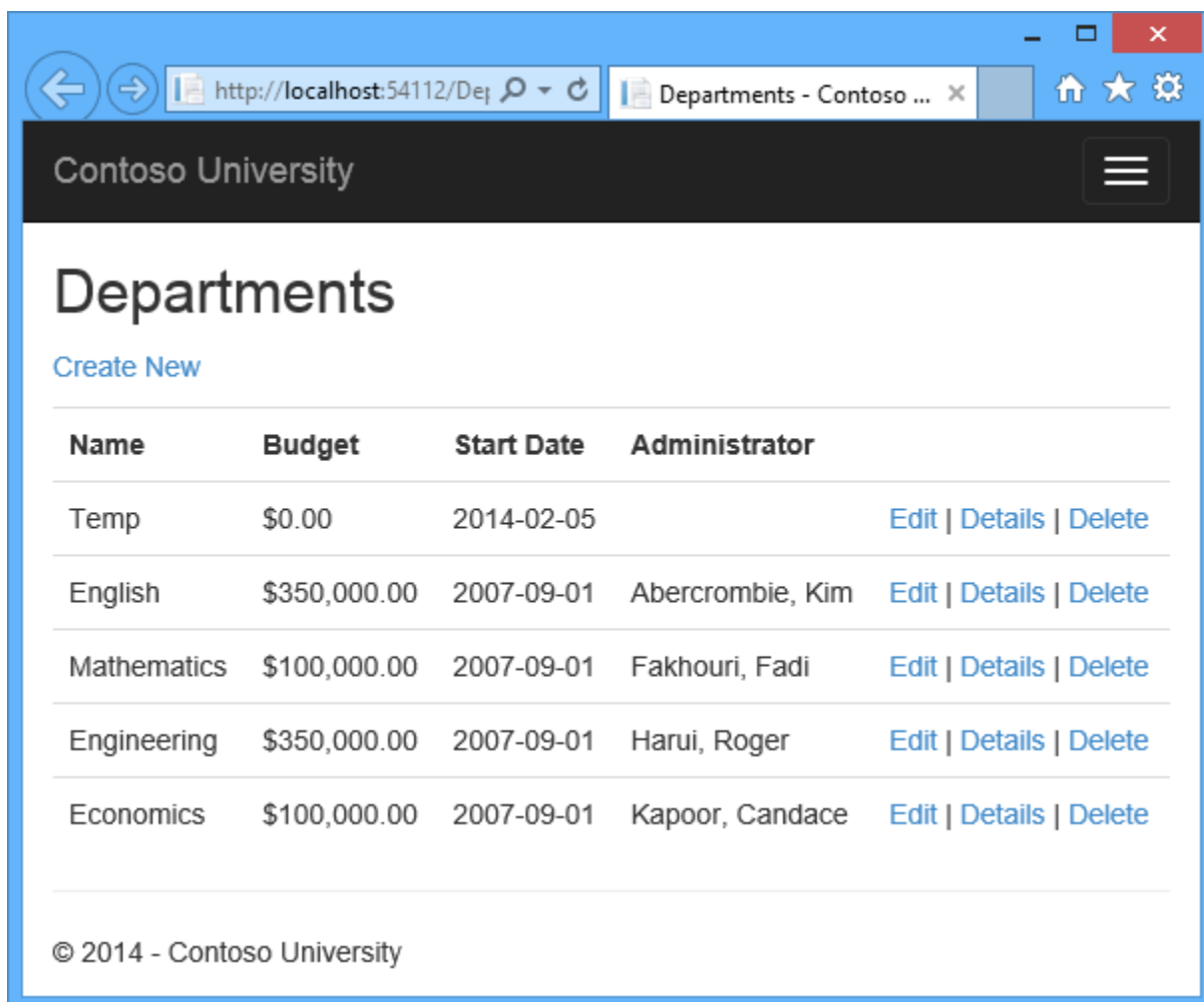
Summary

In this tutorial you saw how to improve server efficiency by writing code that executes asynchronously, and how to use stored procedures for insert, update, and delete operations. In the next tutorial, you'll see how to prevent data loss when multiple users try to edit the same record at the same time.

Handling Concurrency with the Entity Framework 6 in an ASP.NET MVC 5 Application (10 of 12)

In earlier tutorials you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

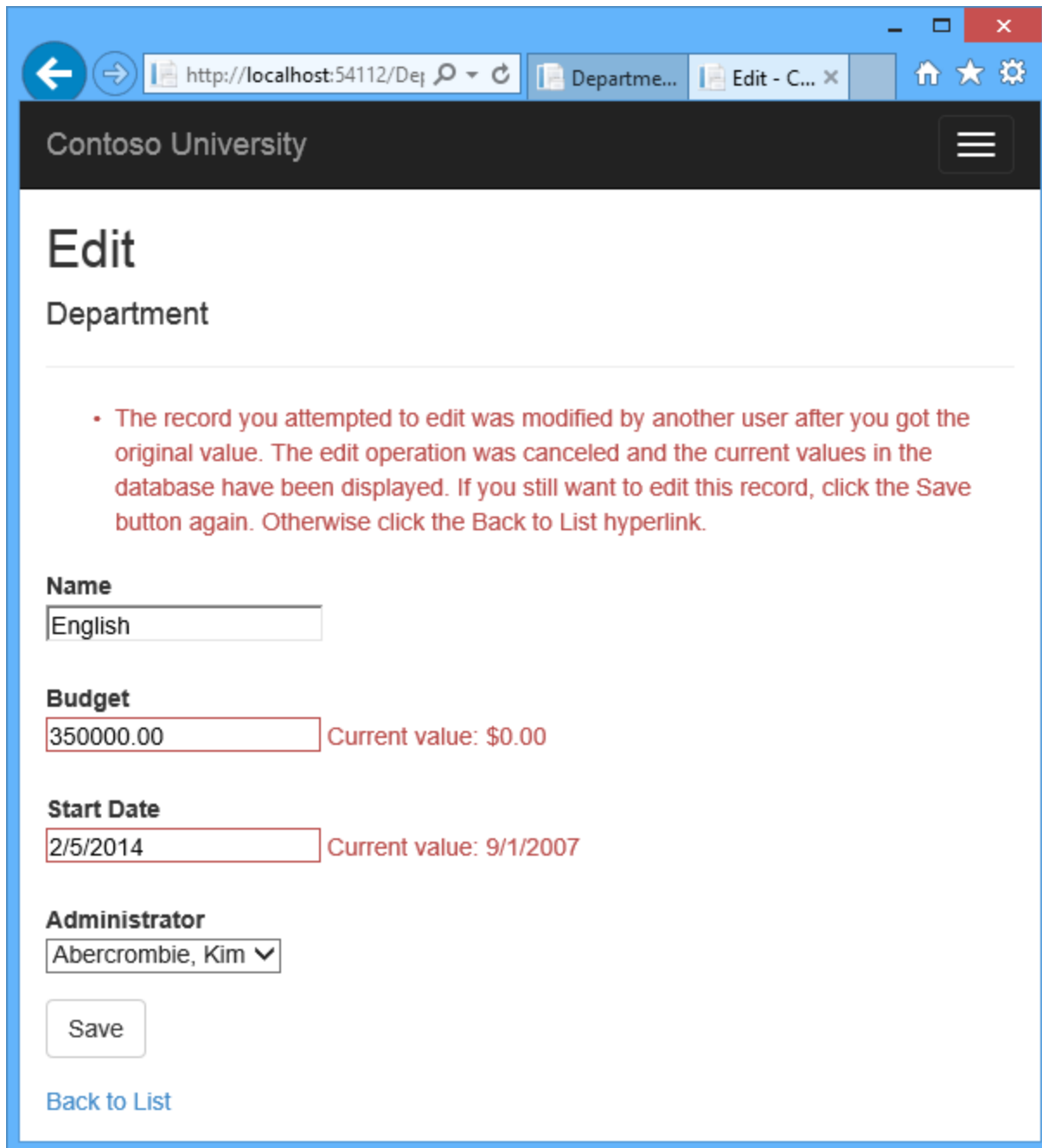
You'll change the web pages that work with the `Department` entity so that they handle concurrency errors. The following illustrations show the Index and Delete pages, including some messages that are displayed if a concurrency conflict occurs.



The screenshot shows a web browser window with the URL `http://localhost:54112/Dej`. The page title is "Departments - Contoso University". The main heading is "Departments" with a "Create New" link below it. A table lists the following departments:

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University



Concurrency Conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if it isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

Pessimistic Concurrency (Locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the Departments Edit page, changes the **Budget** amount for the English department from \$350,000.00 to \$0.00.

Contoso University

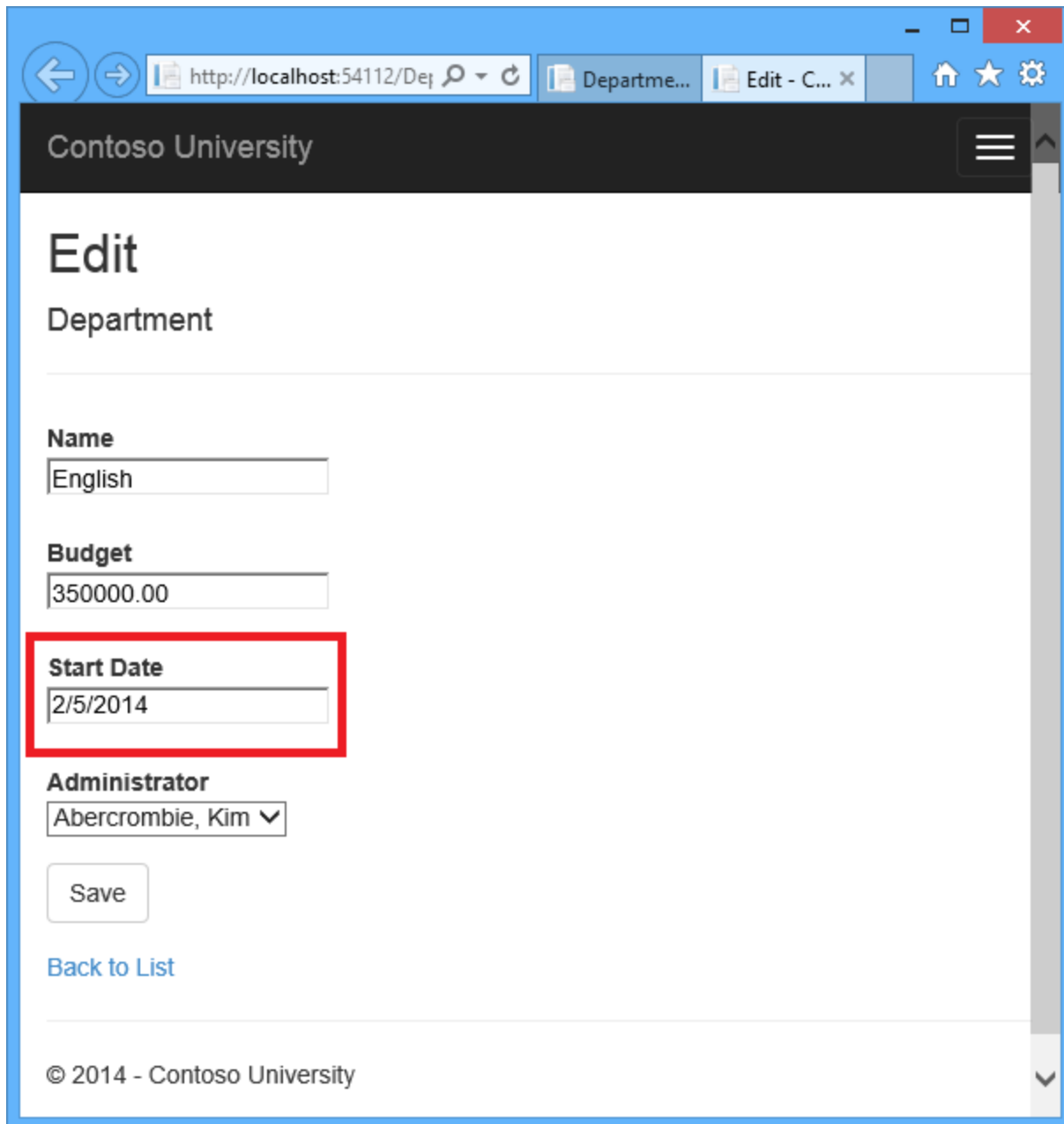
Departments

[Create New](#)

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Before John clicks **Save**, Jane runs the same page and changes the **Start Date** field from 9/1/2007 to 8/8/2013.



John clicks **Save** first and sees his change when the browser returns to the Index page, then Jane clicks **Save**. What happens next is determined by how you handle concurrency conflicts. Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both John's and Jane's changes — a start date of 8/8/2013 and a budget of Zero dollars.

This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let Jane's change overwrite John's change. The next time someone browses the English department, they'll see 8/8/2013 and the restored \$350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.
- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reapply her changes if she still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting Concurrency Conflicts

You can resolve conflicts by handling [OptimisticConcurrencyException](#) exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of `SQL Update` or `Delete` commands.

The data type of the tracking column is typically [rowversion](#). The [rowversion](#) value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the [rowversion](#) column is different than the original value, so the `Update` or `Delete` statement can't find the row to update because of the `Where` clause. When the Entity Framework finds that no rows have been updated by the `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large `Where` clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the [ConcurrencyCheck](#) attribute to them. That change enables the Entity Framework to include all columns in the SQL `WHERE` clause of `UPDATE` statements.

In the remainder of this tutorial you'll add a [rowversion](#) tracking property to the `Department` entity, create a controller and views, and test to verify that everything works correctly.

Add an Optimistic Concurrency Property to the Department Entity

In `Models\Department.cs`, add a tracking property named `RowVersion`:

```
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; }

    [Display(Name = "Administrator")]
    public int? InstructorID { get; set; }

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

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

The [Timestamp](#) attribute specifies that this column will be included in the `Where` clause of `Update` and `Delete` commands sent to the database. The attribute is called [Timestamp](#) because

previous versions of SQL Server used a SQL [timestamp](#) data type before the SQL [rowversion](#) replaced it. The .Net type for [rowversion](#) is a byte array.

If you prefer to use the fluent API, you can use the [IsConcurrencyToken](#) method to specify the tracking property, as shown in the following example:

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

By adding a property you changed the database model, so you need to do another migration. In the Package Manager Console (PMC), enter the following commands:

```
Add-Migration RowVersion  
Update-Database
```

Modify the Department Controller

In *Controllers\DepartmentController.cs*, add a `using` statement:

```
using System.Data.Entity.Infrastructure;
```

In the *DepartmentController.cs* file, change all four occurrences of "LastName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewBag.InstructorID = new SelectList(db.Instructors, "InstructorID",  
"FullName");
```

Replace the existing code for the `HttpPost Edit` method with the following code:

```
[HttpPost]  
[ValidateAntiForgeryToken]  
public async Task<ActionResult> Edit(  
    [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion,  
    InstructorID")]  
    Department department)  
{  
    try  
    {  
        if (ModelState.IsValid)  
        {  
            db.Entry(department).State = EntityState.Modified;  
            await db.SaveChangesAsync();  
            return RedirectToAction("Index");  
        }  
    }  
    catch (DbUpdateConcurrencyException ex)  
    {  
        var entry = ex.Entries.Single();  
        var clientValues = (Department)entry.Entity;  
        var databaseEntry = entry.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: "
                    + String.Format("{0:c}", databaseValues.Budget));
            if (databaseValues.StartDate != clientValues.StartDate)
                ModelState.AddModelError("StartDate", "Current value: "
                    + String.Format("{0:d}", databaseValues.StartDate));
            if (databaseValues.InstructorID != clientValues.InstructorID)
                ModelState.AddModelError("InstructorID", "Current value: "
                    +
db.Instructors.Find(databaseValues.InstructorID).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.");
            department.RowVersion = databaseValues.RowVersion;
        }
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to
write a log.
        ModelState.AddModelError(string.Empty, "Unable to save changes. Try
again, and if the problem persists contact your system administrator.");
    }

    ViewBag.InstructorID = new SelectList(db.Instructors, "ID", "FullName",
department.InstructorID);
    return View(department);
}

```

The view will store the original `RowVersion` value in a hidden field. When the model binder creates the `department` instance, that object will have the original `RowVersion` property value and the new values for the other properties, as entered by the user on the Edit page. Then when the Entity Framework creates a SQL `UPDATE` command, that command will include a `WHERE` clause that looks for a row that has the original `RowVersion` value.

If no rows are affected by the `UPDATE` command (no rows have the original `RowVersion` value), the Entity Framework throws a [DbUpdateConcurrencyException](#) exception, and the code in the `catch` block gets the affected `Department` entity from the exception object.

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

This object has the new values entered by the user in its `Entity` property, and you can get the values read from the database by calling the `GetDatabaseValues` method.

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

The `GetDatabaseValues` method returns `null` if someone has deleted the row from the database; otherwise, you have to cast the object returned to the `Department` class in order to access the `Department` properties.

```
if (databaseEntry == null)
{
    ModelState.AddModelError(string.Empty,
        "Unable to save changes. The department was deleted by another user.");
}
else
{
    var databaseValues = (Department)databaseEntry.ToObject();
```

Next, the code adds a custom error message for each column that has database values different from what the user entered on the Edit page:

```
if (databaseValues.Name != currentValues.Name)
    ModelState.AddModelError("Name", "Current value: " +
        databaseValues.Name);
// ...
```

A longer error message explains what happened and what to do about it:

```
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.");
```

Finally, the code sets the `RowVersion` value of the `Department` object to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

In `Views\Department\Edit.cshtml`, add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property:

```
@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)
    </div>
}
```

Testing Optimistic Concurrency Handling

Run the site and click **Departments**:

Contoso University

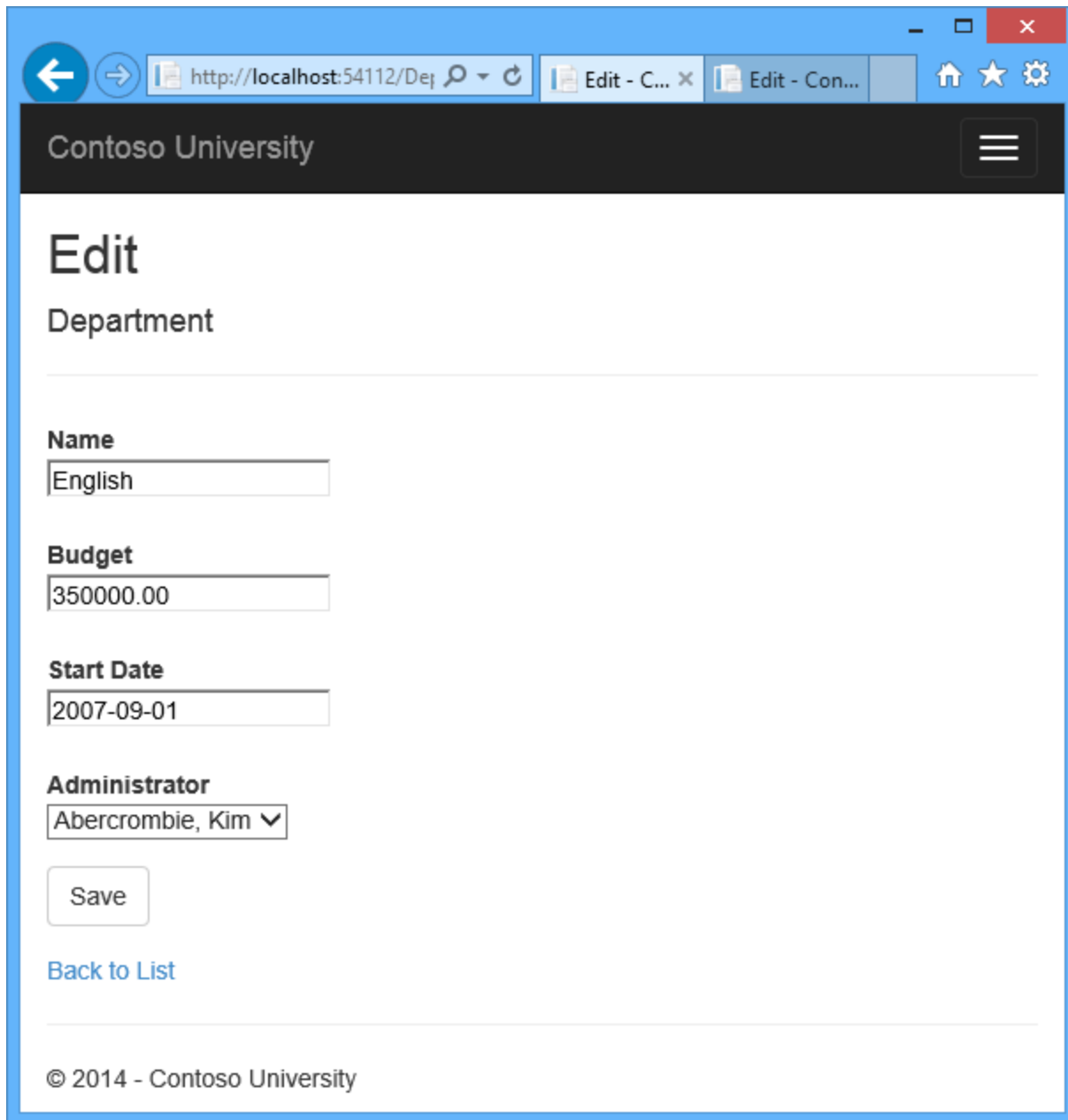
Departments

[Create New](#)

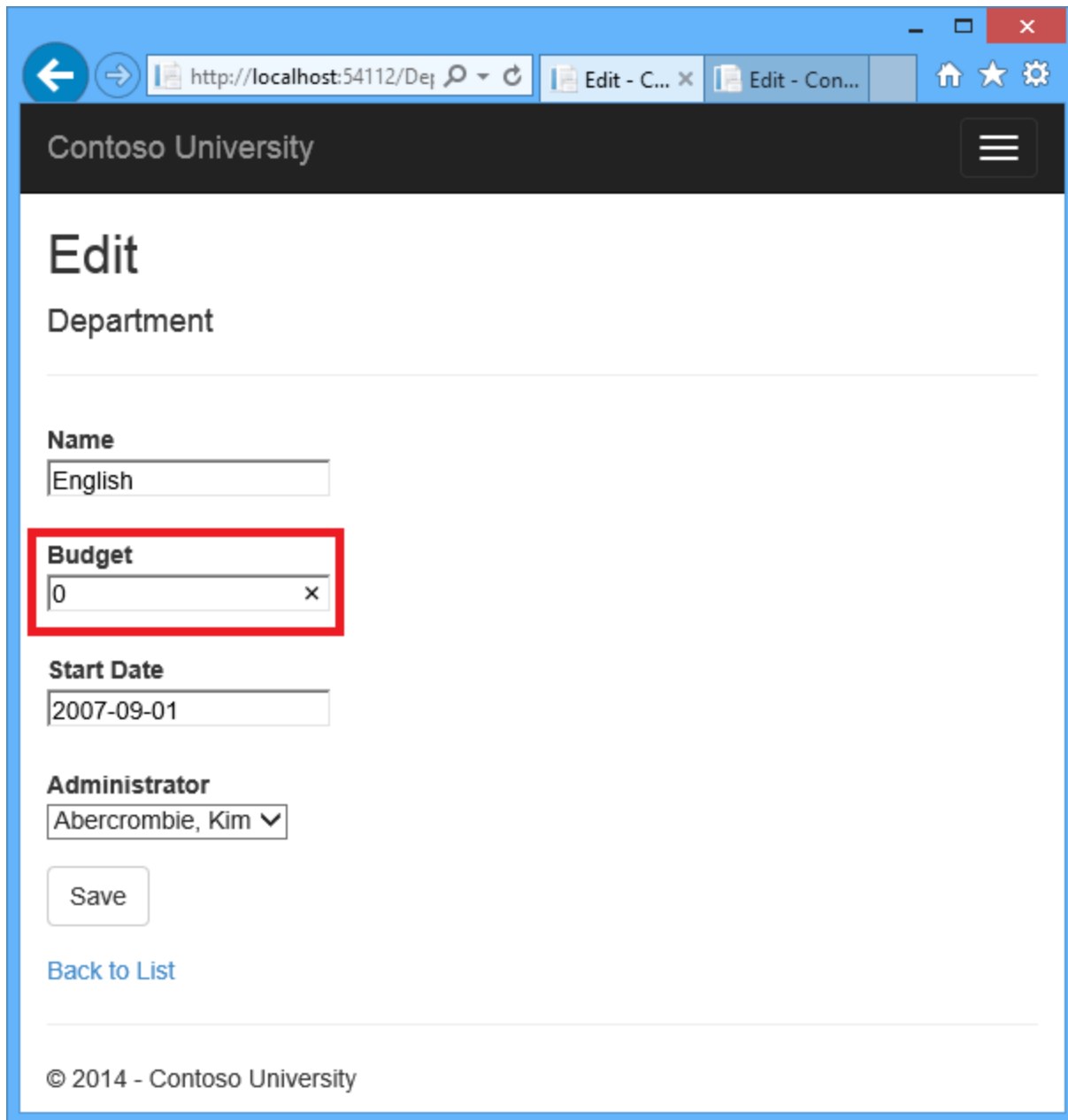
Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$350,000.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Right click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two tabs display the same information.



Change a field in the first browser tab and click **Save**.



The browser shows the Index page with the changed value.

Contoso University

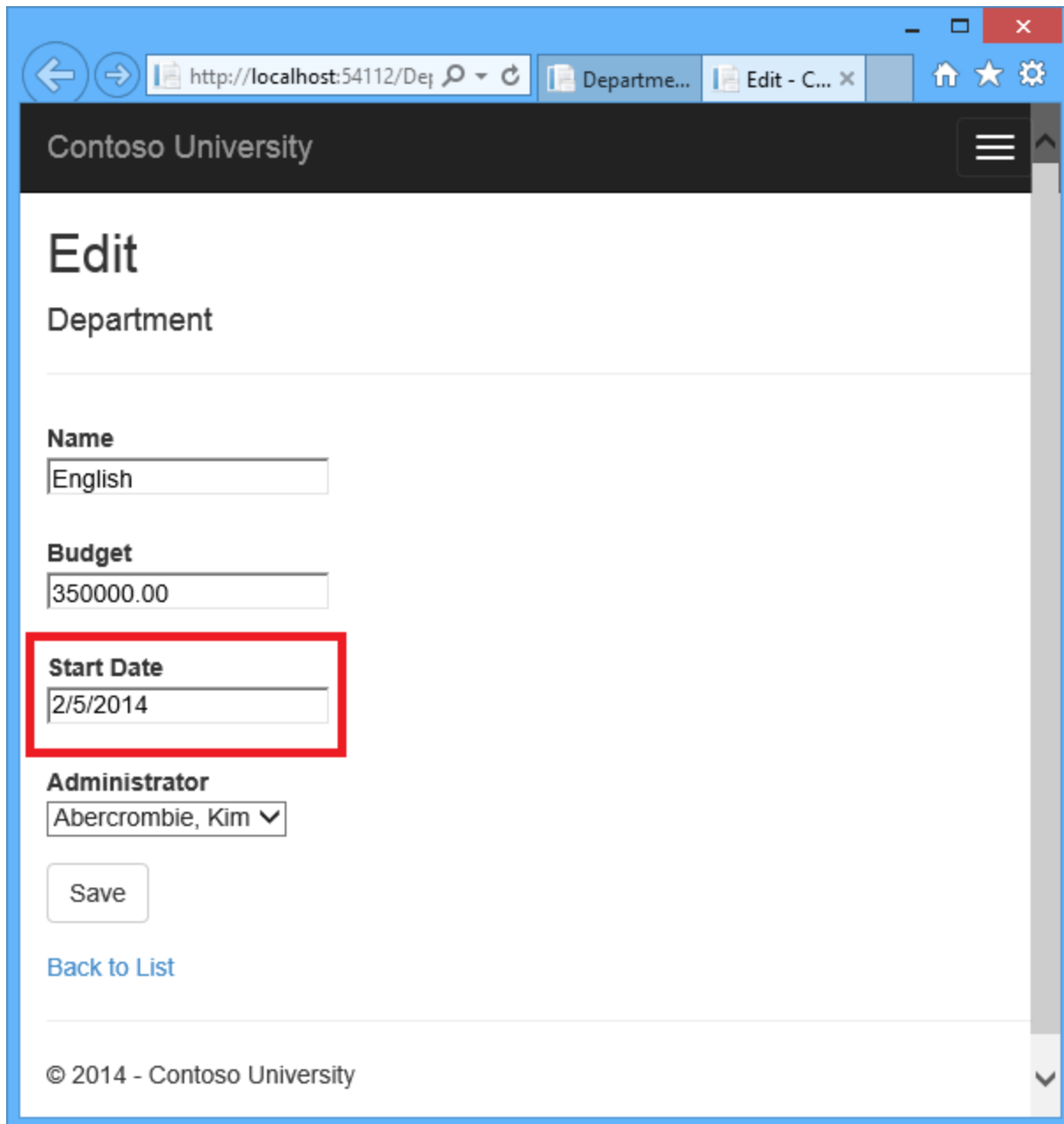
Departments

[Create New](#)

Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2007-09-01	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

Change a field in the second browser tab and click **Save**.



Click **Save** in the second browser tab. You see an error message:

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

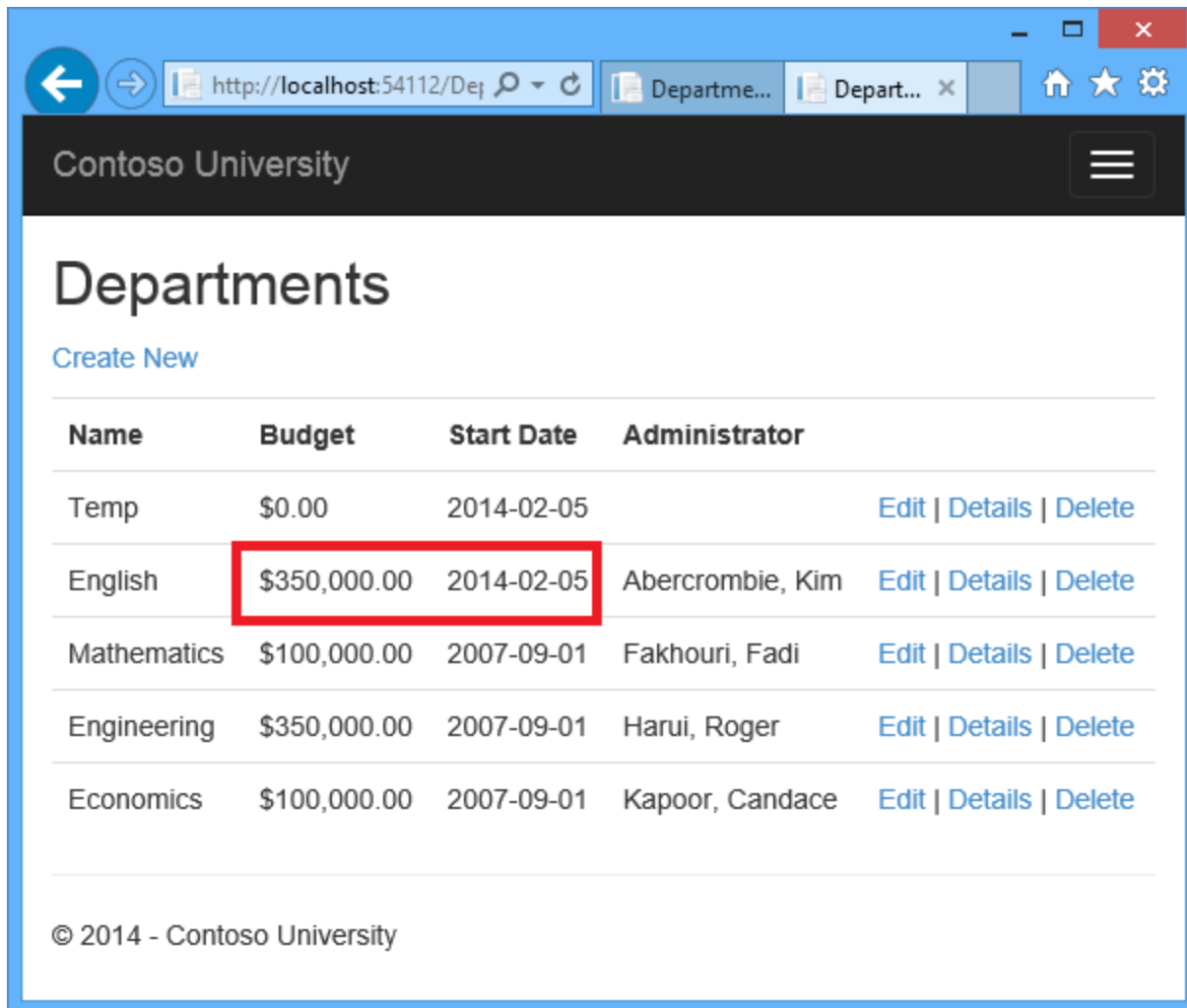
Budget
 Current value: \$0.00

Start Date
 Current value: 9/1/2007

Administrator

[Back to List](#)

Click **Save** again. The value you entered in the second browser tab is saved along with the original value of the data you changed in the first browser. You see the saved values when the Index page appears.



Updating the Delete Page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL `DELETE` command, it includes a `WHERE` clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to `true` in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case a different error message is displayed.

In `DepartmentController.cs`, replace the `HttpGet Delete` method with the following code:

```
public ActionResult Delete(int? id, bool? concurrencyError)
```

```

{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = db.Departments.Find(id);
    if (department == null)
    {
        return HttpNotFound();
    }

    if (ConcurrencyError.GetValueOrDefault())
    {
        if (department == null)
        {
            ViewBag.ConcurrencyErrorMessage = "The record you attempted to
delete "
                + "was deleted by another user after you got the original
values. "
                + "Click the Back to List hyperlink.";
        }
        else
        {
            ViewBag.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);
}

```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is `true`, an error message is sent to the view using a `ViewBag` property.

Replace the code in the `HttpPost Delete` method (named `DeleteConfirmed`) with the following code:

```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(Department department)
{
    try
    {
        db.Entry(department).State = EntityState.Deleted;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}

```

```

    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToAction("Delete", new { concurrencyError=true } );
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name after DataException and
        add a line here to write a log.
        ModelState.AddModelError(string.Empty, "Unable to delete. Try again,
        and if the problem persists contact your system administrator.");
        return View(department);
    }
}

```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public ActionResult DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives you access to the `RowVersion` property value in addition to the record key.

```
public ActionResult Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If a concurrency error is caught, the code redisplay the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

In `Views\Department\Delete.cshtml`, replace the scaffolded code with the following code that adds an error message field and hidden fields for the `DepartmentID` and `RowVersion` properties. The changes are highlighted.

```

@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

<p class="error">@ViewBag.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">

```

```

        <dt>
            Administrator
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>

        <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>
    </dl>

    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()
        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>

```

This code adds an error message between the h2 and h3 headings:

```
<p class="error">@ViewBag.ConcurrencyErrorMessage</p>
```

It replaces LastName with FullName in the Administrator field:

```

<dt>
    Administrator
</dt>

```

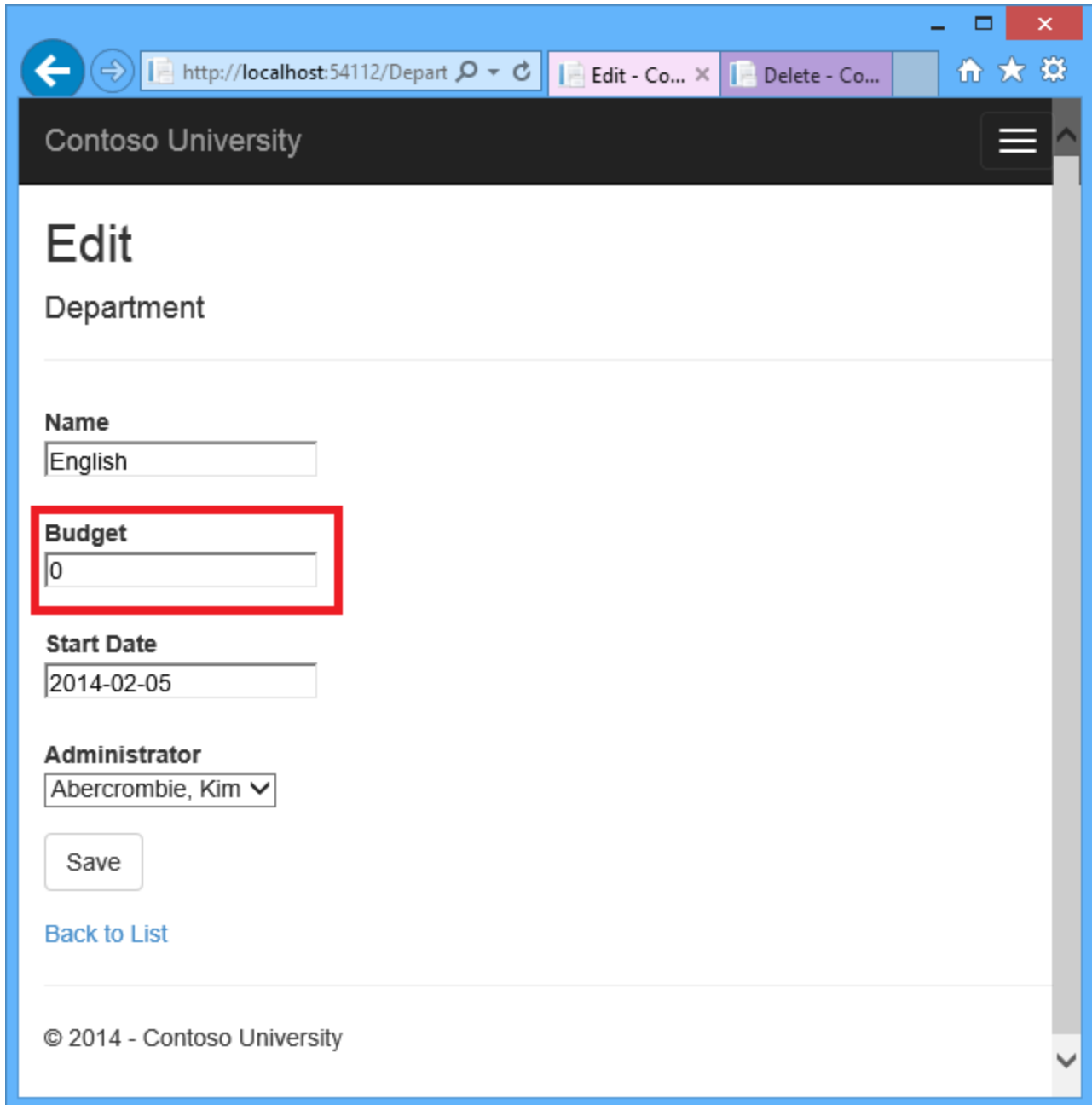
```
<dd>  
    @Html.DisplayFor(model => model.Administrator.FullName)  
</dd>
```

Finally, it adds hidden fields for the `DepartmentID` and `RowVersion` properties after the `Html.BeginForm` statement:

```
@Html.HiddenFor(model => model.DepartmentID)  
@Html.HiddenFor(model => model.RowVersion)
```

Run the `Departments Index` page. Right click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save** :



The Index page confirms the change.

Contoso University

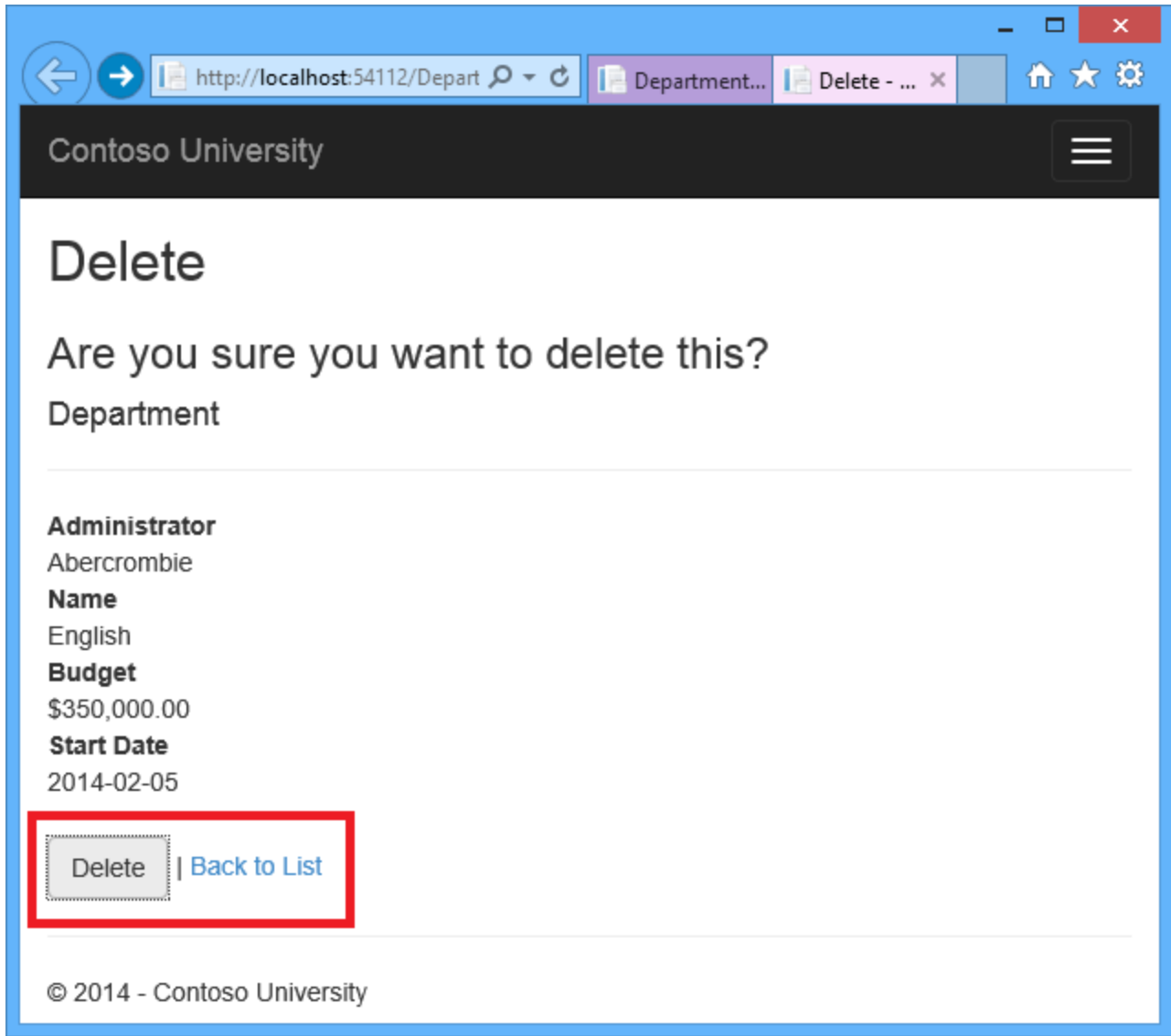
Departments

[Create New](#)

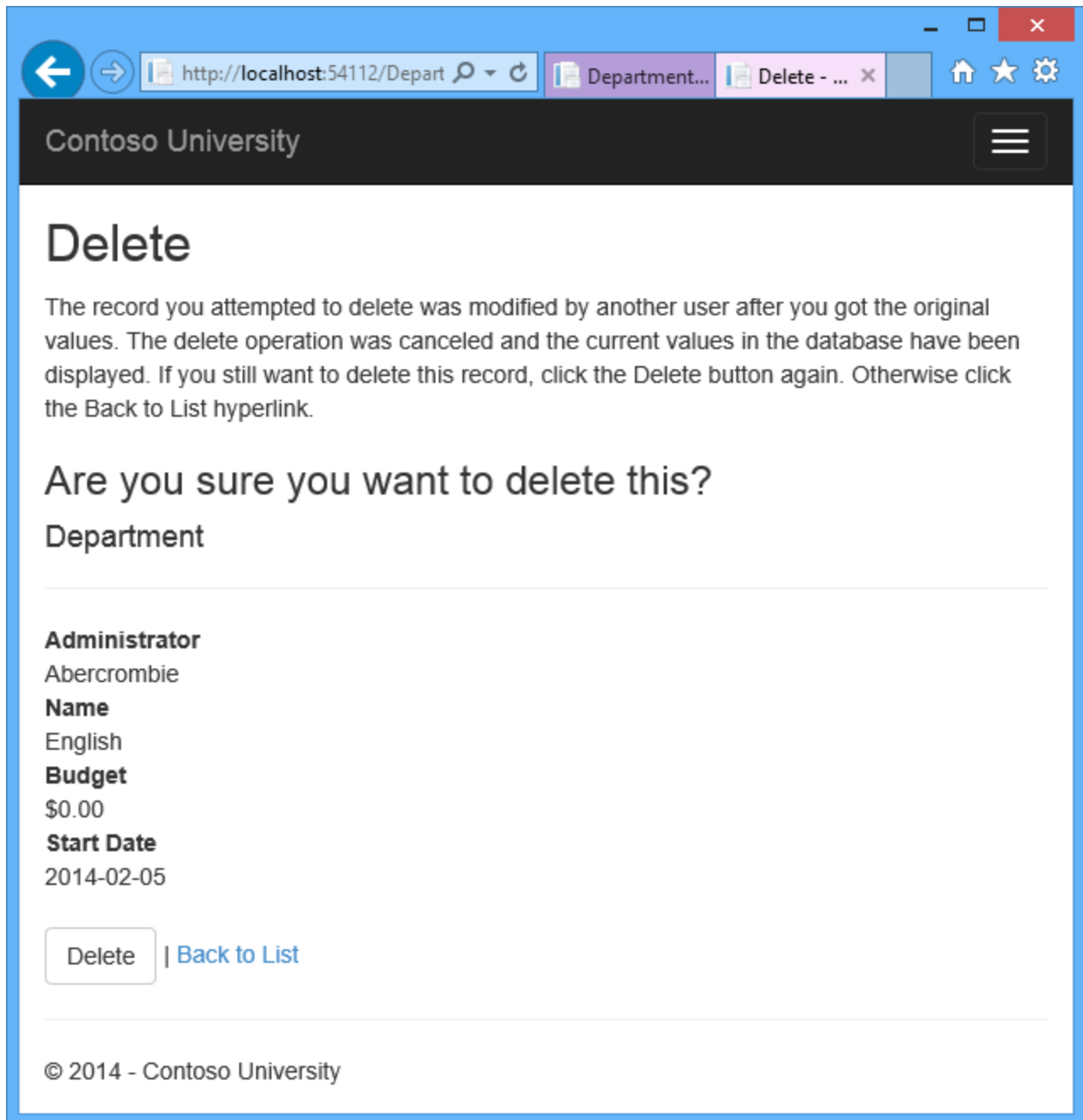
Name	Budget	Start Date	Administrator	
Temp	\$0.00	2014-02-05		Edit Details Delete
English	\$0.00	2014-02-05	Abercrombie, Kim	Edit Details Delete
Mathematics	\$100,000.00	2007-09-01	Fakhouri, Fadi	Edit Details Delete
Engineering	\$350,000.00	2007-09-01	Harui, Roger	Edit Details Delete
Economics	\$100,000.00	2007-09-01	Kapoor, Candace	Edit Details Delete

© 2014 - Contoso University

In the second tab, click **Delete**.



You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Summary

This completes the introduction to handling concurrency conflicts. For information about other ways to handle various concurrency scenarios, see [Optimistic Concurrency Patterns](#) and [Working with Property Values](#) on MSDN. The next tutorial shows how to implement table-per-hierarchy inheritance for the `Instructor` and `Student` entities.

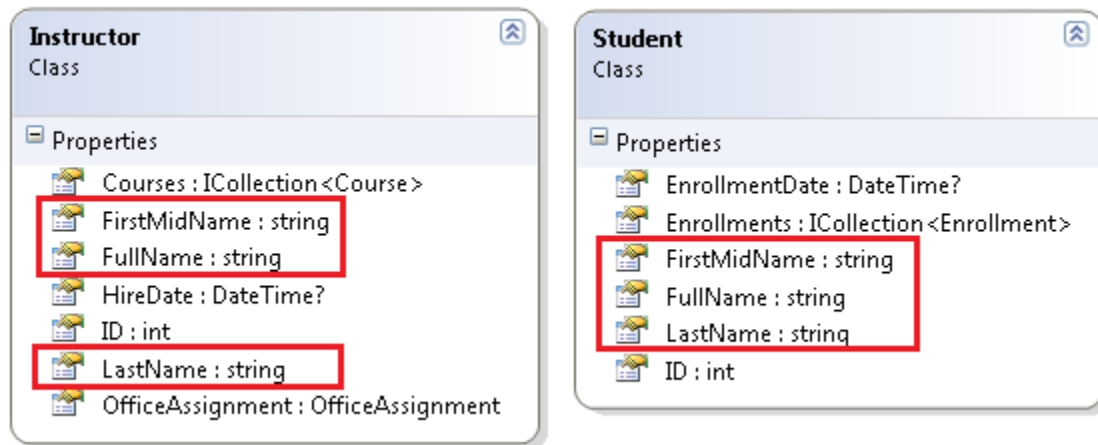
Implementing Inheritance with the Entity Framework 6 in an ASP.NET MVC 5 Application (11 of 12)

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

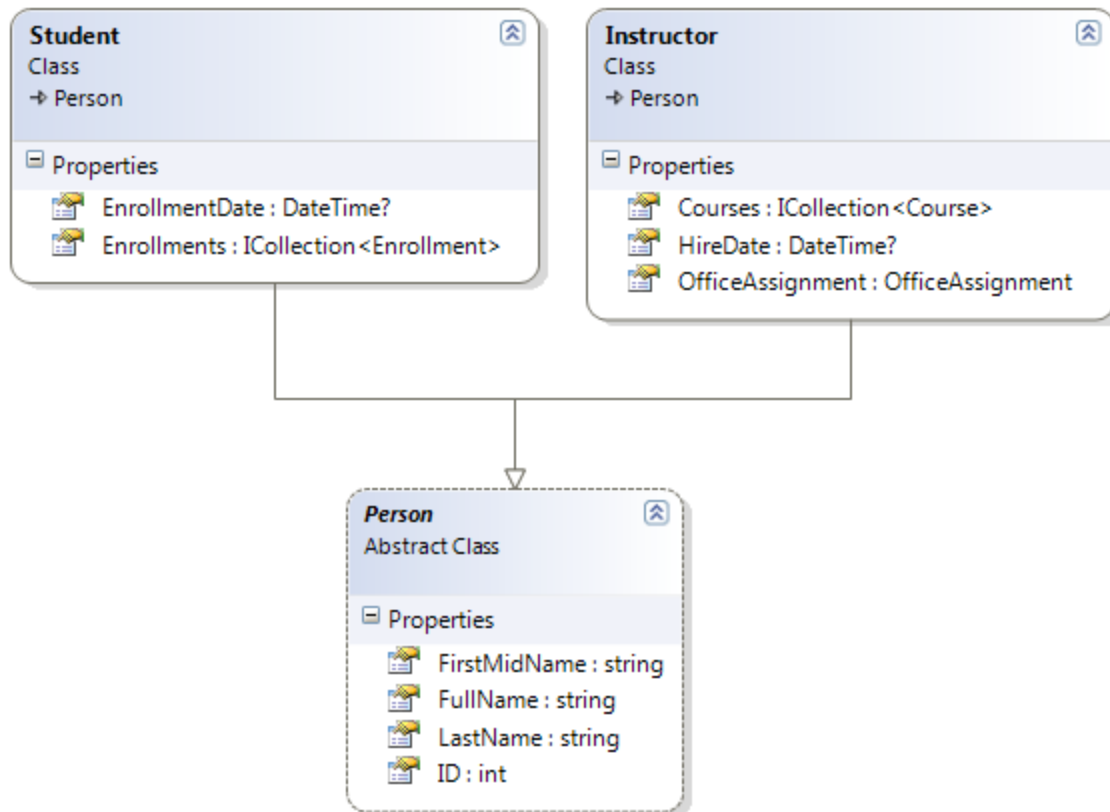
In object-oriented programming, you can use [inheritance](#) to facilitate [code reuse](#). In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

Options for mapping inheritance to database tables

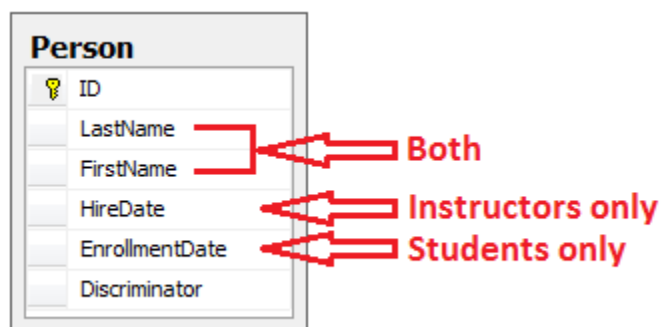
The `Instructor` and `Student` classes in the `School` data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class which contains only those shared properties, then make the `Instructor` and `Student` entities inherit from that base class, as shown in the following illustration:

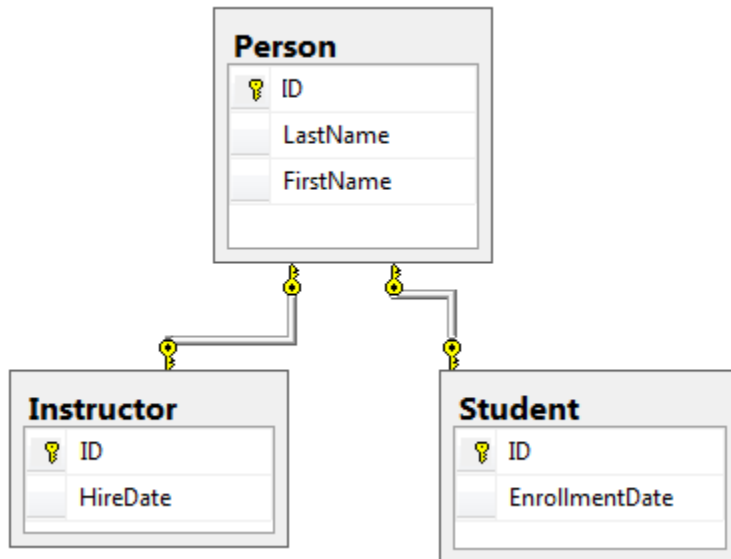


There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), some to both (`LastName`, `FirstName`). Typically, you'd have a *discriminator* column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy* (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.



This pattern of making a database table for each entity class is called *table per type* (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the `Person`, `Student`, and `Instructor` classes as shown earlier, the `Student` and `Instructor` tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the default inheritance pattern in the Entity Framework, so all you have to do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration. (For information about how to implement the other inheritance patterns, see [Mapping the Table-Per-Type \(TPT\) Inheritance](#) and [Mapping the Table-Per-Concrete Class \(TPC\) Inheritance](#) in the MSDN Entity Framework documentation.)

Create the Person class

In the `Models` folder, create `Person.cs` and replace the template code with the following code:

```
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;
            }
        }
    }
}

```

Make Student and Instructor classes inherit from Person

In *Instructor.cs*, derive the `Instructor` class from the `Person` class and remove the key and name fields. The code will look like the following example:

```

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

```

Make similar changes to *Student.cs*. The `Student` class will look like the following example:

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

Add the Person Entity Type to the Model

In *SchoolContext.cs*, add a `DbSet` property for the `Person` entity type:

```
public DbSet<Person> People { get; set; }
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a `Person` table in place of the `Student` and `Instructor` tables.

Create and Update a Migrations File

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

```
Add-Migration Inheritance
```

Run the `Update-Database` command in the PMC. The command will fail at this point because we have existing data that migrations doesn't know how to handle. You get an error message like the following one:

Could not drop object 'dbo.Instructor' because it is referenced by a FOREIGN KEY constraint.

Open *Migrations\<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
public override void Up()
{
```

```

    // Drop foreign keys and indexes that point to tables we're going to
drop.
DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
DropIndex("dbo.Enrollment", new[] { "StudentID" });

RenameTable(name: "dbo.Instructor", newName: "Person");
AddColumn("dbo.Person", "EnrollmentDate", c => c.DateTime());
AddColumn("dbo.Person", "Discriminator", c => c.String(nullable: false,
maxLength: 128, defaultValue: "Instructor"));
AlterColumn("dbo.Person", "HireDate", c => c.DateTime());
AddColumn("dbo.Person", "OldId", c => c.Int(nullable: true));

// Copy existing Student data into new Person table.
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.
Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person
WHERE OldId = Enrollment.StudentId AND Discriminator = 'Student')");

// Remove temporary key
DropColumn("dbo.Person", "OldId");

DropTable("dbo.Student");

// Re-create foreign keys and indexes pointing to new table.
AddForeignKey("dbo.Enrollment", "StudentID", "dbo.Person", "ID",
cascadeDelete: true);
CreateIndex("dbo.Enrollment", "StudentID");
}

```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
 - Adds nullable EnrollmentDate for students.
 - Adds Discriminator column to indicate whether a row is for a student or an instructor.
 - Makes HireDate nullable since student rows won't have hire dates.
 - Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they'll get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `update-database` command again.

(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

Note: It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can continue with the tutorial by changing the connection string in the `Web.config` file or by deleting the database. The simplest approach is to rename the database in the `Web.config` file. For example, change the database name to `ContosoUniversity2` as shown in the following example:

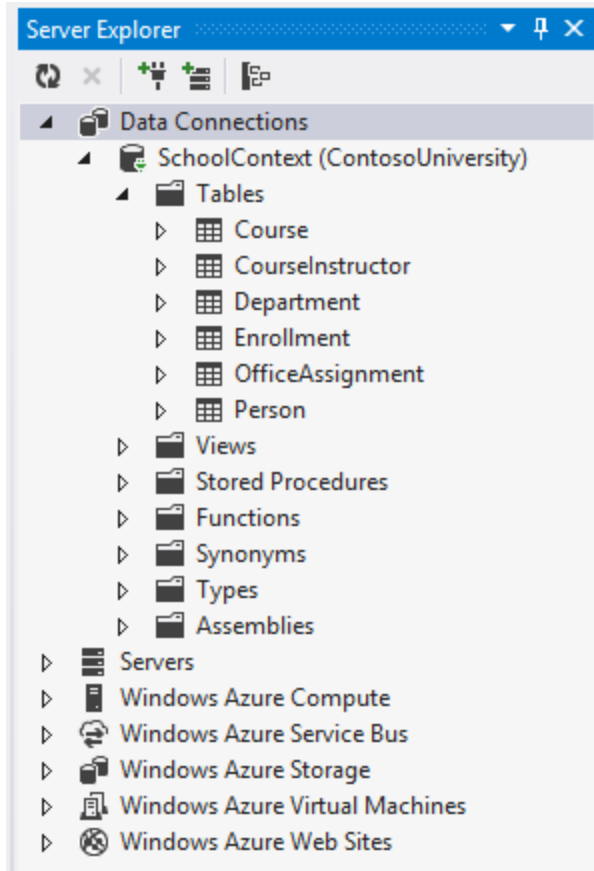
```
<add name="SchoolContext"
      connectionString="Data Source=(LocalDb)\v11.0;Initial
Catalog=ContosoUniversity2;Integrated Security=SSPI;"
      providerName="System.Data.SqlClient" />
```

With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see [How to Drop a Database from Visual Studio 2012](#). If you take this approach in order to continue with the tutorial, skip the deployment step at the end of this tutorial or deploy to a new site and database. If you deploy an update to the same site you've been deploying to already, EF will get the same error there when it runs migrations automatically. If you want to troubleshoot a migrations error, the best resource is one of the Entity Framework forums or [StackOverflow.com](#).

Testing

Run the site and try various pages. Everything works the same as it did before.

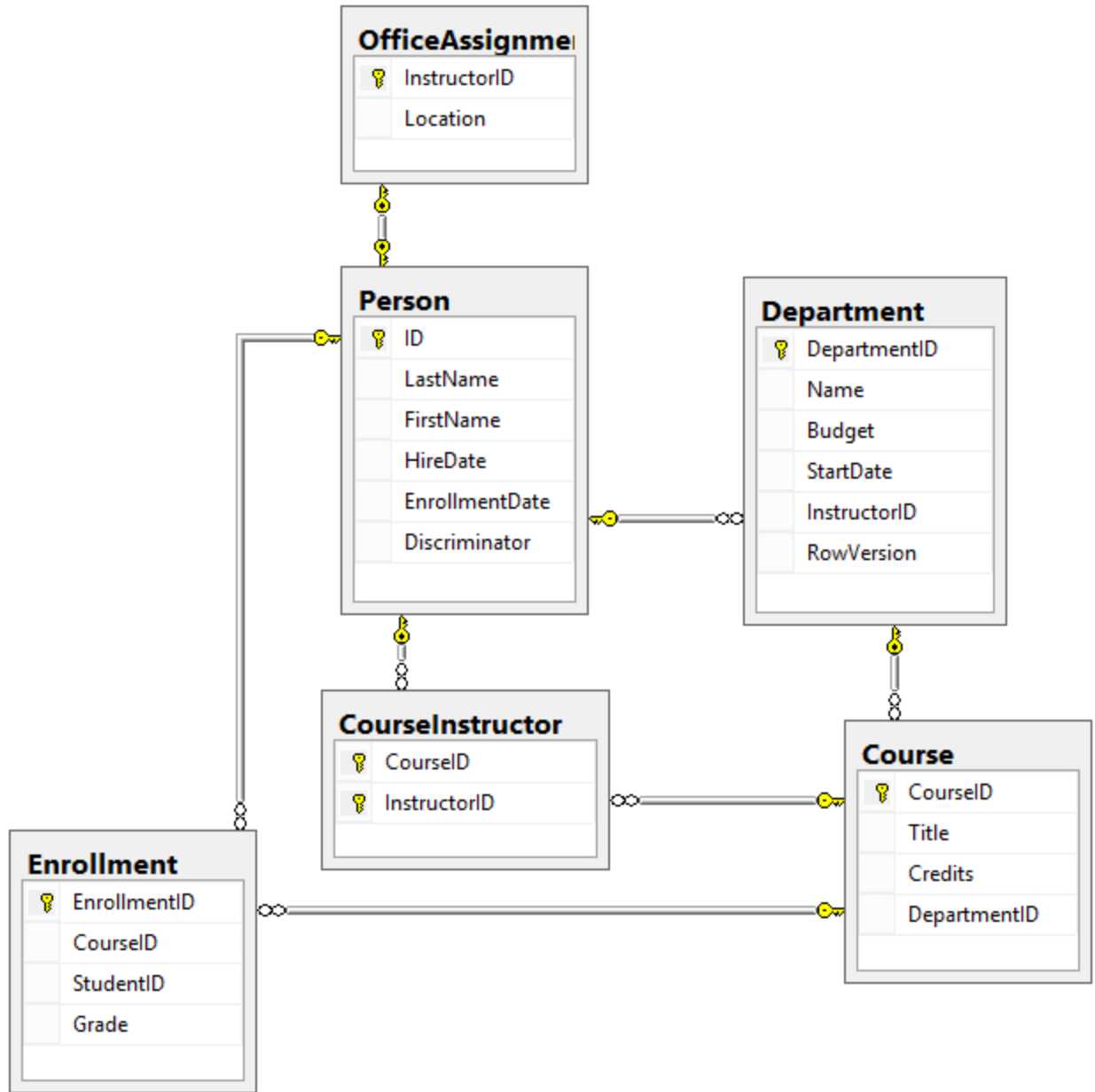
In **Server Explorer**, expand **Data Connections\SchoolContext** and then **Tables**, and you see that the **Student** and **Instructor** tables have been replaced by a **Person** table. Expand the **Person** table and you see that it has all of the columns that used to be in the **Student** and **Instructor** tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.

	PersonID	LastName	FirstName	HireDate	EnrollmentDate	Discriminator
▶	1	Alexander	Carson	NULL	9/1/2010 12:00...	Student
	2	Alonso	Meredith	NULL	9/1/2012 12:00...	Student
	3	Anand	Arturo	NULL	9/1/2013 12:00...	Student
	4	Barzdukas	Gytis	NULL	9/1/2012 12:00...	Student
	5	Li	Yan	NULL	9/1/2012 12:00...	Student
	6	Justice	Peggy	NULL	9/1/2011 12:00...	Student
	7	Norman	Laura	NULL	9/1/2013 12:00...	Student
	8	Olivetto	Nino	NULL	9/1/2005 12:00...	Student
	9	Abercrombie	Kim	3/11/1995 1...	NULL	Instructor
	10	Fakhouri	Fadi	7/6/2002 12:...	NULL	Instructor
	11	Harui	Roger	7/1/1998 12:...	NULL	Instructor
	12	Kapoor	Candace	1/15/2001 1...	NULL	Instructor
	13	Zheng	Roger	2/12/2004 1...	NULL	Instructor
	14	Smith	Joe	NULL	5/23/2013 12:0...	Student
*	NULL	NULL	NULL	NULL	NULL	NULL

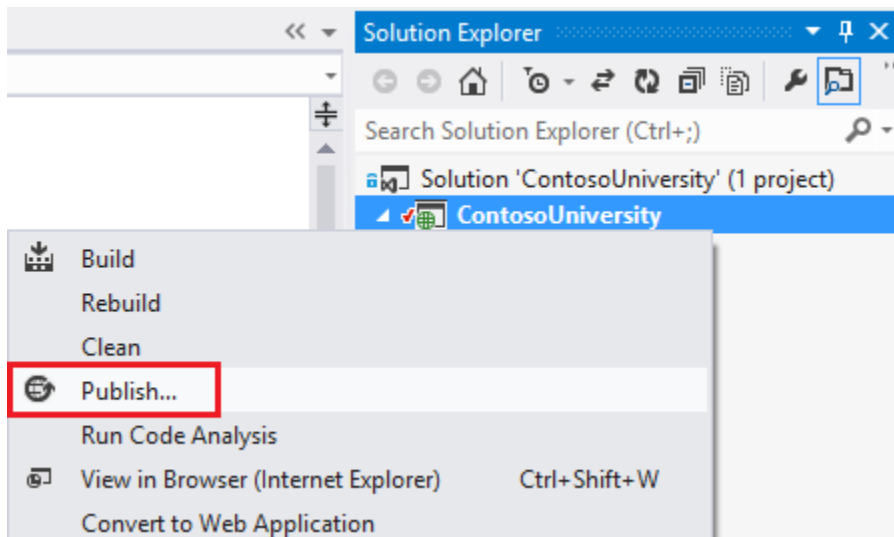
The following diagram illustrates the structure of the new School database:



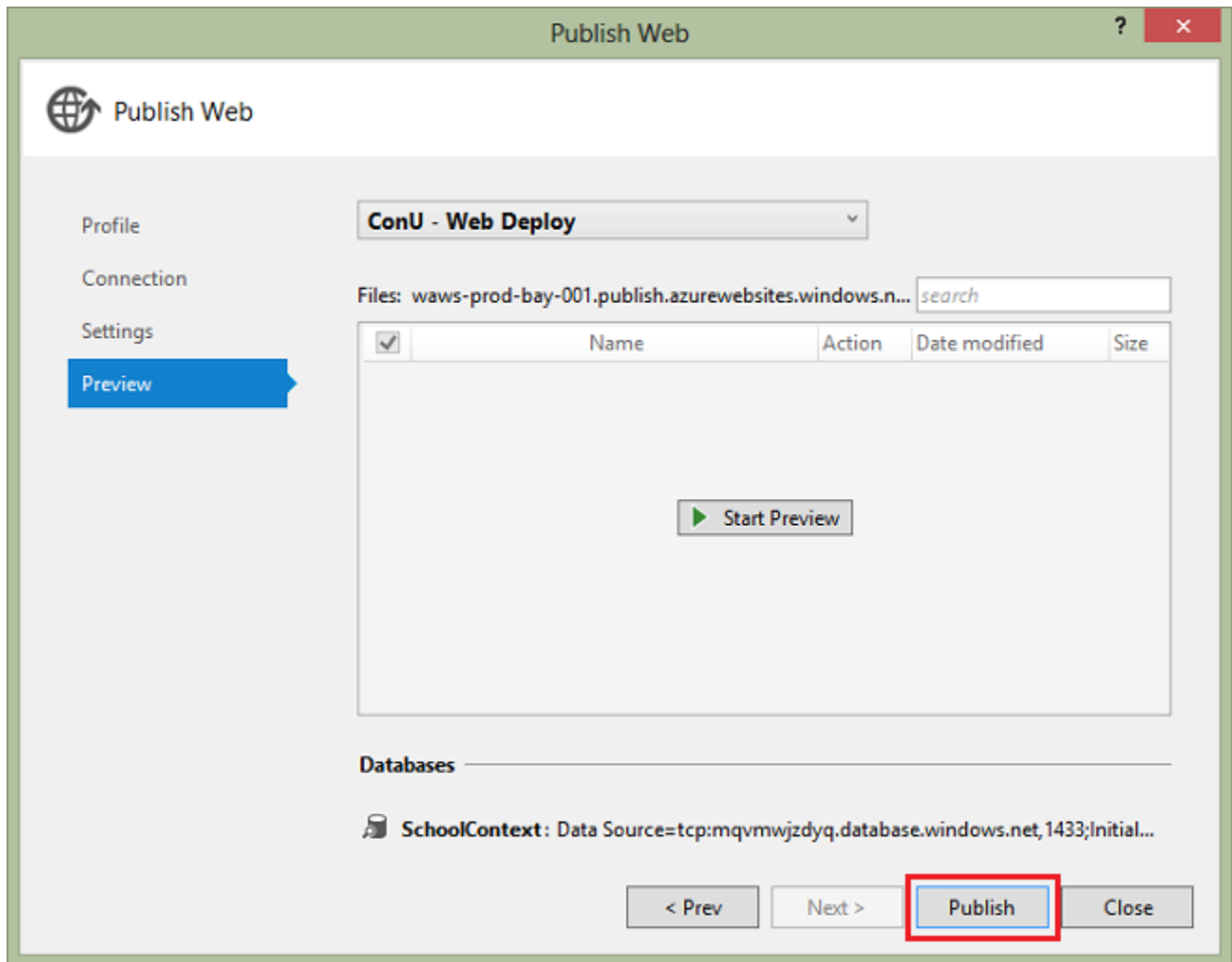
Deploy to Windows Azure

This section requires you to have completed the optional **Deploying the app to Windows Azure** section in [Part 3, Sorting, Filtering, and Paging](#) of this tutorial series. If you had migration errors that you resolved by deleting the database in your local project, skip this step; or create a new site and database, and deploy to the new environment.

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.



2. Click **Publish**.



The Web app will open in your default browser.

3. Test the application to verify it's working.

The first time you run a page that accesses the database, the Entity Framework runs all of the migrations `Up` methods required to bring the database up to date with the current data model.

Summary

You've implemented table-per-hierarchy inheritance for the `Person`, `Student`, and `Instructor` classes. For more information about this and other inheritance structures, see [TPT Inheritance Pattern](#) and [TPH Inheritance Pattern](#) on MSDN. In the next tutorial you'll see some ways to implement the repository and unit of work patterns.

Advanced Entity Framework 6 Scenarios for an MVC 5 Web Application (12 of 12)

In the previous tutorial you implemented table-per-hierarchy inheritance. This tutorial includes introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First. Step-by-step instructions walk you through the code and using Visual Studio for the following topics:

- Performing raw SQL queries
- Performing no-tracking queries
- Examining SQL sent to the database

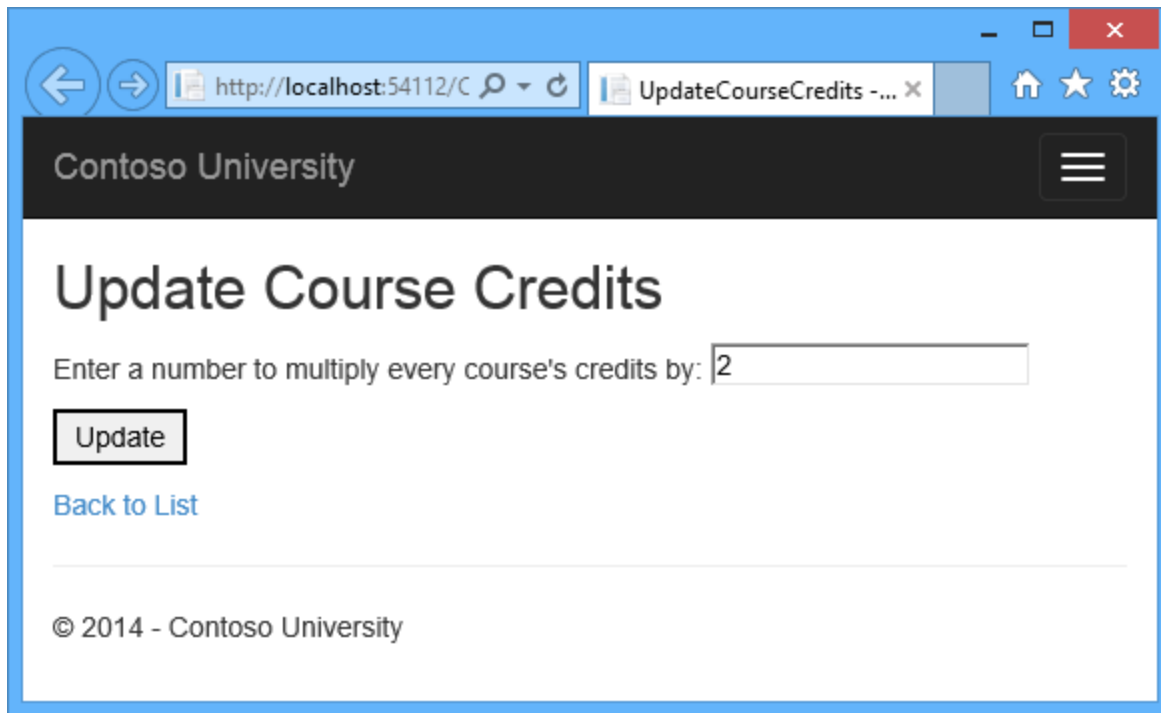
The tutorial introduces several topics with brief introductions followed by links to resources for more information:

- Repository and unit of work patterns
- Proxy classes
- Automatic change detection
- Automatic validation
- EF tools for Visual Studio
- Entity Framework source code

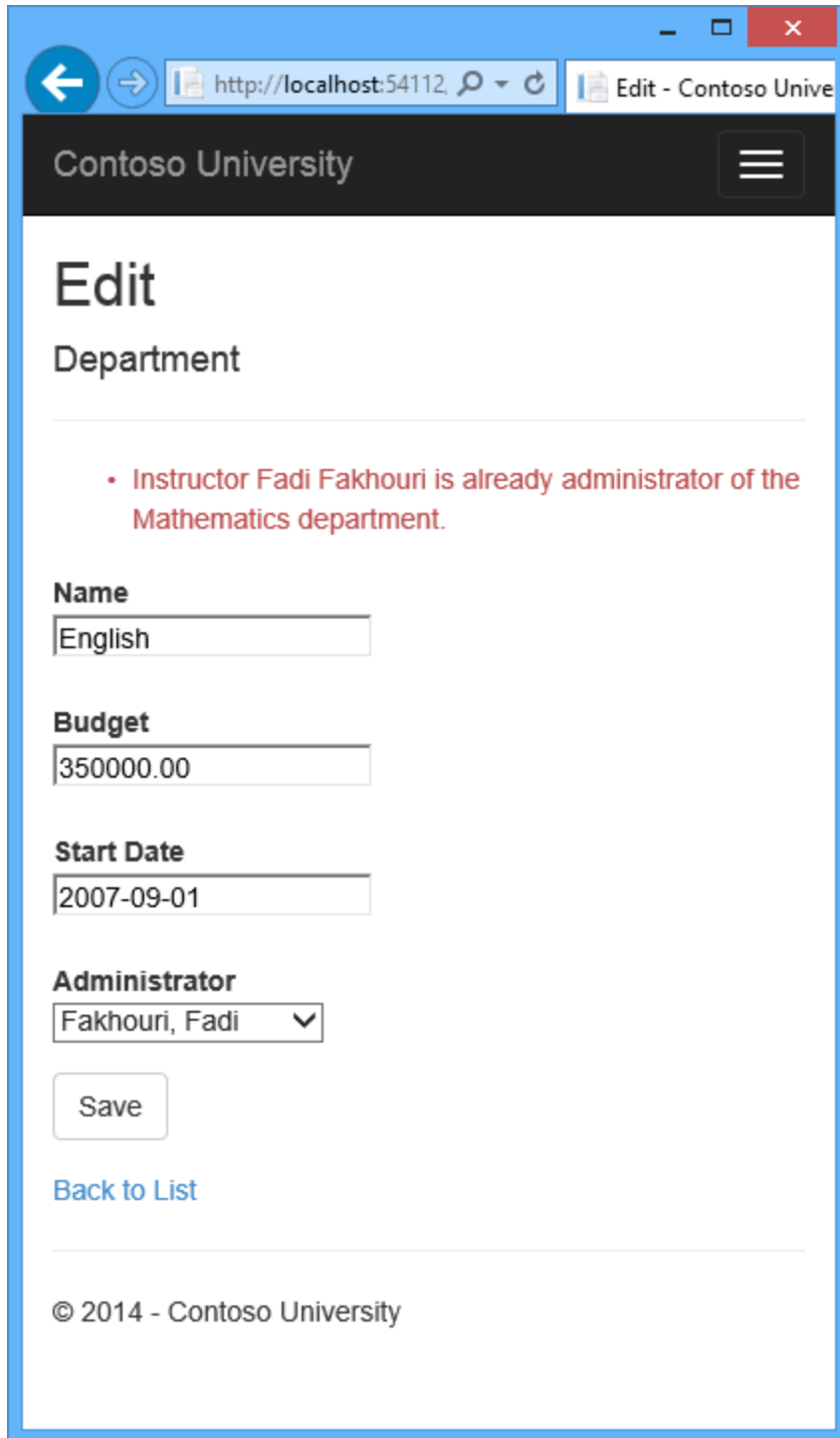
The tutorial also includes the following sections:

- Summary
- Acknowledgments
- A note about VB
- Common errors, and solutions or workarounds for them

For most of these topics, you'll work with pages that you already created. To use raw SQL to do bulk updates you'll create a new page that updates the number of credits of all courses in the database:



And to use a no-tracking query you'll add new validation logic to the Department Edit page:



Performing Raw SQL Queries

The Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options:

- Use the [DbSet.SqlQuery](#) method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you turn tracking off. (See the following section about the [AsNoTracking](#) method.)
- Use the [Database.SqlQuery](#) method for queries that return types that aren't entities. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.
- Use the [Database.ExecuteSqlCommand](#) for non-query commands.

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created, and these methods make it possible for you to handle such exceptions.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Calling a Query that Returns Entities

The [DbSet<TEntity>](#) class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the `Department` controller.

In `DepartmentController.cs`, replace the `db.Departments.Find` method call with a `db.Departments.SqlQuery` method call, as shown in the following highlighted code:

```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

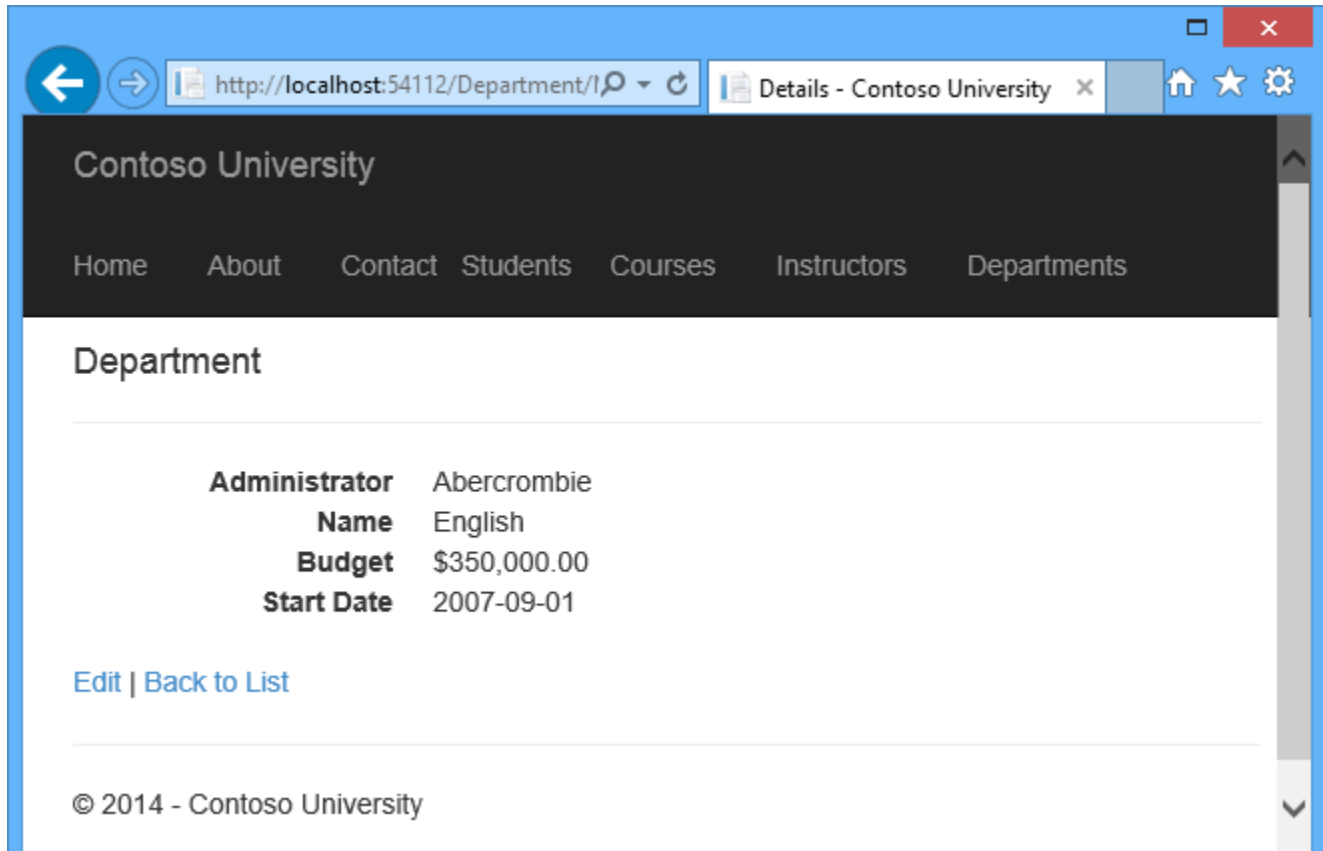
    // Commenting out original code to show how to use a raw SQL query.
    //Department department = await db.Departments.FindAsync(id);

    // Create and execute raw SQL query.
    string query = "SELECT * FROM Department WHERE DepartmentID = @p0";
    Department department = await db.Departments.SqlQuery(query,
id).SingleOrDefaultAsync();

    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

}

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Calling a Query that Returns Other Types of Objects

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. The code that does this in *HomeController.cs* uses LINQ:

```
var data = from student in db.Students
           group student by student.EnrollmentDate into dateGroup
           select new EnrollmentDateGroup()
           {
               EnrollmentDate = dateGroup.Key,
               StudentCount = dateGroup.Count()
           };
```

Suppose you want to write the code that retrieves this data directly in SQL rather than using LINQ. To do that you need to run a query that returns something other than entity objects, which means you need to use the [Database.SqlQuery](#) method.

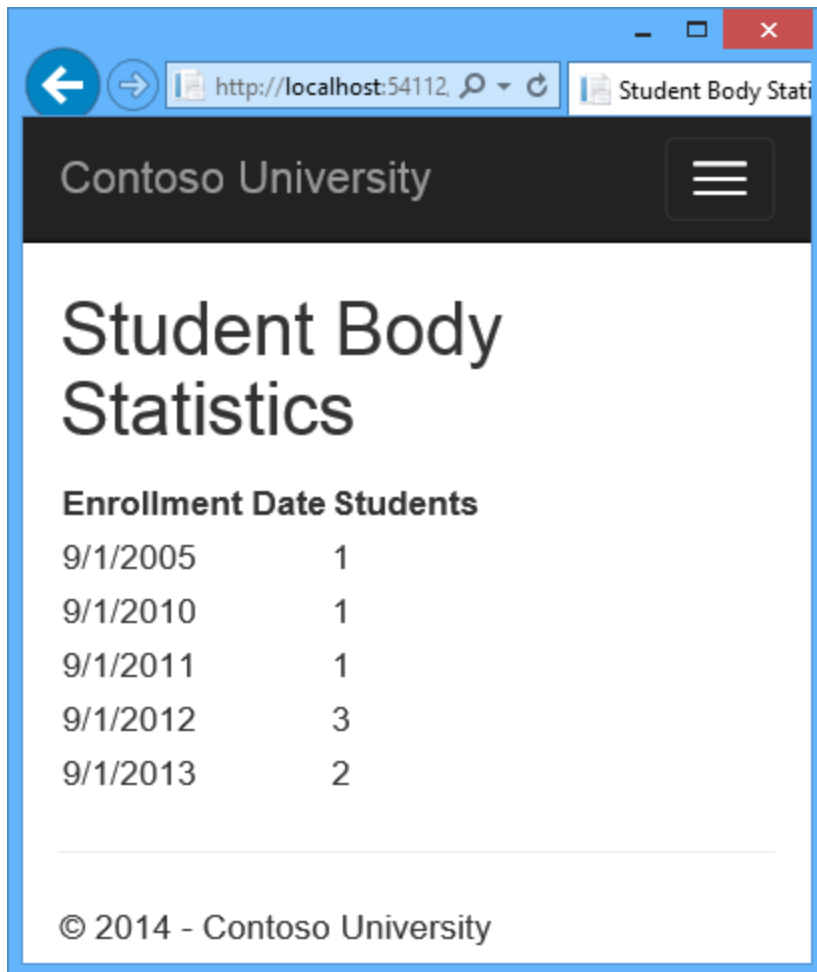
In *HomeController.cs*, replace the LINQ statement in the `About` method with a SQL statement, as shown in the following highlighted code:

```
public ActionResult About()
{
    // Commenting out LINQ to show how to do the same thing in SQL.
    //IQueryable<EnrollmentDateGroup> = from student in db.Students
    //    group student by student.EnrollmentDate into dateGroup
    //    select new EnrollmentDateGroup()
    //    {
    //        EnrollmentDate = dateGroup.Key,
    //        StudentCount = dateGroup.Count()
    //    };

    // SQL version of the above LINQ code.
    string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
        + "FROM Person "
        + "WHERE Discriminator = 'Student' "
        + "GROUP BY EnrollmentDate";
    IEnumerable<EnrollmentDateGroup> data =
    db.Database.SqlQuery<EnrollmentDateGroup>(query);

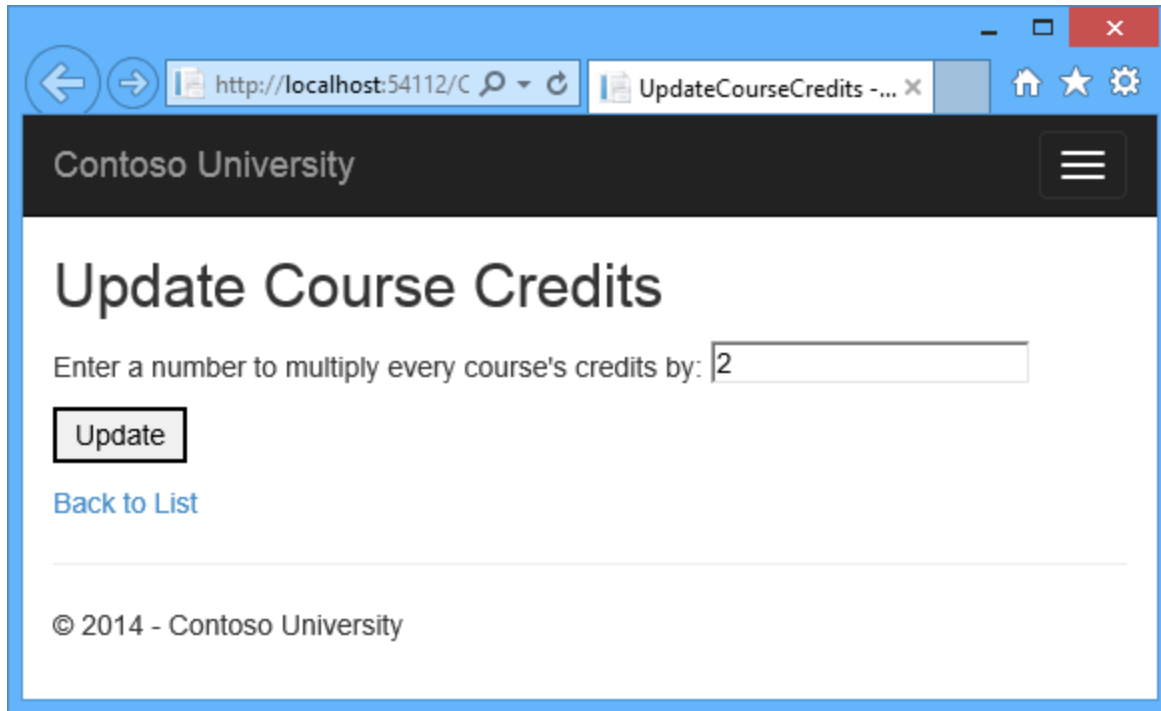
    return View(data.ToList());
}
```

Run the `About` page. It displays the same data it did before.



Calling an Update Query

Suppose Contoso University administrators want to be able to perform bulk changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL `UPDATE` statement. The web page will look like the following illustration:



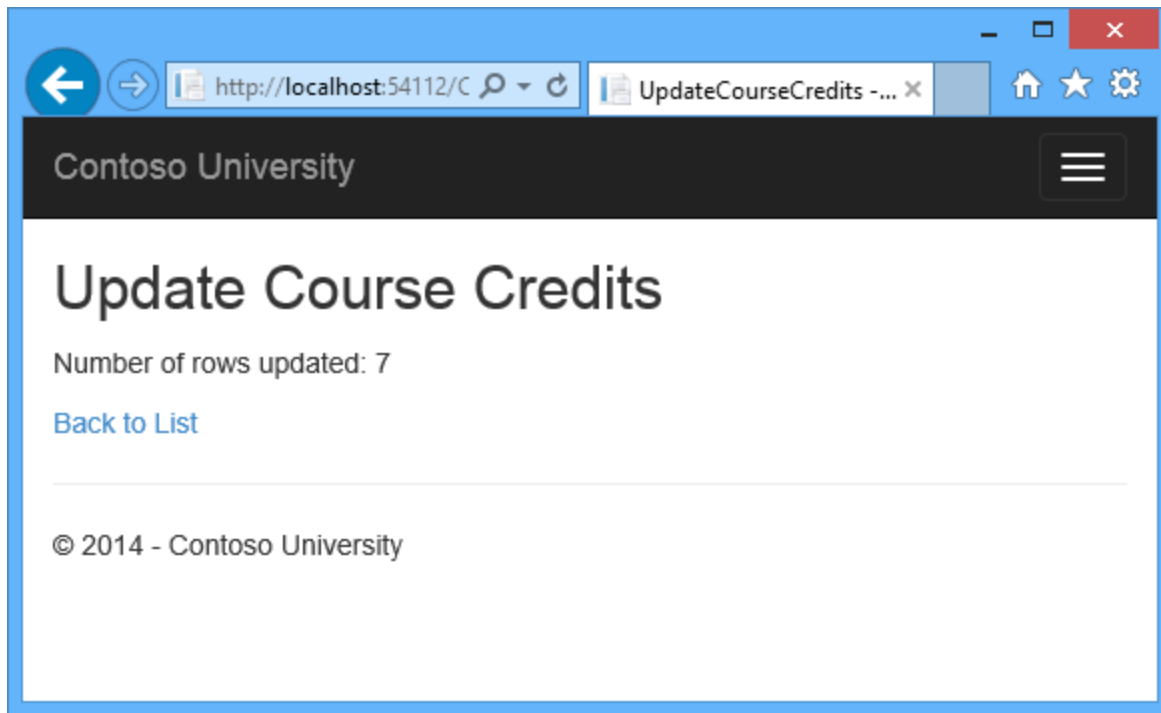
In *CourseContoller.cs*, add `UpdateCourseCredits` methods for `HttpGet` and `HttpPost`:

```
public ActionResult UpdateCourseCredits()
{
    return View();
}

[HttpPost]
public ActionResult UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewBag.RowsAffected = db.Database.ExecuteSqlCommand("UPDATE Course
SET Credits = Credits * {0}", multiplier);
    }
    return View();
}
```

When the controller processes an `HttpGet` request, nothing is returned in the `ViewBag.RowsAffected` variable, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the `HttpPost` method is called, and `multiplier` has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in the `ViewBag.RowsAffected` variable. When the view gets a value in that variable, it displays the number of rows updated instead of the text box and submit button, as shown in the following illustration:



In *CourseController.cs*, right-click one of the `UpdateCourseCredits` methods, and then click **Add**.

In `Views\Course\UpdateCourseCredits.cshtml`, replace the template code with the following code:

```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

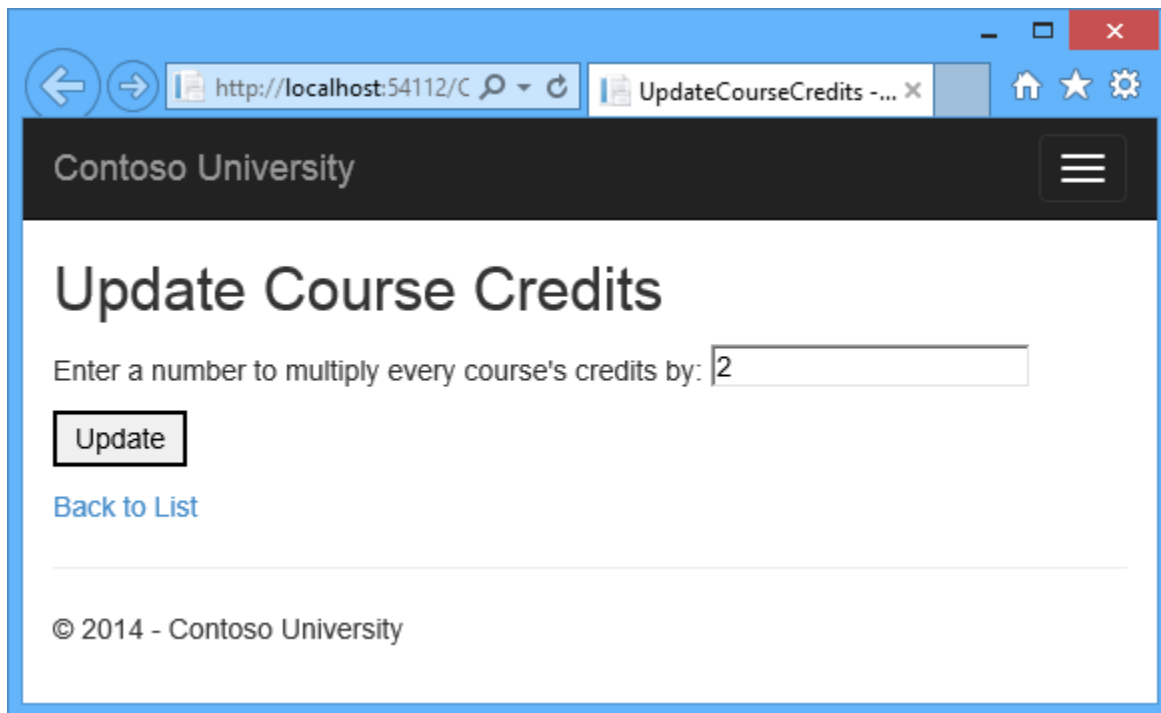
@if (ViewBag.RowsAffected == null)
{
    using (Html.BeginForm())
    {
        <p>
            Enter a number to multiply every course's credits by:
            @Html.TextBox("multiplier")
        </p>
    }
}
```

```

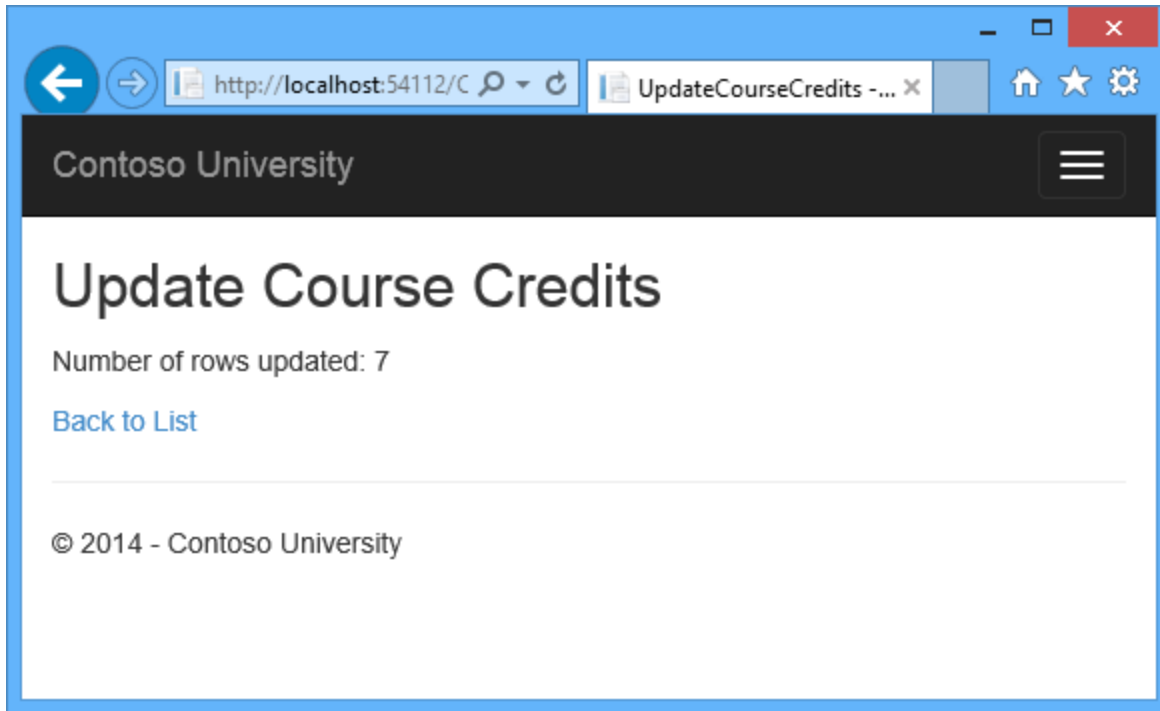
        <p>
            <input type="submit" value="Update" />
        </p>
    }
}
@if (ViewBag.RowsAffected != null)
{
    <p>
        Number of rows updated: @ViewBag.RowsAffected
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

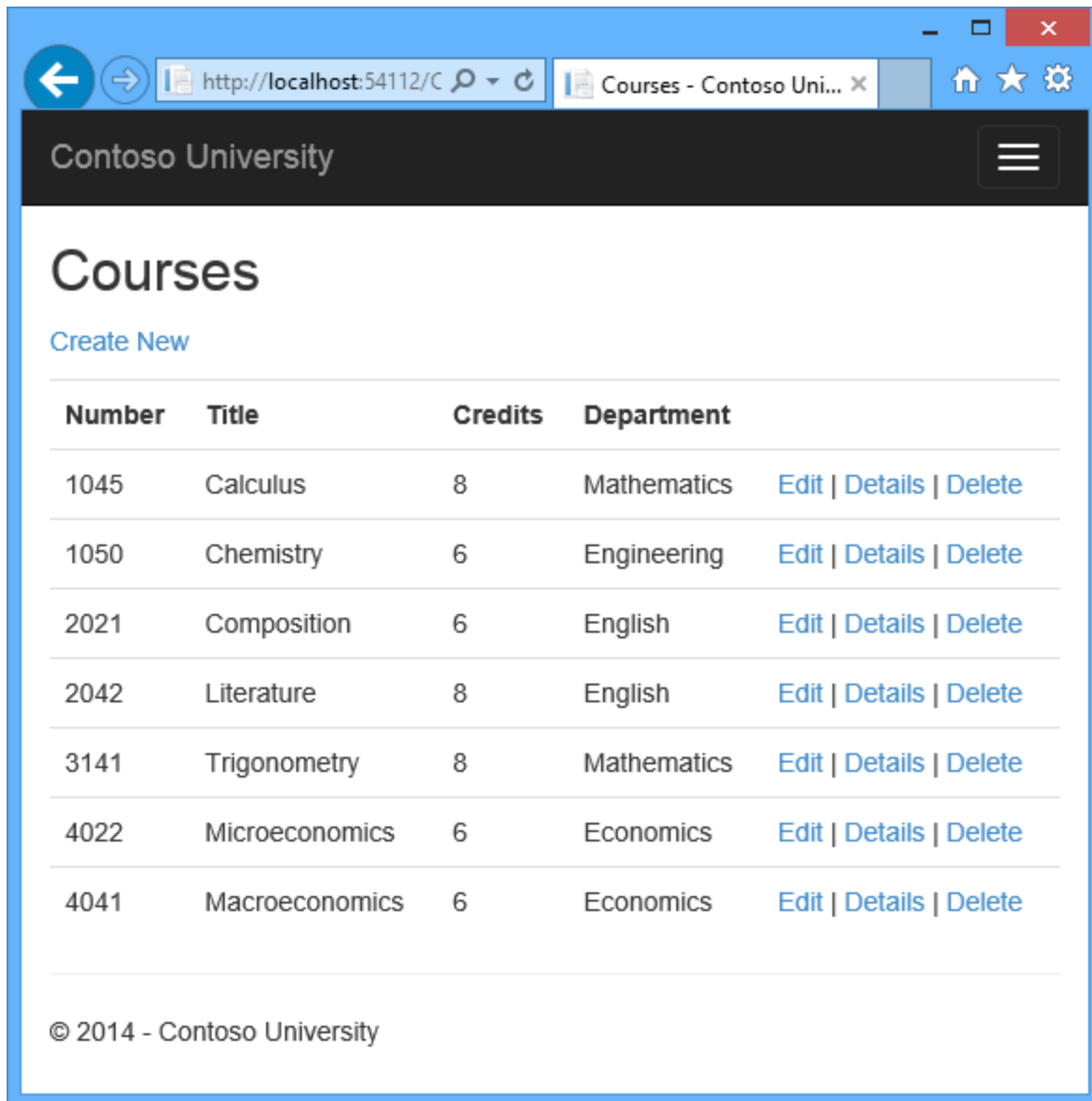
Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding `/UpdateCourseCredits` to the end of the URL in the browser's address bar (for example: `http://localhost:50205/Course/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.



For more information about raw SQL queries, see [Raw SQL Queries](#) on MSDN.

No-Tracking Queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by using the [AsNoTracking](#) method. Typical scenarios in which you might want to do that include the following:

- A query retrieves such a large volume of data that turning off tracking might noticeably enhance performance.
- You want to attach an entity in order to update it, but you earlier retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to use the `AsNoTracking` option with the earlier query.

In this section you'll implement business logic that illustrates the second of these scenarios. Specifically, you'll enforce a business rule that says that an instructor can't be the administrator of more than one department. (Depending on what you've done with the `Departments` page so far, you might already have some departments that have the same administrator. In a production application you would apply a new rule to existing data also, but for this tutorial that isn't necessary.)

In `DepartmentController.cs`, add a new method that you can call from the `Edit` and `Create` methods to make sure that no two departments have the same administrator:

```
private void ValidateOneAdministratorAssignmentPerInstructor(Department
department)
{
    if (department.InstructorID != null)
    {
        Department duplicateDepartment = db.Departments
            .Include("Administrator")
            .Where(d => d.InstructorID == department.InstructorID)
            .FirstOrDefault();
        if (duplicateDepartment != null && duplicateDepartment.DepartmentID
            != department.DepartmentID)
        {
            string errorMessage = String.Format(
                "Instructor {0} {1} is already administrator of the {2}
department.",
                duplicateDepartment.Administrator.FirstMidName,
                duplicateDepartment.Administrator.LastName,
                duplicateDepartment.Name);
            ModelState.AddModelError(string.Empty, errorMessage);
        }
    }
}
```

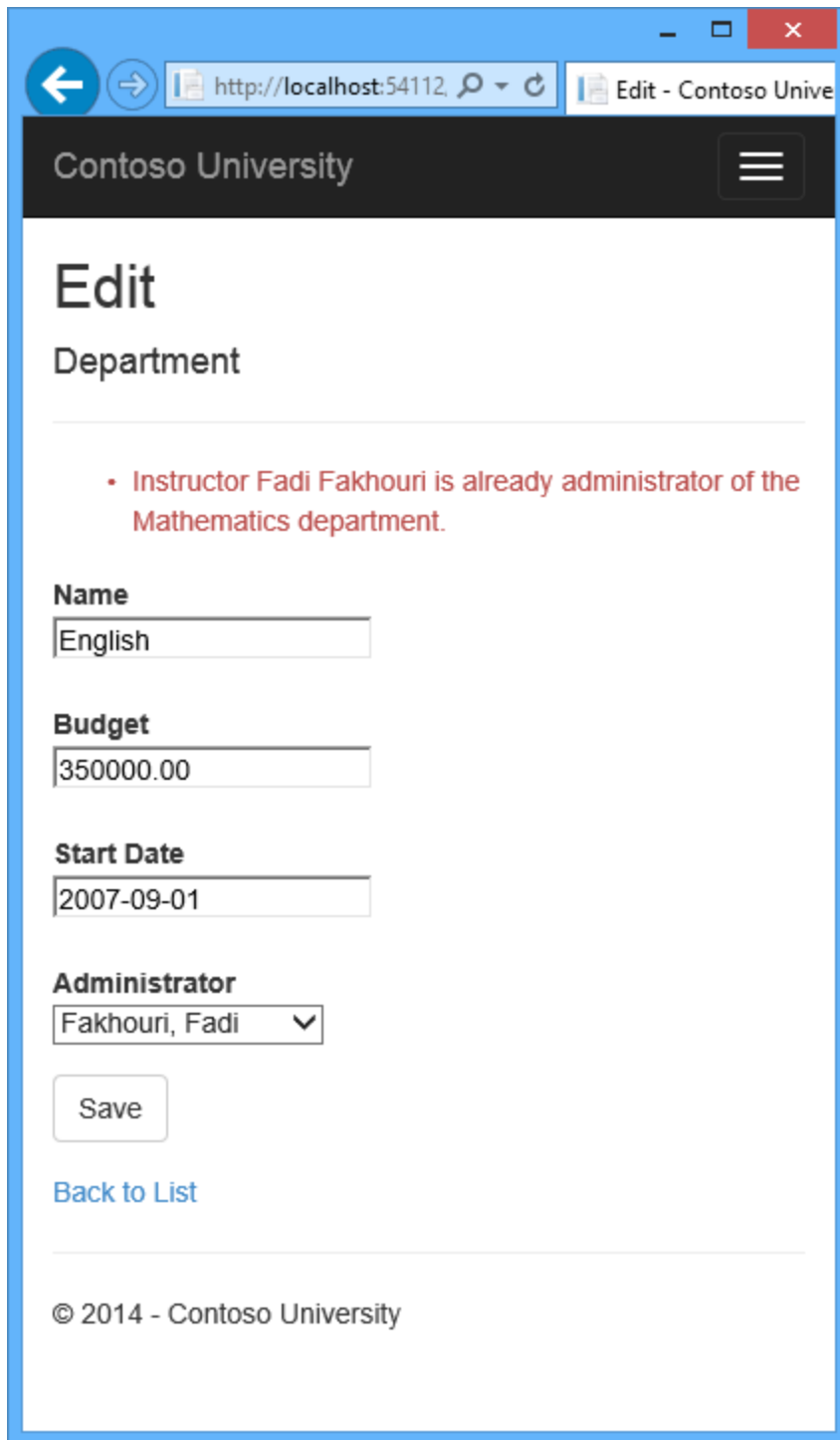
Add code in the `try` block of the `HttpPost Edit` method to call this new method if there are no validation errors. The `try` block now looks like the following example:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
    [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion,
PersonID")]
    Department department)
{
    try
    {
```

```
    if (ModelState.IsValid)
    {
        ValidateOneAdministratorAssignmentPerInstructor(department);
    }

    if (ModelState.IsValid)
    {
        db.Entry(department).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
catch (DbUpdateConcurrencyException ex)
{
    var entry = ex.Entries.Single();
    var clientValues = (Department)entry.Entity;
```

Run the Department Edit page and try to change a department's administrator to an instructor who is already the administrator of a different department. You get the expected error message:



Now run the Department Edit page again and this time change the **Budget** amount. When you click **Save**, you see an error page that results from the code you added in `ValidateOneAdministratorAssignmentPerInstructor`:

Server Error in '/' Application.

Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.InvalidOperationException: Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

Source Error:

```
Line 104:             if (ModelState.IsValid)
Line 105:             {
Line 106:                 db.Entry
(department).State = EntityState.Modified;
Line 107:                 db.SaveChanges();
Line 108:                 return RedirectToAction
("Index");
```

Source File: c:\ContosoUniversity12\ContosoUniversity\Controllers\DepartmentController.cs **Line:** 106

The exception error message is:

Attaching an entity of type 'ContosoUniversity.Models.Department' failed because another entity of the same type already has the same primary key value. This can happen when using the 'Attach' method or setting the state of an entity to 'Unchanged' or 'Modified' if any entities in the graph have conflicting key values. This may be because some entities are new and have not yet received database-generated key values. In this case use the 'Add' method or the 'Added' entity state to track the graph and then set the state of non-new entities to 'Unchanged' or 'Modified' as appropriate.

This happened because of the following sequence of events:

- The `Edit` method calls the `ValidateOneAdministratorAssignmentPerInstructor` method, which retrieves all departments that have Kim Abercrombie as their administrator. That causes the English department to be read. As a result of this read operation, the English department entity that was read from the database is now being tracked by the database context.
- The `Edit` method tries to set the `Modified` flag on the English department entity created by the MVC model binder, which implicitly causes the context to try to attach that entity. But the context can't attach the entry created by the model binder because the context is already tracking an entity for the English department.

One solution to this problem is to keep the context from tracking in-memory department entities retrieved by the validation query. There's no disadvantage to doing this, because you won't be updating this entity or reading it again in a way that would benefit from it being cached in memory.

In `DepartmentController.cs`, in the `ValidateOneAdministratorAssignmentPerInstructor` method, specify no tracking, as shown in the following:

```
Department duplicateDepartment = db.Departments
    .Include("Administrator")
    .Where(d => d.PersonID == department.PersonID)
    .AsNoTracking()
    .FirstOrDefault();
```

Repeat your attempt to edit the **Budget** amount of a department. This time the operation is successful, and the site returns as expected to the Departments Index page, showing the revised budget value.

Examining SQL sent to the database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. In an earlier tutorial you saw how to do that in interceptor code; now you'll see some ways to do it without writing interceptor code. To try this out, you'll look at a simple query and then look at what happens to it as you add options such as eager loading, filtering, and sorting.

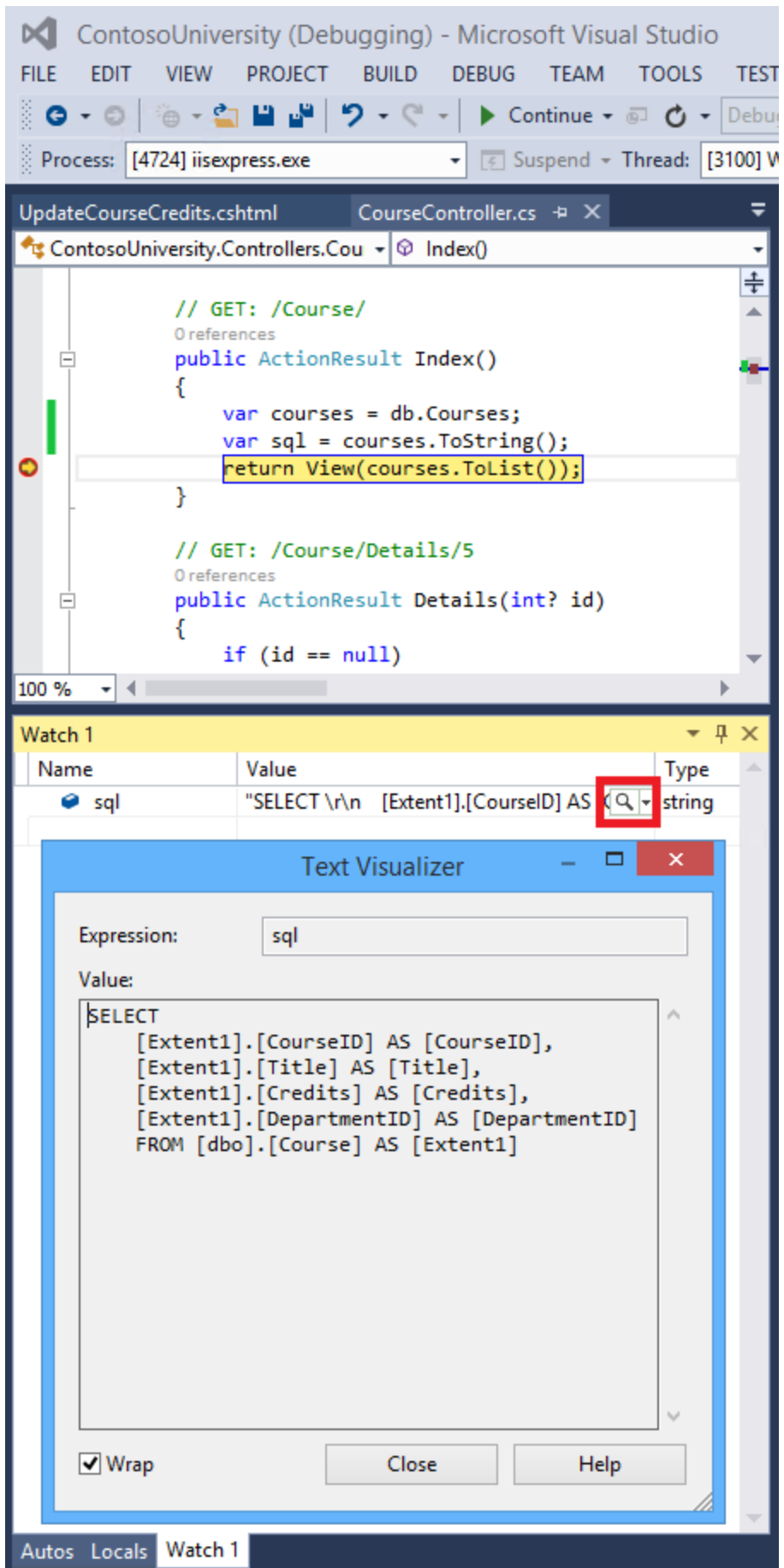
In `Controllers/CourseController`, replace the `Index` method with the following code, in order to temporarily stop eager loading:

```
public ActionResult Index()
{
    var courses = db.Courses;
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Now set a breakpoint on the `return` statement (F9 with the cursor on that line). Press F5 to run the project in debug mode, and select the Course Index page. When the code reaches the breakpoint, examine the `query` variable. You see the query that's sent to SQL Server. It's a simple `Select` statement.

```
{SELECT
[Extent1].[CourseID] AS [CourseID],
[Extent1].[Title] AS [Title],
[Extent1].[Credits] AS [Credits],
[Extent1].[DepartmentID] AS [DepartmentID]
FROM [Course] AS [Extent1]}
```

Click the magnifying glass to see the query in the **Text Visualizer**.



Now you'll add a drop-down list to the Courses Index page so that users can filter for a particular department. You'll sort the courses by title, and you'll specify eager loading for the `Department` navigation property.

In *CourseController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(int? SelectedDepartment)
{
    var departments = db.Departments.OrderBy(q => q.Name).ToList();
    ViewBag.SelectedDepartment = new SelectList(departments, "DepartmentID",
"Name", SelectedDepartment);
    int departmentID = SelectedDepartment.GetValueOrDefault();

    IQueryable<Course> courses = db.Courses
        .Where(c => !SelectedDepartment.HasValue || c.DepartmentID ==
departmentID)
        .OrderBy(d => d.CourseID)
        .Include(d => d.Department);
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Restore the breakpoint on the `return` statement.

The method receives the selected value of the drop-down list in the `SelectedDepartment` parameter. If nothing is selected, this parameter will be null.

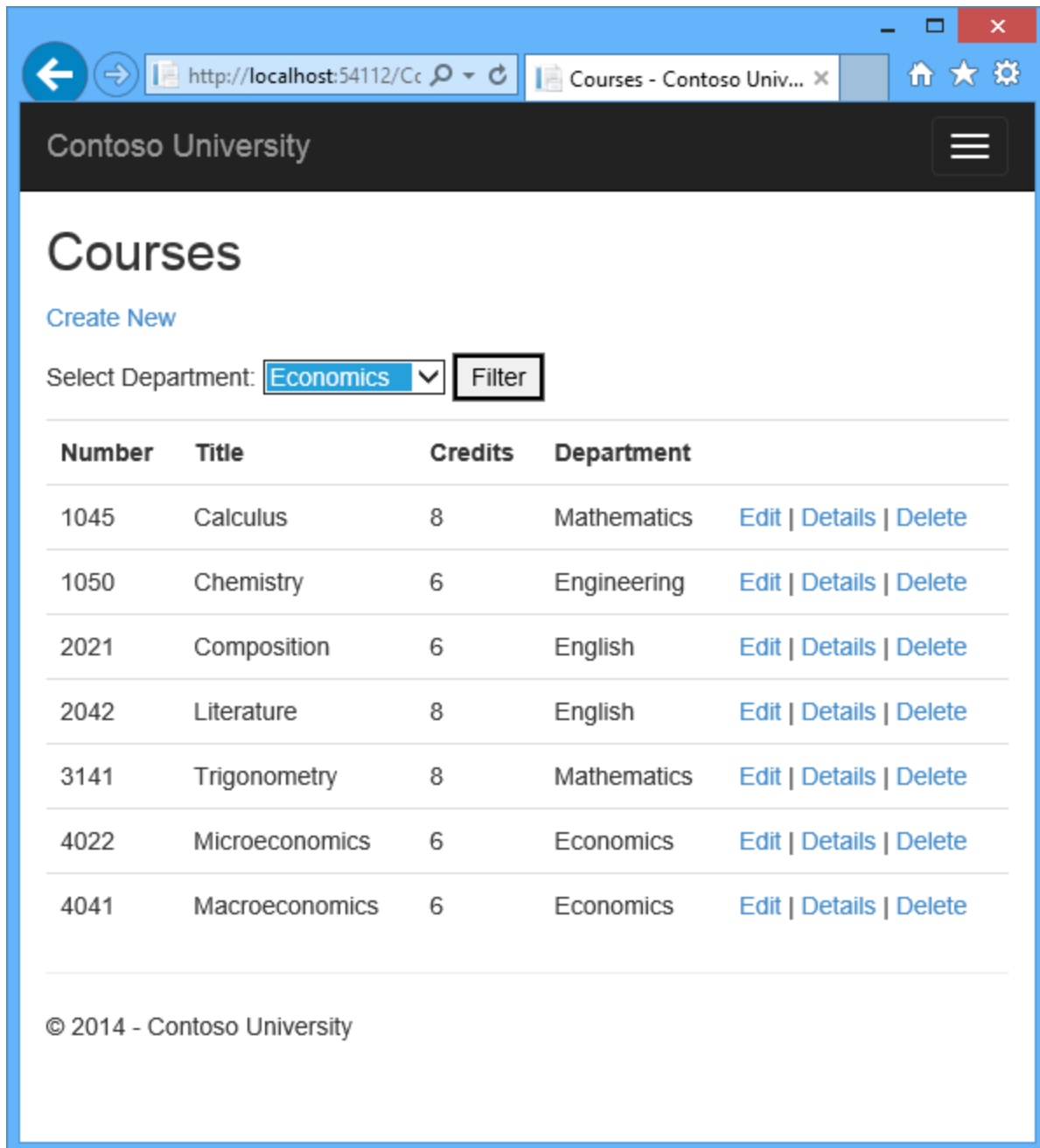
A `SelectList` collection containing all departments is passed to the view for the drop-down list. The parameters passed to the `SelectList` constructor specify the value field name, the text field name, and the selected item.

For the `Get` method of the `Course` repository, the code specifies a filter expression, a sort order, and eager loading for the `Department` navigation property. The filter expression always returns `true` if nothing is selected in the drop-down list (that is, `SelectedDepartment` is null).

In *Views\Course\Index.cshtml*, immediately before the opening `table` tag, add the following code to create the drop-down list and a submit button:

```
@using (Html.BeginForm())
{
    <p>Select Department: @Html.DropDownList("SelectedDepartment","All")
    <input type="submit" value="Filter" /></p>
}
```

With the breakpoint still set, run the Course Index page. Continue through the first times that the code hits a breakpoint, so that the page is displayed in the browser. Select a department from the drop-down list and click **Filter**:



This time the first breakpoint will be for the departments query for the drop-down list. Skip that and view the `query` variable the next time the code reaches the breakpoint in order to see what the `Course` query now looks like. You'll see something like the following:

```
SELECT
    [Project1].[CourseID] AS [CourseID],
    [Project1].[Title] AS [Title],
    [Project1].[Credits] AS [Credits],
    [Project1].[DepartmentID] AS [DepartmentID],
    [Project1].[DepartmentID1] AS [DepartmentID1],
    [Project1].[Name] AS [Name],
```

```

[Project1].[Budget] AS [Budget],
[Project1].[StartDate] AS [StartDate],
[Project1].[InstructorID] AS [InstructorID],
[Project1].[RowVersion] AS [RowVersion]
FROM ( SELECT
    [Extent1].[CourseID] AS [CourseID],
    [Extent1].[Title] AS [Title],
    [Extent1].[Credits] AS [Credits],
    [Extent1].[DepartmentID] AS [DepartmentID],
    [Extent2].[DepartmentID] AS [DepartmentID1],
    [Extent2].[Name] AS [Name],
    [Extent2].[Budget] AS [Budget],
    [Extent2].[StartDate] AS [StartDate],
    [Extent2].[InstructorID] AS [InstructorID],
    [Extent2].[RowVersion] AS [RowVersion]
FROM [dbo].[Course] AS [Extent1]
INNER JOIN [dbo].[Department] AS [Extent2] ON
[Extent1].[DepartmentID] = [Extent2].[DepartmentID]
WHERE @p__linq__0 IS NULL OR [Extent1].[DepartmentID] = @p__linq__1
) AS [Project1]
ORDER BY [Project1].[CourseID] ASC

```

You can see that the query is now a JOIN query that loads Department data along with the Course data, and that it includes a WHERE clause.

Remove the `var sql = courses.ToString()` line.

Repository and unit of work patterns

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

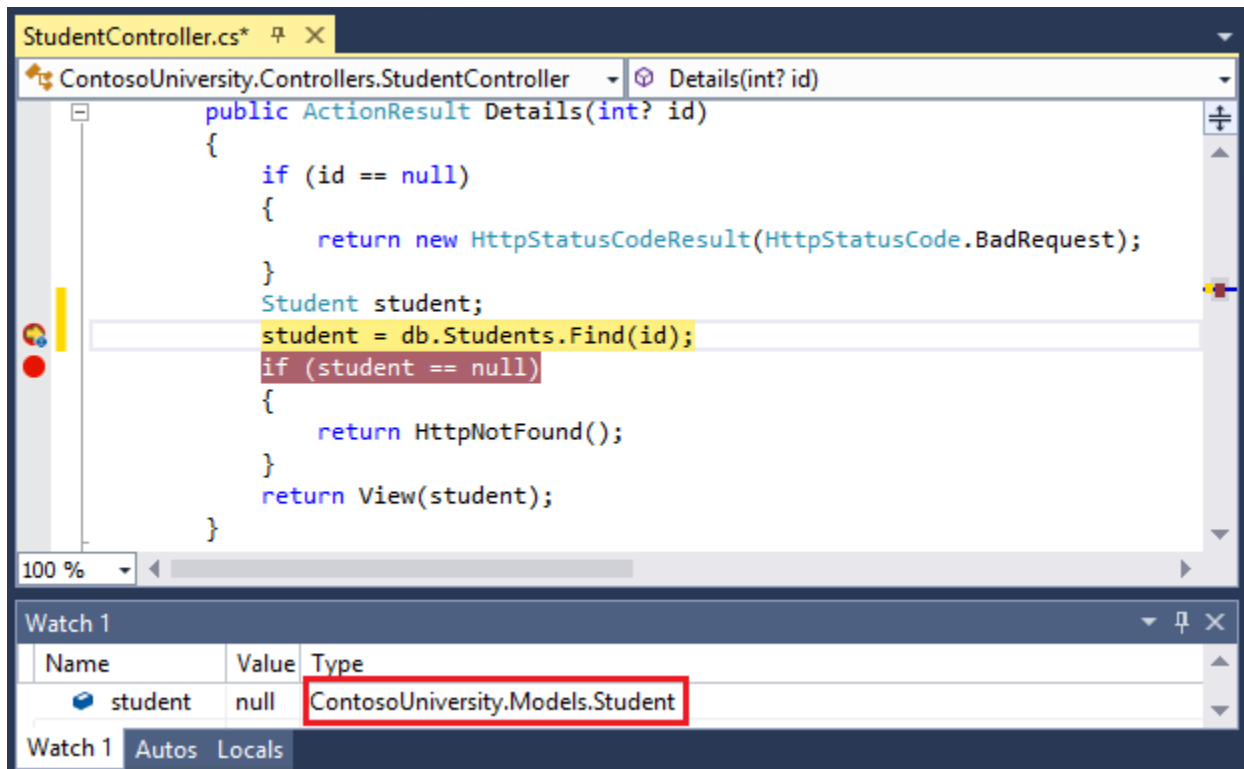
- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- Features introduced in Entity Framework 6 make it easier to implement TDD without writing repository code.

For more information about how to implement the repository and unit of work patterns, see [the Entity Framework 5 version of this tutorial series](#). For information about ways to implement TDD in Entity Framework 6, see the following resources:

- [How EF6 Enables Mocking DbSet more easily](#)
- [Testing with a mocking framework](#)
- [Testing with your own test doubles](#)

Proxy classes

When the Entity Framework creates entity instances (for example, when you execute a query), it often creates them as instances of a dynamically generated derived type that acts as a proxy for the entity. For example, see the following two debugger images. In the first image, you see that the `student` variable is the expected `Student` type immediately after you instantiate the entity. In the second image, after EF has been used to read a student entity from the database, you see the proxy class.



```
StudentController.cs* [X]
ContosoUniversity.Controllers.StudentController Details(int? id)
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Student student;
    student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}

Watch 1
Name      Value      Type
student   {System   ContosoUniversity.Models.Student {System.Data.Entity.DynamicProxies.Student_46F1}
```

This proxy class overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. One function this mechanism is used for is lazy loading.

Most of the time you don't need to be aware of this use of proxies, but there are exceptions:

- In some scenarios you might want to prevent the Entity Framework from creating proxy instances. For example, when you're serializing entities you generally want the POCO classes, not the proxy classes. One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the [Using Web API with Entity Framework](#) tutorial. Another way is to [disable proxy creation](#).
- When you instantiate an entity class using the `new` operator, you don't get a proxy instance. This means you don't get functionality such as lazy loading and automatic change tracking. This is typically okay; you generally don't need lazy loading, because you're creating a new entity that isn't in the database, and you generally don't need change tracking if you're explicitly marking the entity as `Added`. However, if you do need lazy loading and you need change tracking, you can create new entity instances with proxies using the [Create](#) method of the `DbSet` class.
- You might want to get an actual entity type from a proxy type. You can use the [GetObjectType](#) method of the `ObjectContext` class to get the actual entity type of a proxy type instance.

For more information, see [Working with Proxies](#) on MSDN.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbSet.Find`
- `DbSet.Local`
- `DbSet.Remove`
- `DbSet.Add`
- `DbSet.Attach`
- `DbContext.SaveChanges`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`

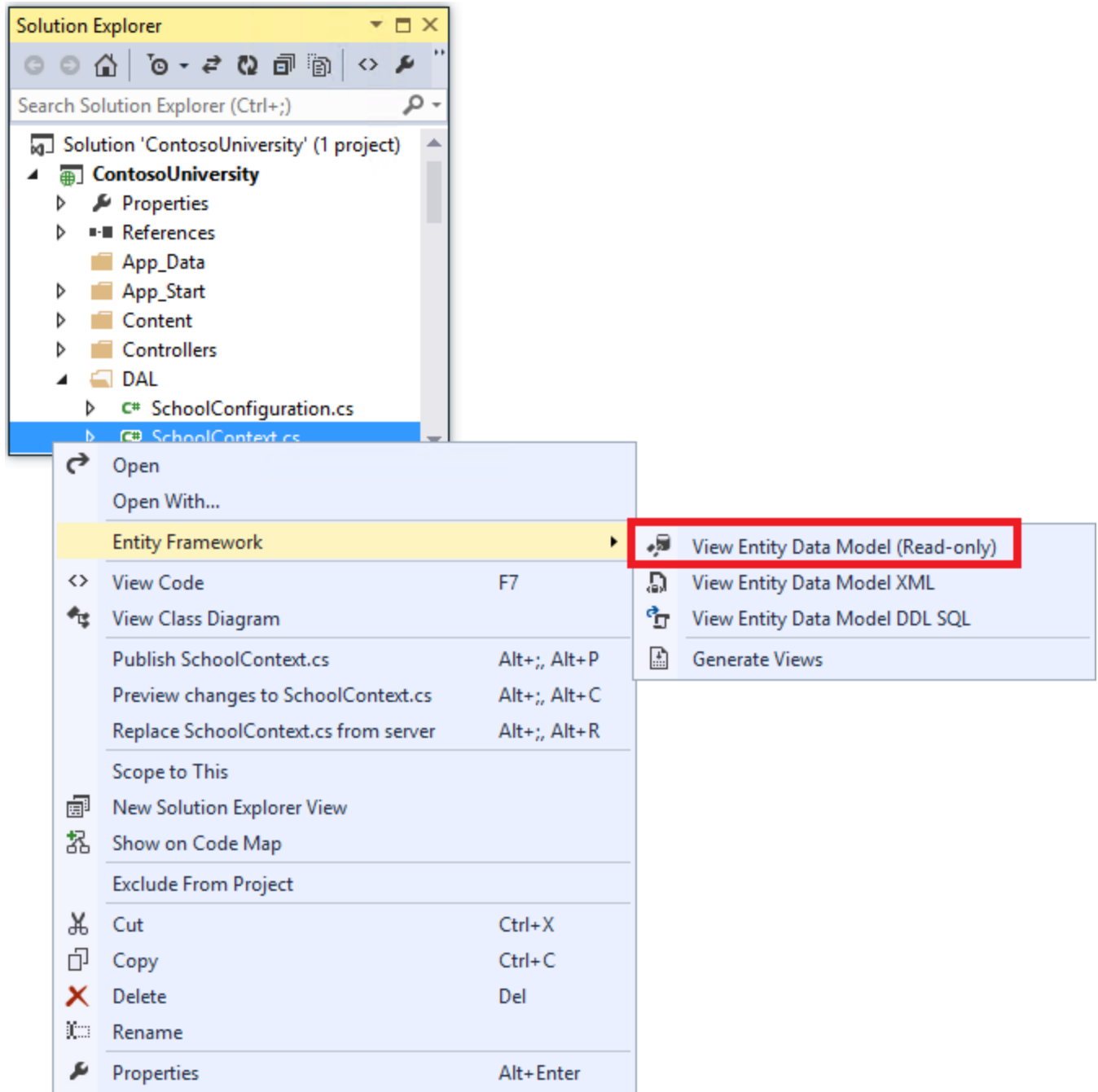
If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the [AutoDetectChangesEnabled](#) property. For more information, see [Automatically Detecting Changes](#) on MSDN.

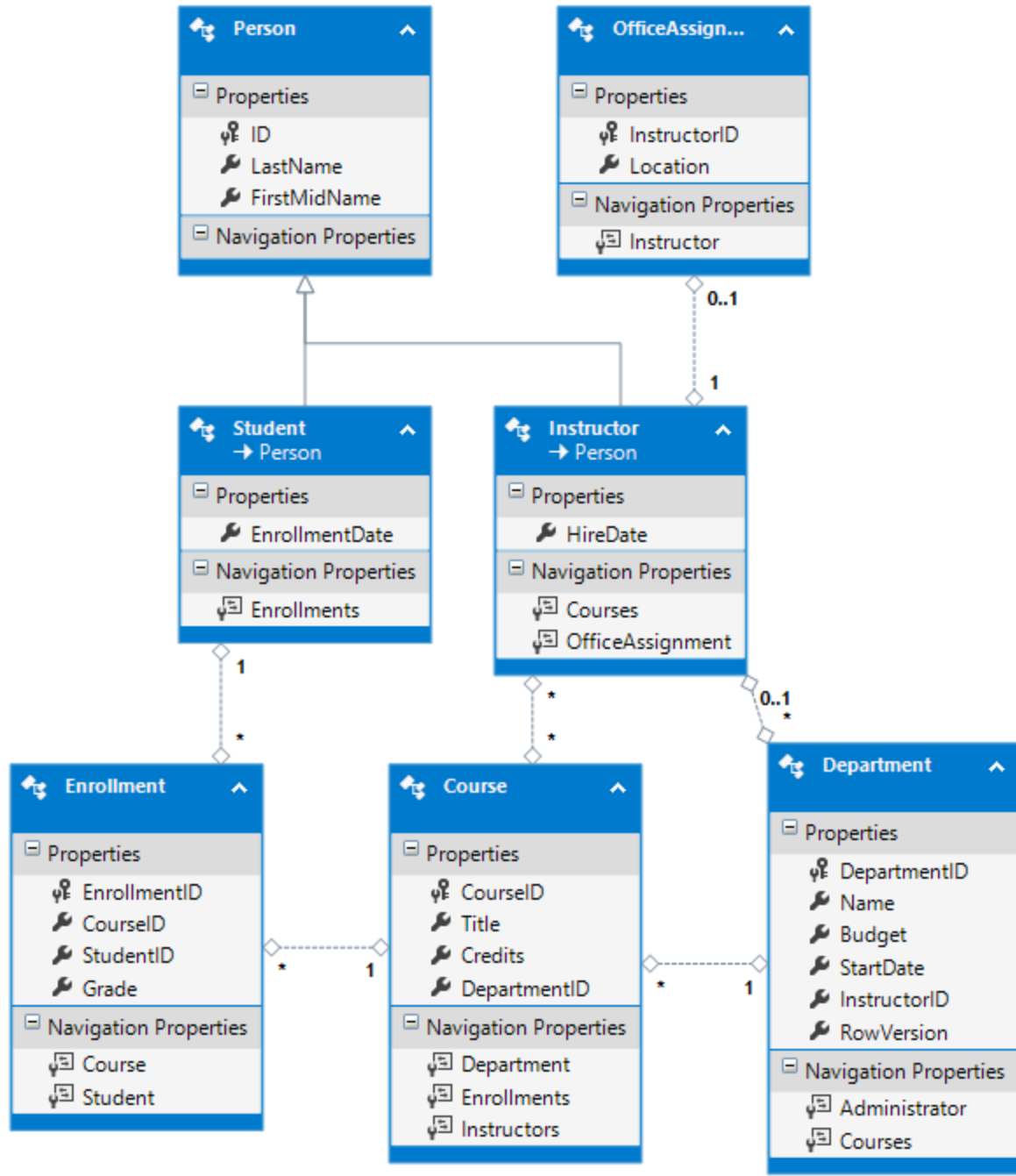
Automatic validation

When you call the `SaveChanges` method, by default the Entity Framework validates the data in all properties of all changed entities before updating the database. If you've updated a large number of entities and you've already validated the data, this work is unnecessary and you could make the process of saving the changes take less time by temporarily turning off validation. You can do that using the [ValidateOnSaveEnabled](#) property. For more information, see [Validation](#) on MSDN.

Entity Framework Power Tools

[Entity Framework Power Tools](#) is a Visual Studio add-in that was used to create the data model diagrams shown in these tutorials. The tools can also do other function such as generate entity classes based on the tables in an existing database so that you can use the database with Code First. After you install the tools, some additional options appear in context menus. For example, when you right-click your context class in **Solution Explorer**, you get an option to generate a diagram. When you're using Code First you can't change the data model in the diagram, but you can move things around to make it easier to understand.





Entity Framework source code

The source code for Entity Framework 6 is available at <http://entityframework.codeplex.com/>. Besides source code, you can get [nightly builds](#), [issue tracking](#), [feature specs](#), [design meeting notes](#), and more. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Summary

This completes this series of tutorials on using the Entity Framework in an ASP.NET MVC application. For more information about how to work with data using the Entity Framework, see the [EF documentation page on MSDN](#) and [ASP.NET Data Access - Recommended Resources](#).

For more information about how to deploy your web application after you've built it, see [ASP.NET Web Deployment - Recommended Resources](#) in the MSDN Library.

For information about other topics related to MVC, such as authentication and authorization, see the [ASP.NET MVC - Recommended Resources](#).

Acknowledgments

- Tom Dykstra wrote the original version of this tutorial, co-authored the EF 5 update, and wrote the EF 6 update. Tom is a senior programming writer on the Microsoft Web Platform and Tools Content Team.
- [Rick Anderson](#) (twitter [@RickAndMSFT](#)) did most of the work updating the tutorial for EF 5 and MVC 4 and co-authored the EF 6 update. Rick is a senior programming writer for Microsoft focusing on Azure and MVC.
- [Rowan Miller](#) and other members of the Entity Framework team assisted with code reviews and helped debug many issues with migrations that arose while we were updating the tutorial for EF 5 and EF 6.

VB

When the tutorial was originally produced for EF 4.1, we provided both C# and VB versions of the completed download project. Due to time limitations and other priorities we have not done that for this version. If you build a VB project using these tutorials and would be willing to share that with others, please let us know.

Common errors, and solutions or workarounds for them

Cannot create/shadow copy

Error Message:

Cannot create/shadow copy '<filename>' when that file already exists.

Solution

Wait a few seconds and refresh the page.

Update-Database not recognized

Error Message (from the `Update-Database` command in the PMC):

The term 'Update-Database' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

Solution

Exit Visual Studio. Reopen project and try again.

Validation failed

Error Message (from the `Update-Database` command in the PMC):

Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

Solution

One cause of this problem is validation errors when the `Seed` method runs. See [Seeding and Debugging Entity Framework \(EF\) DBs](#) for tips on debugging the `Seed` method.

HTTP 500.19 error

Error Message:

HTTP Error 500.19 - Internal Server Error

The requested page cannot be accessed because the related configuration data for the page is invalid.

Solution

One way you can get this error is from having multiple copies of the solution, each of them using the same port number. You can usually solve this problem by exiting all instances of Visual Studio, then restarting the project you're working on. If that doesn't work, try changing the port number. Right click on the project file and then click properties. Select the **Web** tab and then change the port number in the **Project Url** text box.

Error locating SQL Server instance

Error Message:

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution

Check the connection string. If you have manually deleted the database, change the name of the database in the construction string.